Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 61 (2015) 198 – 205

Procedia Computer Science

Complex Adaptive Systems, Publication 5
Cihan H. Dagli, Editor in Chief
Conference Organized by Missouri University of Science and Technology
2015-San Jose, CA

Elastic Scaling of Cloud Application Performance Based on Western Electric Rules by Injection of Aspect-Oriented Code

Konstantin Benz[a]*, Thomas M. Bohnert[b]

[a]*Zürich University of Applied Sciences, Technikumstrasse 9, 8400 Winterthur, Switzerland*
[b]*Zürich University of Applied Sciences, Technikumstrasse 9, 8400 Winterthur, Switzerland*

Abstract

The main benefit of cloud computing lies in the elasticity of virtual resources that are provided to end users. Cloud users do not have to pay fixed hardware costs and are charged for consumption of computing resources only. While this might be an improvement for software developers who use the cloud, application users and consumers might rather be interested in paying for application performance than resource consumption. However there is little effort on providing elasticity based on performance goals instead of resource consumption. In this paper an autoscaling method is presented which aims at providing increased application performance as it is demanded by consumers. Elastic scaling is based on "statistical process monitoring and control" and "Western Electric rules". By demonstrating the architecture of the autoscaling method and providing performance measurements gathered in an OpenStack cloud environment, we show how the injection of aspect-oriented code into cloud applications allows for improving application performance by automatically adapting the underlying virtual environment to pre-defined performance goals. The effectiveness of the autoscaling method is verified by an experiment with a test program which shows that the program executes in only half of the time which is required if no autoscaling capabilities are provided.

Peer-review under responsibility of scientific committee of Missouri University of Science and Technology

Keywords: cloud computing; autoscaling; virtualization; elasticity; scalability; availability; reliability; performance; Western Electric rules; Statistical Process Control; aspect-oriented code

* Corresponding author. Tel.: +41-058-934-7101; fax: +41-058-935-6960.
E-mail address: benn@zhaw.ch

Peer-review under responsibility of scientific committee of Missouri University of Science and Technology
doi:10.1016/j.procs.2015.09.193

1. Motivation

The advance of the cloud computing paradigm in the computer industry has changed the way consumers think about IT services. Computers should be made available on-demand whenever requested, they should perform well even under peak traffic load and still be turned off to keep the costs low. The characteristics of cloud computing services are comparable to utility services like e. g. gas, water or electricity.[1] Like in the utility services industry cloud computing users have high expectations in terms of availability and performance of the services they consume. "High availability" is an important topic in the cloud computing business, since consumers expect that cloud platforms should deliver virtualized environments that are available 99.999% of their total operating time. From this point of view it is evident that availability and performance of cloud computing and its applications should be studied with more care.

"Statistical process monitoring and control" is a method that aims to detect and eliminate variability in production processes.[9] Although SPC has its origins in the manufacturing industry, it can be applied in many other areas too. Its main instrument is the "control chart"[2] which captures means of sample measurements that are taken repetitively from the monitored process. By employing the SPC method, one can react to process changes. The "Western Electric" rules are a set of rules that define when one has to change something in the underlying process.[13]

We employ SPC and WER as a means for automatically rescaling virtual environments that are hosted in a cloud computing environment. Thereby we want to increase availability and performance of applications hosted in virtual machines that run in the cloud.

Nomenclature

AOP	Aspect-oriented programming
CIM	Computation independent model
IT	Information technology
LCL	Lower control limit
MDA	Model-driven architecture
PIM	Platform independent model
PSM	Platform specific model
SPC	Statistical process monitoring and control
UCL	Upper control limit
VM	Virtual machine
WER	Western Electric rules

1.1. Scope

The goal of our study is to show that we can significantly increase performance of a Python software that is hosted in an OpenStack cloud by enabling autoscaling capabilities on the underlying cloud platform. Thereby we run a Python test program in an OpenStack environment that does not have autoscaling capabilities enabled and compare its execution time versus the same Python program when it is running in an OpenStack environment with autoscaling capabilities. A statistical test should reveal if autoscaling is able to significantly increase application performance.

