Performance Optimization of Multi-Tenant Software Systems

Performance Optimization of Multi-Tenant Software Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen
op maandag 14 april 2014 om 10 uur door

Cor-Paul BEZEMER

Master of Science - Informatica
geboren te Den Haag.

Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. A. van Deursen

Copromotor Dr. A.E. Zaidman

Samenstelling promotiecomissie:

Rector Magnificus	voorzitter
Prof. dr. A. van Deursen	Delft University of Technology, promotor
Dr. A.E. Zaidman	Delft University of Technology, copromotor
Prof. dr. M. Di Penta	University of Sannio in Benevento, Italy
Prof. dr. A.E. Hassan	Queen's University, Canada
Prof. dr. S. Brinkkemper	Utrecht University, The Netherlands
Prof. dr. ir. H.J. Sips	Delft University of Technology
Prof. dr. ir. M.J.T. Reinders	Delft University of Technology

This work was carried out as part of the Multi-Tenant Software (MTS) project, under the auspices of the research school IPA (Institute for Programming research and Algorithmics). This project was partially supported by Exact.

Cover: Image 'Hi-tech runner' by Steve A. Johnson.

Printed and bound in The Netherlands by Gildeprint Drukkerijen.

ISBN: 978-90-79982-16-5

It is an immutable law in business that words are words, explanations are explanations, promises are promises but only performance is reality.

– Harold S. Geneen

Acknowledgments

In 2008, I started working on my master's thesis at Exact, under the supervision of Ali Mesbah. While I never expected to pursue an academic career, Ali was able to motivate me and guided me towards my first paper publication. After finishing my master's degree at Exact, I was offered the opportunity to pursue my PhD there as well. Without Ali's supervision during my time at Exact, I probably would have never accepted this opportunity. Thank you Ali, for showing me how much fun doing research can be.

A large part of the fun is due to working with fun colleagues. I had the privilege of having colleagues from three different teams; at Exact, at SERG and during the final 8 months of my PhD, the people from PDS.

First of all, I would like to thank Exact for funding my project. Second, I would like to thank the research and EOL team for embracing me from my first days at Exact and for listening to and helping me with all my problems. Especially, I would like to thank Bart, who was always there for some laughs and listening to my rants when I needed to. Also, I would like to thank Ad, Maarten, Remko and Andre for helping me out during the final phase of my research at Exact.

I would like to thank all my colleagues from SERG for the fun times I have had on the 8^{th} floor. Especially, I would like to thank my office mates, Cuiting and Tiago, for the numerous laughs, post cards and bizarre yet hilarious discussions. I am looking forward to your PhD defenses and am hoping we will stay in touch in the future. I would also like to thank Arie, my promotor, for always offering a critical yet honest opinion. Also, thank you for keeping faith in me and including me whenever there was something official or interesting to do. Last, but definitely not least, I would like to thank Andy, my supervisor / copromotor, for his advise, support, mentorship and the tremendous amount of time he made available to supervise me. Andy, I think it is incredible how much progress I have made under 4 years of your supervision. You were always able to motivate me, especially when

it was much needed after (another) paper rejection. Thank you for keeping the faith in me. I truly enjoyed working together with you and hope that we can work together again some day.

I would like to thank my colleagues from PDS, where I have spent the final days of my PhD. Thank you Johan, for inviting me to the Tribler team and for offering me a postdoc position. I am looking forward to working with you the following years. Also, I would like to thank Elric for descending together with me to the Systemtap hell. Without your help, I would probably not have managed.

Finally, I would like to thank my family and friends, for enduring my monologues and rants about my work. I must have been terrible at times. Mom, dad, Rinze and Nelske, thank you for always supporting me and pushing me to make the best of myself. I love you. My dear Nadia, thank you for always being there for me. The years I have spent with you have been the best years of my life so far and you were right, they have also been the most productive. I love you and I cannot wait to find out what the future will bring us.

Contents

List of Acronyms

CRM Customer Relationship Management

EOL Exact Online

ERP Enterprise Resource Planning

MT Multi-Tenancy/Multi-Tenant

P2P Peer-to-Peer

PIO Performance Improvement Opportunity

RUBiS Rice University Bidding System

SaaS Software-as-a-Service

SC Similarity Coefficient

SFL Spectrum-based Fault Localization

SARatio Slow-to-All-actions-ratio

SLA Service Level Agreement

SLO Service Level Objective

SME Small and Medium Enterprises

1 Introduction

Over the last years, the Internet and its usage have exploded. Reports show that in 2012, 2.4 billion people[1] were using the Internet. In synergy with its popularity, improvements to hardware and bandwidth have been made possible. Amongst other things, these improvements have led to a shift in the way software vendors are offering their products [Mertz et al., 2010].

In the traditional setting, customers buy an application to run on their computer (*on-premise software*). As a result, they pay a relatively large fee for the software product. In addition, they often have expensive maintenance contracts with the software vendor [Dubey and Wagle, 2007].

Improved Internet facilities, and opportunities such as Internet connections on mobile devices, have led to a new licensing scheme, in which the customer uses an application over the Internet through a lightweight client that runs in a web browser without installation. In this so-called *Software-as-a-Service (SaaS)* scheme, customers rent a service from the software vendor. The service delivered by the software vendor consists of hosting and maintaining the software [Kaplan, 2007]. Instead of paying a large fee for the software installation, customers now pay a relatively small monthly fee to the vendor for using the application.

Despite the smaller fee, customers still have the same requirements for their software [Kwok et al., 2008; Bezemer and Zaidman, 2010]:

- **The software should offer the same functionality as if it were running on their own computer** - The customer expects to be able to perform tasks such as saving his work, importing and exporting data and printing reports.
- **The software should offer the same user experience as the desktop version would have** - The customer expects aspects such as security and performance to be at least at the same level as for desktop software.
- **The software should offer the same degree of customizability as the desktop version would have** - The customer wants to be able to customize the software to his needs; this may include simple cosmetic customizations

[1] `www.internetworldstats.com/stats.htm` (last visited: October 22, 2013)

such as theming, but also more complex customization such as the workflow of the application.

An additional characteristic of SaaS is that customers tend to be less loyal[2], due to the low costs of changing vendors. In order to be able to offer software which fulfills these requirements, at a minimal price – so that customers are not attracted by other, cheaper vendors –, vendors must optimize their operational costs. One opportunity for lowering the operational costs is to minimize the hardware used per customer. As a result, methods have been developed for sharing (hardware) resources between customers. These methods can be roughly divided in two groups: the multi-instant and the multi-tenant approaches [Bezemer and Zaidman, 2010; Chong et al., 2006].

The multi-instant approaches, such as virtualization, run several copies (*instances*) of the same application on a server. These approaches provide each customer with an isolated part of the server, on which the application runs. As a result, virtualization imposes a much lower limit on the number of customers per server due to the high memory requirements for every virtual server [Li et al., 2008]. Additionally, it introduces a maintenance challenge, as all instances must be changed in case of a software update.

Multi-tenant approaches try to overcome these challenges by letting customers share the same application and database instance [Kwok et al., 2008]. As a result, only one application and database have to be maintained. Because of the high number of customers that share the same resources in a multi-tenant setting, *performance* is essential. As explained above, customers should not be affected by other customers on the shared resources. In order to let tenants share resources, without negatively affecting their perceived performance, it is necessary to optimize multi-tenant applications as much as possible. Performance optimization can be done at various levels:

- **At the hardware level** - By searching for possible improvements in the infrastructure and task schedule, server or other hardware bottlenecks can be found.
- **At the software level** - By searching for possible improvements in the code, bottlenecks in algorithms or resource hogs can be found.
- **At a combination of the hardware and software level** - By implementing hardware-specific versions of software, hardware-specific optimizations can be made.

[2] `http://businessoverbroadway.com/creating-loyal-customers-for-saas-companies-video`

In this research, we will investigate methods for the first two levels of performance optimization. In our work, we do not address optimization of a combination of these two levels. The main reason for this is that we are interested in finding techniques which are agnostic to the type of hardware used in the system. While this does allow us to propose techniques for finding hardware bottlenecks, it does not allow us to propose hardware-specific optimizations, as this would require a different technique for every type of hardware. In the next section, we will first present our research questions.

1.1 Research Questions

In many situations, performance optimization is done manually by a small team of performance experts. Often, these experts have been with the development team for a long time and have deep knowledge of the application to optimize.

As multi-tenant applications are a relatively new breed of software, knowledge about the applications and infrastructure may be more limited. Semi-automated analysis tools can help provide insight in the application and infrastructure. In addition, they can guide and accelerate the performance optimization process.

The goal of the research presented in this dissertation is to investigate semi-automated methods which assist the performance expert in optimizing the performance of a multi-tenant application. First, we investigate whether multi-tenancy increases the challenge of performance optimization, in comparison to optimizing traditional applications. To get a better understanding of the differences between traditional single-tenant and multi-tenant software which could lead to such an increase, we first focus on the following research question:

> ***RQ1**: What are the differences between a single-tenant and a multi-tenant system?*

In Chapter 2, we will investigate these differences, and analyze the challenges introduced by multi-tenancy. We will do this by converting a single-tenant application into a multi-tenant application. From this process, we will get a better understanding of the consequences of multi-tenancy for performance optimization. With this understanding, we can focus on the following research questions:

> ***RQ2**: How can we assist developers with the optimization of the performance of a multi-tenant system with regard to its hardware?*

We divide RQ2 into three subquestions. To be able to optimize the performance of a system with regard to its hardware, we must be able to detect which hardware components form the bottleneck of the system:

> ***RQ2a**: How can we detect and analyze hardware bottlenecks?*

In Chapter 3, we present an approach which assists performance experts by giving a diagnosis that contains a description of the detected bottlenecks. The next challenge is to find an appropriate method for reporting or visualizing this diagnosis, so that performance experts can quickly interpret the analysis results.

> ***RQ2b**: How can we report and visualize the diagnosis of the bottleneck component(s)?*

In Chapter 4, we present WEDJAT, our open source tool for visualizing the diagnosis given by our bottleneck detection approach.

The level of assistance a performance expert gets from our approach depends on the quality of the diagnosis given. Therefore, we investigate how we can improve the quality of this diagnosis in Chapter 5, in which we focus on the following research question:

> ***RQ2c**: How can we improve the quality of the diagnosis?*

In addition to optimization at the hardware level, a multi-tenant application must be optimized at the software level. In Chapter 6, we present an approach for detecting and analyzing performance regressions. We show that our approach can assist the developer by guiding the performance optimization process. In Chapter 6, we focus on the following research question:

> ***RQ3**: How can we assist developers with the optimization of the performance of a multi-tenant system with regard to its software?*

1.2 Research Context

The research described in this dissertation was done in collaboration with two partners. The first, Exact[3], is a Dutch-based software company, which specializes in enterprise resource planning (ERP), customer relationship management (CRM) and financial administration software. Exact has over 1900 employees working in more than 20 countries. Founded in 1984, Exact has over 25 years of experience in multi-user client/server software and web applications. Since several years, Exact has also been offering a successful multi-tenant Software-as-a-Service solution, called Exact Online[4] (EOL).

The second part of our research was done in collaboration with the Tribler team [Pouwelse et al., 2008], part of the Parallel and Distributed Systems Group[5] at Delft University of Technology. Tribler is a fully decentralized peer-to-peer (P2P)

[3] `http://www.exact.com`
[4] `http://www.exactonline.nl`
[5] `http://www.pds.ewi.tudelft.nl/`

client which allows users to stream video directly via the BitTorrent protocol. Tribler has been in development since 2006 and has received over 1 million downloads since.

1.3 Research Methodology

As explained in the previous section, our research was done in close collaboration with industrial partners. As a result, we followed the 'industry-as-a-laboratory' approach [Potts, 1993] for our research. In this approach, researchers work closely together with industry to identify real problems and construct and evaluate solutions.

To answer our research questions, we have performed a number of case studies in collaboration with our partners. These case studies had the following in common:

- They were either based on real industrial data, or data generated by a well-established benchmark
- The results were evaluated by, or together with performance experts from the team that developed the subject system

We feel it is important for the advancement of research in general to have access to prototypes and implementations of methods described in research papers. Therefore, we have made two of our research prototypes, Wedjat[6] and Spectrapperf[7], available as open source projects. The implementation of our other research projects could not be made available due to the closed source nature of the analyzed projects, especially the data.

1.4 Contributions

In this section we will outline the main contributions of this thesis. Two contributions correspond to multi-tenant systems in general and three correspond to optimizing performance. Our contributions focus on the following aspects:

Multi-Tenant Systems

1. Overview of challenges of developing and maintaining multi-tenant systems.

2. A reengineering pattern for transforming a single-tenant to a multi-tenant system.

In Chapter 2, we give an overview of the challenges of developing and maintaining multi-tenant systems, in contrast to the development of single-tenant systems. In

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

[7] `https://github.com/tribler/gumby`

addition, we present a case study in which we apply our reengineering pattern for transforming an existing single-tenant application into a multi-tenant one on a research prototype.

Performance Optimization

1. An approach for detecting and analyzing performance improvement opportunities at the hardware level.

In Chapter 3, we present an approach which assists the performance expert during the process of finding and diagnosing hardware bottlenecks. We evaluate this approach in two case studies, one of which on real industrial data. We present our technique for improving the diagnostic capabilities of this approach in Chapter 5.

2. An approach for using heat maps to analyze the performance of a system and to find performance improvement opportunities.

In Chapter 4, we extend our approach with the possibility to analyze the performance of a system using heat maps. This resulted in an open source tool called WEDJAT, which was evaluated in a field user study with performance experts from industry.

3. An approach for detecting and analyzing performance improvement opportunities and performance regressions at the software level.

In Chapter 6, we present an approach which assists the performance expert during the process of finding and diagnosing performance regressions. In addition, we show how this approach can be used to find software bottlenecks and guide the performance optimization process. We evaluate this approach in a case study on an open source project.

1.5 Thesis Outline

The outline of this thesis is as follows. Chapter 2 covers our research on the differences between single-tenant and multi-tenant software. In Chapter 3, we present our approach for detecting and analyzing performance improvement opportunities. In Chapter 4 we present our approach for performance optimization with the assistance of heat maps. In this chapter, we also present our open source tool for performance visualization called WEDJAT. In Chapter 5, we discuss our technique for improving the diagnostic capabilities of the approach presented in Chapter 3. In Chapter 6, we discuss our approach for optimizing an application at the software level and we present our open source implementation called SPECTRAPERF. Chapter 7 presents our final conclusions and discusses directions for future work.

1.6 Origin of Chapters

Each of the chapters in this thesis has been published before as, or is based on a peer-reviewed publication or technical report. Therefore, these chapters are mostly self-contained and may contain some redundancy. The following list gives an overview of these publications:

Chapter 2 is based on our papers published in the *26th International Conference on Software Maintenance (ICSM'10)* [Bezemer et al., 2010] and in the *Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution (IWPSE'10)* [Bezemer and Zaidman, 2010].

Chapter 3 contains our work published in the *Journal of Systems and Software* [Bezemer and Zaidman, 2014].

Chapter 4 contains our work published in the proceedings of the *28th International Conference on Software Maintenance (ICSM'12)* [Bezemer et al., 2012].

Chapter 5 contains our work published as technical report *TUD-SERG-2013-015* [Bezemer and Zaidman, 2013].

Chapter 6 contains our work which is submitted for journal publication [Bezemer et al., 2013].

2

Multi-Tenancy

Multi-tenancy is a relatively new software architecture principle in the realm of the Software-as-a-Service (SaaS) business model. It allows to make full use of the economy of scale, as multiple customers – "tenants" – share the same application and database instance. All the while, the tenants enjoy a highly configurable application, making it appear that the application is deployed on a dedicated server. The major benefits of multi-tenancy are increased utilization of hardware resources and improved ease of maintenance, resulting in lower overall application costs, making the technology attractive for service providers targeting small and medium enterprises (SME). Therefore, migrating existing single-tenant to multi-tenant applications can be interesting for SaaS software companies. However, as this chapter advocates, a wrong architectural choice might entail that multi-tenancy becomes a maintenance nightmare. In this chapter we report on our experiences with reengineering an existing industrial, single-tenant software system into a multi- tenant one using a lightweight reengineering approach.[1]

Software-as-a-Service (SaaS) represents a novel paradigm and business model expressing the fact that companies do not have to purchase and maintain their own ICT infrastructure, but instead, acquire the services embodied by software from a third party. The customers subscribe to the software and underlying ICT infrastructure (service on-demand) and require only Internet access to use the services. The

[1]This chapter is based on our papers published in the *26th International Conference on Software Maintenance (ICSM'10)* [Bezemer et al., 2010] and in the *Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution (IWPSE'10)* [Bezemer and Zaidman, 2010].

service provider offers the software service and maintains the application [Kaplan, 2007]. However, in order for the service provider to make full use of the economy of scale, the service should be hosted following a multi-tenant model [Kwok et al., 2008].

Multi-tenancy is an architectural pattern in which a single instance of the software is run on the service provider's infrastructure, and multiple tenants access the same instance. In contrast to the multi-user model, multi-tenancy requires customizing the single instance according to the multi-faceted requirements of many tenants [Kwok et al., 2008]. The multi-tenant model also contrasts the multi-instance model, in which each tenant gets his own (virtualized) instance of the application [Chong et al., 2006].

The benefits of the multi-tenant model are twofold. On the one hand, application deployment becomes easier for the service provider, as only one application instance has to be deployed, instead of hundreds or thousands. On the other hand, the utilization rate of the hardware can be improved, as multiple tenants share the same hardware resources. These two factors make it possible to reduce the overall costs of the application and this makes multi-tenant applications especially interesting for customers in the small and medium enterprise (SME) segment of the market, as they often have limited financial resources and do not need the computational power of a dedicated server.

Because of these benefits, many organizations working with SaaS technology are currently looking into transforming their single-tenant applications into multi-tenant ones. Yet, two barriers are perceived in the adoption of multi-tenant software systems, namely:

- Companies are wary of the initial start-up costs of reengineering their existing single-tenant software systems into multi-tenant software systems [Tsai et al., 2007].
- Software maintainers are worried that multi-tenancy might introduce additional maintenance problems stemming from the fact that these new systems should be highly configurable, in the process effectively eliminating the perceived maintenance advantage that multi-tenancy offers through the fact that updates only have to be deployed and applied once.

This is where this chapter aims to contribute, by providing an overview of challenges and difficulties that software developers and maintainers are likely to face when reengineering and maintaining multi-tenant software applications. To come to this overview, we focus on the following research question presented in Chapter 1:

> ***RQ1**: What are the differences between a single-tenant and a multi-tenant system?*

In addition, we aim to show that migrating from a single-tenant setup to a multi-tenant one can be done (1) easily, in a cost-effective way, (2) transparently for the end-user and (3) with little effect for the developer, as the adaptations are confined to small portions of the system, creating no urgent need to retrain all developers. More specifically, our chapter contains the following contributions:

1. A clear, non-ambiguous definition of a multi-tenant application.
2. An overview of the challenges of developing and maintaining scalable, multi-tenant software.
3. A conceptual blueprint of a multi-tenant architecture that isolates the multi-tenant concern as much as possible from the base code.
4. A case study of applying this approach to an industrial application.

This chapter is further organized as follows. In the next section, we give a definition of multi-tenancy and discuss its benefits and related work. In Section 2.2, we discuss the challenges of multi-tenancy. In Section 2.3, we present our conceptual blueprint of a multi-tenant architecture. In Section 2.4, we describe the industrial target application which we migrated using this pattern. The actual case study is dealt with in Section 2.5. We then discuss our findings and their threats to validity in Section 2.6. Section 2.7 presents our conclusions and ideas for future work.

2.1 Multi-Tenancy

Multi-tenancy is an organizational approach for SaaS applications. Although SaaS is primarily perceived as a business model, its introduction has lead to numerous interesting problems and research in software engineering. Despite the growing body of research in this area, multi-tenancy is still relatively unexplored, despite the fact the concept of multi-tenancy first came to light around 2005[2].

While a number of definitions of a multi-tenant application exist [Warfield, 2007; Weissman and Bobrowski, 2009], they remain quite vague. Therefore, we define a multi-tenant application as the following:

Definition 1. *A **multi-tenant** application lets customers (**tenants**) share the same hardware resources, by offering them one shared application and database instance, while allowing them to configure the application to fit their needs as if it runs on a dedicated environment.*

Definition 2. *A **tenant** is the organizational entity which rents a SaaS application. Typically, a tenant groups a number of users, which are the stakeholders in the organization.*

[2]The Wikipedia entry was first created on November 14th, 2005; `http://en.wikipedia.org/wiki/Multitenancy`.

These definitions focus on what we believe to be the key aspects of multi-tenancy:

1. The ability of the application to share hardware resources [Wang et al., 2008; Warfield, 2007].
2. The offering of a high degree of configurability of the software [Nitu, 2009; Jansen et al., 2010; Müller et al., 2009].
3. The architectural approach in which the tenants (or users) make use of a single application and database instance[3] [Kwok et al., 2008].

In the next two sections (2.1.1 and 2.1.2), we will demarcate multi-tenancy from two other organizational models, namely the multi-user and the multi-instance model. In Section 2.1.3, we will elaborate on the key aspects of multi-tenancy.

2.1.1 Multi-Tenant versus Multi-User

It is necessary to make an important, but subtle distinction between the concepts *multi-tenant* and *multi-user*. In a multi-user application we assume all users are using the same application with limited configuration options. In a multi-tenant application, we assume each tenant has the possibility to heavily configure the application. This results in the situation that, although tenants are using the same building blocks in their configuration, the appearance or workflow of the application may be different for two tenants. An additional argument for the distinction is that the Service Level Agreement (SLA) of each tenant can differ [Lin et al., 2009], while this is usually not the case for users in a multi-user system.

2.1.2 Multi-Tenant versus Multi-Instance

Another contrasting approach is the multi-instance approach, in which each tenants gets his own instance of the application (and possibly also of the database). With the gaining in popularity of virtualization technology and cloud computing, the multi-instance approach is the "easier" way of creating multi-tenant like applications from a development perspective. Yet, the multi-instance approach is better suited if the number of tenants is likely to remain low [Guo et al., 2007], in part because the multi-instance model suffers from an increased maintenance cost. This increased maintenance cost can be attributed to the effort for deploying updates to numerous instances of the application.

A special type of multi-instance software is product line software [van Gurp et al., 2001]. Product line software is created using a set of common shared components. In addition to these common components, instance-specific components

[3] Due to performance and/or legislative reasons, there might be more than one instance, but the number of instances should remain limited.

can be created and added to introduce customization. In the case of dynamic product line software [Hallsteinsen et al., 2008], these instance-specific components can be loaded at runtime. Dynamic product line software can be considered equal to multi-tenant software.

2.1.3 Key Characteristics of Multi-Tenancy

Hardware Resource Sharing

In traditional single-tenant software development, tenants usually have their own (virtual) server. This set-up is similar to the traditional Application Service Provider (ASP) model [Mietzner et al., 2009a]. However, in the SME segment, server utilization in such a model is low. By placing several tenants on the same server, the server utilization can be improved [Wang et al., 2008; Warfield, 2007]. While this can also be achieved through virtualization, virtualization imposes a much lower limit on the number of tenants per server due to the high memory requirements for every virtual server [Li et al., 2008]. Higher utilization of the existing servers will result in lower overall costs of the application, as the total amount of hardware required is lower.

The concept of multi-tenancy comes in different flavours, and depending on which flavour is implemented, the utilization rate of the underlying hardware can be maximized. The following variants of (semi-)multi-tenancy can be distinguished [Chong et al., 2006; Kwok et al., 2008]:

1. Shared application, separate database.
2. Shared application, shared database, separate table.
3. Shared application, shared table (*pure multi-tenancy*).

Throughout this chapter, we will assume the pure multi-tenancy variant is being used, as this variant allows the highest number of tenants per server [Chong et al., 2006; Wang et al., 2008].

High Degree of Configurability

In a single-tenant environment, every tenant has his own, (possibly) customized application instance. In contrast, in a multi-tenant setup, all tenants share the same application instance, although it must appear to them as if they are using a dedicated one. Because of this, a key requirement of multi-tenant applications is the possibility to configure and/or customize the application to a tenant's need, just like in single-tenancy [Mietzner et al., 2009a]. In single-tenant software customization is often done by creating branches in the development tree. In multi-tenancy this is no longer possible and configuration options must be integrated in the product design instead [Nitu, 2009], similar to software product line engineering [Mietzner et al., 2009a].

Because of the high degree of configurability of multi-tenant software systems, it may be necessary to run multiple versions of an application (or parts of an application) next to each other. This situation might arise for reasons of backward compatibility or in situations where the legislation in a particular country changes. Because it is deemed undesirable to deploy different instances of a multi-tenant application, version support should be an integral part of a multi-tenant setup.

Shared Application and Database Instance

A single-tenant application may have many running instances and they may all be different from each other because of customization. In multi-tenancy, these differences no longer exist as the application is runtime configurable.

This entails that in multi-tenancy the overall number of instances will clearly be much lower (ideally it will be one, but the application may be replicated for scalability purposes). As a consequence, deployment is much easier and cheaper, particularly in the area of deploying the updates, as a the number of instances which are touched by the deployment action are clearly much lower.

In addition, new data aggregation opportunities are opened because all tenant data is in the same place. For example, user behaviour traces can be collected much easier, which can help to improve the user experience.

2.1.4 Benefits

From the previous paragraphs a number of reasons for companies to introduce multi-tenancy can be deducted:

1. Higher utilization of hardware resources.
2. Easier and cheaper application maintenance.
3. Lower overall costs, allowing to offer a service at a lower price than competitors.
4. New data aggregation opportunities.

2.1.5 Related Work

Even though SaaS is an extensively researched topic, multi-tenancy has not received a large deal of attention yet in academic software engineering research. A number of researchers [Chong et al., 2006; Guo et al., 2007; Kwok et al., 2008] have described the possible variants of multi-tenancy, as we have described in Section 2.1.3. Wang et al. [Wang et al., 2008] have evaluated these variants for different numbers of tenants and make recommendations on the best multi-tenant variant to use, based on the number of tenants, the number of users and the amount of data per tenant.

Kwok et al. [Kwok et al., 2008] have described a case study of developing a multi-tenant application, in which they emphasize the importance of configurability. This importance is emphasized by Nitu [Nitu, 2009] and Mietzner et al. [Mietzner et al., 2009a] as well.

Guo et al. [Guo et al., 2007] have proposed a framework for multi-tenant application development and management. They believe the main challenge of multi-tenancy is tenant isolation, and therefore their framework contains mainly components for tenant isolation, e.g., data, performance and security isolation. We believe tenant isolation forms a relatively small part of the challenges of multi-tenancy, which is why our chapter focuses on different aspects.

The native support of current database management systems (DBMSs) for multi-tenancy was investigated by Jacobs and Aulbach [Jacobs and Aulbach, 2007]. In their position paper on multi-tenant capable DBMSs, they conclude that existing DBMSs are not capable of natively dealing with multi-tenancy. Chong et al. [Chong et al., 2006] have described a number of possible database patterns, which support the implementation of multi-tenancy, specifically for Microsoft SQL Server.

One problem in multi-tenant data management is tenant placement. Kwok et al. [Kwok and Mohindra, 2008] have developed a method for selecting the best database in which a new tenant should be placed, while keeping the remaining database space as flexible as possible for placing other new tenants.

Finally, Salesforce, an industrial pioneer of multi-tenancy, has given an insight on how multi-tenancy is being handled in their application framework [Weissman and Bobrowski, 2009].

Most research in the field of reengineering in the area of "service-oriented software systems" has focused on approaches to migrate, port and wrap legacy assets to web services. Two notable examples in this context are the works of Sneed and Canfora et al. Sneed reports on an approach to wrap legacy code behind an XML shell [Sneed, 2006]. Sneed's approach allows individual legacy functions to be offered as web services to any external user. The approach has been applied successfully to the integration of both COBOL and C++ programs in a service-oriented system. Canfora et al. presented an approach to migrate form-based software systems to a service [Canfora et al., 2008a]. The approach provides a wrapper that encapsulates the original user interface and interacts with the legacy system which runs within an application server.

We are currently not aware of any research that investigates the reengineering of the first generation of service-oriented systems, an area that we believe to be an important one, as many of the first generation service-based systems have carried over some of the flaws from the systems from which they originate. In particular, we are not aware of any multi-tenancy reengineering strategies.

2.2 Challenges

Unfortunately, multi-tenancy also has its challenges and even though some of these challenges exist for single-tenant software as well, they appear in a different form and are more complex to solve for multi-tenant applications. In this section, we will list the challenges and discuss their specificity with regard to multi-tenancy.

2.2.1 Performance

Because multiple tenants share the same resources and hardware utilization is higher on average, we must make sure that all tenants can consume these resources as required. If one tenant clogs up resources, the performance of all other tenants may be compromised. This is different from the single-tenant situation, in which the behaviour of a tenant only affects himself. In a virtualized-instances situation this problem is solved by assigning an equal amount of resources to each instance (or tenant) [Li et al., 2008]. This solution may lead to very inefficient utilization of resources and is therefore undesirable in a pure multi-tenant system.

2.2.2 Scalability

Because all tenants share the same application and datastore, scalability is more of an issue than in single-tenant applications. We assume a tenant does not require more than one application and database server, which is usually the case in the SME segment. In the multi-tenant situation this assumption cannot help us, as such a limitation does not exist when placing multiple tenants on one server. In addition, tenants from a wide variety of countries may use an application, which can have impact on scalability requirements. Each country may have its own legislation on, e.g., data placement or routing. An example is the European Union's (EU) legislation on the storage of electronic invoicing, which states that electronic invoices sent from within the EU must be stored within the EU as well[4]. Finally, there may be more constraints such as the requirement to place all data for one tenant on the same server to speed up regularly used database queries. Such constraints strongly influence the way in which an application and its datastore can be scaled.

2.2.3 Security

Although the level of security should be high in a single-tenant environment, the risk of, e.g., data stealing is relatively small. In a multi-tenant environment, a security breach can result in the exposure of data to other, possibly competitive, tenants. This makes security issues such as data protection [Guo et al., 2007] very important.

[4] `http://ec.europa.eu/taxation_customs/taxation/vat/traders/invoicing_rules/article_1733_en.htm` (last visited on January 24, 2014)

2.2.4 Zero-Downtime

Introducing new tenants or adapting to changing business requirements of existing tenants brings along the need for constant growth and evolution of a multi-tenant system. However, adaptations should not interfere with the services provided to the other existing tenants. This induces the strong requirement of zero-downtime for multi-tenant software, as downtime per hour can go up to $4,500K depending on the type of business [Ganek and Corbi, 2003].

2.2.5 Maintenance

In the typical evolutionary cycle of software, a challenge is formed by maintenance, e.g. adapting the software system to changing requirements and its subsequent deployment [Jansen et al., 2005]. While it is clear that the multi-tenant paradigm can bring serious benefits for deployment by minimizing the number of application and database instances that need to be updated, the situation for the actual maintenance is not so clear. In particular, introducing multi-tenancy into a software systems will add complexity, which will likely affect the maintenance process. Further research is needed to evaluate whether the hardware and deployment benefits outweigh the increased cost of maintenance.

2.3 Multi-Tenancy Reengineering Pattern

When we started thinking how multi-tenancy affects an application, we came up with the reengineering pattern depicted by Figure 2.1. Here we see that multi-tenancy affects almost all layers of a typical application, and as such, there is high potential for multi-tenancy to become a *cross-cutting concern*. To keep the impact on the code (complexity) low, the implementation of multi-tenant components should be separated from single-tenant logic as much as possible. If not, maintenance can become a nightmare because:

- Mixing multi-tenant with single-tenant code must be done in all application layers, which requires all developers to be reeducated about multi-tenancy.
- Mixing multi-tenant with single-tenant code leads to increased code complexity because it is more difficult to keep track of where multi-tenant code is introduced.

These two problems can be overcome by carefully integrating multi-tenancy in the architecture. The primary goals of our reengineering pattern are the following:

1. Migrate a single-tenant to a multi-tenant application with *minor* adjustments in the existing business logic.
2. Let application developers be unaware of the fact that the application is multi-tenant.

3. Clearly separate multi-tenant components, so that monitoring and load balancing mechanisms can be integrated in the future.

In order to reach our goals, our reengineering pattern requires the insertion of three components in the target application. The remainder of this section will explain the importance and the requirements of each of these components.

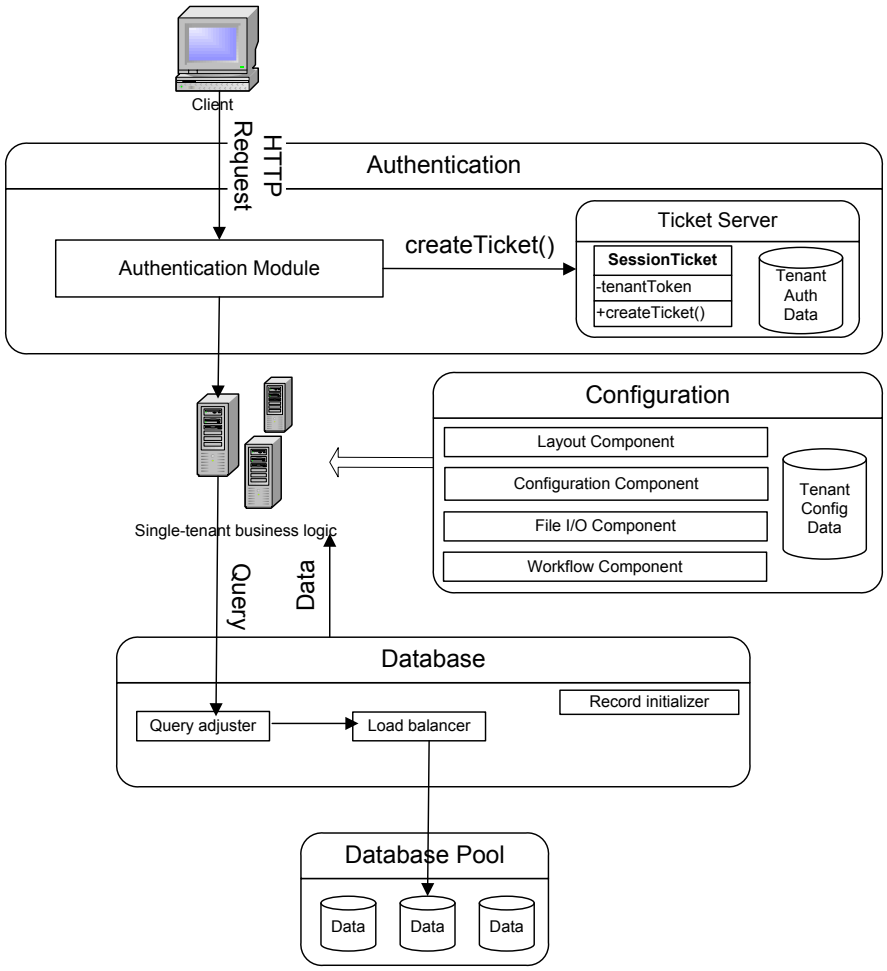

Figure 2.1: Architectural overview for multi-tenancy

2.3.1 Authentication

Motivation. Because a multi-tenant application has one application and database instance, all tenants use the same physical environment. In order to be able to offer customization of this environment and to make sure that tenants can only access their own data, tenants must be authenticated. While user authentication is possibly already present in the target application, a separate tenant-specific authentication mechanism might be required, for two reasons: (1) it is usually much

easier to introduce an additional authentication mechanism, then to change the existing one, and (2) tenant authentication allows a single user to be part of more than one logical organization, which extends the idea of user authentication with "groups". A typical example of such a situation would be a bookkeeper, who works for multiple organizations.

