Delft University of Technology
Software Engineering Research Group
Technical Report Series

Server Overload Detection and Prediction Using Pattern Classification

Cor-Paul Bezemer, Andy Zaidman

Report TUD-SERG-2011-008

TUD-SERG-2011-008

Published, produced and distributed by:

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

ISSN 1872-5392

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

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

Note: Paper accepted as a poster for the 8th International Conference on Autonomic Computing (ICAC 2011)

Server Overload Detection and Prediction Using Pattern Classification

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

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

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Performance

1. INTRODUCTION

One of the key factors in customer satisfaction is the application performance. In traditional settings, it is usually not very difficult to manually detect a performance problem, however, with the advent of ultra-large-scale (ULS) systems [4], manual performance monitoring and prediction becomes tedious and would thus ideally require automation. A typical situation in such a system is depicted by Figure 1, in which a server overload occurs when approximately 500 requests are handled per second by the system. In order to prevent the overloaded state, we should be able to predict this state when the system is handling approximately 400 requests per second, so that it can be scaled up. Automating this prediction is typically hard, because many factors influence performance, and it is typically the human mind that excels at making the right (subjective) decisions based on multiple factors. It is our aim to automate performance prediction, for which we have two distinct goals in mind: (1) warn the system administrator for the need of an impending hardware upscaling and (2) provide an automatic overload prevention mechanism.

An application in which such an automated prediction mechanism is very useful is in *self-adaptive systems*, which are capable of adapting their own behavior according to changes in the environment and the system itself [5]. Having such a mechanism will improve the quality of service as it helps these systems decide when to scale up.

In this paper, we propose an approach for server overload prediction. An important aspect of our overload prediction mechanism is the performance monitoring method. Our performance monitoring is based on measuring a wide variety of so-called *performance counters* [1], such as the `Memory\Available Mbytes` and `Processor\%Processor Time` counters. Rather than defining exact threshold values for the monitored performance counters, we propose to use pattern classification, which can assist with recognizing complex performance counter patterns.

ICAC'11, June 14–18, 2011, Karlsruhe, Germany.
ACM 978-1-4503-0607-2/11/06.

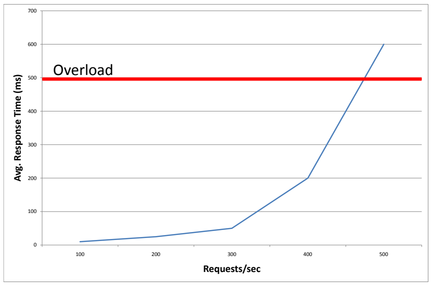

Figure 1: Average response time in an ULS

Problem statement. In the context of ULSs, the application is typically deployed over a number of servers, in order to cope with the performance demand. An interesting problem is then, when the underlying hardware should be scaled up to keep up with increasing demand. Intuitively, we want to scale up the application before the user notices a decrease in performance. Furthermore, because scaling an application takes time, due to factors like hardware relocation, we must be able to predict an overload with a margin in the order of several days to weeks ahead.

Our research will focus on predicting such overload using performance metrics. As customer satisfaction is strongly related to the average response time of an application, we will use this metric as the most important indication of overload. This raises three important challenges: firstly, establishing which response times are deemed acceptable by the users of an application. Secondly, the selection of the right set of performance metrics, that can serve as indicators for overload situations. Thirdly, the definition of thresholds for these metrics, which we consider as non-trivial because overload is likely to be indicated by complex patterns of performance metric values, e.g., a memory overload will exhibit different performance metric values than a CPU overload.

2. APPROACH AND CHALLENGES

To be able to predict overload and to evaluate our approach, it is important that we can detect this state. In this section we will first explain the challenges of overload detection and after this we will propose our approach for predicting overload.

Challenge 1: Defining a response time threshold. The *response time threshold* describes how long the customers are willing to wait for the application to respond. This threshold will be defined by conducting an experiment in which we will introduce delays in an application and ask users how they valued the perceived performance. It is necessary to have multiple thresholds as, e.g., users are willing to wait longer for the generation of a report than for a GUI action.

Challenge 2: Selecting the metrics. To monitor the state of the server we will monitor a set of *performance counters* using PerfMon on our Windows-based servers. Initially we will monitor a large set of performance counters, but as we collect more data, we will use statistics to select the most significant counters to monitor.