1.2. Structure of the paper

In section 2 of this paper we explain the theoretical background of the Western Electric rules on which the autoscaling mechanism is based. Section 3 shows the architecture of the autoscaling mechanism. In section 4 we describe the design of the experiment we make to verify the effectiveness of the autoscaling function. The results of the experiment are discussed in section 5 and some conclusions and inferences of it are made in section 6.

2. Theoretical background

An important part of the autoscaling function is the way how its actions are triggered. One could think of a certain event that acts as a signal that the infrastructure that hosts the application should be replaced by a better performing one. In an OpenStack environment this could mean that the virtual machine which is hosting the application is replaced by a faster VM with more memory and storage capacity. The occurrence of a signaling event should lead to a migration of the running application to a new environment that performs better. The questions that can be brought up are:

- How such signaling events should be defined.
- How trigger signaling can be captured timely and effectively.

The former question is answered in this section, while the latter question includes some architectural considerations which are answered in section 3.

2.1. Statistical process control, control limits and processes

The first question is answered by defining the characteristics of a "signaling event". Autoscaling has the goal to increase performance by detecting and removing issues that occur during runtime of the program. Therefore a "signaling event" must be defined as an event that leads to a "significant" performance decrease.[10] The method that is applied to detect such events is "statistical process monitoring and control" which consists in continual sampling of process results, measuring sample means and detecting statistical deviations between sample means.[10]

In our autoscaling software we run a set of program loops and measure their execution time. This is the "process" we monitor. Thereby we capture mean and the range of the sample measurements. The sample ranges are put into an "R-chart" and the sample means are put into an "$\bar{x}$-chart".[9] Each sample range is one data point in the R-chart and each mean is one data point in the $\bar{x}$-chart. The data points in such "control charts" are used to monitor the program execution.

The rationale behind monitoring processes with control charts is that variability in data points comes from random variability of the underlying process, so changes in the data points reflect changes in the process.[9] The random variability follows a normal distribution with mean μ and standard deviation σ.[9] "Significant" drifts of the data points from the underlying distribution are interpreted as a sign that the monitored process has gone out of control.[9] By the term "significant" we mean that the data points have only a small probability α that they are random derivations of the given normal distribution.[9] Whenever a measurement has a probability lower than the significance level α to be derived from the normal distribution, a process drift is detected.[9] In our case this would mean that an event occurred which should lead to an autoscaling action.

The parameters μ and σ of the underlying normal distribution are usually unknown and must be estimated in advance.[9] This is done in a "phase I" where parameters of the statistical distribution are estimated.[9] Thereby some "control limits" are defined which determine the out-of-control situation.[9] Since process deviations can go in both directions, there is an "upper control limit" and a "lower control limit".[9] The formulae that are used for determining control limits of the R-chart are:

$$UCL = D_3 \cdot \bar{R}; LCL = D_4 \cdot \bar{R} \tag{1}$$

Whereby $\bar{R}$ is the mean of all sample ranges R. D_3 and D_4 are constants that depend on the sample size n. Ranges are checked in order to estimate the standard deviation. The control limits of the $\bar{x}$-chart are:

$$UCL = \bar{x} + A_2 \cdot \bar{R}; LCL = \bar{x} - A_2 \cdot \bar{R} \tag{2}$$

Whereby $\bar{x}$ is the mean of all sample means x, $\bar{R}$ is the mean of all sample ranges R and A_2 is a constant that is only dependent on the size n of the samples. $\bar{x}$ is depicted by a "central line" CL in the control chart. The term $A_2 \cdot \bar{R}$ is an estimation of the parameter σ.[9] Since the difference between the UCL and LCL equals six times the estimated value of the standard deviation σ, SPC techniques are often referenced as "Six Sigma" methods.[3] If a sample measurement is outside of the control limits, a correction of the underlying process is indicated.