Implementation. The authentication component provides the mechanism required to identify a tenant throughout the application, by generating a session ticket after a tenant successfully logs in. The correct application configuration is loaded based on the values in this ticket. Note that this mechanism does not interfere with the authentication logic of the single-tenant application, which means that any security measures implemented in this logic are still in order.

2.3.2 Configuration

Motivation. In a single-tenant environment, every tenant has his own, (possibly) customized application instance. In multi-tenancy, all tenants share the same application instance, although it must appear to them as if they are using a dedicated one. Because of this, a key requirement of multi-tenant applications is the possibility to configure and/or customize the application to a tenant's need [Jansen et al., 2010].

In single-tenant software, customization is often done by creating branches in the development tree. In multi-tenancy this is no longer possible and customization must be made possible through configuration [Nitu, 2009].

Implementation. In order to enable multi-tenancy and let the user have a user-experience as if he were working in a dedicated environment, it is necessary to allow at least the following types of configuration:

Layout Style

Layout style configuration allows the use of tenant-specific themes and styles.

General Configuration

The general configuration component allows the specification of tenant-specific configuration, encryption key settings and personal profile details.

File I/O

The file I/O configuration component allows the specification of tenant-specific file paths, which can be used for, e.g., report generation.

Workflow

The workflow configuration component allows the configuration of tenant-specific workflows. An example of an application in which workflow configuration is required is an ERP application, in which the workflow of requests can vary significantly for different tenants.

2.3.3 Database

Motivation. Because all tenants use the same database instance, it is necessary to make sure that they can only access their own data. In addition, it is necessary to make sure that metrics such as a usage limit for each tenant can be verified.

Implementation. Current off-the-shelf DBMSs are not capable of dealing with multi-tenancy themselves [Jacobs and Aulbach, 2007]. An example of missing functionality is an administrative panel, which provides access to tenant-specific data such as the amount of data used. In addition, developers should be aware that the application is multi-tenant and adjust their database queries accordingly. In our reengineering pattern, the latter is hidden from the developer and should be done in a layer between the business logic and the application's database pool. The main tasks of this layer are as follows:

Creation of new tenants in the database

If the application stores and/or retrieves data, which can be made tenant-specific, in/from a database, it is the task of the database layer to create the corresponding database records when a new tenant has signed up for the application.

Query adaptation

In order to provide adequate data isolation, the database layer must make sure that all queries are adjusted so that each tenant can only access his own records.

Load balancing

To improve the performance of the multi-tenant application, efficient load balancing is required for the database pool. Any Service Level Agreements (SLAs) [Li et al., 2008; Malek et al., 2012] or financial data legislation should be taken into account.

2.4 Exact Codename

Exact[5] is a Dutch-based software company, which specializes in enterprise resource planning (ERP), customer relationship management (CRM) and financial administration software. Exact has over 2200 employees working in more than 40 countries. Founded in 1984, Exact has over 25 years of experience in multi-user client/server software and web applications. Since several years, Exact has also been offering a successful multi-tenant SaaS solution.

Multi-tenancy is an attractive concept for Exact because they target the SME segment of the market. By having the opportunity to share resources between customers, services can be offered to the customers at a lower overall price. In addition, maintenance becomes easier — and thus cheaper — as less different instances must be maintained. While Exact has experience with multi-tenancy,

[5] `http://www.exact.com`

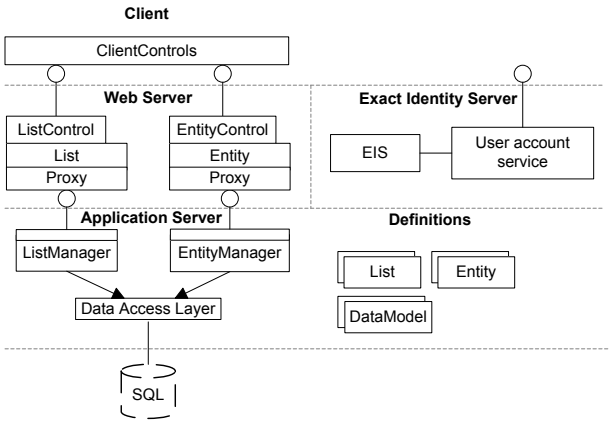

Figure 2.2: Architecture of Exact Codename

they also have existing single-tenant applications that they want to transform into multi-tenant ones.

One of these applications is a research prototype, dubbed Exact Codename. Codename is a proof of concept, single-tenant widget framework that offers the possibility of creating software solutions using widgets as building blocks. The Exact research team has been working for 4 years on Codename and it is the intention to integrate parts of Codename in commercial Exact products in the short to medium term future.

Codename is being developed in C# and ASP.NET and consists of approximately 165K lines of code. Figure 2.2 depicts the (simplified) architecture of Codename.

2.4.1 Architecture of Codename

Codename is built upon two major concepts, the *List* and the *Entity*. A list represents a list of data, such as a list of documents. An entity represents an object, such as News (a news item).

An entity and a list are described using a domain specific language and the descriptions are currently stored in definition files. These definitions are stored separately from the framework code, which allows them to be edited by non-technical domain experts. Such a definition file may contain details about how to retrieve the entity or list from the database, or behaviour. For example, the definition of News contains details on how a News item can be found in the database, and it also tells us that News is a type of Document (which is itself an entity). The default HTML layout of an entity or list is also stored in a (separate) definition file.

Because an entity or list can be created using a definition file only, it is easy for domain experts to add new or edit existing entities or lists.

On the application server, the *ListManager* and *EntityManager* can be used to instantiate a new list or entity. When a new list or entity is created, these man-

agers read the corresponding definition file and generate the required object. All database access is done through the *Data Access Layer*. To allow the use of multiple data sources, possibly in different formats, logical names are used for database columns or tables rather than the physical names. In the Data Access Layer, these logical names are translated to physical names (using the *DataModel* definitions).

The web server communicates with the application server using Windows Communication Foundation (WCF) services and a proxy. The goal of the web server is to generate HTML and JavaScript (JS) representations of the lists and entities for the client. A client can request a list or entity using the *ListControl* or *EntityControl* web services. The client can only retrieve data from or write data to the database using these two services.

2.4.2 Exact Identity Server

A separate component in Codename's architecture is the Exact Identity Server (EIS), which is an implementation of the Microsoft Identity Foundation. In the EIS a token is generated when a tenant successfully logs in to the system. This (encrypted) token contains enough information to identify the tenant throughout the system without contacting the EIS again. This allows single sign-on (SSO) for multiple Exact applications (*relying parties*). The protocol used to do this is SAML 1.1. A token contains several claims, such as the Globally Unique Identifier (GUID) of the user which is logged in. The EIS offers a User Account Service as well, which allows relying parties to add their own users to the EIS.

2.5 Case Study: Codename^{MT}

In this section, we present our case study of enabling multi-tenancy in a single-tenant application using the multi-tenancy reengineering pattern that we discussed in Section 2.3. Our target application is Codename, of which we gave an overview in Section 2.4.

2.5.1 Motivation

In addition to the general advantages of multi-tenancy (Section 2.1), being able to reengineer existing single-tenant applications into multi-tenant ones is interesting for a number of reasons:

1. Existing business logic can be reused with minor adaptations.

2. As our reengineering pattern is lightweight and requires minor adaptations only, most developers will not be aware of the fact that the application is multi-tenant, which means that not all developers need to be trained in multi-tenancy.

3. Lessons learned from applying a pattern may lead to improvements in the architecture of existing multi-tenant products.

2.5.2 Applying the Multi-Tenancy Pattern

In our case study, we will apply our multi-tenancy reengineering pattern to Codename, resulting in a multi-tenant application Codename^{MT}. For transforming Codename into Codename^{MT}, we are introducing the components that we have explained in Section 2.3 into Codename.

Authentication

As identifying to which tenant a user belongs can be done using the tenant's ID only, the existing authentication mechanism could easily be extended. We added Codename^{MT} to the EIS as a relying party, so that we could add users for this application to EIS. After this, we extended the Codename `User` object with a `TenantID` property, which is read from the token after a user successfully logs in. Because the User object is globally available throughout Codename, the `TenantID` is available globally as well. Note that EIS does not keep track of tenant details other than the `TenantID`. Currently this is the task of the relying party.

After our adaptations, an EIS token for the Codename^{MT} application contains a `GUID` and a `TenantID`. The `TenantID` is used to identify the tenant to which the owner of the token belongs. The `GUID` is used to identify the user within Codename^{MT}. Note that the user identification process is unchanged compared to the process in Codename, leaving any values like security levels intact.

Configuration

While applying the pattern to the single-tenant configuration, we limited our case study to the degree of configuration currently possible in Codename. In contrast to the pattern, Codename^{MT} stores all configuration data in the application database, rather than in a separate database.

Layout Style In Codename, the layout style of the application is managed by the following:

- ASP.NET master pages
- ASP.NET themes

The .NET framework offers the possibility to dynamically change these by attaching an event early in the page lifecycle. We have adapted the global.asax[6] file of the application with the code depicted in Figure 2.3, which loads the tenant-specific style for each page request.

[6] In ASP.NET, the (optional) global.asax file is used to access session and application-level events.

```
// attach event
protected void Application_PreRequestHandlerExecute(
 object s, EventArgs e){
   Page p = this.Context.Handler as Page;
   p.PreInit += new EventHandler(page_PreInit);
}

// set tenant-specific theme and master page
protected void page_PreInit(object s, EventArgs e){
   Page p = this.Context.Handler as Page;
   p.Theme = TenantContext.GetTenantTheme();
   p.MasterPageFile = TenantContext.GetTenantMasterpage();
}
```

Figure 2.3: Dynamically setting the tenant-specific style

General Configuration All general configuration, e.g. profile settings, in Codename is stored in the database. This means that making the configuration tables multi-tenant also makes the general configuration multi-tenant.

File I/O The only file I/O used in Codename is to load the definition files on the application server. Originally these definition files were loaded from the `xmd/list` and `xmd/entity` directories. We have adapted this code to check if one of the directories `xmd/tenantID/list` or `xmd/tenantID/entity` contain the requested file. If it exists, the tenant-specific file is loaded, otherwise, a default file is loaded. We have implemented this mechanism to allow tenants to decide whether they want to configure their own lists and entities or use the defaults. Codename also implements a caching system for definition files, which we have adapted to be aware of the existence of tenant-specific definitions.

Workflow In Codename, the application workflow can currently only be configured by physically changing the .aspx page, which describes the process, so that it uses the required library. While tenant-specific workflow configuration using this approach was included in the case study design, the implementation remains future work.

Codename uses a URL rewriting mechanism to allow application users to request URLs which contain less machine code (*friendly URLs*). This leads to better readable URLs such as `docs/person/corpaul` instead of `?page=person&id={12345-abcde-890}`. By altering this rewriting module to load a tenant-specific .aspx page, workflow configuration can be established.

Table 2.1: Multi-tenant query extensions for TenantID '123'

Type of query	Query extension
SELECT	Add Filter('TenantID', 123)
JOIN	Add Filter('TenantID', 123)
UPDATE	Add Filter('TenantID', 123)
DELETE	Add Filter('TenantID', 123)
INSERT	Add Field('TenantID', 123)

Database

All database queries in Codename are generated using the Data Access Layer, so that metadata stored in the data model definitions can always be used during query generation. Because all queries are created in one component, automatically extending them to use the TenantID is straightforward. To prevent unnecessary duplication of data, we added the property IsMultiTenant to the data model. Setting this property to false indicates that data in the table is not tenant-specific, such as country ISO codes or postal shipping rates. This allows us to generate more efficient queries. We added a TenantID column to the tables that were specified as multi-tenant.

After this, we adapted the module which generates the query. For each queried table, the table metadata is retrieved from the data model to see whether the table contains tenant-specific data. If this is the case, the query is extended using the extensions depicted in Table 2.1. Note that for all subqueries and each JOIN clause in a SELECT query, the same occurs. In the Data Access Layer, a Filter adds a criterion to the WHERE clause of a query and a Field adds a column update to the SET clause of a query.

Future work regarding the database component includes adding usage of the TenantID to indexes on tables that contain multi-tenant data.

In this case study, we did not implement automatic creation of new tenants in the database. We plan on doing this when the signup process is linked with the EIS User Account Service. In addition, we did not implement load balancing. This is a very difficult task due to the number and complexity of constraints in financial software, e.g., because of the legislation of several countries on where financial data may be stored. An important requirement is that the Data Access Layer should hide load balancing from the developer. Load balancing in a multi-tenant application will be addressed in future research.

2.5.3 Evaluation

For testing whether our reengineering pattern that transformed Codename into Codename^{MT} did not break any of the major functionalities in Codename, we fol-

lowed a double approach using code reviews and manual tests. As such, we performed a code review together with the third author of this chapter, one of the lead architects of the Exact research team. Furthermore, we created a list of the most important functionality of Codename and manually tested that this functionality still worked correctly in Codename^{MT}. While we consider manual testing to be sufficient for this particular case study, amongst others due to the support from Exact, we do acknowledge that automated testing is a necessity, which is why we aim to investigate an automated test methodology for multi-tenant applications in future research.

For the actual testing of Codename^{MT} we first added two test users with different `TenantID`s on the EIS. Then we created tenant-specific themes and master pages and verified that they were loaded correctly after logging the test users in. After this, we created a number of tenant-specific definition files and verified that the correct ones (including default files) were loaded.

To test the database component, we have assigned different documents to each test user and verified the correct ones were shown in document listings after logging in. In addition, we have verified that queries were extended correctly by manually inspecting a random subset of queries taken from a SQL Server Profiler trace, recorded during usage of the application.

Our double approach where we combined code reviews and manual tests to verify whether Codename^{MT} did not break any of the major functionality from Codename yielded no reports of any faults.

2.6 Lessons Learned & Discussion

In this chapter we have applied our reengineering pattern that guides the reengineering of single-tenant applications into multi-tenant ones and we report on our experiences with the reengineering pattern in an industrial environment. We will now touch upon some of the key lessons that we have learned when applying our reengineering pattern.

2.6.1 Lessons learned

Lightweight reengineering approach We have applied our multi-tenancy reengineering pattern by extending the original Codename code with approximately 100 lines of code, thus transforming it into Codename^{MT}. This shows that our pattern can assist in carrying out the reengineering process in an efficient way, with relatively little effort. In our case study, the reengineering could be done in five days, without prior knowledge of the application, but with the help of domain experts from Exact. The ease by which we were able to reengineer the original Codename into Codename^{MT} is of interest to our industrial partner Exact, and other companies alike, as it shows that even the initial costs of migrating towards multi-tenancy are relatively low and should thus not be seen as a barrier.

Importance of architecture While not surprising, another lesson we learned from the migration was that having a layered architecture is essential, both for keeping our reengineering approach lightweight and for doing the reengineering quickly and efficiently [Laine, 2001]. Without a well-layered architecture, applying our pattern would have taken much more effort.

Automated reengineering proves difficult The ease by which we were able to reengineer Codename automatically raises the question whether it is possible to automate the reengineering process. Unfortunately, we think this is very difficult to achieve, as the reengineering requires a considerable amount of architectural and domain knowledge of the application, which is difficult and costly to capture in a reengineering tool. Furthermore, the integration of the components of our multi-tenancy pattern is strongly dependent on the implementation of the existing application. A similar observation about the difficulty to automate design pattern detection and reengineering approaches was made by Guéhéneuc and Albin-Amiot [2001]. Specifically in our industrial environment, the architectural and domain knowledge of the lead architect of Codename — the third author of this chapter —, proved very valuable for the quick and efficient reengineering of the target application. Capturing this tacit knowledge in an automatic reengineering tool would prove difficult and expensive.

Fully transparent for the end-user An interesting observation is that no changes had to be made to the client side of the application, i.e., in terms of JavaScript. This serves a first indication that the end-user will not be aware of the fact that he is using a multi-tenant application instead of a single-tenant one. Furthermore, the (manual) tests have also shown that the other parts of the user interface have not evolved when going from Codename to Codename^{MT}.

Little effect for the developer Because we could enable multi-tenancy by making small changes only, we expect that most developers can remain relatively uneducated on the technical details. For example, they do not have to take multi-tenancy into account while writing new database queries as these are adapted automatically.

2.6.2 Discussion

In this version of Codename^{MT} we did not implement workflow configuration. The reason for this is that we limited our case study to the degree of configuration currently possible in Codename. A first step towards workflow configuration is to implement the tenant-specific friendly URL mechanism as described in Section 2.5.2. This approach still requires the tenant (or an Exact developer) to develop a custom .aspx page. In a future version of Codename^{MT}, Exact is aiming at making workflow configuration possible by enabling and disabling modules and widgets using

a web-based administration, rather than requiring a tenant to make changes to an .aspx page.

We have applied our pattern by modifying existing single-tenant code. One may argue that multi-tenant code additions should be completely isolated, e.g., by integrating the code using aspect-oriented programming. As typical aspect-oriented programming (following the AspectJ model) does not offer a fine enough pointcut mechanism to target all join points that we would need to change, we decided not to use aspects. Please note however, that using aspect-oriented programming would become applicable after a thorough refactoring of the source code, but this was beyond the scope of the lightweight reengineering pattern that we intended for.

2.6.3 Threats to Validity

We were able to apply our multi-tenancy pattern with relatively little effort. One of the reasons for this is the well-designed and layered architecture of Codename. In addition, the existing integration of the authentication using EIS and the possibility to add a TenantID claim to the token considerably shortened the implementation time for the authentication component. Finally, the database component could be adapted relatively easily as well, as all queries are created in one single component, i.e., the Data Access Layer, which made searching for query generations throughout the application superfluous. As such, we acknowledge that Codename might not be representative for all systems, but we also acknowledge that having intimate knowledge of the application is equally important for the reengineering of single-tenant into multi-tenant applications. Another confounding factor is the complexity of both the source code and the database schema, as both can have a direct influence on the ease by which an existing single-tenant application can be reengineered.

We have manually verified the correctness of the implementation of the functionality in Codename^{MT}. While we are confident that the verification was done thoroughly and was supported by one of the lead architects of Codename, we do see the need for automatic testing in this context. As such, we consider investigating the possibilities of defining a test methodology for multi-tenant applications as future work.

The case study we have conducted is not complete yet. For example, we have not implemented workflow configuration and automated tenant creation. As it is possible that these implementations introduce performance penalties, we did not formally evaluate the performance overhead of our approach. Although we have not encountered performance drawbacks yet, we consider a formal evaluation of the performance as future work.

2.6.4 Multi-Tenancy in the Real World

Although the benefits of multi-tenancy are obvious, there are some challenges which should be considered before implementing it.

Because all tenants use the same hardware resources, a problem caused by one tenant affects all the others. Additionally, the data of all tenants is on the same server. This results in a more urgent requirement for scalability, security and zero-downtime measures than in single-tenant software.

Finally, because multi-tenancy requires a higher degree of configurability, the code inherently becomes more complex, which may result in more difficult software maintenance if not implemented correctly (Section 2.2).

2.7 Conclusion

Because of the low number of instances, multi-tenancy sounds like a maintenance dream. Deployment of software updates becomes much easier and cheaper, due to the fact that a much smaller number of instances has to be updated. However, the complexity of the code does increase. In single-tenant software, challenges like configuration and versioning are solved by creating a branch in the development tree and deploying a separate instance. In multi-tenant software, this is no longer acceptable, which means that features like these must be integrated in the application architecture, which inherently increases the code complexity and therefore makes maintenance more difficult.

We believe that multi-tenancy can be a maintenance dream, despite the increase in code complexity. However, the quality of the implementation is crucial. In a non-layered software architecture, the introduction of multi-tenancy can lead to a maintenance nightmare because of code scattering. On the other hand, in a layered architecture, it is possible to implement multi-tenancy as a relatively isolated cross-cutting concern with little effort, while keeping the application maintainable. This raises the expectation that, for maintenance in a multi-tenant application, refactoring a non-layered architecture into a layered one can be very beneficial.

Our reengineering pattern is a guiding process that allows to quickly and efficiently transform a single-tenant application into a multi-tenant one, thereby also providing capabilities for tenant-specific layout styles, configuration and data management. As multi-tenancy is a relatively new concept, especially in the software engineering world, very little research has been done on this subject. We have defined the multi-tenant components in our approach after having researched existing problems in multi-tenant applications. This research was conducted by analyzing papers, the demand from industrial partners and by reading blog entries (including the comments, which form a source of valuable information as they contain information about the current problems in the SaaS industry).

In this chapter, we have applied our lightweight multi-tenancy reengineering

pattern to Codename, an industrial single-tenant application engineered by Exact. The result is Codename^{MT}, a multi-tenant version of Codename, offering the typical benefits of multi-tenancy, i.e., increased usage of hardware resources and easier maintenance.

From our case study, we learned that our approach:

1. Is lightweight, as implementation was done in about 100 lines of code, which took approximately 5 days to implement. This makes the approach attractive for Exact and other companies, because of the low initial investments. On a side note, we do observe that having a nicely layered architecture is a benefit for doing the migration quickly and efficiently.

2. Is transparent to the end-user, as (1) the look-and-feel of the application does not need to be changed and (2) the end-user does not know that the application is multi-tenant.

3. Does not require all developers working on the project to be trained in multi-tenancy, as the changes to the code are minimal and confined to some small parts.

As important directions for future work, we see the development of a test methodology and a real-time monitoring mechanism for multi-tenant applications. The former is essential when tackling larger reengineering efforts in the realm of multi-tenancy, while the latter will enable to determine the optimal moment for online software evolution in the face of zero-downtime for customers. In addition, we will continue to work with Exact on extending the configuration options for Codename^{MT}, in particular, the workflow configuration support.

3:

Performance Optimization of Deployed SaaS Applications

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 chapter, 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 chapter, we evaluate our technique in two case studies. In the first one, we show that our technique is accurate in detecting the time frame during which a PIO occurs. In the second one, 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.[1]

[1]This chapter contains our work published in the *Journal of Systems and Software* [Bezemer and Zaidman, 2014].

In the ISO standard for software maintenance[2], 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 [Swanson, 1976]. 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 [Hamilton, 2010]. Copying the production system to provide an environment for performance testing will further increase these costs. While we realize cost should be no decisive factor for neglecting performance testing, from our experience we know that it often is in industry. 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 [Woodside et al., 2007], 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 [Thereska et al., 2010], making performance maintenance a tedious task.

Perfecting software performance is typically done by investigating the values of two types of metrics [Thereska et al., 2010]. On one hand, high-level metrics such as response time and throughput [Jain, 1991] are important for getting a general idea of the performance state of a system. On the other hand, information retrieved from lower-level metrics, e.g., metrics for memory and processor usage — so called performance counters [Berrendorf and Ziegler, 1998] —, 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 [Cohen et al., 2004]. This makes manual analysis of performance in large, complex and possibly distributed systems hard.

In this chapter, we present a technique which provides assistance during semi-automated performance analysis. This technique automates locating so-called *per-*

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

formance 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 [Goldszmidt et al., 2005]. Our approach aims to assist experts by analyzing these starting points to give a diagnosis of bottleneck component(s). In this chapter, we address the following research question presented in Chapter 1:

> ***RQ2a**: How can we detect and analyze hardware bottlenecks?*

More specifically, we focus on the following research question:

> ***RQ2a-1**:How can we enhance the interpretation of performance counter values so that they can assist with the identification of the bottleneck components of a system?*

In Chapter 4, we present an evaluation of this technique by performing a user study on an industrial SaaS application. During this preliminary evaluation, we demonstrate 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 time frame 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 chapter is organized as follows. In Section 3.1, we introduce the concept of PIOs and we present our approach for detecting such PIOs. In Section 3.2, we explain our approach for automatically analyzing these PIOs. Section 3.3 discusses the implementation of our approach. Our case studies are presented in Sections 3.4, 3.5 and 3.6. We discuss the results of these case studies and threats to the validity of these results in Sections 3.7 and 3.8. We present related work in Section 3.9 and we conclude our work in Section 3.10.

3.1 Detecting Performance Improvement Opportunities

Performance optimization can be done during the software design phase and after deployment. Techniques such as profiling [Knuth, 1971] 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 [Elbaum and Diep, 2005]. Hence, profiling is not often used in practice [Nistor et al., 2013]. Therefore, it is necessary to use more light-weight

techniques after deployment to get an indication of the bottleneck first. When necessary, more expensive techniques, such as profiling, can then be used 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 time frames 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 chapter, 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 3.1.2, we will present our approach for detecting these PIOs (*Requirement 1*). We will explain our approach for analyzing PIOs (*Requirement 2*) in Section 3.2.

3.1.1 Performance Improvement Opportunities (PIOs)

A *performance improvement opportunity* (PIO) is a situation during which the performance could possibly be improved. Such a situation can be detected by monitoring and analyzing the right performance metrics. A PIO can be represented by the following collection of data:

- Date and time of start of the PIO
- SARATIO metric (Section 3.1.2)
- INTENSITY transformation (Section 3.1.3)
- Rule coverage matrix (Section 3.2.1)

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 3.9). When we improve the performance of a system using the information in a PIO description, we say we *exploit* the PIO. Throughout this chapter we will use the term PIO and PIO description interchangeably.

3.1.2 SARatio Metric

Application performance can be expressed in many different metrics, such as response time, throughput and latency [Jain, 1991]. One of the most important is average response time [Jain, 1991], 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[3] 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.

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 3.8).

From this classification, we construct a metric for performance characterization which fits into our description, namely the ratio SAR_{t} (*Slow-to-All-actions-*

[3] 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 [Koschke and Quante, 2005].

ratio) of the number of slow actions $SLOW_t$ to the total number of actions in time interval t:

$$\text{SARATIO}_t = \frac{|SLOW_t|}{|SLOW_t| + |NORMAL_t|}$$

Because it is a ratio, isolated extreme values have a smaller influence on the metric, which makes it more independent of time interval[4].

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. We consider performance optimization an iterative process; hence, after diagnosing and optimizing (when possible) the slowest 5% situations in the system, we repeat the process and find the 'new' slowest 5% situations. These iterations make the exact percentage that is used for the *HIGH* class less important, as situations which are missed in a previous classification will be detected in a new classification cycle after other situations have been optimized.

Throughout the rest of this chapter, 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 3.1 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

We transform Figure 3.1 into Figure 3.2 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.

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

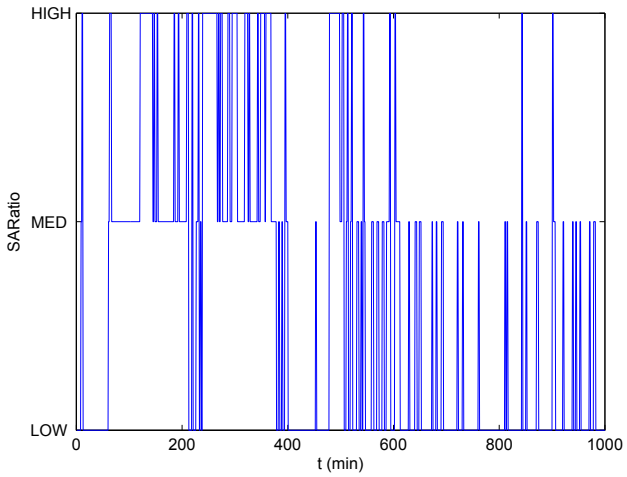

Figure 3.1: SARATIO graph before INTENSITY transformation

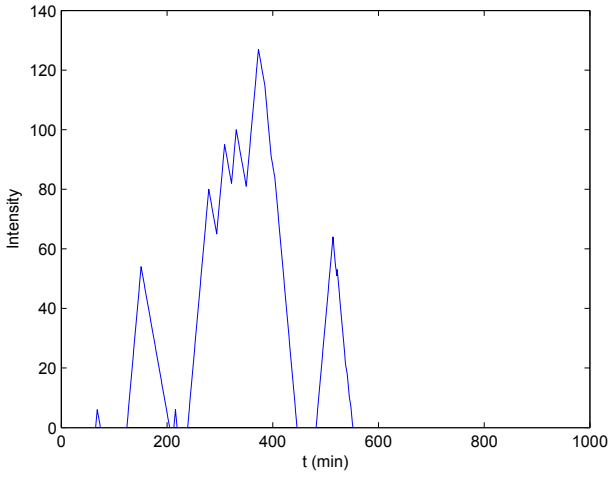

Figure 3.2: SARATIO graph after INTENSITY transformation

3.1.3 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 occurrence of high SARATIO values which are close to each other in time, i.e., longer lasting PIOs. To make such occurrences easier to spot, we perform the transformation described in Algorithm 1 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 (relative) 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 1 (INTENSITYTRANSFORMATION). Figure 3.2 shows the effect of applying this transformation to the data in Figure 3.1. It is easy to see that there are three major PIOs in Figure 3.2. 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 chapter, we assume the INTENSITY threshold is 0. As a result, we can change n to adjust the sensitivity of our approach.

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** $intensity$

3.2 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 [Agrawal et al., 1993] which 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.

3.2.1 PIO Analysis Using Association 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* [Berrendorf and Ziegler, 1998] (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 [Cohen et al., 2004]. 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 satisfies these requirements is the heat map [Wilkinson and Friendly, 2009]. Figure 3.3 depicts an example of a heat map, which could assist during performance analysis. The heat map displays data for two performance counters (*CPU* (CPU utilization) and *MEM* (memory usage)[5]) monitored on five servers (*Server1 – Server5*). In this heat map, darker squares

[5]Note that this heat map is an example. It does not necessarily reflect real situations.

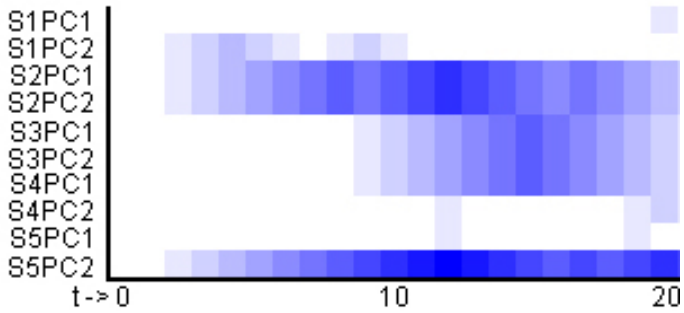

Figure 3.3: Rule Coverage Heat Map

mean that there is a stronger indication that the component on which this performance counter was monitored forms a bottleneck. In Figure 3.3, it is easy to see that server *Server2* and performance counter *MEM* on server *Server5* 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.

The Rule Coverage Matrix

The heat map in Figure 3.3 is a direct visualization of the *rule coverage matrix* depicted by Table 3.1. 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 3.1: Rule Coverage Matrix for Figure 3.3

	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

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. In this section, we describe how to build such a set of rules. An example association rule could be:

```
Server1_CPU ≥ 80% & Server1_MEM ≥ 50% → HIGH
```

This rule signals that during the training phase we observed that on *Server1* 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 detection and (b) PIO analysis.

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 $m^{i,j}_{rcm}$ by 1 if performance counter i is covered by a $HIGH$ rule at measurement j.
- Leave $m^{i,j}_{rcm}$ equal to $m_{i,j-1}$ for a MED rule
- Decrease $m^{i,j}_{rcm}$ by 1 if performance counter i is covered by a LOW rule at measurement j, with a minimum of 0

We consider the MED class a 'buffer-class' between the $HIGH$ and LOW class. A SARATIO classification which falls into the MED class usually indicates that the system is moving from a LOW to $HIGH$ state or vice versa. Therefore, we wait with adjusting the rule coverage matrix until either the LOW or $HIGH$ state is reached. Hence, we do not change the values in m for a MED rule.

Note that the original 'raw' values of the performance counters in m are left untouched in this process. We update the value of every $m^{i,j}_{rcm}$ only once for every

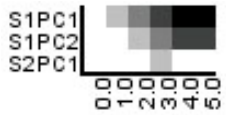

Figure 3.4: Heat map for the rule coverage matrix depicted by Table 3.3

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 3.3 shows the resulting m_{rcm} after applying our approach to the measurements and ruleset of Table 3.2 for a system consisting of two servers *Server1* and *Server2*, which are monitored through three performance counters (*Server1_CPU*, *Server1_MEM* and *Server2_CPU*). 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 *Server1_CPU*. Continuing this process results in the matrix depicted by Figure 3.3.

Figure 3.4 shows the heat map of this matrix. In our simple example we can see a cluster of dark coloured performance counters at server *Server1*, indicating this server may be a bottleneck.

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

Table 3.2: Sample ruleset and performance measurements

Sample association ruleset	**Sample measurements**			
1 Server1_CPU>80 & Server2_CPU<60 → *HIGH*	t	Server1_CPU	Server1_MEM	Server2_CPU
2 Server1_CPU>70 & Server1_MEM>70 → *HIGH*	0	40	60	80
3 Server1_CPU>90 → *HIGH*	1	95	60	80
4 Server1_MEM<30 → *MED*	2	98	80	80
5 else → *LOW*	3	98	95	55
	4	98	80	80
	5	40	45	80

Table 3.3: Coverage matrix for Table 3.2

	0	1	2	3	4	5
Server1_CPU	0	1	2	3	4	4
Server1_MEM	0	0	1	2	3	3
Server2_CPU	0	0	0	1	0	0
covered by rules #	5	3	2,3	1,2,3	2,3	5

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.