Challenge 3: Defining performance counter thresholds. Defining *performance counter thresholds* is more difficult because these are hardware-specific and can interrelate in a complex way. Therefore, we need a more intelligent way of describing performance counter thresholds rather than just using extreme values. In the next paragraph we describe how we will use pattern classification for this.

Overload Detection Using Classification. Classification is an example of pattern recognition, in which a set of input values are assigned to a given set of classes, for example, determine whether a server is or is not in overloaded state. To design a classifier, a training set containing patterns and their known output classes is required. After designing the classifier, it can be used to classify unknown patterns. Our training data will contain triples of the form (*filename*, *performance counter pattern*, *response time*). A performance counter pattern consists of the concatenated values of the monitored performance counters. Note that the response time is used to decide whether the pattern represents and overloaded state.

Generating the Training Data. To correctly train our classifier, we need patterns for normal and overloaded states. An observation is that it may be difficult to get data for overloaded states as we are trying to avoid such states in production environments. Therefore, we have to reproduce the data in a test or isolated environment.

To get this data for a server, which is not in production yet, we will use JMeter[1] to perform a stress test. During this stress test, the load will vary from normal to overload. By monitoring the response time during the stress test, patterns can be gathered for normal and overloaded states.

Because performance counter thresholds are hardware-specific, different servers may have different performance counter patterns in an overloaded state. To avoid running a stress test on a production server, we propose the use of an Application Experience Index, or AppExI. This catalog contains classifiers designed for servers in the past for a certain application. By selecting the classifier for the server in the AppExI which resembles the internal hardware of the production server the closest, we can get an initial version of the classifier of that production server.

Designing the classifier. As we do not know the distribution of our training data yet, we will run various classifier generation algorithms (using the Matlab toolbox PRTools[2]) and select the best working classifier afterwards.

Validation. To validate our classifiers, we will use cross-validation. In addition, we will randomly send a short questionnaire to application users to ask how they valued the performance of the application during that session. If the performance was considered bad, we can use the logged performance patterns to redesign and optimize the classifier.

2.1 Overload prediction

Rather than detecting overload, we want to be able to predict it. To do this, we will automate an approach that is intuitively used by system administrators when monitoring a system. When administrators inspect the performance of a system, they search log files for upward (or downward) trends in performance counter values. A clear example of this is free harddrive space. The longer the `Logical Disk\%Free Space` performance counter exhibits a downward trend, the more likely it is that a performance problem is about to occur, in this case a full harddrive. As we have explained for monitoring, detecting simple trends is relatively easy but detecting complex patterns is difficult.

Therefore, we will use pattern classification for overload prediction as well. Our goal is to train our classifier in such a way that it is able to detect series of patterns which lead to server overload, in other words, consecutive performance states which eventually evolve to a known threshold performance counter pattern.

3. RELATED WORK

Existing research on using pattern classification for overload detection and prediction is mostly focused on preventive mechanisms, such as admission control and load balancing [2, 3]. The main concern we have with using only a prevention mechanism, is that it does not advise us on when to scale a system up. In addition, admission control negatively affects the performance of some customers, which our approach tries to avoid.

4. FUTURE WORK

Future work will consist of the implementation and evaluation of our approach, including an evaluation on an industrial application.

5. REFERENCES

[1] R. Berrendorf and H. Ziegler. 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, 1998.

[2] M. Dantas and A. Pinto. A load balancing approach based on a genetic machine learning algorithm. In *Proc. Int. Symp. on High Performance Computing Systems and Applications (HPCS)*, pages 124 – 130. IEEE, 2005.

[3] R. Fontaine, P. Laurencot, and A. Aussem. Mixed neural and feedback controller for apache web server. *ICGST Int. Journal on Computer Network and Internet Research, CNIR*, 09(1):25–30, 2009.

[4] L. Northrop, P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, and K. Wallnau. *Ultra-Large-Scale Systems - The Software Challenge of the Future.* S.E. Institute, Carnegie Mellon, 2006.

[5] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):14:1–14:42, 2009.

[1] `http://jakarta.apache.org/jmeter/`

[2] `http://www.prtools.org/`

TUD-SERG-2011-008
ISSN 1872-5392