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

3.3 Implementation

Figure 3.5 depicts all the steps required for the implementation of our approach. In this section, we will explain every step taken.

3.3.1 Training Phase

During the training phase (see Section 3.2.1) 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.

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 3.5). 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

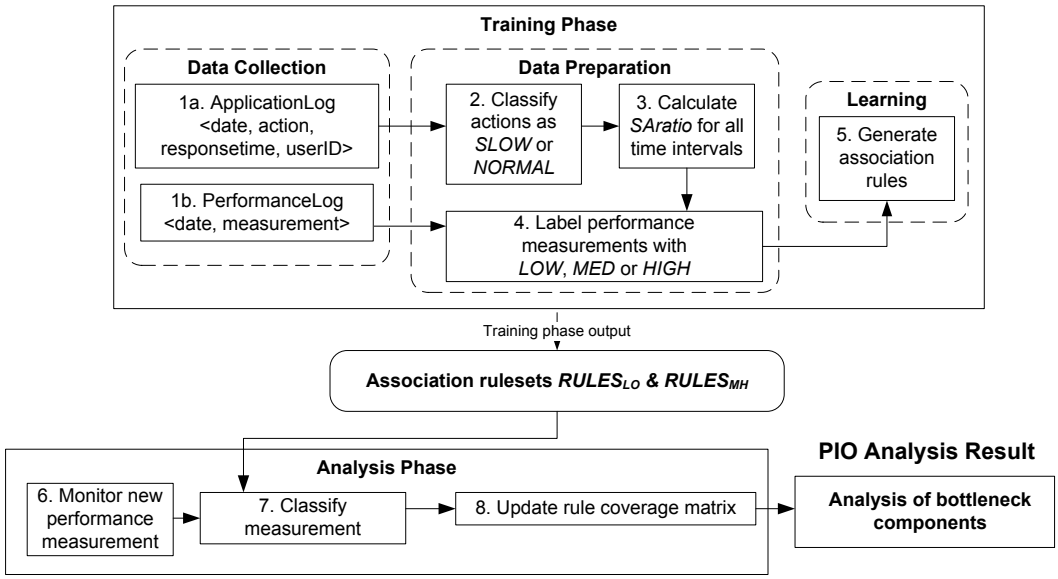

Figure 3.5: Steps of our approach for analyzing PIOs

- 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 chapter we will assume the `ApplicationLog` contains requests made to the application (i.e., the web server log — records will have the format `date`, `page`, `responseTime`, `userID`).

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 3.1.2 (*step 3*). We label all low-level measurements in the `PerformanceLog` with their corresponding load classification (*step 4*).

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 [Witten and Frank, 2005], 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 $RULES_{MH}$.

3.3.2 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.

3.4 Experimental Setup

The goal of our evaluation is to show that our approach is capable of fulfilling the two requirements posed in Section 3.1, namely detecting the time frames 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 3.5), we will show that the SARatio is an accurate metric for detecting time frames 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 3.6), 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:

> ***RQ2a-Eval1**: Is the* SARatio *an accurate metric for detecting the time frames during which the system was slow?* (Case study I)

> ***RQ2a-Eval2**: Is our technique for the estimation of* SARatio *classifications using performance counter measurements accurate?* (Case study I)

***RQ2a-Eval3**: 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.

3.4.1 Case Study Systems

We performed two case studies on SaaS systems: (1) on a widely-used benchmark application running on one server (RUBiS [Cecchet et al., 2002]) and (2) on a real industrial SaaS application running on multiple servers (Exact Online[6]).

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.

Exact Online

Exact Online is an industrial multi-tenant SaaS application for online bookkeeping with approximately 18,000 users[7]. 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.

3.4.2 Process

Training Phase The `ApplicationLog` and `PerformanceLog` are collected using the web server 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 web and database servers used an upper bound limit for timeouts to ensure all requests have a valid response time. The generation of the *LOW*, *MED*, *HIGH* classes 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 [Hall et al., 2009] with its default parameters, which is an implementation of the RIPPERk algorithm [Cohen, 1995]. 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.

[6] `http://www.exactonline.nl`

[7] In fact, there are about 10,000 users with 18,000 administrations, but for clarity we assume 1 user has 1 administration throughout this chapter.

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[8] and JHeatChart[9].

3.5 Proof-of-Concept: Case Study I for SARatio Classification Estimation

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.

3.5.1 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 3.6 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 [Hashemian et al., 2012], 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`.

[8] `http://www.jfree.org/jfreechart/`
[9] `http://www.tc33.org/projects/jheatchart/`

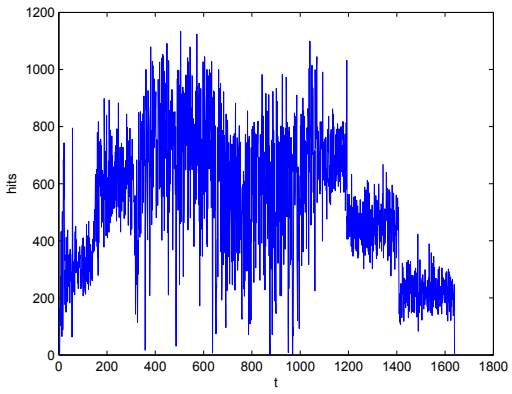

Figure 3.6: Traffic generated for the RUBiS case study

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.

Training Phase

Data Collection Table 3.4 shows the set of performance counters monitored on the server; we used Dstat[10] to log them every second. Together with the Apache *access_log*, we could now create the `ApplicationLog` and `PerformanceLog` SQL databases. These databases have the same structure as those in the EOL proof-of-concept so that the same queries could be used. Table 3.6 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,

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

Table 3.4: Monitored performance counters for RUBiS

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
File system stats	**File locks**
open files, inodes	posix, flock, read, write
IPC stats	
message queue, semaphores	
shared memory	

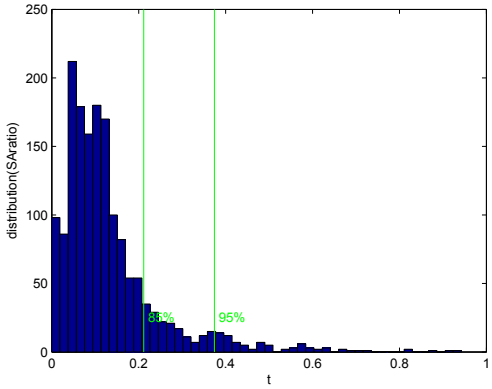

Figure 3.7: Distribution of SARATIO for 30 minutes of RUBiS traffic

user)-tuple but per application instead. Table 3.8 shows the number of *slow* and *normal* requests for these applications. Figure 3.7 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 3.5 shows the generated rules.

Table 3.5: Association rules generated in the RUBiS proof-of-concept

$RULES_{LO}$
(mem/cach ≤ 2175963136) & (mem/used ≥ 1103503360) → OTHER (mem/cach ≥ 1910624256) & (mem/buff ≤ 316026880) → OTHER (mem/buff ≥ 316256256) & (mem/buff ≤ 316358656) & (system/int ≤ 6695) & (dsk/read ≥ 118784) → OTHER (mem/buff ≤ 316497920) & (system/int ≥ 7052) & (mem/used ≤ 1080979456) → OTHER (mem/cach ≤ 2215194624) & (dsk/read ≤ 24576) → OTHER else → LOW
$RULES_{MH}$
(filesystem/files ≤ 2336) → HIGH else → MED

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 association 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

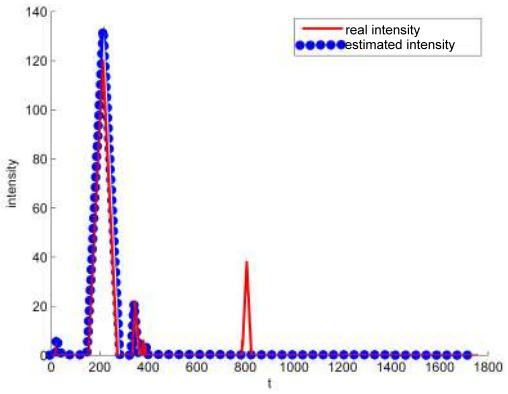

Figure 3.8: Real INTENSITY versus estimated INTENSITY

the RUBiS case. Because the association rules were generated from a subset of the `PerformanceLog` as described in Section 3.3, 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 3.8 shows the graph of the real and estimated INTENSITY[11].

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 3.8. 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 slowdown.

Another interesting observation is that our approach was capable of detecting several known weaknesses in the RUBiS implementation [Pugh and Spacco, 2004], 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.

[11] This graph is best viewed in colour.

3.5.2 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 log files 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

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 3.7 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 3.6 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 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 3.7. To consolidate together the performance counters monitored at varying frequencies, we maintain an array with the last value for each monitored performance counter. Every time a

new value is monitored, we replace the value for that performance counter in the array. The complete array is read at a fixed interval and used to get the SARATIO classification. This mechanism is used as well to deal with missing performance counter values, e.g., due to lost network packets.

Data Preparation To verify that the response times of each application are approximately normally distributed per user, we have inspected the histogram of 10

Table 3.6: Details about the case studies

	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 3.7: Monitored performance counters for EOL (*measurement interval*)

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*)
Web server 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*)

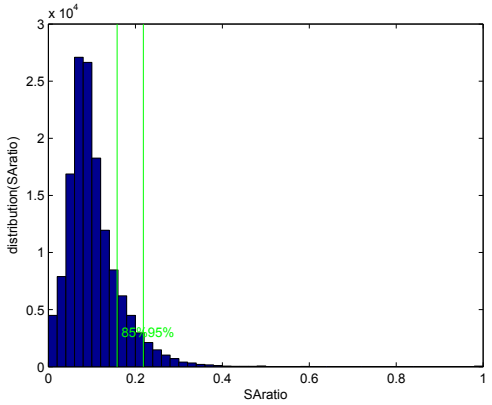

Figure 3.9: Distribution of SARATIO for 64 days of EOL traffic

(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 [1969]. Table 3.8 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 [Dekking et al., 2005].

Figure 3.9 shows the distribution of SARATIO in the EOL case study, together with the 85^{th} and 95^{th} percentile.

Table 3.8: #actions per class for the selected samples

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	

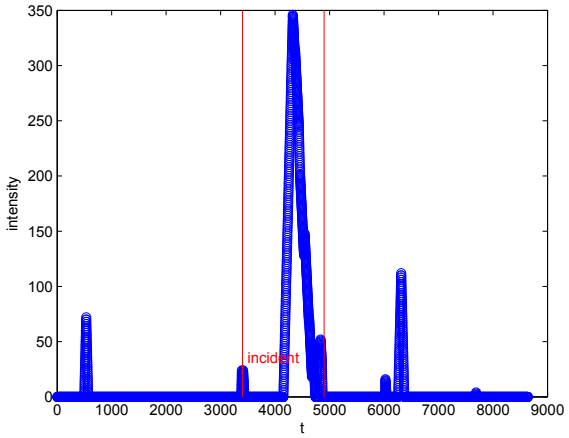

Figure 3.10: INTENSITY graph of the EOL incident based on estimated SARATIO classifications

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[12].

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 rulesets, we have estimated the SARATIO classifications using performance counter measurements. Figure 3.10 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 $t = 4900$.

Evaluation

Figure 3.10 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 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 3.11 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 3.11 shows that shortly after the start of the incident an anomaly was reported, however:

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

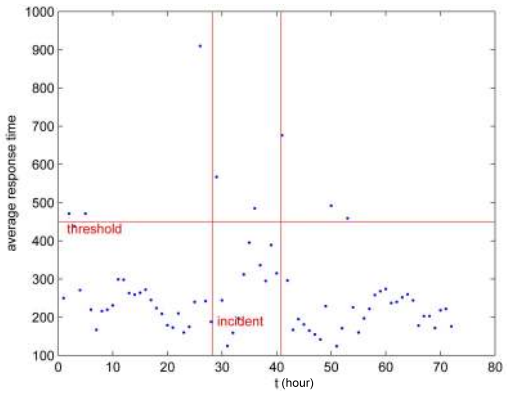

Figure 3.11: Current EOL performance anomaly criterium during incident

- 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 $t = 35$, which is 1-2 hours earlier than the criterium used by the EOL team.

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.

3.6 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?

3.6.1 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 3.5.2. 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 3.5.2). 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 3.9 depicts the number of detected PIOs and overload registrations and the size of the random sample.

Table 3.9: Random sample description

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

3.6.2 Training Phase

The training phase for this case study was equal to the process described in Section 3.5.2, 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 3.10.

Table 3.10: Association rules generated during the random sample verification case study

$RULES_{LO}$

```
(DBclus1/Processor/% Processor Time/_Total ≥ 13.02883)
   & (DBclus1/SQLServer:Latches/Latch Waits/sec/null ≥ 598.898376) → OTHER
(DBclus1/Processor/% Processor Time/_Total ≥ 13.378229)
    & (DBclus1/SQLServer:Buffer Manager/Page life expectancy/null ≤ 1859) → OTHER
(ws2/Processor/% Processor Time/_Total ≥ 11.026497)
   & (ws2/.NET CLR Exceptions/# of Exceps Thrown / sec/_Global_ ≥ 8.948925)
   & (ws6/Process/Handle Count/_Total ≤ 21258) → OTHER
(DBclus1/SQLServer:Buffer Manager/Page life expectancy/null ≤ 4357)
   & (DBclus2/Memory/Available MBytes/null ≤ 5177)
   & (ws5/Processor/% Processor Time/_Total ≥ 2.104228) → OTHER
(DBclus1/SQLServer:Buffer Manager/Page life expectancy/null ≤ 4088)
   & (DBclus1/SQLServer:Latches/Average Latch Wait Time (ms)/null ≥ 1.02296)
   & (DBclus2/LogicalDisk/Avg.  Disk sec/Write/T: ≥ 0.000543) → OTHER
(DBclus1/LogicalDisk/Disk Reads/sec/W: ≥ 20.217216)
   & (IDws1/Paging File/% Usage/_Total ≥ 1.238918)
   & (ws3/ASP.NET Apps v4.0.30319/Requests Timed Out/__Total__ ≥ 1) → OTHER
(ws6/ASP.NET Apps v4.0.30319/Requests Timed Out/__Total__ ≤ 0)
   & (ws4/Processor/% Processor Time/_Total ≥ 13.349845)
   & (ws1/.NET CLR Exceptions/# of Exceps Thrown / sec/_Global_ ≤ 2.83327)
   & (DBclus1/LogicalDisk/Avg.  Disk sec/Write/E: ≥ 0.000446) → OTHER
else → LOW
```

$RULES_{MH}$

```
(ws3/ASP.NET Apps v4.0.30319/Request Bytes In Total/__Total__ ≤ 86408)
   & (DBclus2/LogicalDisk/Avg.  Disk sec/Read/W: ≥ 0.000932)
   & (IDws1/LogicalDisk/Avg.  Disk sec/Write/_Total ≥ 0.001162) → HIGH
(ws3/ASP.NET Apps v4.0.30319/Request Bytes In Total/__Total__ ≤ 70541)
   & (DBclus2/LogicalDisk/Avg.  Disk sec/Write/W: ≥ 0.0005)
   & (DBclus2/Memory/Page Reads/sec/null ≥ 0.046007) → HIGH
(ws4/ASP.NET Apps v4.0.30319/Request Bytes In Total/__Total__ ≤ 81291)
   & (DBclus1/LogicalDisk/Disk Reads/sec/J: ≥ 0.076917)
   & (DBclus1/SQLServer:Buffer Manager/Page life expectancy/null ≤ 131)
   & (contr1/Server/Bytes Total/sec/null ≤ 204.351318) → HIGH
(ws1/ASP.NET Apps v4.0.30319/Request Bytes In Total/__Total__ ≤ 18344)
   & (ws2/ASP.NET Applications/Request Bytes In Total/__Total__ ≤ 7161)
   & (ws1/Server/Bytes Total/sec/null ≤ 2113.22168) → HIGH
(ws6/ASP.NET Apps v4.0.30319/Request Bytes Out Total/__Total__ ≤ 629862)
   & (IDws2/Memory/Pool Paged Bytes/null ≥ 140587008)
   & (IDws2/Memory/% Committed Bytes In Use/null ≤ 19.593651) → HIGH
else → MED
```

3.6.3 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.

3.6.4 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 3.11 and 3.12 show two examples of this process.

Table 3.11: 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.

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 3.13 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.

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

Table 3.12: 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 3.13: 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	-

has grown, these tasks take longer to finish and sometimes their execution overlaps. This causes a slowdown in the system.

Table 3.14 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 3.13 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 3.14: 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%)

3.7 Discussion

3.7.1 The Requirements Revisited

Requirement 1: Detect the time frames during which the system performed relatively slow.

In our evaluation in Section 3.6 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 3.2, 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. We will investigate whether it is possible to automatically combine PIOs in future work.

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

In our evaluation in Section 3.6 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.

3.7.2 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.

3.7.3 Scalability

Our approach is lightweight and transparent; it requires no modification of application code as measurements are done at the operating system level. In addition, we designed our approach to not require knowledge about the structure of the system. In the future, we will do more research on different systems to verify this.

The `PerformanceLog` and `ApplicationLog` analyzed during the case study in Section 3.6 contained respectively 28 million and 135 million records. Preparing the data and training the association rules took approximately 10 minutes on a standard desktop computer. 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.

The data collection was done by default by the EOL application. Therefore, we cannot say anything about the exact overhead it causes. We do know that the amount of overhead is dependent on the frequency with which the performance counters are read and stored.

The case study was done on a system of 18 servers. Due to the low processing times of the different phases of our approach, we expect it to scale well to large environments. However, this should be evaluated in future work.

3.7.4 Limitations

A limitation of our approach is that all performance data must be collected in a centralized location in order to process it, such as a database that is reachable by all components. In 'classical' n-tier systems, such a database is usually easy to implement as it already exists or can be created in the database tier. This may be more challenging for systems without such a centralized location, such as cloud-based systems or service chains. However, our approach should be able to fit such systems, as long as enough data can be gathered.

Because our approach is based on supervised learning, it is more suitable for detecting recurring problems than performance anomalies. In a production system which exhibits anomalies during the training period, we will train a classifier which is specialized in detecting situations which look like the anomalies seen during the training period that caused the greatest slowdown. These anomalies may never (or hardly) happen again (otherwise, they would not be anomalies). In a production system which does not exhibit (dominant) anomalies during the training period, we can get a better view on what needs structural change in order to optimize the system. Therefore, a limitation of our approach is that we assume our system has few (preferably none) performance problems/anomalies during the training period. We do realize that this is hard to verify and that the classifier may require retraining in case of anomalies during the training period. However, anomalies are usually known by performance experts and/or system administrators after they

took place. Therefore, while this assumption may be hard to ensure upfront, it is easy to validate and fix when necessary by retraining the classifier.

3.7.5 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 occurrence of an anomaly is incidental, while the occurrence 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.

3.7.6 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 that 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 3.1.2.

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 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 3.3, *Server2_CPU*, *Server2_MEM* and *Server5_MEM* would more likely be interesting for investigation than *Server1_CPU* and *Server5_CPU*.

We acknowledge that we did not compare our technique to similar state-of-the-art techniques. The main reason for this is the lack of available implementations of those techniques. In addition, it is expensive to perform an industrial case study similar to the one described in this chapter.

3.7.7 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. Finally, shorter PIOs may represent actions which are executed often, making them interesting targets for optimization.

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 3.7.1.

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

3.8 Threats to Validity

3.8.1 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 [Hashemian et al., 2012], 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.

3.8.2 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. In addition, the industrial workload was mostly I/O-intensive. As we monitor both CPU and I/O-related metrics, and do not give preference to any of these, we expect our approach to be agnostic to the type of workload. However, in future work, we will analyze the results of running our approach on an application with a CPU-intensive workload.

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 [Malone et al., 2011], 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.

A threat to validity is that we treat the association rule generation algorithm in our approach as a black-box component. In our evaluation we have used the JRip association rule generation algorithm from WEKA, because the rules generated using this algorithm gave the best crossfold validation results on our dataset, as explained in Section 3.4.2. In future work, we will do a thorough comparison of association rule generation algorithms and their parameters to investigate whether we can select an algorithm that works best for all datasets. In addition, we will investigate if other techniques which generate descriptive classifiers, such as decision trees, can be used.

3.9 Related Work

In this section we discuss 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. [2005] 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. [2008] 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. [2007] 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 [2010] 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. [2001] 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. [2008] 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. [2004]; Zhang et al. [2005] 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. [2011] 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. [2010] 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.

Profiling. Profilers are tools which collect run-time details about software [Knuth, 1971], 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 [Yan et al., 2012]. 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. [1998] 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. [2012] 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 [Elbaum and Diep, 2005; Jovic et al., 2011], 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 [Jovic et al., 2011]. 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 [Jovic et al., 2011] 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 [Fürlinger et al., 2007; Gregg, 2010], 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 [Wu et al., 2004].

3.10 Conclusion

In this chapter 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 chapter 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 question:

RQ2a-1 *How can we enhance the interpretation of performance counter values so that they can assist with the identification of the bottleneck component(s) of a system?* 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. In addition, 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).

3.10.1 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.

4

Visualizing Performance: Wedjat

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 chapter, we extend our previous work on performance anomaly detection with a technique that helps performance experts locate 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 several visualizations, such as heat maps, which were implemented in an open source tool called WEDJAT.

In this chapter, we evaluate our technique and WEDJAT *in a field user study with three performance experts from industry using data from a large-scale industrial application. From our field study we conclude that our technique is useful for speeding up the performance maintenance process and that heat maps are a valuable way of visualizing performance data.*[1]

In the ISO standard for software maintenance[2], 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

[1] This chapter contains our work published in the proceedings of the *28th International Conference on Software Maintenance (ICSM'12)* [Bezemer et al., 2012].

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

software system after delivery. An interesting application of perfective maintenance is *performance maintenance*, as many software performance issues become obvious after deployment only. While a large amount of research has been done on software performance engineering in general [Woodside et al., 2007], only few papers deal with software performance maintenance. In addition, experience from industry shows that performance engineers mainly use combinations of simple and rather inadequate tools and techniques rather than integrated approaches [Thereska et al., 2010], making performance maintenance a tedious task.

Perfecting software performance is typically done by investigating the values of two types of metrics [Thereska et al., 2010]. On one hand, high-level metrics such as response time and throughput [Jain, 1991] are important for getting a general idea of the performance state of a system. On the other hand, information retrieved from lower-level metrics, e.g., metrics for memory and processor usage — so called performance counters [Berrendorf and Ziegler, 1998] —, 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 [Cohen et al., 2004]. This makes manual analysis of performance in large, complex and possibly distributed systems hard.

In Chapter 3, we have presented an approach for detecting and analyzing performance anomalies using performance counter measurements. This approach allows us to detect performance anomalies by identifying system states at which the system is performing relatively slow based on performance counter values. We have also shown that our approach allows faster detection of problems than a 'traditional' threshold setting for the average response time. In this chapter, we extend this approach with a technique that helps performance experts locate spots for possible performance improvements. Our technique describes such spots as *performance improvement opportunities* (PIOs). PIOs give performance experts a starting point for performance improvements, e.g., by pinpointing the bottleneck component. The technique uses a number of visualization methods, amongst which heat maps [Wilkinson and Friendly, 2009], to provide a compact overview of the performance history of a system. Hence, in this chapter, we address the following research question presented in Chapter 1:

> ***RQ2b**: How can we report and visualize the diagnosis of the bottleneck component(s)?*

We have implemented our technique in an open source tool called WEDJAT, which we present and evaluate by conducting a field study with performance experts from industry in this chapter. Our technique is a high-level approach: it works complementary to lower level approaches such as profiling [Knuth, 1971], as it helps narrow down the server or hardware which requires more investigation.

This chapter is organized as follows. In Section 4.1 we discuss relevant background information. In Section 4.2 we present our idea of using heat maps for performance analysis. The implementation of this idea is presented in Section 4.3. We evaluate our approach using a field user study (Section 4.4) and present the results in Section 4.5. Results are discussed in Section 4.6. We present related work in Section 4.7 and we conclude our work in Section 4.8.

4.1 Background

In Chapter 3, we presented an approach that allows to identify performance anomalies, e.g., sudden slowdowns, in a software system using low-level performance measurements only. That work contrasts earlier studies that used the response time as the main indicator of performance anomalies. We have shown that by using low-level performance measurements, we were able to efficiently identify performance anomalies that originate at the server. Furthermore, our technique worked very precisely in that it keeps track of performance profiles per user, so that a user with many database records versus a user with a relatively low number of database records gets treated differently. While we could very precisely identify when a performance anomaly occurred, we had no means of identifying its cause and that is what this chapter adds.

In the next two subsections we briefly describe our approach from Chapter 3. As our approach is a learning-based approach, we first describe the training phase in Section 4.1.1, before we go over to the actual classification phase in Section 4.1.2.

4.1.1 Training Phase

Central to our approach are the SARatio and the Intensity metrics. The SARatio (or *Slow-to-All-actions-ratio*) for a time interval t is defined as:

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

We define an action as slow when it belongs to the 15% slowest actions in terms of response time for a particular user of a particular application (or feature) for a time interval t. We now calculate the SARatio for all time intervals of the training period using a sliding window approach. As we now have a SARatio-value for all monitored time intervals, we can identify intervals during which the system was running relatively slow.

The next step is to define thresholds for the SARatio, such that we can classify system load for each interval as:

- $high$: system load is typically too high, which makes it perform slow (highest 5% of values for SARatio)

- *med*: system load may become or may just have been problematic (medium 10% of values for SARATIO)
- *low*: system load is non-problematic (low 85% of values for SARATIO)

After classifying all intervals as exhibiting *high*, *med*, or *low* load based on the SARATIO, we assign the performance counter data to the time interval during which it was monitored. Next, we use association rule mining to classify the state of the system using performance counters only (e.g., memory≥80, CPU>70 → *high*). In particular, we want to use the low level performance counter measurements rather than the SARATIO directly as they can give a more precise description of the performance, which can assist in giving a diagnosis.

4.1.2 Classification Phase

During the classification phase we classify new performance counter measurements. However, because we are not interested in isolated spikes in the performance of a system, but rather in situations in which the system is relatively slow for longer periods of time, we use a sliding-window approach to filter out these isolated spikes.

For a sliding window of size n, we determine which association rules match for the measured performance counter values. Next, we count how many of those roles are classified as *low*, *med* and *high* load (see Section 4.1.1). Finally, we determine the INTENSITY metric as follows:

$$\text{If } >30\% \text{ of the classifications in the window } \begin{cases} \text{high} & \rightarrow & \text{intensity+2} \\ \text{med} & \rightarrow & \text{intensity-1} \\ \text{low} & \rightarrow & \text{intensity-2} \end{cases}$$

This results in a series of values for the INTENSITY of the load on a system during the classification phase, in which new performance counter measurements are classified. Ideally, this metric is zero; whenever it is larger, we have an indication that the system is running relatively slow.

Both the association rules used to calculate the INTENSITY metric and the INTENSITY metric itself will form the basis for the approach presented in this chapter.

4.2 Approach

In this section, we present our approach for locating performance improvement opportunities (PIOs), which is an extension of our approach for performance anomaly detection explained in Section 4.1. A PIO is a snapshot of the system during a period of time at which the performance of the system could possibly be improved. This snapshot is described by the following for that period of time:

- Date and time of start of the PIO

- Date and time of end of the PIO
- INTENSITY graph (Section 4.1.2 and Figure 4.2)
- Raw metric value matrix (Section 4.2.1)
- Rule coverage matrix (Section 4.2.1)
- Missing value matrix (Section 4.3.5)

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 4.7).

Our PIO detection approach exploits the association rules used during the classification process of the anomaly detection to detect starting points for exploring possible performance improvement opportunities. The goal of our approach is to analyze the information in the rules matched by a measurement and detect clusters of performance counter metrics that help us to decide on which server or hardware we must start looking for possible performance improvements.

Table 4.1 shows a sample set of association rules used to characterize the load on a system into the classes low, med and $high$. The system consists of server $S1$ with performance counters $PC1$ and $PC2$ and server $S2$ with performance counter $PC1$. Table 4.1 contains a set of sample measurements for these performance counters as well. As defined in Chapter 3, a $high$ load often represents a performance anomaly, which indicates a possible PIO. We exploit this property to get an indication of the bottleneck component.

Table 4.1: Sample rule set and performance measurements

Sample association rule set	**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	25	80

4.2.1 The Rule Coverage Matrix

Our approach 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.

In addition, we maintain a so-called coverage matrix m'. The rows of this matrix contain the performance counters, the columns depict measurements. The first column, representing the first measurement is initialized to 0. Each time a

new measurement is received, the last column of m' is copied and the following algorithm is applied:

- Increase $m'_{i,j}$ if performance counter i is covered by a $high$ rule at measurement j.
- Leave $m'_{i,j}$ equal to $m'_{i,j-1}$ for a med rule
- Decrease $m'_{i,j}$ 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 $m'_{i,j}$ 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 performance counter values are in general relatively stable, which means 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 this way, higher values in m' will appear because these values will be increased for performance counters which occur in adjacent measurements.
4. Eventually, clusters of higher values in m' for performance counters on specific hardware will appear.
5. These clusters can be used to do performance maintenance, e.g., by pinpointing a bottleneck component.

The following example illustrates this. Table 4.2 shows the resulting m' after applying our approach to the measurements and ruleset of Table 4.1. Figure 4.1 shows a visual representation of this matrix in the form of a heat map [Wilkinson and Friendly, 2009]. In such a map darker colours represent higher values. In our simple example we can see a cluster of dark coloured performance counters at server $S1$, indicating this server may be a bottleneck. Note that the order of the performance counters within the matrix influences the ease with which such

Table 4.2: Rule coverage matrix for Table 4.1

	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

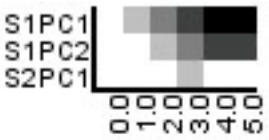

Figure 4.1: Rule Coverage Heat Map

clusters can be detected. In the remainder of this chapter, we assume that the performance counters are grouped per server within the matrix.

In the next section we present WEDJAT, a tool which implements the rule coverage matrix and heatmap. In addition, in WEDJAT we combine a number of methods for visualizing performance counter data.

4.3 Tool Implementation: Wedjat

In our approach, the starting point for all investigations is the INTENSITY metric discussed in Section 4.1. Whenever the INTENSITY is larger than 0, the performance data requires more detailed inspection. Our tool WEDJAT[3] gives an overview of performance data deviating from its normal behaviour, helping to quickly identify possible bottlenecks.

In WEDJAT, we combine our ideas of using a rule coverage matrix and heat maps to visualize performance data. WEDJAT offers several views, which can be used on their own or complementary to each other. In the remainder of this section, WEDJAT and its views are presented.

4.3.1 Time Slider

One of the main components in WEDJAT is the time slider, which enables a time interval for the heat maps to be selected. This allows a user to zoom (out) on events.

4.3.2 Intensity Graph View

Goal The INTENSITY graph is intended to be displayed in a performance monitor or dashboard and its purpose is to serve as a trigger for starting investigation with

[3]The Wedjat is an ancient Egyptian symbol of protection and good health.

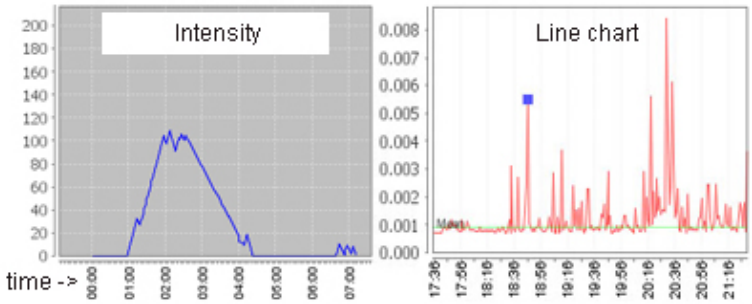

Figure 4.2: Example INTENSITY and line chart

WEDJAT. We have included the INTENSITY graph in WEDJAT itself as well; the time slider allows to zoom in or out on this graph.

Interpretation Whenever this graph shows a peak (value greater than zero), a possible PIO exists and the performance data requires deeper investigation.

Example Figure 4.2 shows the INTENSITY graph generated during a large and a small performance incident in a system.

4.3.3 Rule Coverage Heat Map View

Goal The rule coverage heat map is the visualization of the rule coverage matrix as described in Section 4.2. The goal of this heat map is to give an indication of where to start the investigation for possible performance improvements.

Interpretation In this heat map, adjacent dark squares indicate that this counter occurred in matched association rules for a longer period of time.

Example Figure 4.3 depicts a rule coverage heat map in which the rule:

(`sv4/Memory/Available MBytes/null` ≥ 691) `and`
(`ws4/Processor/% Processor Time/_Total` ≤ 53.519428)

was matched repeatedly[4].

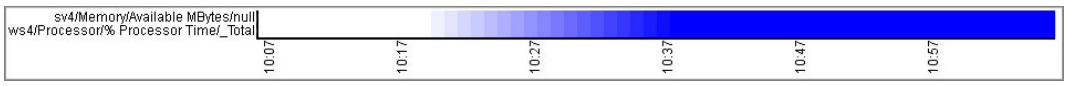

Figure 4.3: Rule coverage heat map (rule matched repeatedly: (`sv4/Memory/Available MBytes/null` ≥ 691) and (`ws4/Processor/% Processor Time/_Total` ≤ 53.519428)

[4]In this chapter, we use the format *ServerName/CounterCategory/PerformanceCounter/Instance* to describe a performance counter.

4.3.4 Deviation-from-mean Heat Map View

Goal The goal of this heat map is to easily identify groups of abnormal values for performance counters.

Interpretation In the deviation-from-mean heat map, the intensity of the colour of its squares is calculated based upon the number of standard deviations from the mean value of the performance counter represented by that row. In this heat map, adjacent dark squares indicate that this counter exhibited an unusually low or high value for a longer period of time. WEDJAT offers two possibilities of filtering deviation-from-mean heat maps: show only the counters in the rule coverage heat map and show all counters for a specific server. It is possible to display the heat map for two servers at the same time, for example, to compare them to verify a load balancer is behaving properly.

Example The deviation-from-mean heat map for server `ws4` is partly displayed in Figure 4.4. In this heat map it is clear to see that there was a period of approximately 20 minutes during which the monitored processor time counter exhibited abnormal behaviour (see first line between approximately 10:20 and 10:40).

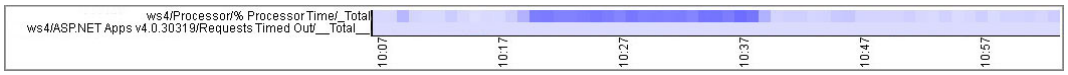

Figure 4.4: Deviation-from-mean heat map for the server `ws4`

4.3.5 Missing Values Heat Map View

Goal During the classification phase, we classify a set of performance counter values. In some cases, this set is not complete, i.e., it does not contain values for all performance counters expected based on the configuration of the monitor. The reason for this may be a configuration error or a problem with hardware or a server. The goal of the missing values heat map is to detect such configuration and hardware issues.

Interpretation A value *MissingIntervals* is maintained for every performance counter, which is initialized to 0 and increased when we do not receive a value for this performance counter in a measurement interval. After a value is received *MissingIntervals* is reset to 0. In the missing values heat map, the intensity of the colour of a square is based on the value of *MissingIntervals*. As such, adjacent dark squares indicate that this counter did not exhibit a value for a longer period of time, which indicates either a configuration or hardware issue.

Example Figure 4.5 shows the missing values heat map when there was a problem with the `sv2` server and a configuration error on the SQL cluster.

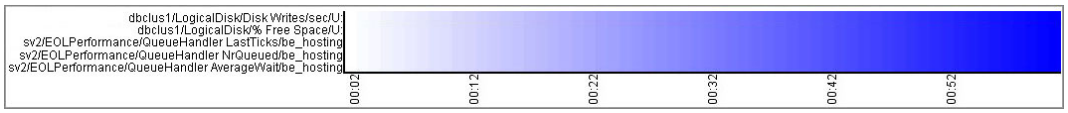

Figure 4.5: Missing values heat map

4.3.6 Line Chart View

Goal After clicking on a square in the deviation-from-mean heat map, a line chart is plotted displaying the values of the performance counter in the selected time interval. In addition, the mean calculated during the training phase is shown. The goal of displaying the line chart is to allow users of the tool to combine the information in the heat maps with views they are more accustomed to.

Interpretation This chart can be used to see trends, e.g., whether the counter is increasing or decreasing rapidly.

Example Fig. 4.2 shows a line chart for counter `dbclus1/LogicalDisk/Avg. Disk sec/Read/W:`.

4.3.7 Histogram View

Goal After clicking on a square in the deviation-from-mean heat map, two histograms are plotted as well. The goal of the histograms is to give the user a quick overview of normal values for the selected performance counter.

Interpretation The first histogram shows the values of the performance counter observed during the training phase. This histogram shows the user a quick overview of regular values for this performance counter. The second one shows the histogram of the values observed during the selected time interval. This histogram allows the user to compare the distribution of the values of this performance counter with those observed during the training period.

Example Figure 4.6 depicts these two histograms for the `dbclus/LogicalDisk/Avg. Disk sec/Read/U:` performance counter. The dark line in the histograms depicts the value of the selected square. From these histograms becomes clear[5] that during the trainingsperiod values between 0.025 and 0.005 rarely occured, while these occured frequently during the selected period. This could indicate that the counter is exhibiting abnormal behaviour during the selected period.

4.4 Design of the Field User Study

In this section, we evaluate Wedjat and our idea of using heat maps for software performance maintenance in a field user study. In our study, we are looking for an answer to the following research questions:

[5]Note that the scale of the axes of the histograms is different.

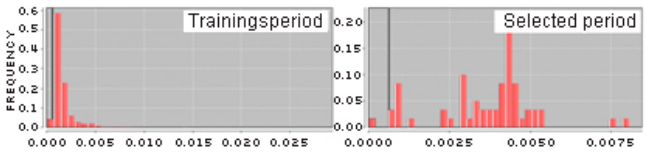

Figure 4.6: Histograms for the `dbclus/LogicalDisk/Avg. Disk sec/Read/U:` performance counter

> ***RQ2b-Eval1**: Does the rule coverage matrix provide a good starting point for performance maintenance?*
>
> ***RQ2b-Eval2**: Are heat maps an appropriate way of visualizing performance data?*

The outline of this section is as follows. First, we will describe the field setting of the user study. After this, we will discuss the setup of the study and the profile of the participants. In the next sections, we will present and discuss the findings of the field user study.

4.4.1 Field Setting

In order to find out how useful the rule coverage matrix and heat maps are for indicating starting points for improving performance bottlenecks, we set up a field user study with three performance experts from Exact[6]. Exact is a Dutch-based software company, which specializes in enterprise resource planning (ERP), customer relationship management (CRM) and financial administration software. Exact has over 1900 employees working in more than 20 countries. Founded in 1984, Exact has over 25 years of experience in multi-user client/server software and web applications. Since several years, Exact has also been offering a successful multi-tenant (see Chapter 2) Software-as-a-Service solution, called Exact Online[7] (EOL), which is the target of our field study. Exact Online uses ASP.NET and SQL Server.

4.4.2 Participant profile

The participants are the performance experts in the EOL team. The first part of the questionnaire consisted of a set of general questions about their experience, knowledge and the current process at EOL. Their experience is summarized in Table 4.3. Participant I and II have 16+ years of experience with software engineering and 10 years of experience with performance analysis. Participant III does not have direct experience as a software engineer, but does have 10 years of performance analysis experience from a product management point of view. All participants indicate

[6] `http://www.exact.com`
[7] `http://www.exactonline.nl`

their knowledge of the EOL infrastructure and of performance analysis in general as excellent.

Table 4.3: Years Experience of the Participants

Participant	Role	Soft. Eng.	Perf. Analysis	EOL
PI	Senior Research Engineer	16	10	7
PII	Principal Research Engineer	18	10	3
PIII	Product Manager	0	10	7

In the current situation at EOL, performance problems are noticed from log reports or follow from customer complaints about application speed and availability. Analysis of these problems and other possible opportunities to improve performance is done using a set of standard non-integrated tools such as Microsoft's PerfMon and SQL Profiler. All participants spend 5–48 hours per week improving the performance of Exact Online, depending upon the number of problems reported.

4.4.3 Field User Study Setup

In a short preliminary study an initial evaluation of the usability of WEDJAT and its enclosed heat maps was done. In two sessions, three participants were requested to investigate the performance issue discussed in Section 3.5.2 using WEDJAT only. A large amount of feedback and improvements to the usability of the heat maps and WEDJAT were gathered from the participants. These improvements were implemented and evaluated during the field study. In the field study, three participants were requested to investigate a performance issue different from the one investigated during the preliminary study. Two of the participants to the field study took part in the preliminary study, while the third had never worked with WEDJAT before.

The performance data used during the field study was monitored during a period of one month on part of the Exact Online infrastructure. The association rule set used by WEDJAT was trained on the data monitored during the first week of that period. The monitored part of the infrastructure consists of 6 webservers, 3 service machines, 2 SQL clusters and 2 virtual domain controllers, exhibiting a total of 140 performance counter values every minute[8].

The process followed during the field study is discussed in the remainder of this section.

Step 1: Demo of Wedjat

All sessions were started with a short demo of WEDJAT, demonstrating all its features.

[8]For a complete list see `http://www.st.ewi.tudelft.nl/~corpaul/eol_list.txt`

Step 2: Assigned Task

The participants were asked to investigate a real performance issue in WEDJAT for around 1.5 hours. We gave assistance during the investigation and asked them to fill in a questionnaire at the end of the session about the usability.

The task the experts are asked to solve is the following:

1. Load the data set for April 27 2012.
2. List the times of PIOs found on that date.
3. Investigate the PIO found between 10 AM and 11 AM.
4. Indicate the bottleneck component(s) found in that PIO.
5. Elaborate on how you selected the bottleneck(s).

The bottleneck component in the PIO between 10 AM and 11 AM was formed by the processor of a web server. This was expressed in WEDJAT by the following:

1. The `Processor/% Processor Time` performance counter of `ws4` (web server 4) showed a cluster of dark squares in the rule coverage heat map (see Figure 4.3).
2. The `Processor/% Processor Time` performance counter of `ws4` showed a cluster of dark squares in the deviation-from-mean heat map (see Figure 4.4).

Step 3: Questionnaire

After the investigation, the participants were asked to fill in a questionnaire consisting of 5 open questions and 56 questions that could be answered on a 5-point Likert-scale, ranging from 1 for 'strongly agree' to 5 for 'strongly disagree'. During the questionnaire, all participants regularly switched back to WEDJAT to answer the questions as accurately as possible. The questionnaire and the participants' answers are listed in Table 4.4.

Step 4: Contextual Interview

A large amount of information and feedback was elicited from the participants through the use of a contextual interview [Holtzblatt and Jones, 1995]. During such an interview, the participants are requested to solve an assigned task while they are being questioned constantly when working with the application for suggestions and feedback based on their actions. This allows them to articulate their normal or preferred work practices [Holtzblatt and Jones, 1995] and to get involved in the design of the application. During the session with two participants, we encouraged them to cooperate and discuss their ideas out loud to elicit more detailed feedback. The contextual interview was actually not a separate step in the experiment but was held throughout step 2 and 3, based on the actions and

questions of the participants. When interesting opportunities arose after an action or question, the participants were asked to explain why they performed that action or asked that question. From the discussions that followed, a large amount of feedback could be extracted.

After filling out the questionnaire, the answers of the participants in that session were compared and discussed, especially when they were different from each other.

Table 4.4: Questionnaire results

		P I	P II	P III
General (1 = strongly disagree, 5 = strongly agree)				
Q1	The tool was easy to use.	4	3	2
Q2	A tool like Wedjat will save me time.	5	3	5
Q3	There is added value in using heatmaps for performance analysis.	5	4	5
Q4	A tool like Wedjat will help me understand and diagnose performance problems better.	4	4	5
Q5	Switching between different datasets in Wedjat was easy.	4	3	4
Rule coverage heat map				
Q6	The meaning of this heatmap was clear to me.	4	3	3
Q7	The heatmap was intuitive.	4	3	3
Q8	The heatmap provides new information.	4	3	4
Q9	The heatmap improves the speed with which problems can be analyzed.	4	3	4
Q10	Adding the rules heatmap view to a performance dashboard for EOL will save me time.	4	1	5
Deviation-from-mean heat map				
Q11	The meaning of this heatmap was clear to me.	4	3	3
Q12	The heatmap was intuitive.	5	3	3
Q13	The heatmap provides new information.	4	3	4
Q14	The heatmap improves the speed with which problems can be analyzed.	4	3	4
Q15	I clicked in the heatmap to get more details about a value.	4	4	5
Q16	The extra information revealed after clicking was useful.	5	4	5
Q17	Adding the raw values heatmap view to a performance dashboard for EOL will save me time.	2	1	4
Deviation-from-mean heat map per server				
Q18	The meaning of this heatmap was clear to me.	4	3	3

Q19	The heatmap was intuitive.	4	3	3
Q20	The heatmap provides new information.	3	3	4
Q21	The heatmap improves the speed with which problems can be analyzed.	3	3	4
Q22	I clicked in the heatmap to get more details about a value.	4	3	5
Q23	The extra information revealed after clicking was useful.	4	3	5
Q24	Adding the raw values per server heatmap view to a performance dashboard for EOL will save me time.	3	1	4
Missing values heat map				
Q25	The missing values tab was intuitive.	2	4	3
Q26	The missing values tab was useful.	2	4	2
Line chart				
Q27	The meaning of this chart was clear to me.	5	4	5
Q28	The chart was intuitive.	5	4	5
Q29	The chart provides new information.	4	4	3
Q30	The chart improves the speed with which problems can be analyzed.	4	4	5
Q31	Adding the line chart to a performance dashboard for EOL will save me time.	3	1	4
Histogram				
Q32	I used the extra information provided in the histogram tab.	4	3	5
Q33	Adding the histogram tab to a performance dashboard for EOL will save me time.	2	1	3
Intensity				
Q34	I used the extra information provided in the intensity tab.	4	3	5
Q35	Adding the intensity tab to a performance dashboard for EOL will save me time.	4	5	4
Applicability: bottleneck detection (1 = WEDJAT is very unuseful for this, 5 = WEDJAT is very useful for this)				
Q36	Detecting the server that forms the bottleneck.	5	5	5
Q37	Detecting the load balancer that forms the bottleneck.	3	1	1
Q38	Detecting the instance that forms the bottleneck.	5	4	5
Q39	Detecting the ASPX that forms the bottleneck.	1	1	1
Q40	Detecting the network connection that forms the bottleneck.	1	1	1
Q41	Detecting the hardware that forms the bottleneck.	5	4	3
Q42	Detecting the CPU that forms the bottleneck.	5	4	5
Q43	Detecting the HDD that forms the bottleneck.	5	4	5
Q44	Detecting the memory that forms the bottleneck.	5	4	5

Q45	Detecting other hardware that forms the bottleneck.	5	1	3
Q46	Detecting a thread pool that forms the bottleneck.	1	3	1
Applicability: failure detection				
Q47	Detecting a server failure.	4	3	5
Q48	Detecting a load balancer failure.	3	1	1
Q49	Detecting an instance failure.	4	3	5
Q50	Detecting an ASPX failure.	1	1	1
Q51	Detecting a network failure.	3	1	4
Q52	Detecting hardware failure.	4	3	5
Q53	Detecting a CPU failure.	4	3	5
Q54	Detecting a HDD failure.	4	3	5
Q55	Detecting a memory failure.	4	3	5
Q56	Detecting other hardware failures.	3	1	4
Q57	**Favorite features in WEDJAT:**			
PI	Deviation-from mean heat map, intensity graph			
PII	Rule coverage heat map, intensity graph			
PIII	Rule coverage heat map, deviation-from mean heat map			

4.5 Results of the Field User Study

As discussed in the previous section, we asked the participants (P I, P II and P III) to solve a task and to answer the questions listed in Table 4.4. In a subsequent step, we compared the responses of the participants and tried to bridge differing opinions. In all cases where we identified differing opinions, this was due to a different interpretation of the question. In this section, we discuss the most interesting highlights from the task, questionnaire and anecdotes elicited during the interview.

4.5.1 The Assigned Task

All participants could easily solve the assigned task. An interesting result from the process was that participant I and II found additional bottlenecks during the investigation, which were not directly obvious from the rule coverage heat map. From our heat map, it was only obvious that the processor of web server 3 was overloaded. However, because the participants have experience with the EOL infrastructure, they decided to start an investigation for the other web servers as well. The reason for this is that the load of the application is balanced over the web servers and therefore, they wanted to see whether the processor of the other web servers was overloaded as well. As an investigation they compared the deviation-from-mean heat maps of the different web servers to find out if any of them exhib-

ited abnormal behaviour. They found out that all web server processors had the same problem due to a software error. We found out that our approach did not detect a problem on all these servers because it is trained using supervised learning. This means that for a problem to be detected and analyzed correctly, a similar case should be in the training set, which is difficult for anomalies. This shows that our association rule set either requires better training or a different generation process. We consider this future work.

4.5.2 Questionnaire Highlights

General

The participants' general opinion about WEDJAT was that it is useful and would help them to analyze and diagnose performance problems better and quicker (*Q1, Q2, Q4*, see Table 4.4), even though the usability of the tool has to improve (*Q1*). An example of a usability issue WEDJAT currently has is the following. When the heat maps for a large period of time are being displayed, they are horizontally scrollable. Because the names of the performance counters are on the left side of the heat map, these disappear from the screen as the heat map is being scrolled. While this does not make WEDJAT unusable, it would be more comfortable to have the performance counter names in sight at all times. Improving the usability of WEDJAT by fixing such issues is future work. A promising result is that all participants felt very optimistic about the use of heat maps for performance analysis (*Q3*).

For all views, we asked whether they should be included in a performance dashboard for Exact Online. The answers to these questions were different because the participants had different opinions about what the purpose of a performance dashboard should be. Participant III, the product manager, felt that such a dashboard should contain performance analysis tools while the research engineers believed it should contain a trigger to start performance analysis only (*Q10, Q17, Q24, Q31, Q33, Q35*). A reason for this could be that it is easier for the research engineers to access additional performance data when necessary, while this may be more difficult for the product manager. All participants agreed that the INTENSITY graph is the most important starting point for performance maintenance, as it gives the initial indication that the system is performing relatively slow (*Q35*).

In an open question (*Q57*) we asked the participants what their two favourite WEDJAT features were. It is promising to see that all participants liked the new views on the data (rule coverage heat map, deviation-from-mean heat map and the INTENSITY graph) in WEDJAT best. This indicates WEDJAT offers new and useful information.

Heat Maps

All participants agreed that the heat map views offer new information which helps them analyze performance problems faster (*Q8, Q9, Q13, Q14, Q20, Q21*). During the discussion we found out that the participants thought the heat maps were also

useful for communicating and explaining performance problems to the management and system administrators. An interesting result was that participant I found the heat maps very intuitive, while the other participants were slightly less positive about this (*Q7, Q12, Q19*). From the contextual interview we found out that participant I had a background in statistics, which made him more comfortable with less traditional visualization techniques. An exception to this is the missing values heat map (*Q25, Q26*). Only participant II found this heat map intuitive and useful. The main reason for this is that participant II was the actual participant who requested this feature after the preliminary study. The other participants found this heat map difficult to grasp without extra information, e.g., they would like to receive a 'trigger' when a particular value has been missing, rather than inspect the heat map themselves.

All participants agreed that using the 'traditional' line chart (Section 4.3.6) and histogram (Section 4.3.7) in combination with the heat maps improves the usability of WEDJAT (*Q30, Q32*). In addition, they all considered the deviation-from-mean heat maps to be the most useful ones (*Q11-Q16, Q18-Q23*). This is interesting, as our expectation was that the experts would prefer the rule coverage heat map as it gives a direct indication of the bottleneck component. However, during the experiment we found out that the experts preferred to directly view the deviation-from-mean heat maps of servers on which they have seen performance problems occur before, and inspect those for abnormal behaviour (e.g, dark spots in the heat map). If there was no abnormal behaviour on these servers, they referred to the rule coverage heat map to see if it would give another indication of where to look.

Applicability

All participants agreed mostly on what they believe WEDJAT is and is not useful for. According to them, WEDJAT is most useful for detecting which server (*Q36*), instance (*Q38*) or hardware (*Q41-Q45*) forms a bottleneck, using the rule coverage and deviation-from-mean heat maps. While they believed that WEDJAT currently is not capable of detecting which load balancer (*Q37*) or network connection (*Q40*) forms a bottleneck, the participants agreed the cause for this was that we currently did not monitor performance counters for that hardware. The participants expected WEDJAT to be capable of detecting such bottlenecks after the set of monitored performance counters would be extended. We have also asked whether the participants believed WEDJAT could detect which ASP.NET file forms a bottleneck (*Q39*). They agreed WEDJAT was not suitable for this without extra information. We consider detecting application bottlenecks with WEDJAT as future work.

The participants believed the missing values heat map was most useful for detecting hardware failures (*Q47-Q56*). However, the same limitations as with the bottleneck detection apply: load balancer (*Q48*) and network connection (*Q51*) failures are difficult to detect currently, as no performance counters are monitored for them.

4.5.3 Additional Insights Obtained From The Interview

In this section we will briefly discuss some of the additional insights that we gained during the contextual interview with the participants. The first insight we obtained was that the participants did not like to cope with graphs that exhibit more than one metric at the same time. For example, the deviation-from-mean heat map used more than one colour in the initial version of WEDJAT (green for increasing and red for decreasing values). All participants agreed that this heat map was overly complex and confusing because of the large amount of data. Therefore, we changed the heat map to use one colour instead to represent the absolute deviation from the mean, which was preferred by the participants.

The second insight we obtained was that the participants would like a better integration with existing performance tools such as their own dashboard. In addition, they would like to see error log files such as those from the SQL server and web server integrated with WEDJAT, to display more information for a selected time interval.

Finally, there was some concern with the generation of the association rules and the strength of their diagnosis. It is likely that the generated rules do not cover all performance counters and therefore may give incomplete diagnosis of a problem. A possible solution is to offer an additional heat map which gives an overview of performance counters that exhibited abnormal values for a longer period of time. Such a heat map would solve at least part of the problem of supervised learning (see Section 4.6). Another solution would be to use a continuous training scheme in which the training set is constantly improved through user feedback. The implementation and evaluation of this heat map and a continuous training scheme will be addressed in future work.

4.6 Discussion

4.6.1 The Evaluation Research Questions Revisited

RQ2b-Eval1*: Does the rule coverage matrix provide a good starting point for performance maintenance?*

In our field user study we have seen that the participants appreciated the rule coverage matrix as a starting point for investigating PIOs. The rule coverage heat map was considered useful especially in combination with the deviation-from-mean heat map, which provided some extra information on top of the rule coverage heat map. A possibility to improve the rule coverage matrix is to improve the coverage of the association rule set as discussed in Section 4.5.1. A final result from the field user study is that users prefer the INTENSITY metric as the initial trigger for starting a PIO investigation, while the rule coverage matrix provides the starting point for pinpointing the bottleneck component.

***RQ2b-Eval2**: Are heat maps an appropriate way of visualizing performance data?*
The results of our field study show that the participants were very enthusiastic about using heat maps for performance maintenance. They all believed using heat maps for performance maintenance will speed up the process. In addition, they were optimistic about using heat maps to explain performance problems to non experts such as hosting companies and product management.

4.6.2 Threats to Validity

In this section we discuss the threats to validity of our field user study and our approach for detecting PIOs. For a discussion of the threats to validity of the INTENSITY metric, we refer to Section 3.8.

External validity. We have performed our field study on one industrial multi-server SaaS application. Due to its outright scale and set-up, this application is likely to be representative of other large-scale SaaS applications.

Only three participants participated in our field user study, however, all three are performance experts and have many years of experience with performance maintenance.

Internal validity. We use supervised learning to train our association rule set. While this may form a threat to the validity of anomaly detection, we believe this is actually an advantage for the detection of PIOs. Supervised learning implies we will investigate PIOs which occur more often only, making them more interesting for actual maintenance.

The complexity of the assigned task might have been too easy. While the task assigned is indeed relatively easy to solve, the main focus of the study was to see whether performance experts appreciated the new view on the performance data. Future work will consist of evaluating WEDJAT using more complex tasks in a controlled experiment setting [Cornelissen et al., 2011]. We acknowledge that the low number of participants should be considered a serious threat. However, for the current application, we involved all available performance experts. As such, as future work we consider to extend our study to other systems as well.

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.

Reliability validity. WEDJAT and our implementation for calculating the INTENSITY metric are available for download at `http://swerl.tudelft.nl/bin/view/Main/MTS`.

4.6.3 Lessons Learned

We have learned several lessons from our research:

Performance maintenance is usually done using simple tools while more integrated approaches are desired by the experts (introduction of this chapter). Experts prefer approaches that can be integrated with their existing tools and like to use the same environment as much as possible.

Combining traditional with novel methods for performance data visualization can help experts to understand the novel visualizations quicker (Section 4.5.2).

Experts do not like to cope with graphs that exhibit more than one metric at the same time (Section 4.5.3). In our effort to combine several metrics in one heat map, we found out that the experts did not appreciate this as they found the learning curve for such maps too steep. Instead, they preferred several maps displaying one metric.

It is important not to rely on the rule coverage matrix only when investigating a PIO (Section 4.5.1) in order to prevent being too dependent on the assocation rule set. It is possible that this rule set is not complete. Instead, it is better to combine the results with those from the deviation-from-mean heat maps to get a broader result.

4.7 Related Work

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.

Munawar et al. [2008] 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. [2004]; Zhang et al. [2005] 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. [2011] 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. [2010] 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. [2009] 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. [2012] 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 details about software [Knuth, 1971], 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 [Yan et al., 2012]. 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.

Bergel et al. [2012] 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 [Elbaum and Diep, 2005; Jovic et al., 2011], 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 [Jovic et al., 2011]. 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.

Using Heat Maps for Performance Maintenance. Heat maps have been used for performance analysis before [Fürlinger et al., 2007; Gregg, 2010], but we have evaluated our approach in an industrial setting and on multi-server data.

4.8 Conclusion

In this chapter we have proposed a technique for detecting performance improvement opportunities (PIOs) using association rules and performance counter measurements. We have implemented this technique, together with several novel techniques for the visualization of performance data, in an open source tool called Wed-

JAT. We have evaluated WEDJAT, the novel visualization methods and our approach in a field user study with three performance experts from industry for a large-scale industrial SaaS application. The results of the user study show that WEDJAT helps them to perform performance maintenance easier and faster. In short, this chapter makes the following contributions:

- An approach for using heat maps to analyze the performance of a system and exploit performance improvement opportunities
- The open source tool WEDJAT, which assists during the performance maintenance process
- A field user study in which WEDJAT and the idea of using heat maps for performance analysis are evaluated by three performance experts from industry

In future work we will focus on improving the usability of WEDJAT and the coverage of the used rule set. In addition, we will keep on extending WEDJAT with several new visualization methods. Finally, we will perform a more extended evaluation of our approach in which we will (a) do a true/false positive/negative analysis of our results and (b) assign more complex tasks to solve to the experts.

5

Improving the Diagnostic Capabilities of a Performance Optimization Approach

Understanding the performance of a system is difficult because it is affected by every aspect of the design, code and execution environment. Performance maintenance tools can assist in getting a better understanding of the system by monitoring and analyzing performance data. In previous work, we have presented an approach which assists the performance expert in obtaining insight into and subsequently optimizing the performance of a deployed application. This approach is based on the classification results made by a single classifier. Following results from literature, we have extended this approach with the possibility of using a set (ensemble) of classifiers, in order to improve the classification results. While this ensemble is maintained with the goal of optimizing its accuracy, the completeness (or coverage) is neglected.

In this chapter, we present a method for improving both the coverage and accuracy of an ensemble. By doing so, we improve the diagnostic capabilities of our existing approach, i.e., the range of possible causes it is able to identify as the bottleneck of a performance issue. We present several metrics for measuring coverage and comparing two classifiers. We evaluate our approach on real performance data from a large industrial application. From our evaluation we get a strong indication that our approach is capable of improving the diagnostic capabilities of an ensemble, while maintaining at least the same degree of accuracy.[1]

[1]This chapter contains our work published as technical report *TUD-SERG-2013-015* [Bezemer and Zaidman, 2013].

Understanding the performance of a system is difficult because it is affected by every aspect of the design, code and execution environment [Woodside et al., 2007]. Therefore, tools are being developed which assist experts in understanding the performance of a system. A subset of these tools [Rao and Xu, 2008; Cohen et al., 2004; Zhang et al., 2005], rely on machine learning techniques to correlate low-level system measurements with high-level service objectives (SLOs). A part of these techniques, the so-called supervised learning techniques [Witten and Frank, 2005], use part of the monitored data as input for training the *classifier* used in the performance maintenance system. This classifier can then analyze newly monitored input and reason about, or predict the performance of the system. The quality of the analysis is dependent on the classifier. On the one hand, it is important that the classification made by the classifier is accurate. On the other hand, the range of possible diagnoses the classifier can make, its *coverage*, defines the applicability of the approach on real world problems.

Zhang et al. [2005] and Tan et al. [2010] showed that the quality of a classification can be improved by using an ensemble of classifiers. In an ensemble, classifiers work together to take a better decision than they would on their own. For example, Zhang et al. select the most suitable classifier for classifying a set of measurements, which works well for analyzing performance anomalies. While Zhang et al. and Tan et al. propose methods for increasing the accuracy of the classifier ensemble, these methods do not take the diagnostic capabilities of the ensemble into account. In this chapter, we present an approach which aims at improving both the coverage and accuracy of a classifier ensemble. By improving the coverage of the ensemble, we improve its diagnostic capabilities and we also improve the level of understanding an expert can get from the performance analysis results. In this chapter, we focus on the following research question presented in Chapter 1:

> ***RQ2c**: How can we improve the quality of the diagnosis?*

In short, we make the following contributions:

1. A set of metrics for defining and calculating the coverage of a classifier or ensemble
2. An ensemble maintenance approach for improving both accuracy and coverage of a classifier ensemble
3. An industrial case study in which we evaluate our approach and show that it is capable of improving the diagnostic capabilities of an ensemble, while

maintaining approximately the same degree of accuracy for detecting performance bottlenecks in the system

We do so by extending our approach for performance optimization of deployed applications (Chapter 3). Because we focus on finding performance optimizations instead of anomalies, we let classifiers work together in an ensemble by letting them vote and subsequently select the classification which has received the most votes. As a result, the ensemble is more likely to find structural bottlenecks.

In the rest of this chapter, we first present our problem statement in Section 5.1. In Section 5.2, we give a brief overview of the approach we extended to come to our contributions. We discuss our approach which aims at improving both coverage and accuracy of a classifier ensemble in Section 5.3. We present and discuss the evaluation results of our industrial case study in Section 5.4, 5.5 and 5.6. We present related work in Section 5.7 and conclude our work in Section 5.8.

5.1 Problem Statement

Many performance maintenance approaches are based on supervised learning techniques, i.e., they must be trained using labeled data to be able to reason about the performance of a system. During the training, a (trained) *classifier* is generated which is able to classify new performance data into certain classes. An advantage of using supervised learning instead of unsupervised learning, is that you have more control over what should be considered abnormal cases in the training data. This is interesting for performance optimization approaches, as it allows you to find structural performance issues, which are possibly missed by unsupervised learning.

A classifier can be trained using any training set specified by a performance expert, but the quality of the classification results can be very different. As the reasoning (or *diagnosing*) part of the approach often relies on this classification, the quality of the diagnosis depends on the quality of the classification as well. The quality of a diagnosis is defined by:

1. The *accuracy*, i.e., whether the diagnosed issue is indeed an issue.
2. The *completeness*, i.e., whether the diagnosis provides a complete description of the cause of the issue, giving the expert enough details to start an investigation.

Existing methods for increasing the accuracy of a diagnosis by using an ensemble of classifiers instead of a single classifier [Zhang et al., 2005; Tan et al., 2010] do not consider the completeness of the diagnosis. In this chapter, we propose a method which is based on Zhang et al.'s method for maintaining a classifier ensemble. However, our method aims at improving both the completeness and accuracy of the diagnosis. A metric in which the completeness of a diagnosis can be expressed is coverage.

The coverage of a classifier or ensemble is the set of features it uses to take a decision. In a performance monitoring system, this set may, for example, consist of a subset of the monitored performance metrics. The coverage of a classifier tells us something about its diagnostic capabilities: the more features a classifier or ensemble covers, the wider the range of possible diagnoses it can make, hence, the more complete the diagnosis can be. We focus on the following research question:

> ***RQ2c-1**: How can we improve the coverage of a classifier ensemble, while maintaining at least the same degree of accuracy?*

In Chapter 3, we have presented an approach for performance optimization of deployed Software-as-a-Service (SaaS) applications. In this chapter, we extend this approach with support for a classifier ensemble maintained using our approach for accuracy and coverage improvement. In the next section, we first give background information about our performance optimization approach.

5.2 Background

In Chapter 3, we presented an approach that allows to identify and analyze possible *performance improvement opportunities* (PIOs) in a software system using performance metrics [2] only. To do this, our approach uses a system-specific classifier, generated from data collected during the training phase. The performance of a system is often characterized by unexpected combinations of performance metric values, rather than following simple rules of thumb [Cohen et al., 2004]. Therefore, we generate association rules [Cohen, 1995] from the monitored data[3], as these are capable of representing such complex combinations.

Table 5.1 depicts a rule set and a set of new measurements for a sample (fictituous) system. The sample system consists of server $S1$ with two monitored performance metrics CPU (% CPU Utilization) and MEM (% Memory Usage), and server $S2$ with one monitored performance metric CPU. In Chapter 3, we devised an approach which generates such association rules based on the response time. Central to our approach is the SARatio metric. The SARatio (or *Slow-to-All-actions-ratio*) for a time interval t is defined as:

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

We define an action[4] as slow (i.e., a member of $SLOW_t$) when it belongs to the 15% slowest actions in terms of response time for a particular user of a particular

[2] See `http://www.st.ewi.tudelft.nl/~corpaul/eollist.txt` for the list used in our case study.

[3] See Chapter 3 for a thorough discussion of the approach.

[4] 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 [Koschke and Quante, 2005].

application (or feature) for a time interval t. We consider the other 85% of the actions as a member of the $NORMAL_t$ class. We calculate the response time per user and per application to make sure that actions that have greater resource demands are not automatically considered slow. 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.

We calculate the SARATIO for all time intervals of the training period using a sliding window approach, in which the window contains all actions made during time interval t. As we now have a SARATIO-value for all monitored time intervals, we can identify when the system was running relatively slow. We define the following thresholds for the SARATIO, such that we can classify system load for each interval:

- $high$: system load is typically too high, which makes it perform slow (top 5% SARATIO values)
- med: system load may become or may just have been problematic (medium 10% SARATIO values)
- low: system load is non-problematic (low 85% SARATIO values)

From this definition, classifications which fall into the $high$ class form the most interesting situations, the PIOs, with respect to performance optimization. Using the SARATIO class and the performance metric values monitored during a time interval, we use the JRip algorithm from the WEKA toolkit [Hall et al., 2009] to mine association rules which classify performance measurements into one of the three SARATIO classes.

Our PIO analysis approach exploits the association rules used during the classification process of the PIO detection to find starting points for exploring possible performance improvement opportunities. The goal of our approach is to analyze the information in the rules matched by a measurement and detect clusters of performance metrics that help decide on which server, hardware or software components we must start looking for performance improvements.

Our approach uses a so-called *rule coverage matrix m*. The rows of this matrix contain the performance metrics, the columns depict measurements. The first column, representing the first measurement is initialized to 0. Each time a new measurement is received, the last column of m is copied and the following algorithm is applied:

- Increase $m_{i,j}$ if performance metric i is covered by a $high$ rule at measurement j.

Table 5.1: Sample rule set and performance measurements

Sample association rule set	**Sample measurements**			
1 S1_CPU>80 & S2_CPU>60 → $high$	t	S1_CPU	S1_MEM	S2_CPU
2 S1_CPU>70 & S1_MEM>70 → $high$	0	40	60	50
3 S1_CPU>90 → $high$	1	95	60	50
4 S1_MEM<30 → med	2	98	80	50
5 else → low	3	98	95	80
	4	98	80	50
	5	40	25	50

- Leave $m_{i,j}$ equal to $m_{i,j-1}$ for a med rule
- Decrease $m_{i,j}$ if performance metric i is covered by a low rule at measurement j, with a minimum of 0

We update the value of every $m_{i,j}$ only once for every measurement, even though multiple covering rules may contain the same performance metric. The rationale behind building the rule coverage matrix this way is the following:

1. The rule set 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 performance metric values are in general relatively stable, which means that abnormal values of (combinations of) performance metrics will be exhibited for a longer period of time, i.e., throughout the PIO.
3. When entering this into the rule coverage matrix this way, higher values in m will appear because these values will be increased for performance metrics which occur in adjacent measurements.
4. Eventually, clusters of higher values in m for performance metrics on specific hardware will appear.
5. These clusters can be used to do performance reengineering, e.g., pinpointing a bottleneck component.

The following example illustrates this. Table 5.2 shows the resulting m after applying our approach to Table 5.1. We can see a cluster of higher values at server $S1$ at $t = 4$ and $t = 5$, indicating this server may be a bottleneck.

Our approach for PIO analysis relies on the contents of the rule set used by the classifier. If this rule set contains rules that cover two servers, we can identify one of those two servers as a bottleneck only. Therefore, a higher rule set coverage can lead to a better bottleneck diagnosis.

Table 5.2: Rule coverage matrix for Table 5.1

metric \ t	0	1	2	3	4	5
S1_CPU	0	1	2	3	4	4
S1_MEM	0	0	1	2	3	3
S2_CPU	0	0	0	1	0	0
covered by rules #	5	3	2,3	1,2,3	2,3	4

5.3 Our Approach

In this section, we present our approach for improving both accuracy and coverage of a classifier ensemble. First, we will discuss the metrics used to calculate the *accuracy*, and *coverage* of a classifier and ensemble. In addition, we will present metrics for the *contribution* of a classifier. The contribution is a metric for describing how much a classifier would improve the coverage of an ensemble, if it were a member of the ensemble. After this, we will present our approach for accuracy and coverage improvement of a classifier ensemble, which uses these metrics.

5.3.1 Accuracy

The accuracy of a measurement system is the degree of closeness of measurements of a quantity to that quantity's actual (true) value [BIPM et al., 2008]. We make a distinction between *accuracy* and *balanced accuracy*. Accuracy is the number of times the classifier or ensemble classified a measurement correctly. For systems in which the occurence of certain classes is rare, this metric can give a false judgement of the system. For example, in our approach, the *low* class represents approximately 85% of all classifications, as we expect the load on a system to be non-problematic in most cases (Chapter 3). A classifier which would always classify a measurement as *low*, would have an accuracy of 85%, but it would still be unusable in a production system. Hence, we use the balanced accuracy [Brodersen et al., 2010], which is capable of dealing with such imbalances in the data set by taking the true and false positives (*TP* and *FP*) and true and false negatives (*TN* and *FN*) into account. The balanced accuracy `BA` can be calculated as follows:

$$\texttt{BA} = \frac{0.5 * TP}{TP + FN} + \frac{0.5 * TN}{TN + FP}$$

The classifier mentioned which has an accuracy of 85%, would have a `BA` of approximately 0.5.

5.3.2 Coverage

In order to improve the coverage of a classifier ensemble, we must be able to compare the coverage of two classifiers, so that we can decide whether adding

(or removing) a classifier to the ensemble will increase (or decrease) its coverage. In this section, we introduce metrics for defining the *coverage* of a classifier or ensemble (Section 5.3.2) and for the *contribution* of a classifier (Section 5.3.2).

Throughout this section, we will use the association rule sets in Table 5.3 as an example to illustrate the metrics. These rule sets were synthetically crafted for demonstration purposes. Note that we use the terms association rule set and classifier interchangeably throughout this chapter.

Table 5.3: Sample rule sets

Complete set of monitored performance metrics (M_{ALL}): S1_CPU, S1_MEM, S2_CPU, S3_CPU **Complete set of monitored servers (S_{ALL}):** S1, S2, S3	
Rule set/Classifier A	**Rule set/Classifier B**
1 S1_CPU>80 & S2_CPU>60 → $high$	1 S1_CPU>80 & S2_CPU>60 → $high$
4 S1_MEM<30 → med	1 S3_CPU>90 → $high$
5 else → low	else → low

Classifier Coverage Metrics

An important property of a classifier is the set of performance metrics or hardware components it covers. As explained in Section 5.2, the contents of the rule set are used to perform bottleneck analysis on the system. As a result, the larger the set a classifier covers, the more precise the bottleneck diagnosis can be. We propose to exploit this to improve coverage of an ensemble. For example, two classifiers which cover a disjoint set of servers could be complementary.

Metrics Coverage Percentage (`MCP`) The `MCP` is the percentage of performance metrics of the complete set of monitored metrics M_{ALL} that a rule set R covers. To calculate this percentage, we compose set M_R containing all performance metrics used in rule set R. After this, we calculate which percentage of the complete set M_R covers:

$$\texttt{MCP} = \frac{|M_R|}{|M_{ALL}|} * 100$$

For our example rule sets, this gives:

$$\begin{aligned} M_A &= \{S1\_CPU, S1\_MEM, S2\_CPU\} \\ \texttt{MCP for rule set A} &= \frac{3}{4} * 100 = 75\% \\ M_B &= \{S1\_CPU, S2\_CPU, S3\_CPU\} \\ \texttt{MCP for rule set B} &= \frac{3}{4} * 100 = 75\% \end{aligned}$$

Server Coverage Percentage (SCP) The SCP is the percentage of servers of the set of monitored servers S_{ALL} a rule set R covers. SCP is defined similarly to MCP:

$$\text{SCP} = \frac{|S_R|}{|S_{ALL}|} * 100$$

For our example rule sets, this gives:

$$\begin{aligned} S_A &= \{S1, S2\} \\ \text{SCP for rule set A} &= \frac{2}{3} * 100 = 66,7\% \\ S_B &= \{S1, S2, S3\} \\ \text{SCP for rule set B} &= \frac{3}{3} * 100 = 100\% \end{aligned}$$

The MCP and SCP are an indication of the completeness of a rule set. Note that they are not a metric for describing how accurate the rule set can classify new performance measurements. However, they can help with improving the coverage of an ensemble, as the ideal ensemble can cover up to 100% of M_{ALL} and S_{ALL}. Therefore, we define $\text{MCP}_{ensemble}$ as the percentage of metrics of the complete set of all monitored metrics that all classifiers in the ensemble cover together. Likewise, we define $\text{SCP}_{ensemble}$ as the percentage of servers the ensemble covers. To calculate $\text{MCP}_{ensemble}$, we take the union of M_R of every classifier R in the ensemble. We can calculate $\text{SCP}_{ensemble}$ by taking the union of S_R of every classifier R in the ensemble. The $\text{MCP}_{ensemble}$ and $\text{SCP}_{ensemble}$ can be used to decide whether the coverage of the ensemble increases after adding a classifier. Throughout the rest of this chapter, we will address the monitored performance metrics and servers as features.

Classifier Coverage Vector (CCV) To be able to compare classifiers with each other, we propose to use a vector representation of their contents. The classifier coverage vector (CCV) is such a representation. For a classifier A, we create a vector V_c with $|M_{ALL}|$ elements with value zero, and set value V_{m_i} to $true$ (1) if M_{ALL_i} is in M_A. In addition, we create a vector V_s with $|S_{ALL}|$ elements with value zero, and set value V_{s_i} to 1 if S_{ALL_i} is in S_A. We then concatenate V_c and V_s to create CCV_A. For our example set A this gives:

$$\begin{aligned} V_c &= [1;1;1;0] \\ V_s &= [1;1;0] \\ \text{CCV}_A &= [V_c; V_s] = [1;1;1;0;1;1;0] \end{aligned}$$

Ensemble Coverage Frequency Vector (ECFV) To maintain a classifier ensemble, we must know what the current ensemble coverage is. To keep track of this, we maintain the ECFV, which contains the number of classifiers in the ensemble that cover a feature. We start with an empty ECFV, i.e., all elements are set to zero. Everytime a classifier is added to the ensemble, we add its CCV to the ECFV. Likewise, when we remove a classifier from the ensemble, we subtract its CCV from the ECFV. For our example rule sets, this gives:

$$\begin{aligned} \texttt{ECFV} &= [0;0;0;0;0;0;0] \\ \texttt{CCV}_A &= [1;1;1;0;1;1;0] \\ \texttt{ECFV} + \texttt{CCV}_A &= [1;1;1;0;1;1;0] \\ \texttt{CCV}_B &= [1;0;1;1;1;1;1] \\ \texttt{ECFV} + \texttt{CCV}_A + \texttt{CCV}_B &= [2;1;2;1;2;2;1] \end{aligned}$$

Note that in contrast to the CCV where the elements represent booleans, the elements in ECFV represent the number of classifiers in the ensemble that cover a certain feature. By maintaining the ECFV, we can easily keep an administration of the coverage of the ensemble.

Classifier Contribution

When selecting the best out of two candidate classifiers for addition to the ensemble, we want to select the classifier which makes the greatest contribution to the coverage of the ensemble. When one of the classifiers adds more new features to the ensemble than the other, i.e., causes the largest increase in $MCP_{ensemble}$ and $SCP_{ensemble}$, the decision is clear. Note that this number is the difference in the number of zeros before and after adding CCV to ECFV for a new candidate, and after subtracting CCV from ECFV for a removal candidate.

However, when both classifiers cause an equal increase in the coverage of the ensemble, we must find out which classifier provides the most valuable contribution. This is the classifier which covers the features with the lowest frequency in the ECFV, because adding this classifier would allow us more flexibility when removing a classifier later without affecting $MCP_{ensemble}$ and $SCP_{ensemble}$. Imagine the following $\texttt{CCV}_C$ and $\texttt{CCV}_D$:

$$\begin{aligned} \texttt{ECFV} &= [2;1;2;1;2;2;1] \\ \texttt{CCV}_C &= [1;0;0;0;0;0;0] \\ \texttt{CCV}_D &= [0;1;0;0;0;0;0] \end{aligned}$$

Adding classifier C to the ensemble would increase the coverage of $S1\_CPU$ (the first element of ECFV), which is already covered by two classifiers. Adding

classifier D to the ensemble would increase the coverage of $S1\_MEM$ (the second element of ECFV), which is covered by one classifier in the ensemble only. Therefore, classifier D would make the greatest contribution to the ensemble.

We use the cosine similarity [Baeza-Yates and Ribeiro-Neto, 1999] for finding the classifier with the greatest contribution. The cosine similarity measures the cosine between two vectors to describe their similarity; it is 1 for vectors that are exactly the same ($\cos 0$) and -1 for vectors that are exactly the opposite ($\cos 180$). We chose to use the cosine similarity over other similarity measures for its easy interpretation. By calculating the cosine similarity of the ECFV and the CCV of the candidate classifiers, we can calculate which CCV is the least similar to the ECFV. Because this vector covers features which are covered less in the ECFV than the other would, its contribution is greater. The (cosine) $similarity$ for two vectors V_1 and V_2 is defined as follows:

$$similarity_{V_1,V_2} = \cos\theta \quad = \quad \frac{V_1 \cdot V_2}{\left|\left|V_1\right|\right| \left|\left|V_2\right|\right|}$$

with $V_1 \cdot V_2$ being the Euclidean dot product formula and $\left|\left|V_1\right|\right|$ the norm for vector V_1:

$$V_1 \cdot V_2 = \sum_{i=1}^{n} V_{1_i} V_{2_i} \qquad \left|\left|V_1\right|\right| = \sqrt{\sum_{i=1}^{n} (V_{1_i})^2}$$

For classifiers C and D, $similarity_{\mathtt{ECFV},\mathtt{CCV}_C}$ and $similarity_{\mathtt{ECFV},\mathtt{CCV}_D}$ are calculated as follows:

$$\begin{array}{ll} \mathtt{ECFV} \cdot \mathtt{CCV}_C = 2 & ||\mathtt{ECFV}|| = \sqrt{19} \\ \mathtt{ECFV} \cdot \mathtt{CCV}_D = 1 & \left|\left|\mathtt{CCV}_C\right|\right| = \sqrt{1} \\ & \left|\left|\mathtt{CCV}_D\right|\right| = \sqrt{1} \end{array}$$

$$similarity_{\mathtt{ECFV},\mathtt{CCV}_C} = \tfrac{2}{\sqrt{19}} \qquad similarity_{\mathtt{ECFV},\mathtt{CCV}_D} = \tfrac{1}{\sqrt{19}}$$

This shows that $\mathtt{CCV}_D$ is indeed less similar to ECFV than $\mathtt{CCV}_C$, making D the classifier to select for addition as it makes the greatest contribution to ECFV.

5.3.3 Ensemble Maintenance

Whenever we monitor new performance data, this data may be used to generate a new classifier. Upon generation of such a new classifier, we must decide whether we should add it to the ensemble or not. Algorithm MaintainEnsemble depicts the steps necessary for maintenance of the ensemble. A data buffer is being kept which contains all the newly monitored data. Whenever the data buffer is full, i.e. it has reached the configured size (for example one day or one week), a new

Algorithm 2 MaintainEnsemble(e, d, n)

Require: Ensemble e, DataBuffer d, maximum ensemble size n (0 = unlimited)
Ensure: Trains a new classifier c whenever d is full, and adds it to e when c can increase the coverage of e.

1: **while** $!d.isFull()$ **do**
2: $\quad d.waitForNewData()$
3: **end while**
4: $c = d.trainNewClassifier()$
5: AddClassifier(c, e, n)
6: $d.resetBuffer$

Algorithm 3 CalcNewFeatures(c, e)

Require: Classifier c, Ensemble e
Ensure: Returns the number of new features c contributes to e

1: **if** $c \in e$ **then**
2: $\quad ecfvOld = e.ecfv - c.ccv$
3: $\quad$ **return** $cntZeros(ecfvOld) - cntZeros(e.ecfv)$
4: **else**
5: $\quad ecfvNew = e.ecfv + c.ccv$
6: $\quad$ **return** $cntZeros(e.ecfv) - cntZeros(ecfvNew)$
7: **end if**

classifier c is trained using the data in the buffer as training set. This classifier is then evaluated on a set of test data and its balanced accuracy BA is calculated. This BA is compared with the BA of the classifiers in the ensemble. If the BA of the new candidate is higher than that of any of the classifiers in the ensemble, it replaces that classifier. If the selection of the classifier to remove or add (i.e., the BA is equal to that of one in the ensemble) is ambiguous, we select the classifier which would give the greatest contribution in coverage if it were in the ensemble. We then select this classifier as the winning candidate and make it a member of the ensemble. Algorithm AddClassifier depicts this process.

An important aspect is the amount of space available for the ensemble: this may be either unlimited or limited, e.g., in an embedded system. Note that the ensemble may consist of only one classifier. We will now discuss the maintenance algorithm in the case of unlimited and limited space.

Maintenance of an Ensemble with Unlimited Space

In the case of unlimited space (or when there is still space left in the ensemble), the algorithm for ensemble maintenance is depicted by lines 1 to 5 of Algorithm AddClassifier. In this case we add the classifier if and only if its BA is at least as high as the BA of one of the classifiers in the ensemble.

Algorithm 4 AddClassifier(c, e, n)

Require: Candidate classifier c, ensemble e, maximum ensemble size n (0 = unlimited)
Ensure: If there is enough space in the ensemble, and c has high enough BA, c is added. If there is not enough space, and the BA of c is at least as high as another classifier in the ensemble, a tiebreak is started to decide which classifier should be in the ensemble.

```
 1: if c.getBA() < e.getMinBA() then
 2:     return
 3: end if
 4: if e.size() < n or n == 0 then
 5:     e.addClassifier(c)
 6: else
 7:     for all cl ∈ e.getMinBAClassifiers() do
 8:         feats[] = (cl, CalcNewFeatures(cl, e))
 9:     end for
10:     remCandidate = findMinContr(feats)
11:     if c.getBA() > remCandidate.getBA() then
12:         e.replaceClassifier(remCandidate, c)
13:     else if c.getBA() == remCandidate.getBA() then
14:         if CalcNewFeatures(c, e) > remCandidate.contr then
15:             e.replaceClassifier(remCandidate, c)
16:         else if CalcNewFeatures(c, e) == remCandidate.contr then
17:             if cosSim(c, e) < cosSim(remCandidate, e) then
18:                 e.replaceClassifier(remCandidate, c)
19:             end if
20:         end if
21:     end if
22: end if
```

Maintenance of an Ensemble with Limited Space

Lines 6-22 of AddClassifier depict the case where the ensemble is full. We must then select a candidate for removal from the set of classifiers with the lowest BA in the ensemble (*e.getMinBAClassifiers()*). To do this, we must first calculate the number of new features each classifier in the set of removal candidates contributes to the ensemble using CalcNewFeatures. The number of new features is calculated as described in Section 5.3.2 by counting the number of zeros in the ECFV before and after adding the CCV of the classifier. If the BA of these classifiers is lower than the BA of the classifier c we are trying to add to the ensemble, we can simply replace the classifier with the lowest contribution with c. Otherwise, we calculate the number of new features the candidate classifier c would contribute. If this contribution is greater than that of the removal candidate, we replace that classifier with the new candidate c. If the removal candidate contributes as much new features as c, we must perform a 'tiebreak' to find out which classifier must be in the ensemble. Therefore, we add c only if its cosine similarity with the ECFV is smaller (Sect. 5.3.2). This principle is also used to find the removal candidate which contributes the least in line 10 (only we select the classifier with the largest cosine similarity with ECFV as this classifier contributes the least).

5.4 Experimental Setup

Subject System We performed a case study on data monitored in Exact Online[5] (EOL), an industrial multi-tenant SaaS application for online bookkeeping with approximately 18,000 users. The application currently runs on several web, application and database servers. It is written in VB.NET and uses Microsoft SQL Server 2008.

Process For a period of 65 days, we collected the data of in total 255 performance metrics[6] on 15 servers, every minute in the same database using a .NET background task. In addition, all requests, including the user who made them and the application that was targeted by the request, were logged. We selected 51 days as training data for the classifier ensemble, and reserved 7 days as new test data to calculate the accuracy of every classifier. The final 7 days were used to calculate the accuracy of the ensemble. The labels for the test set were estimated using the SARatio (see also Section 5.2).

The classifier ensemble maintenance approach was implemented as an extension to the performance optimization approach presented in Chapter 3. This approach and the extension are implemented in Java and use the WEKA API [Hall et al., 2009] (in particular the *JRip* algorithm [Cohen, 1995]) to perform the classification. The voting mechanism was implemented as follows. We let each classifier in the ensemble classify the data of every minute that we want to analyze. Each classification made by a classifier for a specific minute counts as a vote. Then, for each minute, we calculate the number of votes for each SARatio class and we select the class with the most votes as the voted classification. The voted classification is the classification made by letting the classifiers in the ensemble work together. In the case of an equal number of votes, we give a preference to the lowest classification. For example, if there are 20 votes, and the med and $high$ class both have received 10 votes, we select med.

Balanced Accuracy In our evaluation we focus on the (mis)classifications of the $high$ class, as this class triggers an action in our approach (Section 5.2). Therefore, we consider measurements wrongly classified as $high$ to be false positives, and measurements which should have been $high$ but were classified differently false negatives. Summarizing, we use the following definitions throughout our evaluation:

- TP: correctly classified as H
- FP: wrongly classified as H
- TN: correctly classified as not H
- FN: wrongly classified as not H

[5] http://www.exactonline.nl

[6] See `http://www.st.ewi.tudelft.nl/~corpaul/eol_list.txt` for a complete list

We use these definitions to calculate the balanced accuracy (Section 5.3.1).

5.5 Evaluation Results

To evaluate our MaintainEnsemble algorithm, we generated ensembles for each maximum size n ranging from $n = 1$ to $n = 51$. We used 51 classifiers trained on one day of training data to populate the ensemble. We calculated the accuracy and BA for the classifiers by classifying 7 days of test data.

We evaluated our algorithm in 8 situations. Each situation represents a combination of the following parameters:

- *The ensemble maintenance approach*: using only accuracy or using a combination of accuracy and coverage (our approach). In the case of using only accuracy, the oldest classifier was removed from the classifier in case of multiple removal candidates. This parameter is used to investigate whether our algorithm indeed improves the coverage of an ensemble, while keeping the accuracy similar.

- *Precision of classifier accuracy*: 1 or 2 digits after the decimal point. With lower precision, classifiers are more likely to have equal accuracy. As a result, the tiebreak in our algorithm is used more often.

- *The type of accuracy used in our algorithm*: accuracy or BA. As explained in Section 5.3, BA is more appropriate for our approach. However, as we designed our algorithm to be generic, it should perform with the accuracy metric as well.

In this section, we discuss the coverage and accuracy of the ensembles generated using the described parameters.

5.5.1 Coverage Evaluation

In each situation, we calculated $SCP_{ensemble}$ and $MCP_{ensemble}$ at the end of every ensemble generation. The results are plotted in Figure 5.1 to 5.4 and summarized in Table 5.4. The average difference is calculated for those cases in which the values for $SCP_{ensemble}$ and $MCP_{ensemble}$ are not equal.

In Figure 5.1 and Figure 5.3, we can see that for 2 digits precision, $SCP_{ensemble}$ and $MCP_{ensemble}$ do not change much. The reason for this is that the high precision results in few conflicts when adding a classifier to the ensemble. Hence, the coverage optimization part of the algorithm is rarely used and the ensembles are generated based on (balanced) accuracy only. In Table 5.4, the average decrease in $SCP_{ensemble}$ looks larger than the increase of $MCP_{ensemble}$ for the case of 2 digits precision accuracy. However, this is due to the different number of servers and

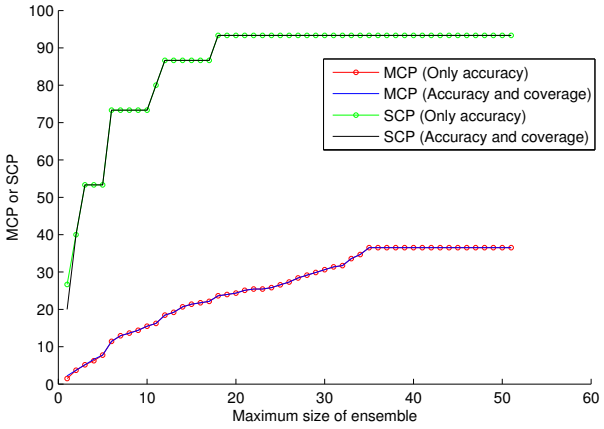

Figure 5.1: MCP / SCP (accuracy with 2 digits precision used in algorithm)

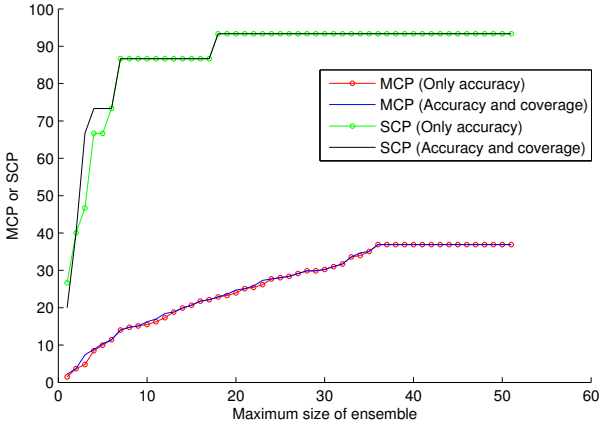

Figure 5.2: MCP / SCP (accuracy with 1 digit precision used in algorithm)

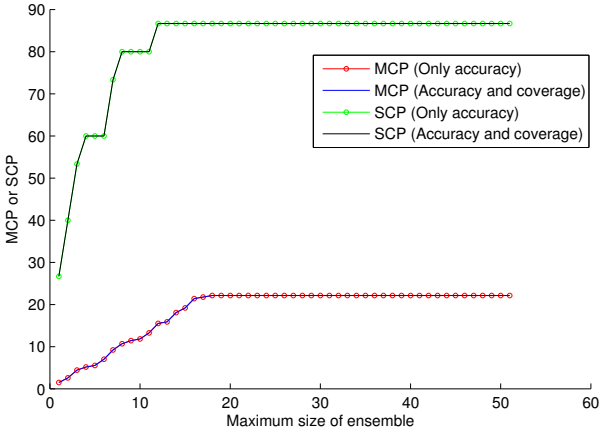

Figure 5.3: MCP / SCP (BA with 2 digits precision used in algorithm)

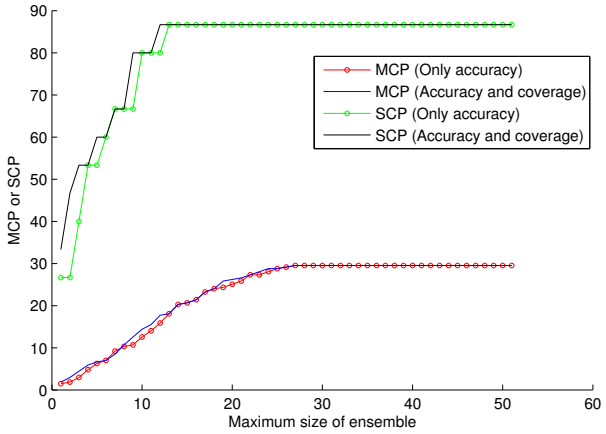

Figure 5.4: MCP / SCP (BA with 1 digit precision used in algorithm)

Table 5.4: Differences of MCP and SCP between approaches

Situation		acc. & cov. wins	acc. & cov. loses	equal	avg. diff.
1 digit precision acc.	SCP	3	1	47	6.6625
	MCP	12	0	39	0.8292
2 digits precision acc.	SCP	0	1	50	-6.6700
	MCP	2	0	49	0.5500
1 digit precision BA	SCP	6	0	45	11.1100
	MCP	15	1	35	0.9913
2 digits precision BA	SCP	0	0	51	0.0000
	MCP	0	0	51	0.0000

metrics analyzed. As a result, a difference of 6.67% for $SCP_{ensemble}$ means a difference of 1 feature, while a difference of 0.55 for $MCP_{ensemble}$ means approximately 1.4 feature. Because our algorithm tries to optimize the total number of covered features, it may give preference to a classifier which covers more metrics but less servers than the classifiers currently in the ensemble.

For the situations with 1 digit precision, the coverage improvement is done more often. In Figure 5.2 and 5.4, we see an improvement of $SCP_{ensemble}$ and $MCP_{ensemble}$ in the ensembles maintained using our approach. For 1 digit precision accuracy, in 4 cases the ensembles generated using our approach covered approximately 1 server more than the ensembles generated using accuracy only. In addition, the ensembles generated using our approach covered an average of approximately 2 metrics more in 12 cases. For 1 digit precision BA, in 6 cases the ensembles generated using our approach covered approximately 1.7 servers more than the ensembles generated using accuracy only. In addition, the ensembles generated using our approach covered an average of approximately 2.5 metrics more in 16 cases. An explanation for the difference in coverage between the ensembles maintained using accuracy and using BA could be that classifiers with high BA

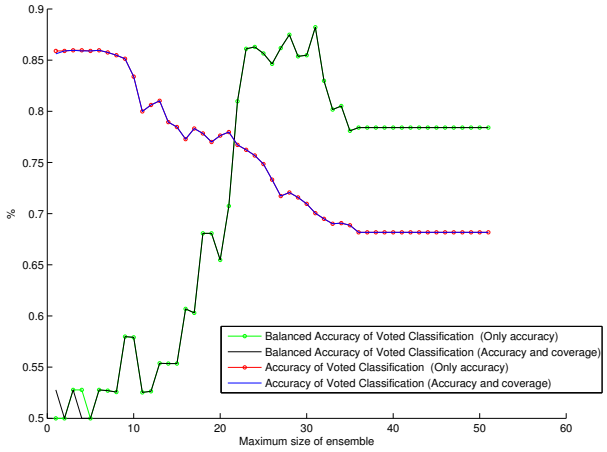

Figure 5.5: Accuracy / BA of the voted classification (accuracy with 2 digits precision used in algorithm)

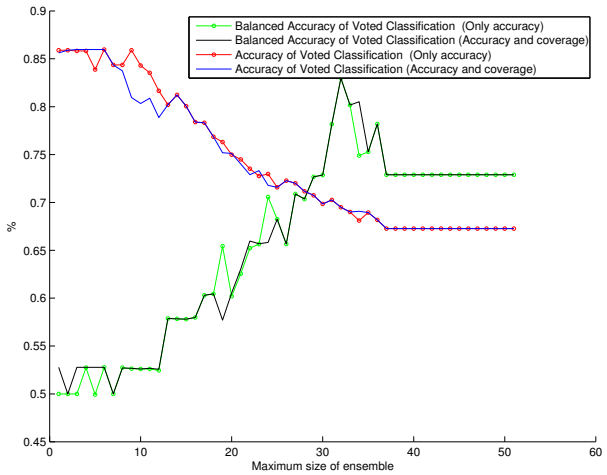

Figure 5.6: Accuracy / BA of the voted classification (accuracy with 1 digit precision used in algorithm)

have higher coverage than classifiers with high accuracy. In future work, we will investigate whether these metrics are related.

5.5.2 Approach Effectiveness Evaluation

To evaluate the effectiveness of our approach, we let each ensemble classify the 7 days of test data using the voting process explained in Section 5.4. We calculate the accuracy and BA of the classification chosen after the voting process was executed by the ensemble. Figure 5.5 to 5.8 depict the results of these calculations. The results are summarized in Table 5.5.

The first observation we make is that for the situations with 2 digits precision,

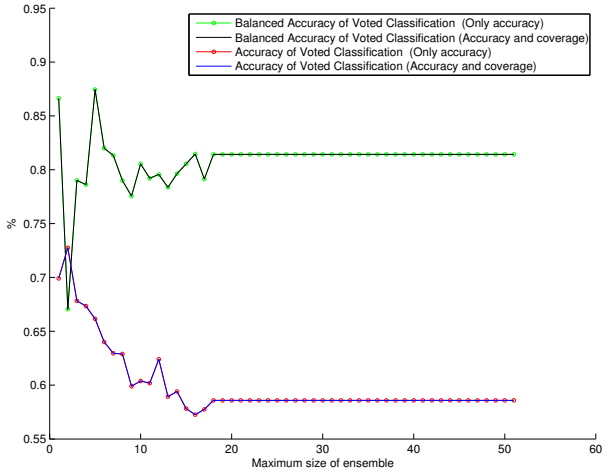

Figure 5.7: Accuracy / BA of the voted classification (BA with 2 digits precision used in algorithm)

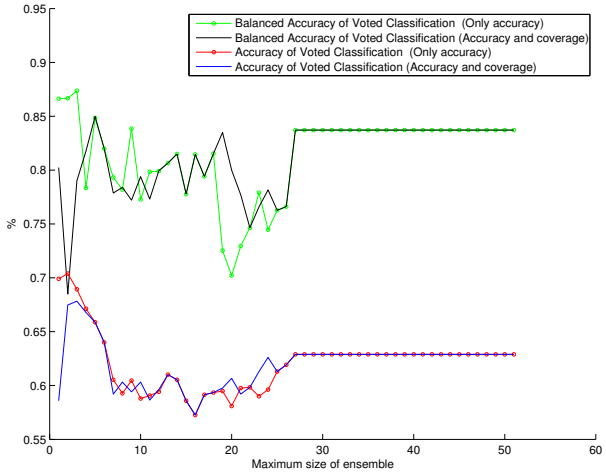

Figure 5.8: Accuracy / BA of the voted classification (BA with 1 digit precision used in algorithm)

Table 5.5: Differences for accuracy and BA of voted classification between approaches

Situation		acc. & cov. wins	acc. & cov. loses	equal	avg. diff.
1 digit precision acc.	Acc.	6	10	35	-0.0091
	BA	12	3	36	0.0023
2 digits precision acc.	Acc.	0	2	49	-0.0017
	BA	1	1	49	0.0000
1 digit precision BA	Acc.	7	8	36	0.0059
	BA	9	7	35	0.0157
2 digits precision BA	Acc.	0	0	51	-0.0054
	BA	0	0	51	-0.0061

there are few differences between the approach which uses accuracy only and our approach. As explained in Section 5.5.1, this is due to the low number of conflicts when adding classifiers. For the cases with 1 digit precision, Table 5.5, Figure 5.5 and 5.7 show that there are more differences. However, the average difference for the ensemble sizes with unequal accuracy and BA is small (at most 1.6%).

Another observation we make, is that the graphs for accuracy and BA are not continuously increasing or decreasing. The reason for this is that we measure the accuracy and BA of the voted classification. Even though we try to increase the average accuracy or BA of the ensemble when adding a classifier, there is no guarantee that a classifier with high accuracy or BA will increase the accuracy or BA of the voted classification. The reason for this is that despite the high accuracy of a classifier, it may accurately classify a different subset of classifications than the classifiers in the ensemble. Hence, it will not always vote for the same classification as the other classifiers in the ensemble during the voting process. As a result, a classifier with high accuracy or BA will not necessarily increase the accuracy or BA of the voted classification, unless the accuracy or BA of the classifier is 100% (which is difficult to achieve). To optimize this, we would have to calculate the accuracy or BA of the voted classification for all possible combinations of classifiers, every time we have trained a new classifier. This would be impractical due to the number of combinations and therefore, we settle for suboptimal results using our approach.

The third observation we make, is the difference between the accuracy and BA of the voted classification. As explained in Section 5.3.1, this is because we have an imbalanced data set, in which a classifier that classifies all measurements as *low* would have high accuracy but low BA. As a result, when maintaining the ensemble using accuracy, classifiers with high accuracy but low BA will be added first and vice versa when using the BA during ensemble maintenance.

5.5.3 Conclusion

From our evaluation, we get a strong indication that our approach is capable of improving the coverage of an ensemble, while maintaining a similar degree of ac-

curacy and BA for the voted classification. Table 5.4 shows the coverage clearly increases, compared to the approach which neglects coverage, while Table 5.5 shows that the difference of the accuracy and BA on average is very small.

5.6 Discussion

5.6.1 The Research Question Revisited

***RQ2c-1**: 'How can we improve the coverage of a classifier ensemble, while maintaining at least the same degree of accuracy?'*

In this chapter, we have presented an approach which aims at improving both accuracy and coverage of an ensemble. In an evaluation on an industrial data set, we have shown that ensembles generated using our algorithm, always cover at least the same number of features, compared to an approach which only tries to improve classifier accuracy. In the cases in which $MCP_{ensemble}$ and $SCP_{ensemble}$ improved, the improvement was on average larger than 1 feature.

In addition, we showed in our evaluation that the average accuracy and BA did not change significantly because of our coverage improvement algorithm. Hence, we can conclude, that for this data set, our approach is able of improving the coverage of a classifier ensemble, while maintaining at least the same degree of accuracy.

5.6.2 Scalability

Our approach is lightweight and transparent; it requires no modification of application code as measurements are done at the operating system level. The only knowledge our approach needs about the system is the set of features which are being monitored to calculate SCP and MCP.

The data set used was 17 GB in size, containing 163 million records. Running the experiment in which 7 days of data were classified by 51 classifiers and 7 days of data were classifed by the various ensembles took approximately 8 hours. The experiment was ran from code which was not optimized for bulk processing, but is used in production for single classifications. Parallelization offers interesting speed-up opportunities for our approach, as the task performed by the classifiers is completely independent. We did not monitor resource usage statistics during the experiment.

5.6.3 Limitations

A limitation of our approach is the fact that we treat features equally when calculating the number of new features contributed by a classifier. While we did not have preference for any type of feature in this experiment, it is possible to assign weights to the types of features.

The fact that we consider all classifiers equal is another limitation of our approach. Our evaluation results could possibly be improved by treating classifiers generated on days during the week and during the weekend differently. Future work must provide more insight into this.

In our work, we used a simple voting algorithm, which selects the classification with the most votes. The same voting algorithm was used throughout the whole experiment. We did not compare other voting algorithms as this is not the core concern of this chapter.

5.6.4 Different Applications

We assume the classifiers use association rule sets, but any type of classifier (e.g., a decision tree) which explicitly states its decision variables can be used. It is not necessary that classifiers within the ensemble are of the same type.

We have implemented our ensemble maintenance approach in a performance optimization system, but we expect it is applicable for any type of classifier ensemble in which the complete set of monitored features is known.

5.6.5 Threats to Validity

We have performed our evaluation on data monitored in an industrial multi-server SaaS application. Due to its outright scale and set-up, this application is likely to be representative of other large-scale SaaS applications, making the monitored data set representative as well. Even so, in future work we will evaluate our approach on other data sets.

Concerning the internal validity of our approach, we acknowledge that we focused on (mis)classifications of the $high$ class during the evaluation. The reason for this is that the $high$ class triggers an action in our approach (Chapter 3), making false positives and false negatives expensive. Therefore, we focused on these properties by evaluating the BA of the ensembles resulting from our approach.

In our evaluation we did not investigate whether our test set contained any anomalies. While this may influence the accuracy of the classifiers or the ensembles, maintaining a representative test set is difficult in real-world systems due to the manual labour involved [Ryu et al., 2012]. Therefore, we decided to select a week of data and estimate the load classifications using the SARatio. As all classifiers and ensembles were using the same test set, we expect that any anomalies in the set will be filtered out by the voting process.

In addition, we did not evaluate whether the diagnosis itself actually improves due to the improved coverage. This should be done in a user study, similar to the one in Chapter 3, making it costly in time. Hence, we will address this in future research.

While we are aware of the existence of different metrics such as support and confidence [Dudek, 2010] for comparing rules within rule sets, we did not use

these as we were interested in the coverage of our ensemble. In future work, we will investigate if these metrics can contribute to our approach.

5.7 Related Work

To the best of our knowledge, there is no other work related to performance maintenance in which a classifier ensemble was maintained based on both coverage and accuracy. In this section we discuss some of the research related most to ours.

Zhang et al. [2005] present a technique for detecting violations of service level objectives using an ensemble of models. Cohen et al. [2005] use this approach to automatically extract a signature from a running system. The ensemble maintenance method used by Zhang et al. and Cohen et al. focuses on classifier accuracy. After training a new classifier, they evaluate it on a synthetic test set, to see if it is more accurate than the classifiers in the ensemble. If the accuracy of the new classifier is higher than all classifiers in the ensemble, it is added to the ensemble. When a classification has to be made, they select the best fitting classifier from the ensemble to make the classification. In this approach, the authors assume unlimited space in the ensemble. As we use a voting process to make a decision, we do not make this assumption, as the number of classifiers in an ensemble influences the voting process. In contrast, using our approach, we are able to find structural bottlenecks, rather than anomalies. In addition, our approach improves the coverage of the ensemble, which is something Zhang et al. and Cohen et al. neglect.

Ryu et al. [2012] propose a technique for building classifiers from new data based on changes in the distribution of the monitored data. Their method is based on the assumption that monitored data which belongs to the current distribution of training data can be classified by the ensemble correctly. In future work we will investigate whether this approach is complementary to ours.

ALERT [Tan et al., 2010] uses an ensemble of classifiers to predict and diagnose performance anomalies. In contrast to ALERT, our system focuses on detecting structural performance improvement opportunities. In addition, ALERT does not improve the coverage of the ensemble.

On the issue of rule set quality improvement, much research has been done in the field of association rule mining [Shankar and Purusothaman, 2009]. Several metrics for comparing association rule sets were defined by Dudek [2010]. However, these metrics are tailored towards association rule sets with binary features. In addition, the metrics are meant for evaluating the quality of association rule set mining algorithms, instead of comparing the quality of the actual mined rule set. In future work, we will investigate if the metrics presented by Dudek can contribute to our approach.

5.8 Conclusion

In this chapter we have proposed a technique for improving both coverage and accuracy of a classifier ensemble. The goal of this is to improve the diagnostic capabilities of the ensemble, by broadening the set of possible diagnoses it can make, while keeping at least the same degree of accuracy. Concretely, this means that the set of association rules, which helps an expert understand the performance of a system, gets extended. As a result, the extended rule set can provide a better understanding of the performance as the possibility of giving a more precise diagnosis is likely to increase.

In an evaluation on a large industrial data set we showed that our approach is capable of improving the coverage of an ensemble. Additionally, our approach does this without affecting the ensemble accuracy. We make these contributions:

1. A set of metrics for defining and calculating the coverage of a classifier or ensemble
2. An ensemble maintenance approach for improving both accuracy and coverage of a classifier ensemble
3. An industrial case study in which we evaluate our approach and show that it is capable of improving the diagnostic capabilities of an ensemble, while maintaining approximately the same degree of accuracy for detecting performance bottlenecks in the system

In future work we will investigate whether the improved diagnostic capabilities indeed result in an improved diagnosis, by performing an expert user study.

6

Detecting and Analyzing Performance Regressions Using a Spectrum-Based Approach

Regression testing can be done by re-executing a test suite on different software versions and comparing the outcome. For functional testing, this is straightforward, as the outcome of such tests is either pass (correct behaviour) or fail (incorrect behaviour). For non-functional testing, such as performance testing, this is more challenging as correct and incorrect are not clearly defined concepts for these types of testing.

In this chapter, we present an approach for detecting performance regressions using a spectrum-based technique. Our method is supplemental to existing profilers and its goal is to analyze the effect of source code changes on the performance of a system. The open source implementation of our approach, SPECTRAPERF*, is available for download.*

We evaluate our approach in a field user study on Tribler, an open source peer-to-peer client. In this evaluation, we show that our approach can guide the performance optimization process, as it helps developers to find performance bottlenecks on the one hand, and on the other allows them to validate the effect of performance optimizations.[1]

[1]This chapter contains our work submitted for journal publication [Bezemer et al., 2013].

Regression testing is performed on a modified program to instill confidence that changes are correct and have not adversely affected unchanged portions of the program [Rothermel and Harrold, 1996]. It can be done by re-executing a test suite on different software versions and comparing the test suite outcome. For functional testing, this is straightforward, as the functionality of a program is either correct or incorrect. Hence, the outcome of such tests is either pass or fail. For non-functional testing, this is more challenging, as correct and incorrect are not clearly defined concepts for these types of testing [Chung et al., 2000].

An example of non-functional testing is performance testing. Two possible reasons for performance testing are:

1. To ensure the software behaves within the limits specified in a service-level agreement (SLA)

2. To find bottlenecks or validate performance optimizations

SLA limits are often specified as hard thresholds for execution/response time, i.e., the maximum number of milliseconds a certain task may take. The main reason for this is that execution time influences the user-perceived performance the most [Jain, 1991]. For performance optimizations, such a limit is not precisely defined, but follows from comparison with the previous software version instead, as the goal is to make a task perform as fast or efficient as possible. Hence, we are interested in finding out whether a specific version of an application runs faster or more efficiently than its predecessor.

As a result, including performance tests in the regression testing process may provide opportunities for performance optimization. In fact, in this chapter we will show that the outcome of these tests can guide the optimization process. Note that we do not distinguish between unexpected (i.e., a bug) and expected performance regressions (e.g., by adding a new feature) in this chapter.

Performance optimization can be done on various metrics. Execution time, which is the most well-known, can be analyzed using traditional profilers. Other examples of metrics which can be optimized are the amount of I/O, memory usage and CPU usage. These metrics are difficult to analyze for software written in higher-level languages, such as Python, due to the lack of tools. Hence, the understanding of how software written in such languages behaves regarding these metrics is often low [Reiss, 2009]. In addition, understanding the performance of a system in general is difficult because it is affected by every aspect of the design, code and execution environment [Woodside et al., 2007].

In this chapter, we propose a method which helps performance experts understand how the performance, including the metrics mentioned above, changes over

the different versions of their software. We address the following research question presented in Chapter 1:

> ***RQ3**: How can we assist developers with the optimization of the performance of a multi-tenant system with regard to its software?*

Our method is supplemental to existing profilers and its goal is to analyze the effect of source code changes on the performance of a system. We achieve this by monitoring the execution of a specific test by two versions of an application and comparing the results using an approach based on *spectrum-based analysis* [Abreu et al., 2007]. The result of our approach is a report which helps a performance expert to:

1. Understand the impact on performance of the changes made to the software on a function-level granularity
2. Identify potential performance optimization opportunities by finding regressions or validate fixes

We evaluate our approach in a field user study on a decentralized peer-to-peer (P2P) BitTorrent client, Tribler [Pouwelse et al., 2008]. In the first part of our study, we analyze the performance history of a component in Tribler by analyzing its unit test suite. In the second part, we analyze the effect of nondeterminism on our approach, by analyzing a 10 minute run of Tribler in the wild.

The outline of this chapter is as follows. In the next section, we first give two motivational examples for our approach. In Section 6.2, we present our problem statement. In Section 6.3, we explain spectrum-based fault localization, a technique which forms the basis for our approach. In Section 6.4, we present our approach for spectrum-based performance analysis. We present the implementation of our approach, called SPECTRAPERF, in Section 6.5. In Section 6.6 and 6.7, we present the setup and results of our user study. We discuss these results and the limitations of our approach in Section 6.8. In Section 6.9, we discuss related work. We conclude our work in Section 6.10.

6.1 Motivational Examples

In this section, we give two real-world motivational examples for the approach presented in this chapter.

Monitoring I/O: In a database system, some queries require the creation of a temporary table[2]. The creation of such a file is often done silently by the database system itself, but is intensive in terms of I/O usage. Finding out which function

[2]For example, for SQLite: `http://www.sqlite.org/tempfiles.html`

causes the temporary table creation can help reduce the I/O footprint of an application. Because I/O takes time, we can detect this behaviour using a traditional profiler, which is based on execution time. However, there is no information available about whether the function resulted in the creation of a temporary table, or that the high execution time was caused by something else. This makes the issue hard to diagnose and optimize. In addition, if a developer has found the cause of the temporary table generation, a fix is difficult to validate due to the same reasons. Using an approach which can automate this process, we can see if a function has started generating temporary tables since the previous version. Then, after fixing it, we can validate if our optimization had the desired effect.

Memory Usage: In many applications, custom caching mechanisms are used. Understanding the impact of these mechanisms on memory and disk I/O is often difficult. By being able to compare versions of software with implementations of different caching mechanisms, we can improve our understanding of their behaviour better. Through this better understanding, we can select, evaluate and optimize the most suitable caching mechanism for an application.

6.2 Problem Statement

By including performance testing in the regression testing process, developers can get feedback about the way their changes to the code impact the performance of the application. This feedback can be used to (1) be warned of undesired negative effects or (2) validate the positive effect of a performance bug fix. To give this feedback, we must do the following:

1. Define which metrics we want to analyze and combine this set of metrics into a *performance profile*, which describes the performance behaviour of a revision
2. Generate such a performance profile for every source code revision
3. Compare the most recent profile with the profile(s) of the preceding revision(s)
4. Analyze which source code change(s) caused the change(s) in performance

In this chapter, we focus on the following research question:

> ***RQ3-1**: How can we guide the performance optimization process by doing performance regression tests?*

To answer this research question, we divide it into the subquestions discussed in the remainder of this section.

> ***RQ3-1a**: How can we monitor performance data and generate a comparable profile out of this data?*

Depending on which metric we want to analyze, we must find a suitable monitor (or profiler) to monitor performance data. Ideally, we want to be able to monitor without needing to change the source code of the application. An additional challenge is that an application may use libraries written in different programming languages, making it more difficult to get fine-grained information about, for example, I/O.

A challenge is formed by the fact that monitoring the same test twice may result in slightly different performance profiles, due to variations in, for example, data contents and current memory usage [Larres et al., 2013]. As such, we must devise a method for comparing these profiles:

> ***RQ3-1b**: How can we compare the generated performance profiles?*

Finally, we must be able to analyze the differences between profiles and report on the functions most likely to cause the change in performance:

> ***RQ3-1c**: How can we analyze and report on the differences between profiles?*

In this chapter, we investigate an approach based on spectrum-based fault localization (see Section 6.4). In this study, we focus on detecting and analyzing performance regression caused by write I/O. We expect that our approach can easily be adapted to work for other performance metrics, which we will verify in future work.

6.3 Spectrum-Based Fault Localization (SFL)

Spectrum-based fault localization (SFL) is a technique that automatically infers a diagnosis from symptoms [Chen et al., 2012]. The *diagnosis* is a ranking of faulty components (block, source code line, etc.) in a system, with the most likely faulty one ranked on top. To make this ranking, observations are made during test execution. These observations express the involvement of components during that specific test case in *block-hit spectra* (hence the name of the technique). These spectra contain a binary value for each component, which represents whether it was executed during that test case. Together with the outcome of a test case (*pass/fail*), these observations form so-called *symptoms*. The outcome of all test cases ($0 =$ *pass,* $1 =$ *fail*) is represented by the *output vector*.

All observations combined with the output vector form the activity matrix, which gives an overview of how component involvement is spread over the execution of a test suite. For every row in the activity matrix, the *similarity coefficient*

Table 6.1: Illustration of SFL [Chen et al., 2012]

Component	Character counter	t_1	t_2	t_3	t_4	t_5	t_6	SC
	def count(string)	[Activity Matrix]						
C_0	let = dig = other = 0	1	1	1	1	1	1	0.82
C_1	string.each_char { |c|	1	1	1	1	1	1	0.82
C_2	if c===/[A-Z]/	1	1	1	1	0	1	0.89
C_3	**let += 2**	1	1	1	1	0	0	1.00
C_4	elsif c===/[a-z]/	1	1	1	1	0	1	0.89
C_5	let += 1	1	1	0	0	0	0	0.71
C_6	elsif c===/[0-9]/	1	1	1	1	0	1	0.89
C_7	dig += 1	0	1	0	1	0	0	0.71
C_8	elsif not c===/[a-zA-Z0-9]/	1	0	1	0	0	1	0.58
C_9	other += 1 }	1	0	1	0	0	1	0.58
C_{10}	return let, dig, other	1	1	1	1	1	1	0.82
	end							
	Output vector (verdicts)	1	1	1	1	0	0	

of that row and the output vector is calculated. The idea behind this is that the row with the highest similarity coefficient indicates the component most likely to be faulty, as this component was executed during most of the failed test cases.

As the similarity coefficient, any similarity coefficient can be used, but Ochiai was proven to give the best results [Abreu et al., 2006], hence we will use it throughout this study. This technique mimics how a human would diagnose an error by looking which parts of the system were active during the failed tests. The Ochiai similarity coefficient (SC) for two binary vectors v_1 and v_2 is defined as:

$$SC = \sqrt{\frac{a}{a+b} * \frac{a}{a+c}} \tag{6.1}$$

with a the number of items in both vectors, b the number of items in v_1 that are not in v_2 and c the number of items that are in v_2 but not in v_1. Table 6.1 illustrates the use of SFL for a function which counts the characters in a string, which is tested by test cases t_1 to t_6. The SC column shows the similarity coefficient, calculated against the output vector, for each line of code. In this example, line C_3 is the line most likely to be faulty as it has the highest SC. In this case, it is clear to see that this is correct as `let` should be increased by 1 instead of 2. In the remainder of this chapter, we present our approach for using spectrum-based analysis for detecting performance regressions.

6.4 Approach

The goal of our approach is to analyze the effect of source code changes on the performance of a system. Ideally, we would like to be able to generate a report explaining per function how much a performance metric changed, compared to the previous source code revision. In this section, we explain our approach for

generating such a report. The idea of our approach is that we summarize the behaviour of an application during the execution of a certain test execution in a *profile*. After an update, we compare the behaviour of our application during the execution of the same test using that profile.

6.4.1 Profile Generation

To be able to report on a function-level granularity, we must also monitor data on this granularity. Therefore, we first automatically instrument (see Section 6.5) all functions in our application that perform I/O writes. The instrumentation code writes an entry to the log for every write action, containing the number of bytes written, the name of the function and the location of the file being written to.

Second, we let the instrumented code execute a test, which generates a log of all write actions made during that execution. This test can be any existing, repeatable test (suite), for example, a unit test or integration test suite. The write actions made to the log are filtered out from this process.

To lessen the effect of variation within the program execution [Larres et al., 2013], for example, due to data content and current memory usage, we execute the test several times for each revision and combine the logged data into a performance profile. The number of times the test must be executed to get an accurate profile is defined by a tradeoff between accuracy and test execution time. Ideally, we would like to run the test many times to get a more precise profile, but this may be impractical, depending on the execution time. A profile is generated by:

- For every function:
  - Calculate the average number of bytes a function writes per call during a test execution (hence: divide the total number of bytes written by that function during the test execution by the total number of calls to that function during the test execution)
  - For every test execution, this will result in a number. Define the highest and lowest values for this number as the accepted range for that revision

Table 6.2 demonstrates this idea. The profile can be read as: 'During revision 1, `flushToDatabase()` wrote an average of 900 to 1500 bytes per call. The function `generateReport()` wrote an average of 1200 to 1604 bytes per call.'

6.4.2 Profile Analysis

In order to assess the changes in performance of a revision, we compare its profile with the profile of the previous revision. While this can be done manually, this is a tedious process and prone to mistakes. As explained in Section 6.3, SFL is a technique which closely resembles the human diagnosis process. Therefore, we propose to automate the comparison using a spectrum-based technique. Another

Table 6.2: Illustration of the profile generation idea

Revision: 1	Avg. # bytes written per call					Profile
Function	t_0	t_1	t_2	t_3	t_4	
flushToDatabase()	**900**	1000	1200	1100	**1500**	[900-1500]
generateReport()	**1200**	1500	1359	**1604**	1300	[1200-1604]

Table 6.3: Illustration of profile comparison

Revision: 2	Average # bytes written			Matrix			SC
Function	t_0	t_1	t_2	(t_0)	(t_1)	(t_2)	
flushToDatabase()	1000	1200	1100	1	1	1	1
generateReport()	2200	2000	1600	0	0	1	0.58
writeCache()	10000	12000	8000	0	0	0	0
Output vector				1	1	1	

advantage of automating this comparison, is that we can use the technique in automated testing environments, such as continuous integration environments. To the best of our knowledge, we are the first to apply spectrum-based analysis to performance.

For every test execution t_i, we record the I/O write data as described in Section 6.4.1. After this, we verify for every function whether the recorded average number of bytes written falls in (1) or outside (0) the accepted range of the profile of the previous revision. As a result, we get a binary vector in which every row represents a function. If we place those vectors next to each other, we get a matrix looking similar to the activity matrix described in Section 6.4.1. Table 6.3 shows sample data and the resulting matrix for three test executions t_i, after comparing them with the profile of Table 6.2. We use three executions here for brevity, but this may be any number.

The analysis step now works as follows. When performance did not change after the source code update, all monitored values for all functions should fall into the accepted ranges of the profile of the previous revision. For three test executions, this is represented by the row [1 1 1] for every function. Any deviations from this mean that the average number of bytes written for that function was higher or lower than the accepted range. By calculating the SC for each row and the 'ideal' vector [1 1 1], we can see whether the average number of bytes written for that function has changed (SC close to 0) or that it is similar to the previous profile (SC close to 1). Using the SC, we can make a ranking of the functions most likely to have been affected by the update. When all SC's are close or equal to 1, the average number of bytes written did not change for any function after the update. [3] The functions with SC closer to 0 are likely to have been affected by the

[3] Note that this terminology is different from that in Section 6.3, in which a SC close to 1 means

update. In Table 6.3, from the *SC* column we can conclude that the performance of the `generateReport()` and `writeCache()` functions were likely to have been affected by the changes made for revision 2.

While the *SC* allows us to find *which* functions were affected by the update, it does not tell us *how* they were affected. For example, we cannot see if `writeCache()` started doing I/O in this version, or that the amount of I/O increased or decreased. Therefore, we append the report with the average number of bytes the monitored values were outside the accepted range (`Impact`). We also display the average number of calls and the `TotalImpact`, which is calculated by the average number of calls to that function multiplied with `Impact`. This allows us to see if the performance decreased or increased and by how much. In addition, we display the difference of the highest and lowest value in the range (`RangeDiff`). The goal of this is to help the performance expert understand the ranking better. For example, when a monitored value is 100 bytes outside the accepted range, there is a difference whether the range difference is small (e.g., 50 bytes) or larger (e.g., 50 kilobytes). Additionally, we display the number of test executions out of the total number of test executions for this revision during which this function wrote bytes. This is important to know, as a function does not necessarily perform I/O in all executions, for example, an error log function may be triggered in only a few of the test executions. A final extension we make to our report is that we collect data for a complete stack trace instead of a single function. The main reasons for this are that (1) the behaviour of a function may be defined by the origin from which it was called (e.g., a database `commit()` function) and (2) this makes the optimization process easier, as we have a more precise description of the function behaviour.

Summarizing, the final report of our analysis contains a ranking of stack traces. In this ranking, the highest ranks are assigned to the traces of which the write behaviour most likely has changed due to the source code changes in this revision. The ranking is made based on the *SC* (low to high) and the `TotalImpact` (high to low). In this way, the stack traces which were impacted the most, and were outside the accepted range in most test executions, are ranked on top. These stack traces are the most likely to represent performance regressions.

Table 6.4 shows the extended report. Throughout this chapter, we will refer to this type of report as the *similarity report* for a revision. From the similarity report, we can see that the average number of bytes written by `generateReport()` has increased relatively a lot compared to revision 1: the value for `Impact` is larger than the difference of the range in the profile. However, as `SC` and `TotalImpact` indicate, this was not the case for all test executions and the average total impact was low. Additionally, we can immediately see from this report that `writeCache()`

the component is likely to be faulty. We do not use the terminology 'faulty', as the effect of an update may be positive or negative. Hence, in this case we feel the more intuitive explanation of a high *SC* is the high similarity compared to the previous version.

Table 6.4: Similarity report for Table 6.3

Revision: 2						
Function	*SC*	# calls	`Impact`	`TotalImpact`	`RangeDifference`	Runs
flushToDatabase()	1	50	0	0	600	3/3
generateReport()	0.58	50	496 B	24.8 KB	404	3/3
writeCache()	0	500	10 KB	5 MB	N/A	3/3

was either added to the code, or started doing I/O compared to the previous version, as there was no accepted range defined for that function. In this case, `Impact` represents the average number of bytes written by that function. We can also see that the `TotalImpact` of the additional write traffic is 5MB, which may be high or low, depending on the test suite and the type of application.

6.5 Implementation

In this section, we present the implementation of our approach called SPECTRAPERF. SPECTRAPERF is part of the open-source experiment runner framework GUMBY[4], and is available for download from the GUMBY repository. Our implementation consists of two parts, the data collection and the data processing part.

6.5.1 Data Collection

To collect data on a function-level granularity, we must use a profiler or code instrumentation. In our implementation, we use Systemtap [Prasad et al., 2005], a tool to simplify the gathering of information about a running Linux system. The difference between Systemtap and traditional profilers is that Systemtap allows dynamic instrumentation of both operating system (*system calls*) and application-level functions. Because of the ability of monitoring system calls, we can monitor applications which use libraries written in different languages. In addition, by instrumenting system calls, we can monitor data which is normally hidden from higher-level languages such as the number of bytes written or allocated.

These advantages are illustrated by the following example. We want to monitor the number of bytes written by application-level functions of an application that uses libraries written in C and in Python, so that we can find the functions that write the most during the execution of a test. Libraries written in C use different application-level functions for writing files than libraries written in Python. If we were to instrument these libraries at the application level, we would have to instrument all those functions. In addition, we would have to identify all writing functions in all libraries. However, after compilation or interpretation, all these functions use the same system call to actually write the file. Hence, if we could

[4] `http://www.github.com/tribler/gumby`

instrument that system call and find out from which application-level function it was called, we can obtain the application-level information with much less effort.

By combining application-level and operating system-level data with Systemtap, we can get a detailed profile of the writing behaviour of our application and any libraries it uses. Systemtap allows dynamic instrumentation [Prasad et al., 2005] by writing *probes* which can automatically instrument the entry or return of functions. Listing 6.1 shows the workflow through (a subset of) the available probe points in a Python function which writes to a file. Note that, if we want to monitor other metrics such as memory usage, we must probe other system calls[5].

The subject system of our user study (see Section 6.6), Tribler, is written in Python. Therefore, we implemented a set of probes to monitor the number of bytes written per Python function. Listing 6.2 shows the description of this set of probes [6]. By running these probes together with any Python application, we can monitor write I/O usage on a function-level granularity.

```
(begin)
=> python.function.entry
  => syscall.open.entry
  <= syscall.open.return
  => syscall.write.entry
  <= syscall.write.return
<= python.function.return
(end)
```

Listing 6.1: Set of available probe points in a writing Python function.

```
probe begin {
  /* Print the CSV headers */
}

probe python.function.entry{
  /* Add function name to the stack trace */
}

probe syscall.open.return{
  /* Store the filehandler and filename of the opened file */
}

probe syscall.write.return {
  /* Add the number of bytes written */
}

probe python.function.return{
  /* Print the python stack trace and the number of bytes written */
}
```

Listing 6.2: Description of probes for monitoring Python I/O write usage.

While Systemtap natively supports C and C++, it does not include native support for probing Python programs. Therefore, we use a patched version of Python,

[5] See `http://asm.sourceforge.net/syscall.html` for a (partial) list of system calls in Linux

[6] See the GUMBY source code for the exact implementation.

which allows Systemtap to probe functions. This version of Python can be automatically installed using GUMBY.

To monitor write actions, we count the number of bytes written per stack trace. To maintain a stack trace, for every Python function we enter (*python.function.entry*), we add the function name to an array for that thread. Then, for all the writes done during the execution of that function, we sum the total number of bytes written per file (*syscall.open.entry* and *syscall.write.entry*). After returning from the Python function (*python.function.return*), we output the number of bytes written per file for the function and the stack trace to that function in CSV format. As a result, we have a CSV file with the size and stack traces of all write actions during the test execution.

6.5.2 Data Processing

After collecting the data, we import it into a SQLite[7] database using R[8] and Python. From this database, we generate a report for each test execution (the *test execution report*) which shows:

1. The stack traces with the largest total number of bytes written.
2. The stack traces with the largest number of bytes written per call.
3. The filenames of the files to which the largest total number of bytes were written.

The test execution report helps with locating the write-intensive stack traces for this execution. In addition, when we have monitored all test executions for a revision, we generate a profile as described in the previous section. We use this profile as a basis to analyze test executions for the next revision.

6.6 Design of the Field User Study

We evaluate our approach in a field user study. The goal of our study is to determine whether performance bottlenecks can be found and optimizations can be verified using our approach. In particular, we focus on these research questions:

> ***RQ3-Eval1**: Does our approach provide enough information to detect performance regressions?*
>
> ***RQ3-Eval2**: Does our approach provide enough information to guide the performance optimization process?*

[7] `http://www.sqlite.org/`
[8] `http://www.r-project.org/`

***RQ3-Eval3**: Does our approach provide enough information to verify the effect of made performance optimizations?*

***RQ3-Eval4**: How does our approach work for test executions which are influenced by external factors?*

In this section, we present the experimental setup of our field user study.

Field Setting: The subject of our study is Tribler [Pouwelse et al., 2008], a fully decentralized open source BitTorrent client. Since its launch in 2006, Tribler was downloaded over a million times. Tribler is an academic prototype, developed by multiple generations of students, with approximately 100 KLOC. Tribler uses Dispersy [Zeilemaker et al., 2013] as a fully decentralized solution for synchronizing messages over the network. Tribler has been under development for 9 years. As a result, all 'low-hanging fruit' performance optimizations have been found with the help of traditional performance analysis tools. One of the goals for the next version is to make it run better on older computers. Therefore, we must optimize the resource usage of Tribler. In the first part of our study, we analyze the unit test suite of Dispersy. In the second part, we analyze a 10 minute idle run of Tribler, in which Tribler is started without performing any actions in the GUI. However, because of the peer-to-peer nature of Tribler, actions will be performed in the background as the client becomes a peer in the network after starting it.

Participant Profile: The questionnaire was filled in by two participants. Participant I is a PhD student with 4 years of experience with Tribler. Participant II is a scientific programmer with 5 years of experience with Tribler, in particular with the Dispersy component. Both participants describe their knowledge of Tribler and Dispersy as very good to excellent.

Experimental Setup: Tribler and Dispersy are being maintained through GitHub [9]. We implemented a script in GUMBY which does the following for each of the the last n commits:

1. Execute the required test 5 times[10], together with the Systemtap probes
2. Load the monitored data into a SQLite database
3. Generate a test execution report for each test execution as explained in Section 6.5.2
4. Compare the output of each run with the previous revision and add this result to the activity matrix m
5. Calculate SC for every row in m

[9] http://www.github.com/tribler

[10] Note that these numbers were chosen based on the execution time of the tests. We have no statistical evidence that this is indeed an optimal value.

6. Generate a similarity report from the activity matrix as displayed in Table 6.4

7. Generate a profile to compare with the next revision

After all commits have been analyzed, the data is summarized in an *overview* report. The overview report shows a graph (e.g., Figure 6.1) of the average number of total bytes written for the test executions of a revision/commit and allows the user to drill down to the reports generated in step 3 and 6, i.e., each data point in the graph acts as a link to the similarity report for that commit. Each similarity report contains links to the test execution reports for that commit. In addition, we added a link to the GitHub diff log for each commit, so that the participants could easily inspect the code changes made in that commit.

In the Dispersy case study, we will analyze the unit test suite of Dispersy for the last 200 revisions[10]. In the Tribler case study, we will analyze a 10 minute idle run of Tribler for the last 100 revisions[10]. Tribler needs some time to shutdown. If for some reason, Tribler does not shutdown by itself, the instance is killed after 15 minutes using a process guard.

Questionnaire: To evaluate our approach, we asked two developers from the Tribler team to rate the quality and usefulness of the reports. We presented them with the reports for the Dispersy and Tribler case study and asked them to do the following:

1. To select the 3 most interesting areas (5-10 data points) on the graphs and rate them 1 (first to investigate) to 3 (third to investigate)

2. To mark with 1-3 the order of the points they would investigate for each area

Then, for each area/phenomenon and each selected data point, we asked them to answer the following:

1. Which position shows the stack trace you would investigate first/second/third, based on the report?

2. Does this lead to an explanation of the phenomenon, and if so, which one?

3. If not, please drill down to the separate test execution reports. Do these reports help to explain the phenomenon?

Finally, we asked them general questions about the reports concerning the usability and whether they expect to find new information about Tribler and Dispersy using this approach. In the next section, we present the results of our study.

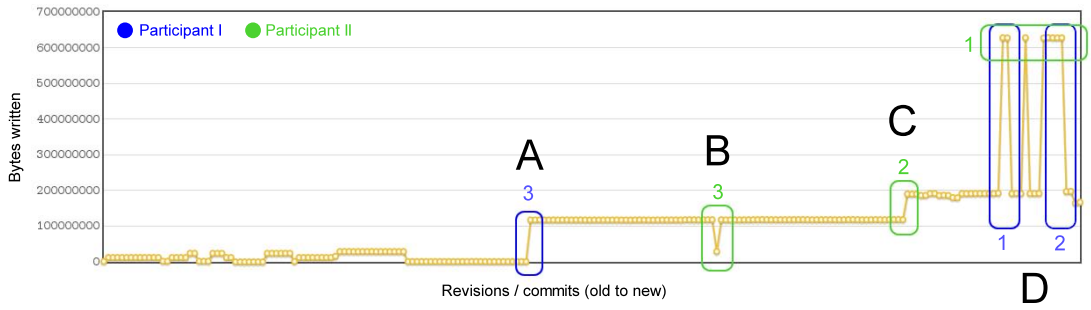

Figure 6.1: Average number of bytes written during an execution of the Dispersy unit test suite for each commit

Table 6.5: Overview of Dispersy evaluation results

Phenomenon	Participant	# Ranking	Helpful?
A	I	1	Yes
B	II	84	No
		test execution reports	No
C	II	1	Partly
		18	Yes
D	I (area 1)	1	Yes
	I (area 2)	1	Yes
	II	1	Yes

6.7 Evaluation

6.7.1 Case Study I: Dispersy Unit Test Suite

Figure 6.1 contains the graph generated during the Dispersy study. In the graph, we highlighted the areas marked by the participants (including their rankings for the most interesting ones). Both participants selected phenomenon D as the most interesting to investigate, due to the increased writes of over 400 MB. Participant I considered the peaks as separate phenomena, while participant II considered them as one event. Furthermore, participant II expected that the cause of phenomenon A was the addition of test cases which resulted in more I/O, hence he selected different phenomena to investigate. Next, we discuss each phenomenon and the way the participants investigated them. Table 6.5 gives an overview of which ranked position the participants analyzed and whether the information provided was useful.

Phenomenon A

The increase was caused by a bugfix. Before this bugfix, data was not committed to the database. *Participant's Analysis:* Participant I indicated that our ranking correctly showed that the database commit function started doing I/O or was called since the previous commit.

Phenomenon B

The drop in writes is due to the order in which the git commits were traversed. Git allows branching of code. In this case, the branch was created just before phenomenon A and merged back into the main branch in phenomenon B. In git, a pull request can contain multiple subcommits. When requesting the git log, git returns a list of all commits (including subcommits) in topological order. This means that every merge request is preceded directly by its subcommits in the log. Hence, these commits were traversed by us first. Figure 6.2 shows an example for the traversal order of a number of commits. This pitfall when mining a git repository is explained in more detail by Bird et al. [2009].

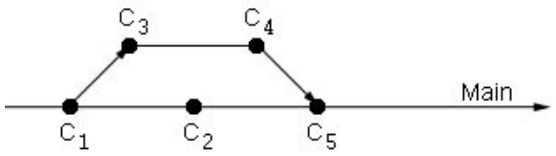

Figure 6.2: Order of traversal of commits in git log (C_1 to C_5)

Likewise, the drop during phenomenon B was caused by testing 'old' code, which lead to a confusing report. This can be avoided by testing only merge requests on the main branch, without subcommits. However, this would also make the analysis of the cause more difficult as the number of changes to the code is larger when subcommits are combined.

Participant's Analysis: Participant II was not able to explain this effect from the report. However, after explaining this effect, the phenomenon was clear to him.

Phenomenon C

In the updated code, a different test class was used which logged more info.

Participant's Analysis: Participant II indicated that he inspected the similarity reports for the highest and the lowest point of the phenomenon. From the report for the highest point, he suspected the #1 ranked stack trace caused the phenomenon. However, as he was not convinced yet, he used the report for the lowest point to verify his suspicions, in which this stack trace was ranked #18. From the combination of the reports, he concluded the number of calls changed from 270 to 400, causing the phenomenon. After inspecting the code changes using the GitHub diff page, he concluded that the different test class was the cause for the increase in the number of calls.

Because the participant was not convinced by the #1 ranked stack trace by itself, we marked this stack trace as 'partly useful' in Table 6.5. Following the advice from Participant II, the reports were extended with the `CallsDiff` metric after the user study. This metric shows the difference in the number of calls to each stack trace, compared to the previous revision.

Phenomenon D

A new test case creates 10k messages and does a single commit for every one of these messages, introducing an additional 435 MB of writes.

Participant's Analysis: Participant I marked this phenomenon as two separate events, for the same reason as explained for phenomenon B. Both participants were able to explain and fix the issue based on the highest ranked stack trace in the report. This was the trace in which a message is committed to the database, had a *SC* of 0 and a `TotalImpact` of 435MB. As the number of calls was 10k, the issue was easy to fix for the participants. The fix was verified using our approach. From the graph, we could see that the total writes decreased from 635MB to approximately 200MB. From the similarity report, we could see that the number of calls to the stack trace decreased from 10k to 8.

6.7.2 Case Study II: Tribler Idle Run

Figure 6.3 contains the graph generated during the Tribler case study. We have marked the areas selected by the participants. It is obvious that this graph is less stable than the Dispersy graph. The reason for this is that the behaviour during the idle run (i.e., just starting the application) is influenced by external factors in Tribler. Due to its decentralized nature, an idle client may still be facilitating searches or synchronizations in the background. As a result, the resource usage is influenced by factors such as the number of peers in the network. Despite this, the participants both selected phenomena C and D as interesting. Participant II explained later that the difference in the choice for A and B was because he preferred investigating more recent phenomena, as their cause is more likely to still exist in the current code. In the remainder of this section, we discuss the phenomena and the participants' evaluations. Table 6.6 summarizes these results for the Tribler case study.

Phenomenon A

During 2 out of 5 test executions, Tribler crashed for this commit. Hence, less messages were received, resulting in a lower average of bytes sent. The actual explanation for this crash cannot be retrieved from these reports, but should be retrieved from the application error logs.

Participant's Analysis: From the reports, participant I was able to detect that less messages were received, but he was not able to detect the actual cause for this. Therefore, he granted the behaviour to noise due to external factors.

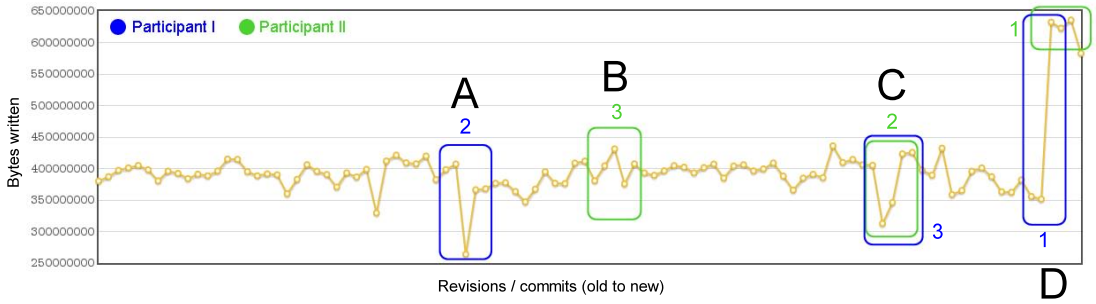

Figure 6.3: Average number of bytes written during an execution of the Tribler idle run of 10 minutes for each commit

Table 6.6: Overview of Tribler evaluation results

Phenomenon	Participant	# Ranking	Helpful?
A	I	45	Yes
B	II	-	No
C	I	1	No
	I	65	Yes
	II	-	No
D	I	1	Yes
	II	24, 26, 27	No
	II	17, 31, 2	Yes

Phenomenon B

No significant changes were found, the variation was due to external factors.

Participant's Analysis: Participant II correctly diagnosed this as noise due to external factors.

Phenomenon C

There was no clear explanation for the drop in resource usage. It was probably due to less active users in the network during the test execution.

Participant's Analysis: Both participants concluded that less messages were received and that the phenomenon did not require further investigation.

Phenomenon D

The reason for the large increase in writes is that the committed code made part of Tribler crash. As a result, the idle run had to be killed after 15 minutes by the process guard. This allowed the part of Tribler that still was running to collect data longer than during the other runs, with the high peak in the graph as the result.

Participant's Analysis: Both participants correctly indicated that more messages

were received and they could both identify the function which caused the large number of writes. They did not directly indicate the partial crash as the cause. Both participants advised to include (1) the actual duration of the execution and (2) a link to the application logs in the report, in order to be able to diagnose such cases better in the future.

In addition, the participants agreed that the function causing the large number of writes used too much resources. This resulted in a performance optimization, which was validated using our approach. From the reports of the validation we could see that the total number of written bytes decreased by 340MB after the fix and from the similarity reports, we could see that the the stack trace disappeared from the report. This means that the function stopped doing write I/O.

6.7.3 Evaluation Results

From our evaluation, we get an indication that our approach is useful for finding performance optimizations. Especially in the case of a test which is repeatable, such as the Dispersy test suite, our approach leads to detection of performance regressions which can be optimized. For test suites which are influenced by external factors, such as the Tribler idle run, our analysis results require deeper investigation and may show more phenomena which are either difficult to explain using our reports, or simply do not lead to performance optimizations.

Even so, the participants were able to correctly analyze and diagnose 3 out of 4 phenomena in the Dispersy report and 2 out of 4 phenomena in the Tribler report. The participants indicated, that with little more information, they would have been able to correctly diagnose all phenomena. These results are summarized in Table 6.7. Together with the participants, we concluded that the reports miss the following information:

1. The `CallsDiff` metric, which displays the difference in the number of calls to a function via the path showed in the stack trace between two commits
2. A link to the application log, so that the user of the report can check for the exit code and whether any exceptions occurred during the test execution
3. The total duration of the test execution
4. An explanation of (or solution to) the 'git log order' effect, explained in Section 6.7.1

After the user study, two phenomena out of two that could be optimized, were optimized after the case study based on our reports. In addition, both of these optimizations could be validated using our approach after they were made. During the case study, a phenomenon was also correctly explained to be a validation of a performance bugfix. Finally, according to the participants, four out of the

Table 6.7: Summary of field user study results

Dispersy	Participant	Correct?	Tribler	Participant	Correct?
A	I	Yes	A	I	Partly
	II	-		II	-
B	I	-	B	I	-
	II	No		II	Yes
C	I	-	C	I	Yes
	II	Yes		II	Yes
D	I	Yes	D	I	Partly
	II	Yes		II	Partly

five phenomena which did not represent a performance regression, were easy to diagnose.

In Table 6.5 and 6.6, we see that in the Dispersy study the problem was indicated by the top ranked stack trace in most cases. In the Tribler study, this is not the case, but the lower ranked stack traces were selected because of their high negative impact. If we would rank the traces by the *SC* and absolute value of `TotalImpact` (instead of exact value), the traces would have had a top 3 rank as well. Hence, we can conclude that the ranking given by our approach is useful after a small adjustment. An observation we made was that the participants all used the `TotalImpact` as a guideline for indicating whether the change in behaviour of a stack trace was significant enough to investigate further. After this, they checked the *SC* to see in how many test executions the behaviour was different. This indicates that the ranking should indeed be made based upon a combination of these two metrics, and not by the *SC* or `TotalImpact` alone.

6.8 Discussion

6.8.1 The Evaluation Research Questions Revisited

***RQ3-Eval1**: Does our approach provide enough information to detect performance regressions?*

From our evaluation, we conclude that our reports provide, after adding the information explained in Section 6.7.3, enough information for detecting performance regressions. In our study, two out of two detected regressions were diagnosed correctly by the participants.

***RQ3-Eval2**: Does our approach provide enough information to guide the performance optimization process?*

Our evaluation showed that our approach provides enough information for guiding the performance optimization process as this user study alone resulted in two

optimizations (Dispersy phenomenon D and Tribler phenomenon D) that have immediately been carried through in the respective projects.

***RQ3-Eval3**: Does our approach provide enough information to verify the effect of made performance optimizations?*

Our approach provides enough information to validate the two optimizations made after the user study. In addition, the participants were able to validate a performance fix made in the history of Dispersy. The participants indicated the optimizations would have been easier to validate if the difference in number of calls for each stack trace was shown in the reports, hence, we will add this in future work.

***RQ3-Eval4**: How does our approach work for test executions which are influenced by external factors?*

From our Tribler case study, we get an indication that our approach can deal with influence from external factors, as the participants were able to completely explain 2 out of 4 performance phenomena and partly explain the remaining 2. However, the results should be treated with more care than for a test which is not influenced by external factors, as they are more likely to represent noise due to those factors. In future work, we will do research on how we can minimize the effect of external factors.

6.8.2 Scalability & Limitations

For the moment, the overhead of our approach is considerable, mostly due to the monitoring by Systemtap. However, to the best of our knowledge, Systemtap is the only available option for monitoring Python code with such granularity. In addition, our approach is meant to run in testing environments and as we do not take execution time into account in our analysis, we do not see overhead as a limitation. In future work, we will investigate other monitoring tools, including existing profilers such as cProfile[11]. However, these tools are limited in the metrics they can monitor, i.e., they cannot monitor I/O traffic with the granularity Systemtap can.

In this chapter, we focused on write I/O. We set up our tooling infrastructure such that the monitoring component can easily be exchanged for another component that is able to monitor different metrics. Hence, by changing the monitoring component, our approach can analyze other performance metrics. In addition, we will investigate how we can rank stack traces on a combination of these metrics, rather than on one metric only. This would help in making a trade-off between the various performance metrics while optimizing.

Another limitation is that we compare a version with its predecessor only. In future work, we will investigate if comparing with more versions can lead to new insights, such as the detection of performance degradation over longer periods.

[11] `http://docs.python.org/2/library/profile.html`

In our approach we do not deal with errors that occurred during the test executions. When no profile could be generated for a revision, we simply compare with the last revision that has a profile. In future work, we will investigate how we can inform the user about errors better, for example by using information from the application logs in our reports.

6.8.3 Threats to Validity

We have performed our field study on an application which has been under development for 9 years and is downloaded over a million times. This application is well-developed and 'low-hanging fruit' optimizations are already done, because of the importance of performance for Tribler due to its peer-to-peer nature. The user study was carried out with developers who have considerable experience with the application.

Concerning the internal validity of our approach, we acknowledge that using the range of monitored values in the profiles is not a statistically sound method. However, due to the low number of test executions, we feel that using a value such as the standard deviation does not add to the reliability of the profiles. In addition, our evaluation shows that we can achieve good results with this approach.

A threat to the validity of our evaluation is that we tested all commits instead of just the merge commits. As a result, we encountered crashing code more often, as these commits do not necessarily provide working code. In addition, it added some phenomena which are difficult to explain without knowing this effect (see Section 6.7). However, after making the participants aware of this effect, they both agreed it would be easy to detect in future investigations.

6.9 Related Work

Spectrum-based analysis has been successfully used before for fault localization [Abreu et al., 2007; Chen et al., 2012]. To the best of our knowledge, we are the first to apply spectrum-based analysis to performance.

Comparison of execution profiles and the detection of performance regressions have received surprisingly little attention in research. Savari and Young [2000] has proposed a method which works for frequency-based profiling methods. Our approach works for any type of metric on a function-level granularity.

Bergel et al. [2012] have proposed a profiler for Pharo which compares profiles using visualization. In their visualization, the size of an element describes the execution time and number of calls. Alcocer [2012] extends Bergel's approach by proposing a method for reducing the generated callgraph. These visualizations require human interpretation, which is difficult when the compared profiles are very different [Bergel et al., 2012]. Our approach provides a textual ranking, which we expect to be easier to interpret. However, we believe that the work of Bergel

et al., Alcocer and our approach can be supplemental to each other, and we will investigate this in future work.

Foo et al. [2010] present an approach for detecting performance regressions by mining performance repositories. Their approach to detect performance regressions is similar to the approach for PIO analysis presented by us in Chapter 3. Both approaches extract association rules from a historical data set. However, their approach then calculates the difference in confidence of the rules between the historical data set and the new data set to detect changes in the performance metrics. Our approach uses the historical data set directly to try and find situations in which the system was performing relatively slow. As indicated in Section 5.6, we will investigate how using rule confidence can improve our approach in the future.

Jiang et al. [2009] analyze readily available execution log files to see if the results of a new load test deviate from previous ones. The advantage of this approach is that it does not introduce extra monitoring overhead. However, this also limits the granularity with which regression analysis can be performed. This is also demonstrated by the granularity of their case studies: in three conducted case studies, they analyze system and application-wide tasks such as finding the optimal DBMS configuration. Our approach does not have such a limitation. However, this comes at the cost of increased overhead.

Nguyen et al. [2012] propose an approach for detecting performance regressions using statistical process control techniques. Nguyen et al. use control charts to decide whether a monitored value is outside an accepted range. The violation ratio defines the relative number of times a value is outside this range. Control charts and the violation ratio are similar to our profile approach. The approach of Nguyen is more statistically sound than our approach, however, we expect that this is not necessarily an improvement when using a small number of test executions. The main difference in the approach used by Nguyen and our approach is the granularity. Their approach identifies performance regressions in system-level metrics, while our approach identifies regressions on the function-level, making analysis of the regression easier. In future work, we will investigate how our approach and Nguyen's approach can complement each other.

Horky et al. [2013] and Heger et al. [2013] propose approaches for integrating performance tests into the unit test suite. Their approaches require the generation of new unit tests, while our approach can be attached to existing test suites.

6.10 Conclusion

In this chapter, we proposed a technique for detecting and analyzing performance regressions using a spectrum-based approach. By comparing execution profiles of two software versions, we report on the functions of which the performance profile changed the most. This report can be used to find regressions or to validate performance optimizations. In this chapter, we focused on optimizing write I/O,

but our approach can easily be extended to other metrics such as read I/O, memory and CPU usage by changing the monitoring component.

In a field user study, we showed that our approach provides adequate information to detect performance regressions and guides the performance optimization process. In fact, our field user study resulted in two optimizations made to our subject system. To summarize, we make the following contributions:

1. An approach for the detection and analysis of performance regressions
2. An open-source implementation of this approach, called SPECTRAPERF
3. A field user study in which we show that our approach guides the performance optimization process

Revisiting our research questions:

***RQ3-1a**: How can we monitor performance data and generate a comparable profile out of this data?* We have proposed an approach using Systemtap to monitor data and we have showed how to generate a comparable profile from this data.

***RQ3-1b**: How can we compare the generated performance profiles?* We have presented our approach for using a spectrum-based technique to compare performance profiles, and provide a ranking of the stack traces which were most likely to have changed behaviour. This ranking is made based on the *similarity coefficient* compared to the previous performance profile, and the *total impact* of a source code change on performance. In our user study, we showed the ranking was useful in 6 out of 8 cases and helped the participants find two optimizations.

***RQ3-1c**: How can we analyze and report on the differences between profiles?* We have showed how we report on the data and we have evaluated this reporting technique in a field user study. During this study, we analyzed the performance history of the open-source peer-to-peer client Tribler and one of its components, Dispersy. The field user study resulted in two optimizations, which were also validated using our approach. During the user study, we found that our approach works well for repeatable tests, such as a unit test suite, as the participants were able to explain 3 out of 4 performance phenomena encountered during such a test using our approach. We also received indication that it works well for a test which was influenced by external factors, as the participants were able to explain 2 out of 4 performance phenomena completely and could partly explain the remaining 2 for such a test.

***RQ3-1**: How can we guide the performance optimization process by doing performance regression tests?* We have showed that our approach for spectrum-based performance analysis can guide the performance optimization process by detecting performance regressions. The results of our field user study alone, resulted in two optimizations to Tribler and Dispersy.

In future work, we will focus on extending our approach to monitor different performance metrics such as memory and CPU usage. Additionally, we will investigate how we can report on trade-offs between these metrics.

7 Conclusion

In this thesis, we have focused on performance optimization of multi-tenant applications. Our research was done in three parts. In the first, we investigated the differences between multi-tenant and single-tenant software, to find the consequences of multi-tenancy for software performance. From this investigation, we found that a multi-tenant application must be optimized at two levels:

1. At the hardware level, due to the increased number of customers (or tenants) sharing the same hardware
2. At the software level, due to the increased number of customers (or tenants) sharing the same software

In the second and third part of our research, we focused on investigating methods that assist the performance expert with finding and analyzing performance bottlenecks at the hardware and software level. Multi-tenancy comes in various variants, some of which closely resemble the multi-instance approach. Therefore, performance analysis and optimization approaches for multi-tenant applications should support these variants. As a result, the approaches presented by us are applicable to a wide range of applications, including multi-tenant and multi-instance applications. We evaluated our approaches in several industrial case studies, for which we worked closely together with performance experts from industry.

7.1 Summary of Contributions

The main contributions of this thesis are:

- A clear, non-ambiguous definition of a multi-tenant application. (Chapter 2)
- An overview of the challenges of developing and maintaining scalable, multi-tenant software. (Chapter 2)
- A conceptual blueprint of a multi-tenant architecture that isolates the multi-tenant concern as much as possible from the base code. (Chapter 2)

- A case study of applying this approach to an industrial application. (Chapter 2)

- An approach for detecting and analyzing performance improvement opportunities (PIOs) using association rules, performance counters and the SARATIO metric. (Chapter 3)

- A proof-of-concept case study in which we show that the SARATIO can be estimated using association rules and performance counters. (Chapter 3)

- An evaluation of our approach for PIO analysis done by a performance expert. (Chapter 3)

- An approach for using heat maps to analyze the performance of a system and exploit performance improvement opportunities. (Chapter 4)

- The open source tool WEDJAT, which assists during the performance maintenance process. (Chapter 4)

- A field user study in which WEDJAT and the idea of using heat maps for performance analysis are evaluated by three performance experts from industry. (Chapter 4)

- A set of metrics for defining and calculating the coverage of a classifier or ensemble. (Chapter 5)

- An approach for improving both accuracy and coverage of a classifier ensemble. (Chapter 5)

- An industrial case study in which we evaluate our approach and show that it is capable of improving the diagnostic capabilities of an ensemble while maintaining approximately the same degree of accuracy. (Chapter 5)

- An approach for the detection and analysis of performance regressions. (Chapter 6)

- An open-source implementation of this approach, called SPECTRAPERF. (Chapter 6)

- A field user study in which we show that our approach guides the performance optimization process. (Chapter 6)

7.2 The Research Questions Revisited

7.2.1 RQ1: What are the differences between a single-tenant and a multi-tenant system?

In Chapter 2, we have presented the key characteristics of multi-tenancy: hardware resource sharing, high degree of configurability and shared application and database instance. From these characteristics, we have deducted the main benefits of multi-tenancy:

- Higher utilization of hardware resources.
- Easier and cheaper application maintenance.
- Lower overall costs, allowing to offer a service at a lower price than competitors.
- New data aggregation opportunities.

Unfortunately, multi-tenancy also has its challenges and even though some of these challenges exist for single-tenant software as well, they appear in a different form and are more complex to solve for multi-tenant applications. These challenges are:

- Performance
- Scalability
- Security
- Zero-downtime
- Maintenance

Keeping these challenges in mind, we have come up with a reengineering pattern for transforming a single-tenant to a multi-tenant application. In a case study on an industrial research prototype, we showed that our pattern forms a guiding process for quickly and efficiently transforming a single-tenant into a multi-tenant application.

7.2.2 RQ2: How can we assist developers with the optimization of the performance of a multi-tenant system with regard to its hardware?

Because tenants share hardware resources in a multi-tenant application, it is important to optimize the application at the hardware level. We have divided RQ2 into three subquestions, which will be revisited in this section.

RQ2a: How can we detect and analyze hardware bottlenecks?

In Chapter 3, we have presented our approach for detecting performance improvement opportunities (PIOs), situations during which the performance of an application could possibly be improved. In this approach, we focused on detecting which hardware components form the bottleneck of a system. We do this by analyzing system-wide performance metrics at the times at which an application is running relatively slow. To perform this analysis, we monitor such metrics for a period of time and use an association rule mining algorithm to generate a set of association rules. These association rules then assist us to classify the performance of the system and to find hardware bottlenecks.

We have evaluated our approach in a case study on Exact Online, in which we analyzed 271 performance metrics, monitored on 18 servers during 66 days of normal execution. Together with a performance expert from Exact, we investigated a random sample of the detected PIOs. In this random sample of 12 detected PIOs, we confirmed 10 cases as real PIOs, which comes down to a precision of 83%. In addition, we showed that in 4 out of 12 cases, the diagnosis given by our approach was completely correct and in 6 out of 12 cases it was partly correct. We compared this to the overload detection mechanism currently implemented by Exact, which uses a threshold for the average response time. We manually analyzed 5 situations classified as overload by this mechanism. Two of these situations were actual overload situations, which means that this mechanism had a precision of 40% in our case study. In contrast, our PIO analysis approach correctly classified all 5 of these situations.

RQ2b: How can we report and visualize the diagnosis of the bottleneck component(s)?

In Chapter 4, we present a visualization method for this approach which uses heat maps. By using heat maps, performance experts can quickly get an indication of (1) the problematic component(s) and (2) for how long these components have been problematic. We have implemented this visualization method in an open source tool, called WEDJAT.

In a field user study, we have evaluated WEDJAT by letting three performance experts from industry investigate a performance problem in a real system using WEDJAT only. First, we let them investigate the issue for 1.5 hours using WEDJAT. During this investigation, we constantly questioned them for suggestions and feedback based on their actions (a so-called *contextual interview*). In addition, we asked the participants to cooperate and discuss their ideas out loud to elicit more detailed feedback. After this, we asked them to fill in a questionnaire. After filling out the questionnaire, the answers of the participants were compared and discussed with them, especially when they were different from each other.

All participants were able to solve the assigned task. From the results of the questionnaire followed that there is added value in using heat maps and WEDJAT

for performance analysis, especially in combination with traditional visualization techniques such as line charts and histograms. An additional result was that the participants selected the new views on performance data as their favourite feature of WEDJAT. All participants expected WEDJAT would help them investigate performance issues easier in the future.

Two of the participants found additional bottlenecks during the investigation. However, these were more difficult to detect using WEDJAT only. From our evaluation, we concluded that this was due to the quality of the association rule set used to detect and analyze PIOs, which leads to the research done to address research question RQ2c.

RQ2c: How can we improve the quality of the diagnosis?

The approaches presented in Chapter 3 and 4 rely on the use of association rules to detect the bottleneck component(s). In Chapter 5, we present a method to improve the diagnostic capabilities of these association rules. We do this by extending the association rule set so that it covers more metrics and servers. With this broadened set of metrics covered, the association rule set is more likely to give a more detailed diagnosis of the bottleneck.

First, we extended our PIO analysis approach with the capability of using multiple classifiers, which work together in an *ensemble*. Existing methods for maintaining such an ensemble focus on improving the accuracy of the ensemble, while neglecting the range of possible diagnoses it may make. Therefore, in Chapter 5, we presented an approach which aims at both improving the coverage and accuracy of a classifier ensemble. We have introduced metrics for defining the coverage of a classifier or ensemble and for the contribution of a classifier. Using these metrics, we can compare classifiers and select the classifier which would contribute the most if it were added to the ensemble.

We have evaluated our approach in a case study on an industrial data set, in which we compared ensembles which were maintained using accuracy only and which were maintained using both coverage and accuracy. We have compared the algorithms in 8 situations in which the algorithm parameters for the classifier accuracy precision, ensemble maintenance approach and the type of accuracy used varied. From this evaluation followed that ensembles generated using our algorithm, which is based on the combination of accuracy and coverage, always cover at least the same number of features, compared to an approach which only tries to improve classifier accuracy.

In addition, we showed that the accuracy of the ensembles generated using our approach did not differ significantly from the ensembled generated using the accuracy-only approach. Therefore, we get strong indication that our approach is capable of increasing the coverage and hence, the diagnostic capabilities of our PIO analysis approach.

7.2.3 RQ3: How can we assist developers with the optimization of the performance of a multi-tenant system with regard to its software?

On the one hand, multi-tenant applications must be optimized at the hardware level. On the other hand, it is important to optimize multi-tenant applications at the software level, as many tenants share the same application instance. In Chapter 6, we present our approach for detecting performance regressions. By detecting performance regressions, we can (1) make sure performance does not decrease after a software update and (2) validate the effect of a performance fix. Our approach uses spectrum-based analysis to find functions of which the performance profile has changed since the previous version. These findings are summarized in a report, which can be used by the developer to analyze and fix performance regressions.

In a user study on a peer-to-peer BitTorrent client, Tribler, we show that our approach assists developers with the detection of software bottlenecks. In this user study, we analyzed (1) the performance history of the unit test suite of Dispersy, a module in Tribler, and (2) the performance history of a 1-hour idle run in Tribler. To evaluate our approach, we asked two developers from the Tribler team to rate the quality and usefulness of the reports generated by our approach, using a questionnaire. In this questionnaire, the participants were asked to select and analyze the most interesting phenomena in the reports.

The participants were able to correctly analyze and diagnose 3 out of 4 phenomena in the Dispersy report and 2 out of 4 phenomena in the Tribler report. The participants indicated, that with little more information, such as the difference in the number of calls and information from the application log, they would have been able to correctly diagnose all phenomena.

After the user study, two out of two phenomena that could be optimized, were optimized based on our reports. This lead to two actual optimizations to the production code of Tribler. These optimizations were validated using our approach after inspecting the reports generated for those code changes.

To summarize, in our field user study we have showed that our approach guides the performance optimization process, as with our approach, the developers were able to (1) find opportunities for performance optimization and (2) validate the optimizations made.

7.3 Recommendations for Future Work

A multitude of interesting open issues are worth investigating. In the following we suggest what we believe to be the most important recommendations for future work.

7.3.1 Multi-Tenancy

In the field of multi-tenancy, we recommend the following future work:

1. Automated Test Methodology

In Chapter 2, we transformed a single-tenant application into a multi-tenant one. We manually tested the resulting application to verify that the multi-tenant requirements were fulfilled and correctly implemented. A (semi-) automated approach should be developed, which assists developers with the testing of a multi-tenant application. This approach should also be capable of testing more complex properties of a multi-tenant application, such as tenant placement and workflow configuration.

Tsai et al. [2010, 2013] have published promising results on parts of such a testing framework. However, their work should be evaluated in an industrial setting.

Search-based approaches for stress test generation form a promising starting point for automated performance testing of multi-tenant applications [Garousi et al., 2008; Briand et al., 2005; Di Penta et al., 2007].

2. Zero-Downtime

As software evolves, it is necessary to perform software updates. A common approach to deploy these updates is by using planned downtime [Momm and Krebs, 2011]: take the system offline and perform the necessary updates. This downtime takes place at moments during which the least customers are in the system, e.g., in weekends. In a multi-tenant application, customers may come from different places in the world, making it difficult to find a time during which system utilization is low. Therefore, it is necessary to come up with approaches, which do not require planned downtime, for online evolution of multi-tenant applications. Self-adaptive systems and dynamic SOA binding [Canfora et al., 2008b] form promising starting points for tackling the zero-downtime challenge.

3. Workflow Configuration

In Chapter 2, we posed workflow configuration as a key challenge of multi-tenancy. While work has been done to address this challenge, e.g., [Pathirage et al., 2011; Mietzner et al., 2009b], existing research lacks an evaluation of the proposed techniques in a large-scale industrial setting. Such evaluations should be done in order to reassess this challenge.

7.3.2 Bottleneck Detection and Analysis

In Chapter 3, we presented an approach for detecting and analyzing performance improvement opportunities (PIOs). To improve this approach, we recommend future work in the following areas:

1. Case Studies on Different Systems

We have focused on one industrial multi-tenant application (Exact Online) in our evaluation. While we expect our case study subject to be representative of other multi-tenant applications, more case studies should be done on different applications, in order to generalize our results. Additionally, our approach should be evaluated using other types of workloads, such as scientific workloads.

2. Application Bottlenecks

In our PIO detection approach, we have focused on detecting hardware bottlenecks. An interesting opportunity could be formed by using our approach to detect software bottlenecks. This could be done by monitoring and analyzing software performance metrics. Such metrics could, for example, be calculated using application heartbeats [Hoffmann et al., 2010]. Future research should give more insight on whether our approach can detect software bottlenecks.

7.3.3 Diagnosis Visualization

In Chapter 4, we have presented our visualization technique for PIOs, which uses heat maps. This technique is implemented in the open source tool WEDJAT, for which we recommmend the following:

1. Improving Wedjat

As pointed out in the discussion of the results of the field user study in Chapter 4, the usability of WEDJAT should be improved. This should be done together with a user experience expert, in order to make the PIO analysis as clear and fast as possible. Additionally, the visualization should be extended with support for diagnoses made by ensembles, as suggested in Chapter 5.

7.3.4 Diagnosis Quality

The quality of the diagnosis of the approach for PIO analysis presented in Chapter 3, on which Chapter 4 and 5 are based, strongly depends on the quality of the trained classifier used. In Chapter 5, we have proposed an algorithm for improving this quality, however, we expect there is still a lot to gain.

1. Training Data Selection

An important part of the equation is formed by the data used to train the classifier. While we now simply use new data as it is being monitored, we should use more clever ways to train classifiers. This can be done, for example, by preprocessing the data to filter any anomalous events. Another possibility is to selectively use new data as it becomes available, as suggested by Ryu et al. [2012]. More research should be done on how the training data used by our approach can be selected in a more educated manner.

An interesting opportunity may be formed by using user feedback to improve the quality of the diagnosis. By using user feedback, it becomes possible to include or exclude specific periods from the training data. In addition, classification results can be included or excluded from the result set based on this user feedback. By combining this with machine learning, it may be possible to train better classifiers. More research should be done on the inclusion of user feedback in the training data selection process.

2. Type of Classifier Used

Another part which influences the quality of the diagnosis is the type of classifier used. In this thesis, we used association rules generated using the JRip algorithm from the WEKA tool kit, as experimentation showed that this algorithm yields good results for our data sets. While we designed our approach to be flexible towards the type of classifier used, a more thorough evaluation should be done to find how different data sets work with different types of classifiers or association rule sets generated by algorithms other than JRip.

3. (Semi-)Automated Diagnosis Validation

More research should be done on how diagnosis quality can be validated in a (semi-)automated manner. Currently, the only method for validating the quality of a diagnosis is by an expert, which is costly and time-consuming. While this applies to performance in general, it is especially the case in an industrial setting.

4. Heterogeneity of Classifiers

In our ensemble maintenance approach, we have treated all classifiers as equal. This means that classifiers trained using data (partly) monitored on different types of days such as weekdays, days in the weekend and holidays, are treated equally. More research should be done on how these classifiers perform and whether treating them unequally can improve the diagnosis made by a classifier ensemble.

7.3.5 Spectrum-Based Performance Analysis

In Chapter 6, we have presented our approach for the detection of performance regressions used a spectrum-based technique. For this approach, we recommend the following future research:

1. Comparison with Multiple Versions

In our approach, we compared each revision with its direct predecessor only. This makes it difficult to detect performance degradation of longer periods. Future research should show whether comparing a revision with more versions can lead to new insights.

2. Report Extension

From the field user study described in Chapter 6, we concluded that the reports generated by our approach could be improved by extending them with metrics such as the difference in the number of calls and the inclusion of data from the application log. In addition, more performance metrics, such as CPU and memory usage, should be monitored and analyzed. When more performance metrics are included in the report, developers must make the tradeoff between these metrics when optimizing their software. Future work should lead to an approach which helps developers make this tradeoff.

3. Minimize Effect of External Factors

In the case study on Tribler, the test suite was heavily influenced by noise due to external factors. While this is primarily caused by the design of the test suite and the nature of the application, future research should investigate how the influence of external factors can be minimized in the generated reports.

Bibliography

Abreu, R., Zoeteweij, P., and van Gemund, A. (2006). An evaluation of similarity coefficients for software fault localization. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 39–46.

Abreu, R., Zoeteweij, P., and Van Gemund, A. J. (2007). On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION)*, pages 89–98. IEEE.

Agrawal, H., Alberi, J., Horgan, J., Li, J., London, S., Wong, W., Ghosh, S., and Wilde, N. (1998). Mining system tests to aid software maintenance. *Computer*, 31(7):64 –73.

Agrawal, R., Imieliński, T., and Swami, A. (1993). Mining association rules between sets of items in large databases. In *Proceedings of the SIGMOD international conference on Management of data (SIGMOD)*, pages 207–216. ACM.

Alcocer, J. P. S. (2012). Tracking down software changes responsible for performance loss. In *Proceedings of the International Workshop on Smalltalk Technologies (IWST)*, pages 3:1–3:7. ACM.

Appleby, K., Fakhouri, S., Fong, L., Goldszmidt, G., Kalantar, M., Krishnakumar, S., Pazel, D., Pershing, J., and Rochwerger, B. (2001). Oceano-SLA based management of a computing utility. In *International Symposium on Integrated Network Management (IM)*, pages 855–868. IEEE.

Baeza-Yates, R. A. and Ribeiro-Neto, B. (1999). *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Bergel, A., Bañados, F., Robbes, R., and Binder, W. (2012). Execution profiling blueprints. *Software: Practice and Experience*, 42(9):1165–1192.

Berrendorf, R. and Ziegler, H. (1998). PCL – the performance counter library: A common interface to access hardware performance counters on microprocessors. Technical Report FZJ-ZAM-IB-9816, Central Institute for Applied Mathematics – Research Centre Juelich GmbH.

Bezemer, C.-P., Milon, E., Zaidman, A., and Pouwelse, J. (2013). Detecting and analyzing performance regressions using a spectrum-based approach. Technical Report TUD-SERG-2013-020, Delft Univ. of Technology.

Bezemer, C.-P. and Zaidman, A. (2010). Multi-tenant SaaS applications: maintenance dream or nightmare? In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, pages 88–92. ACM.

Bezemer, C.-P. and Zaidman, A. (2013). Improving the diagnostic capabilities of a performance optimization approach. Technical Report TUD-SERG-2013-015, Delft Univ. of Technology.

Bezemer, C.-P. and Zaidman, A. (2014). Performance optimization of deployed software-as-a-service applications. *Journal of Systems and Software*, 87(0):87 – 103.

Bezemer, C.-P., Zaidman, A., Platzbeecker, B., Hurkmans, T., and 't Hart, A. (2010). Enabling multi-tenancy: An industrial experience report. In *Proceedings of the 26th International Conference on Software Maintenance (ICSM)*, pages 1–8. IEEE C.S.

Bezemer, C.-P., Zaidman, A., van der Hoeven, A., van de Graaf, A., Wiertz, M., and Weijers, R. (2012). Locating performance improvement opportunities in an industrial software-as-a-service application. In *Proceedings of the 28th International Conference on Software Maintenance (ICSM)*, pages 1–10. IEEE C.S.

BIPM, IEC, IFCC, ILAC, IUPAC, IUPAP, ISO, and OIML (2008). International vocabulary of metrology - basic and general concepts and associated terms (VIM), 3rd edn. JCGM 200: 2008.

Bird, C., Rigby, P., Barr, E., Hamilton, D., German, D., and Devanbu, P. (2009). The promises and perils of mining git. In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pages 1–10.

Breitgand, D., Henis, E., and Shehory, O. (2005). Automated and adaptive threshold setting: Enabling technology for autonomy and self-management. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*, pages 204 –215. IEEE.

Briand, L. C., Labiche, Y., and Shousha, M. (2005). Stress testing real-time systems with genetic algorithms. In *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, GECCO '05, pages 1021–1028. ACM.

Brodersen, K. H., Ong, C. S., Stephan, K. E., and Buhmann, J. M. (2010). The balanced accuracy and its posterior distribution. In *International Conference on Pattern Recognition (ICPR)*, pages 3121–3124. IEEE.

Canfora, G., Fasolino, A. R., Frattolillo, G., and Tramontana, P. (2008a). A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. *Journal of Systems and Software*, 81(4):463–480.

Canfora, G., Penta, M. D., Esposito, R., and Villani, M. L. (2008b). A framework for QoS-aware binding and re-binding of composite web services. *Journal of Systems and Software*, 81(10):1754 – 1769.

Cecchet, E., Marguerite, J., and Zwaenepoel, W. (2002). Performance and scalability of EJB applications. In *Proceedings of the 17th SIGPLAN Conference on OO-programming, systems, languages, and applications (OOPSLA)*, pages 246–261. ACM.

Chen, C., Gross, H.-G., and Zaidman, A. (2012). Spectrum-based fault diagnosis for service-oriented software systems. In *International Conference Service-Oriented Computing and Applications (SOCA)*, pages 1–8. IEEE.

Cherkasova, L., Ozonat, K., Mi, N., Symons, J., and Smirni, E. (2008). Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 452 –461. IEEE.

Chong, F., Carraro, G., and Wolter, R. (2006). Multi-tenant data architecture. `http://msdn.microsoft.com/en-us/library/aa479086.aspx`.

Chung, L., Nixon, B., Yu, E., and Mylopoulos, J. (2000). *Non-functional Requirements in Software Engineering*. Kluwer Academic Publishers.

Cohen, I., Goldszmidt, M., Kelly, T., Symons, J., and Chase, J. S. (2004). Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *Proceedings of the Symposium on Operating Systems Design & Implementation*, pages 231–244. USENIX Association.

Cohen, I., Zhang, S., Goldszmidt, M., Symons, J., Kelly, T., and Fox, A. (2005). Capturing, indexing, clustering, and retrieving system history. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 105–118. ACM.

Cohen, W. W. (1995). Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 115–123. Morgan Kaufmann.

Cornelissen, B., Zaidman, A., and van Deursen, A. (2011). A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering*, 37(3):341–355.

Correa, S. and Cerqueira, R. (2010). Statistical approaches to predicting and diagnosing performance problems in component-based distributed systems: An experimental evaluation. In *Proceedings of the International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 21–30. IEEE.

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

Di Penta, M., Canfora, G., Esposito, G., Mazza, V., and Bruno, M. (2007). Search-based testing of service level agreements. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1090–1097. ACM.

Dubey, A. and Wagle, D. (2007). Delivering software as a service. *The McKinsey Quarterly*, 6:1–12.

Dudek, D. (2010). Measures for comparing association rule sets. In *Artificial Intelligence and Soft Computing*, volume 6113 of *LNCS*, pages 315–322. Springer.

Elbaum, S. and Diep, M. (2005). Profiling deployed software: assessing strategies and testing opportunities. *IEEE Transactions on Software Engineering*, 31(4):312–327.

Foo, K. C., Jiang, Z. M., Adams, B., Hassan, A. E., Zou, Y., and Flora, P. (2010). Mining performance regression testing repositories for automated performance analysis. In *International Conference on Quality Software (QSIC)*, pages 32–41. IEEE.

Fuchs, E. and Jackson, P. E. (1969). Estimates of distributions of random variables for certain computer communications traffic models. In *Proceedings of the first symposium on Problems in the optimization of data communications systems*, pages 205–230. ACM.

Fürlinger, K., Gerndt, M., and Dongarra, J. (2007). On using incremental profiling for the performance analysis of shared memory parallel applications. In *Proceedings of the International Euro-Par Conference*, volume 4641 of *LNCS*, pages 62–71. Springer.

Ganek, A. G. and Corbi, T. A. (2003). The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18.

Garousi, V., Briand, L. C., and Labiche, Y. (2008). Traffic-aware stress testing of distributed real-time systems based on {UML} models using genetic algorithms. *Journal of Systems and Software*, 81(2):161 – 185.

Goldszmidt, M., Cohen, I., Fox, A., and Zhang, S. (2005). Three research challenges at the intersection of machine learning, statistical induction, and systems. In *Proceedings of the 10th conference on Hot Topics in Operating Systems - Volume 10*, HOTOS'05, pages 10–10, Berkeley, CA, USA. USENIX Association.

Gregg, B. (2010). Visualizing system latency. *ACM Communications*, 53(7):48–54.

Guéhéneuc, Y.-G. and Albin-Amiot, H. (2001). Using design patterns and constraints to automate the detection and correction of inter-class design defects. In *Proceedings of the International Conference on Technology of Object-Oriented Languages (TOOLS)*, pages 296–306. IEEE C.S.

Guo, C. J., Sun, W., Huang, Y., Wang, Z. H., and Gao, B. (2007). A framework for native multi-tenancy application development and management. In *Proceedings of the 9th International Conference on E-Commerce Technology (CEC) and the 4th International Conference on Enterprise Computing, E-Commerce, and E-Services (EEE)*, pages 551–558. IEEE CS.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA data mining software: an update. *SIGKDD Exploration Newsletter*, 11:10–18.

Hallsteinsen, S., Hinchey, M., Park, S., and Schmid, K. (2008). Dynamic software product lines. *Computer*, 41(4):93–95.

Hamilton, J. (2010). Overall data center costs. `http://perspectives.mvdirona.com/2010/09/18/OverallDataCenterCosts.aspx` (last visited on May 20th, 2010).

Hashemian, R., Krishnamurthy, D., and Arlitt, M. (2012). Web workload generation challenges - an empirical investigation. *Software: Practice and Experience*, 42(5):629–647.

Heger, C., Happe, J., and Farahbod, R. (2013). Automated root cause isolation of performance regressions during software development. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 27–38.

Hoffmann, H., Eastep, J., Santambrogio, M. D., Miller, J. E., and Agarwal, A. (2010). Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *Proceedings of the 7th international conference on Autonomic computing*, ICAC, pages 79–88. ACM.

Holtzblatt, K. and Jones, S. (1995). Human-computer interaction. chapter Conducting and analyzing a contextual interview (excerpt), pages 241–253. Morgan Kaufmann.

Horky, V., Haas, F., Kotrc, J., Lacina, M., and Tuma, P. (2013). Performance regression unit testing: A case study. In *Computer Performance Engineering*, volume 8168 of *LNCS*, pages 149–163. Springer.

Jacobs, D. and Aulbach, S. (2007). Ruminations on multi-tenant databases. In *Datenbanksysteme in Business, Technologie und Web (BTW), 12. Fachtagung des GI-Fachbereichs Datenbanken und Informationssysteme (DBIS), Proceedings 7.-9. März*, volume 103 of *LNI*, pages 514–521. GI.

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

Jansen, S., Brinkkemper, S., Ballintijn, G., and van Nieuwland, A. (2005). Integrated development and maintenance of software products to support efficient updating of customer configurations: A case study in mass market ERP software. In *Proceedings of the International Conference Soft. Maintenance (ICSM)*, pages 253–262. IEEE.

Jansen, S., Houben, G.-J., and Brinkkemper, S. (2010). Customization realization in multi-tenant web applications: Case studies from the library sector. In *Web Engineering*, volume 6189 of *Lecture Notes in Computer Science*, pages 445–459. Springer Berlin Heidelberg.

Jiang, Z. M., Hassan, A., Hamann, G., and Flora, P. (2009). Automated performance analysis of load tests. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 125–134. IEEE.

Jovic, M., Adamoli, A., and Hauswirth, M. (2011). Catch me if you can: performance bug detection in the wild. In *Proceedings of the International Conference on Object oriented programming systems languages and applications (OOPSLA)*, pages 155–170. ACM.

Kaplan, J. M. (2007). SaaS: Friend or foe? In *Business Communications Review*, pages 48–53. http://www.webtorials.com/abstracts/BCR125.htm.

Knuth, D. E. (1971). An empirical study of FORTRAN programs. *Software: Practice and Experience*, 1(2):105–133.

Koschke, R. and Quante, J. (2005). On dynamic feature location. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 86–95. ACM.

Kwok, T. and Mohindra, A. (2008). Resource calculations with constraints, and placement of tenants and instances for multi-tenant SaaS applications. In *Proceedings of the International Conference on Service-Oriented Computing (ICSOC)*, volume 5364 of *LNCS*, pages 633–648.

Kwok, T., Nguyen, T., and Lam, L. (2008). A software as a service with multi-tenancy support for an electronic contract management application. In *Proceedings of the International Conference on Services Computing (SCC)*, pages 179–186. IEEE C.S.

Laine, P. (2001). The role of SW architecture in solving fundamental problems in object-oriented development of large embedded SW systems. In *Proceedings of the IEEE/IFIP Working Conference on Software Architecture (WICSA)*, pages 14–23. IEEE C.S.

Larres, J., Potanin, A., and Hirose, Y. (2013). A study of performance variations in the mozilla firefox web browser. In *Proceedings of the Thirty-Sixth Australasian Computer Science Conference - Volume 135*, ACSC, pages 3–12. Australian Computer Society, Inc.

Li, X. H., Liu, T., Li, Y., and Chen, Y. (2008). Spin: Service performance isolation infrastructure in multi-tenancy environment. In *Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC)*, volume 5364 of *LNCS*, pages 649–663. Springer.

Lin, H., Sun, K., Zhao, S., and Han, Y. (2009). Feedback-control-based performance regulation for multi-tenant applications. In *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS)*, pages 134–141. IEEE.

Malek, S., Medvidovic, N., and Mikic-Rakic, M. (2012). An extensible framework for improving a distributed software system's deployment architecture. *Software Engineering, IEEE Transactions on*, 38(1):73–100.

Malik, H., Jiang, Z. M., Adams, B., Hassan, A., Flora, P., and Hamann, G. (2010). Automatic comparison of load tests to support the performance analysis of large enterprise systems. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, pages 222–231. IEEE.

Malone, C., Zahran, M., and Karri, R. (2011). Are hardware performance counters a cost effective way for integrity checking of programs. In *Proceedings of the sixth ACM workshop on Scalable trusted computing (STC)*, pages 71–76. ACM.

Mertz, S., Eschinger, C., Eid, T., Swinehart, H., Pang, C., Wurster, L., Dharmasthira, Y., and Pring, B. (2010). Forecast analysis: Software as a service, worldwide, 2009-2014. *Gartner*, G00201597.

Mietzner, R., Metzger, A., Leymann, F., and Pohl, K. (2009a). Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In *Proceedings of the ICSE Workshop on Principles of Engineering Service Oriented Systems (PESOS)*, pages 18–25. IEEE.

Mietzner, R., Metzger, A., Leymann, F., and Pohl, K. (2009b). Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In *Proceedings of the ICSE Workshop on Principles of Engineering Service Oriented Systems*, PESOS, pages 18–25. IEEE Computer Society.

Momm, C. and Krebs, R. (2011). A qualitative discussion of different approaches for implementing multi-tenant SaaS offerings. In *Software Engineering (Workshops)*, volume 184 of *LNI*, pages 139–150. GI.

Müller, J., Krüger, J., Enderlein, S., Helmich, M., and Zeier, A. (2009). Customizing enterprise software as a service applications: Back-end extension in a multi-tenancy environment. In *Proceedings of the 11th International Conference on Enterprise Information Systems (ICEIS)*, volume 24 of *Lecture Notes in Business Information Processing*, pages 66–77. Springer.

Munawar, M. A., Jiang, M., and Ward, P. A. S. (2008). Monitoring multi-tier clustered systems with invariant metric relationships. In *Proceedings of the International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 73–80. ACM.

Nguyen, T. H., Adams, B., Jiang, Z. M., Hassan, A. E., Nasser, M., and Flora, P. (2012). Automated detection of performance regressions using statistical process control techniques. In *Proceedings of the joint WOSP/SIPEW International Conference on Performance Engineering (ICPE)*, pages 299–310. ACM.

Nistor, A., Jiang, T., and Tan, L. (2013). Discovering, reporting, and fixing performance bugs. In *Proceedings of the Tenth International Workshop on Mining Software Repositories (MSR)*, pages 237–246. IEEE Press.

Nitu (2009). Configurability in SaaS (software as a service) applications. In *Proceedings of the 2nd annual India Software Engineering Conference (ISEC)*, pages 19–26. ACM.

Pathirage, M., Perera, S., Kumara, I., and Weerawarana, S. (2011). A multi-tenant architecture for business process executions. In *International Conference on Web Services (ICWS)*, pages 121–128. IEEE.

Potts, C. (1993). Software-engineering research revisited. *IEEE Software*, 10(5):19–28.

Pouwelse, J. A., Garbacki, P., Wang, J., Bakker, A., Yang, J., Iosup, A., Epema, D. H., Reinders, M., Van Steen, M. R., and Sips, H. J. (2008). Tribler: a social-based peer-to-peer system. *Concurrency and Computation: Practice and Experience*, 20(2):127–138.

Prasad, V., Cohen, W., Eigler, F., Hunt, M., Keniston, J., and Chen, B. (2005). Locating system problems using dynamic instrumentation. In *Proceedings Ottawa Linux Symposium*, pages 49–64.

Pugh, B. and Spacco, J. (2004). RUBiS revisited: why J2EE benchmarking is hard. In *Companion to the 19th annual SIGPLAN Conference on OO-programming systems, languages, and applications (OOPSLA)*, pages 204–205. ACM.

Rao, J. and Xu, C.-Z. (2008). Online measurement of the capacity of multi-tier websites using hardware performance counters. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 705–712.

Reiss, S. P. (2009). Visualizing the Java heap to detect memory problems. In *International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 73–80. IEEE.

Rothermel, G. and Harrold, M. J. (1996). Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551.

Ryu, J. W., Kantardzic, M. M., and Kim, M.-W. (2012). Efficiently maintaining the performance of an ensemble classifier in streaming data. In *Convergence and Hybrid Information Technology*, pages 533–540. Springer.

Savari, S. A. and Young, C. (2000). Comparing and combining profiles. *Journal of Instruction-Level Parallelism*, 2.

Shankar, S. and Purusothaman, T. (2009). Utility sentient frequent itemset mining and association rule mining: A literature survey and comparative study. *International Journal of Soft Computing Applications*, 4:81–95.

Sneed, H. M. (2006). Integrating legacy software into a service oriented architecture. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, pages 3–14. IEEE C.S.

Swanson, E. B. (1976). The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering (ICSE)*, pages 492–497. IEEE Computer Society Press.

Syer, M., Adams, B., and Hassan, A. (2011). Identifying performance deviations in thread pools. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 83–92. IEEE.

Tan, Y., Gu, X., and Wang, H. (2010). Adaptive system anomaly prediction for large-scale hosting infrastructures. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC)*, pages 173–182. ACM.

Thereska, E., Doebel, B., Zheng, A. X., and Nobel, P. (2010). Practical performance models for complex, popular applications. In *Proceedings of the International Conference on Measurement and modeling of computer systems (SIGMETRICS)*, pages 1–12. ACM.

Tsai, C.-H., Ruan, Y., Sahu, S., Shaikh, A., and Shin, K. G. (2007). Virtualization-based techniques for enabling multi-tenant management tools. In *18th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM)*, volume 4785 of *LNCS*, pages 171–182. Springer.

Tsai, W., Li, Q., Colbourn, C., and Bai, X. (2013). Adaptive fault detection for testing tenant applications in multi-tenancy SaaS systems. In *Cloud Engineering (IC2E), 2013 IEEE International Conference on*, pages 183–192.

Tsai, W.-T., Shao, Q., Huang, Y., and Bai, X. (2010). Towards a scalable and robust multi-tenancy SaaS. In *Proceedings of the Second Asia-Pacific Symposium on Internetware*, Internetware '10, pages 8:1–8:15. ACM.

van Gurp, J., Bosch, J., and Svahnberg, M. (2001). On the notion of variability in software product lines. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 45–54.

Wang, Z. H., Guo, C. J., Gao, B., Sun, W., Zhang, Z., and An, W. H. (2008). A study and performance evaluation of the multi-tenant data tier design patterns for service oriented computing. In *Proceedings of the International Conference on e-Business Engineering (ICEBE)*, pages 94–101. IEEE C.S.

Warfield, B. (2007). Multitenancy can have a 16:1 cost advantage over single-tenant. `http://smoothspan.wordpress.com/2007/10/28/multitenancy-can-have-a-161-cost-advantage-over-single-tenant/` (last visited on May 20th, 2010).

Weissman, C. D. and Bobrowski, S. (2009). The design of the force.com multi-tenant internet application development platform. In *Proceedings of the 35th SIGMOD International Conference on Management of data (SIGMOD)*, pages 889–896. ACM.

Wilkinson, L. and Friendly, M. (2009). The history of the cluster heat map. *The American Statistician*, 63(2):179–184.

Witten, I. H. and Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition*. Morgan Kaufmann.

Woodside, M., Franks, G., and Petriu, D. (2007). The future of software performance engineering. In *Future of Softw. Engineering (FOSE)*, pages 171–187. IEEE.

Wu, J., Holt, R., and Hassan, A. (2004). Exploring software evolution using spectrographs. In *Proceedings of the Working Conference Reverse Engineering (WCRE)*, pages 80–89. IEEE.

Yan, D., Xu, G., and Rountev, A. (2012). Uncovering performance problems in Java applications with reference propagation profiling. In *Proceedings of the International Conference Software Engineering (ICSE)*, pages 134–144. IEEE CS.

Zeilemaker, N., Schoon, B., and Pouwelse, J. (2013). Dispersy bundle synchronization. Technical Report PDS-2013-002, TU Delft.

Zhang, Q., Cherkasova, L., Mathews, G., Greene, W., and Smirni, E. (2007). R-capriccio: a capacity planning and anomaly detection tool for enterprise services with live workloads. In *International Conference on Middleware*, pages 244–265. Springer.

Zhang, S., Cohen, I., Goldszmidt, M., Symons, J., and Fox, A. (2005). Ensembles of models for automated diagnosis of system performance problems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 644 – 653. IEEE.

Summary

Performance Optimization of Multi-Tenant Software Systems
– Cor-Paul Bezemer –

Multi-tenant software systems are Software-as-a-Service systems in which customers (or *tenants*) share the same resources. The key characteristics of multi-tenancy are hardware resource sharing, a high degree of configurability and a shared application and database instance. We can deduct from these characteristics that they lead to challenges compared to traditional software schemes. To better understand these challenges, we have come up with a reengineering pattern for transforming an existing single-tenant application into a multi-tenant one. We have done a case study in which we transform a single-tenant research prototype into a multi-tenant version. This case study showed that in a layered application, this transformation could be done in less than 100 lines of code.

With a better understanding of the challenges inflicted by multi-tenancy, we have focused on one of these challenges in this thesis, namely performance. Because tenants share resources in multi-tenant applications, it is necessary to optimize these applications on two levels: (1) at the hardware level and (2) at the software level.

Hardware-level Optimization

To optimize an application at the hardware level, we must be able to detect bottlenecks. We have proposed a method for detecting hardware bottlenecks which is based on the average response time per user per action. Hence, our method can deal with applications that have large and small customers, which may have different requirements. Our approach assists performance experts by detecting and analyzing *performance improvement opportunities* (PIOs), situations during which the performance could possibly be improved. Our approach uses supervised learning to generate association rules, which are used by a classifier to classify moni-

tored performance measurements into one of the load classes low, med or $high$. The $high$ class indicates the system is running relatively slow and hence, could be optimized. We have evaluated our approach in an industrial case study on Exact Online, a multi-tenant solution for online bookkeeping by Exact. In this case study, we analyzed and verified a subset of detected PIOs together with a performance expert from Exact. From the evaluation results, we found that the diagnosis given by our approach is accurate, but could benefit from extending the association rule set to improve its completeness. Therefore, we extended our approach with the possibility of using a set (*ensemble*) of classifiers, in which the classifiers work together to make a classification. We proposed an algorithm which aims at improving both accuracy and coverage of the ensemble when adding new classifiers. As a result, we can generate classifier ensembles which have improved diagnostic capabilities, as they are able to give a broader range of diagnoses.

In addition, we have proposed a visualization technique for our approach, which uses heat maps. In a field user study with performance experts from Exact, we showed that these visualizations assisted the experts in finding performance bottlenecks quicker and easier. The visualization techniques are implemented in an open source tool called WEDJAT.

Software-level Optimization

In addition to optimization at the hardware level, software-level optimization is important to ensure that the application runs as efficiently as possible. An important part of this process is verifying that software updates do not (unexpectedly) decrease the performance of an application. We have presented an approach for detecting and analyzing performance regressions. Our approach attaches to existing test suites and uses spectrum-based analysis to analyze whether the performance behaviour of these test suites has changed since the previous version of the application. In a case study on a peer-to-peer BitTorrent client, called Tribler, we show that our approach is capable of guiding the optimization process. On the one hand, bottlenecks can be found, and on the other hand, performance fixes can be validated using our approach. Feedback from two developers who have years of experience with Tribler showed that our approach is accurate and useful, as the case study resulted in two optimizations to Tribler. Our approach is implemented in the open source tool SPECTRAPERF.

Samenvatting

Performance Optimization of Multi-Tenant Software Systems
– Cor-Paul Bezemer –

Multi-tenant software systemen zijn Software-as-a-Service systemen waarin bronnen gedeeld worden door klanten (of *tenants*). De belangrijkste eigenschappen van multi-tenancy zijn het delen van hardware bronnen, een hoge graad van configureerbaarheid en het delen van één applicatie en database instantie. Vergeleken met traditionele software, leiden deze eigenschappen tot een aantal uitdagingen. Om deze uitdagingen beter te begrijpen, hebben we een sjabloon bedacht om een bestaande single-tenant applicatie om te bouwen naar een multi-tenant applicatie. We hebben een case study gedaan waarin we een single-tenant onderzoeksprototype ombouwen naar een applicatie met ondersteuning voor multi-tenancy. Uit deze case study bleek dat deze transformatie in dit gelaagde prototype in minder dan 100 regels code gedaan kon worden.

Met een beter begrip van de uitdagingen die bij multi-tenancy komen kijken, hebben we ons onderzoek in deze thesis op één van deze uitdagingen gericht, namelijk op performance. Omdat klanten bronnen delen binnen een multi-tenant applicatie, is het noodzakelijk om deze applicaties op twee niveaus te optimaliseren: (1) op het hardware niveau en (2) op het software niveau.

Optimalisatie op het Hardware Niveau

Om een applicatie op het hardware niveau te optimaliseren, moeten we knelpunten kunnen herkennen. We hebben een methode voorgesteld om knelpunten te vinden in hardware. Deze methode is gebaseerd op de gemiddelde responstijd per gebruiker per actie. Hierdoor kan onze methode omgaan met applicaties die zowel grote als kleine klanten heeft, aangezien deze verschillende eisen kunnen stellen aan de performance. Onze methode assisteert performance experts met het vinden en analyseren van zogenaamde *performance improvement opportunities*

(PIOs), oftewel situaties waarin de performance mogelijk verbeterd zou kunnen worden. Onze aanpak gebruikt begeleid leren om associatie regels te genereren, die dan weer gebruikt worden door een classifier om gemeten data te classificeren in een van de drie klassen *low*, *med* of *high*. De *high* klasse geeft aan dat het system relatief traag draait en dus waarschijnlijk kan worden geoptimaliseerd. We hebben onze aanpak geevalueerd in een industriële case study op Exact Online, een multi-tenant oplossing voor online boekhouden van Exact. In deze case study, hebben we een subset van gevonden PIOs geanalyseerd en gecontroleerd met een performance expert van Exact. Uit de resultaten van deze evaluatie bleek dat de diagnose gegeven door onze aanpak accuraat is, maar verbeterd kan worden door de gebruikte set van associatie regels uit te breiden. Daarom hebben we onze aanpak uitgebreid met de mogelijkheid een set (*ensemble*) van classifiers te gebruiken, waarin de classifiers samen werken om een classificatie te maken. We hebben een algoritme voorgesteld dat zich richt op zowel het verbeteren van de precisie als op de compleetheid van de diagnose tijdens het onderhouden van het ensemble. Dit heeft als resultaat dat we classifier ensembles kunnen genereren die verbeterde mogelijkheden voor het stellen van een diagnose hebben, aangezien ze een bredere variatie aan diagnoses kunnen stellen.

Daarnaast hebben we een visualisatietechniek gepresenteerd voor onze aanpak, die gebruik maakt van zogenaamde heat maps. In een gebruikersstudie met performance experts van Exact, hebben we laten zien dat deze visualisaties de experts hielpen met het sneller en eenvoudiger vinden van knelpunten. De visualisatietechnieken zijn geïmplementeerd in een open source applicatie genaamd WEDJAT.

Optimalisatie op het Software Niveau

Naast optimalisatie op het hardware niveau, is optimalisatie op het software niveau noodzakelijk om er zeker van te zijn dat de applicatie zo efficient mogelijk draait. Een belangrijk deel van dit proces is het controleren dat software updates niet (onverwachts) de performance van een applicatie verslechteren. We hebben een aanpak voor het detecteren en analyseren van performance regressies gepresenteerd. Onze aanpak kan aan bestaande test suites gekoppeld worden en gebruikt spectrum-gebaseerde analyse om te analyseren of de performance van deze test suites veranderd is sinds de vorige versie van de software. In een case study op een peer-to-peer BitTorrent client, Tribler, laten we zien dat onze aanpak het optimalisatie proces kan leiden. Aan de ene kant kunnen er knelpunten gevonden worden met onze aanpak, en aan de andere kant kunnen oplossingen voor deze knelpunten gecontroleerd worden met onze aanpak. Uit commentaar van twee ontwikkelaars uit het Tribler team kunnen we afleiden dat onze aanpak accuraat en bruikbaar is, aangezien de case study in twee optimalisaties in Tribler resulteerde. Onze aanpak is geïmplementeerd in de open source applicatie SPECTRAPERF.

Curriculum Vitae

Education

2009 – 2013: Ph.D., Computer Science
Delft University of Technology, Delft, The Netherlands. Under the supervision of dr. A.E. Zaidman.

2007 – 2009: M.Sc., Computer Science
Delft University of Technology, Delft, The Netherlands.
Master's thesis title: *Automated Security Testing of AJAX Web Widget Interactions* (in collaboration with Exact)

2002 – 2007: B.Sc., Computer Science
Delft University of Technology, Delft, The Netherlands.

Work Experience

October 2013 – present: Postdoctoral Researcher
Parallel and Distributed Systems Department, Delft University of Technology. Mekelweg 4, 2628CD Delft, The Netherlands.

1998 – present: Software developer
Collect4all / Howell Holding BV. Zuiderparklaan 133, 2574 HD The Hague, The Netherlands.

July 2009 – October 2013: Assistant in Opleiding (AIO). *Research Trainee*
Software Technology Department, Delft University of Technology. Mekelweg 4, 2628 CD Delft, The Netherlands.

2007 – 2009: Web developer
Maxcode. Parkstraat 83, 2514 JG, The Hague

Titles in the IPA Dissertation Series since 2008

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code*

Generation with Templates. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes*. Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains*. Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations*. Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice*. Faculty of Science, Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars*. Faculty of Science, UvA. 2011-21

M. Izadi. *Model Checking of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-22

L.C.L. Kats. *Building Blocks for Language Workbenches*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

S. Kemper. *Modelling and Analysis of Real-Time Coordination Patterns*. Faculty of Mathematics and Natural Sciences, UL. 2011-24

J. Wang. *Spiking Neural P Systems*. Faculty of Mathematics and Natural Sciences, UL. 2011-25

A. Khosravi. *Optimal Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2012-01

A. Middelkoop. *Inference of Program Properties with Attribute Grammars, Revisited*. Faculty of Science, UU. 2012-02

Z. Hemel. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03

T. Dimkov. *Alignment of Organizational Security Policies: Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

S. Sedghi. *Towards Provably Secure Efficiently Searchable Encryption*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

F. Heidarian Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference*. Faculty of Science, Mathematics and Computer Science, RU. 2012-06

K. Verbeek. *Algorithms for Cartographic Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2012-07

D.E. Nadales Agut. *A Compositional Interchange Format for Hybrid Systems: Design and Implementation*. Faculty of Mechanical Engineering, TU/e. 2012-08

H. Rahmani. *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2012-09

S.D. Vermolen. *Software Language Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10

L.J.P. Engelen. *From Napkin Sketches to Reliable Software*. Faculty of Mathematics and Computer Science, TU/e. 2012-11

F.P.M. Stappers. *Bridging Formal Models – An Engineering Perspective*. Faculty of Mathematics and Computer Science, TU/e. 2012-12

W. Heijstek. *Software Architecture Design in Global and Model-Centric Software Development*. Faculty of

Mathematics and Natural Sciences, UL. 2012-13

C. Kop. *Higher Order Termination*. Faculty of Sciences, Department of Computer Science, VUA. 2012-14

A. Osaiweran. *Formal Development of Control Software in the Medical Systems Domain*. Faculty of Mathematics and Computer Science, TU/e. 2012-15

W. Kuijper. *Compositional Synthesis of Safety Controllers*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16

H. Beohar. *Refinement of Communication and States in Models of Embedded Systems*. Faculty of Mathematics and Computer Science, TU/e. 2013-01

G. Igna. *Performance Analysis of Real-Time Task Systems using Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2013-02

E. Zambon. *Abstract Graph Transformation – Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2013-04

G.T. de Koning Gans. *Outsmarting Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2013-05

M.S. Greiler. *Test Suite Comprehension for Modular and Dynamic Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

L.E. Mamane. *Interactive mathematical documents: creation and presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2013-07

M.M.H.P. van den Heuvel. *Composition and synchronization of real-time components upon one processor*. Faculty of Mathematics and Computer Science, TU/e. 2013-08

J. Businge. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins*. Faculty of Mathematics and Computer Science, TU/e. 2013-09

S. van der Burg. *A Reference Architecture for Distributed Software Deployment*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-10

J.J.A. Keiren. *Advanced Reduction Techniques for Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2013-11

D.H.P. Gerrits. *Pushing and Pulling: Computing push plans for disk-shaped robots, and dynamic labelings for moving points*. Faculty of Mathematics and Computer Science, TU/e. 2013-12

M. Timmer. *Efficient Modelling, Generation and Analysis of Markov Automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-13

M.J.M. Roeloffzen. *Kinetic Data Structures in the Black-Box Model*. Faculty of Mathematics and Computer Science, TU/e. 2013-14

L. Lensink. *Applying Formal Methods in Software Development*. Faculty of Science, Mathematics and Computer Science, RU. 2013-15

C. Tankink. *Documentation and Formal Mathematics — Web Technology meets Proof Assistants*. Faculty of Science, Mathematics and Computer Science, RU. 2013-16

C. de Gouw. *Combining Monitoring with Run-time Assertion Checking*. Faculty of Mathematics and Natural Sciences, UL. 2013-17

J. van den Bos. *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics*. Faculty of Science, UvA. 2014-01

D. Hadziosmanovic. *The Process Matters: Cyber Security in Industrial Control Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-02

A.J.P. Jeckmans. *Cryptographically-Enhanced Privacy for Recommender Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-03

C.-P. Bezemer. *Performance Optimization of Multi-Tenant Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2014-04