2.2. Western Electric rules

The "Western Electric rules" go a step further than just detecting out-of-control situations: they try to anticipate significant process deviations by spotting trends in the control chart. [13]

WER consists in rules that describe out-of-control situations. These are:

- One data point in the control chart lies above the UCL threshold or below the LCL.
- Two out of three **consecutive** data points lie above the threshold $UCL_{2/3}$ or below the threshold $LCL_{2/3}$ which are defined by:

$$UCL_{2/3} = \bar{x} + \frac{2}{3} \cdot A_2 \cdot \bar{R}; LCL_{2/3} = \bar{x} - \frac{2}{3} \cdot A_2 \cdot \bar{R} \quad (3)$$

- 4 out of 5 **consecutive** data points lie outside the area between $LCL_{1/3}$ or below the threshold $UCL_{1/3}$ which are defined as:

$$UCL_{1/3} = \bar{x} + \frac{1}{3} \cdot A_2 \cdot \bar{R}; LCL_{1/3} = \bar{x} - \frac{1}{3} \cdot A_2 \cdot \bar{R} \quad (4)$$

- 7 out of 8 **consecutive** points lie either below or above $\bar{x}$.

The important part of these rules is that consecutive data points are investigated. The reason for investigating consecutive data points is that a trend pattern should be discovered which makes SPC more efficient.[14]

3. Architecture of the autoscaling mechanism

The question how to implement a mechanism to capture significant deviations from the normal distribution, is not so trivial since the autoscaling software must be capable to measure its own performance during runtime and react to measurements. In our program code we must integrate a reactive part that builds and monitors the control charts and reacts to signaling events in the control charts. Our approach is to use the "aspect-oriented programming" paradigm.

3.1. Autoscaling process

Given the SPC method and the Western Electric rules, the runtime logic of the autoscaling software can be defined quite straightforward. It consists in four steps.

- The Python program is executed on a VM that is hosted in the OpenStack environment. The program consists in multiple steps. During each step, it measures the execution time of the step and stores it in a log file.
- After n steps the range of the last n execution times is calculated and stored as a data point in an R-chart. The same calculation is performed for the mean of the last n steps. The mean is then stored as data point in the $\bar{x}$-chart. The sample which is used in the control charts is the set of execution times of the last n program steps.
- Every time the two new data points are put into the control charts, the control chart monitoring component of the program checks if there is a process shift according to Western Electric rules.
- If a performance decline has been detected, the program must move to a more powerful VM. If the performance is unusually good, the program is migrated to a slower virtual machine. Migration operations are only performed if the target VM is available.

The definition of the autoscaling process makes it clear that the Python program should execute a task which is dividable into multiple steps (otherwise sampling would not be possible). Furthermore it should receive some input which can be dynamically directed to another program. This is a requirement, because otherwise there would be no way to migrate the program to another VM.

3.2. Aspect-oriented programming

AOP is a paradigm that aims at separating "cross-cutting concerns" in a software program from other components of the software.[7] A "cross-cutting concern" is said to be "cross-cutting", because it is composed in several different ways and appears at different locations in the code of a program, but still needs to be coordinated.[7] A cross-cutting concern is encapsulated in an "aspect" which is a program part which are not well-localized and composed in a procedure.[7] AOP coordinates aspects in a program by defining "join points" where the program logic of the aspect is encapsulated.[7] In the case of autoscaling AOP is useful, because things like performance measurements have to be taken at several locations in the program code.

The implementation of join points in a Python program is quite effective by adding decorators to certain Python functions. These decorators can wrap classes or methods that are called at the time when the decorated class or method is called. Thereby they implement the aspect oriented programming logic of join points that direct program logic to interwoven code that handles cross-cutting concerns. Therefore some units in the program code are implemented as decorator functions and classes.

3.3. Architecture models

The architecture of the autoscaling function is designed by applying the "model-driven architecture" approach. MDA is favoured to other modelling methods, because its models are linked to each other.[8] In MDA the architect first models the system from an end user perspective and designs a "computation independent model" that describes how the tool should work.[8] In the next step a "platform independent model" is derived from the CIM by defining components that should implement the process steps described in the CIM. [8] The last step consists in creating a "platform specific model" which links the components of the PIM to concrete technologies that implement them. [8]

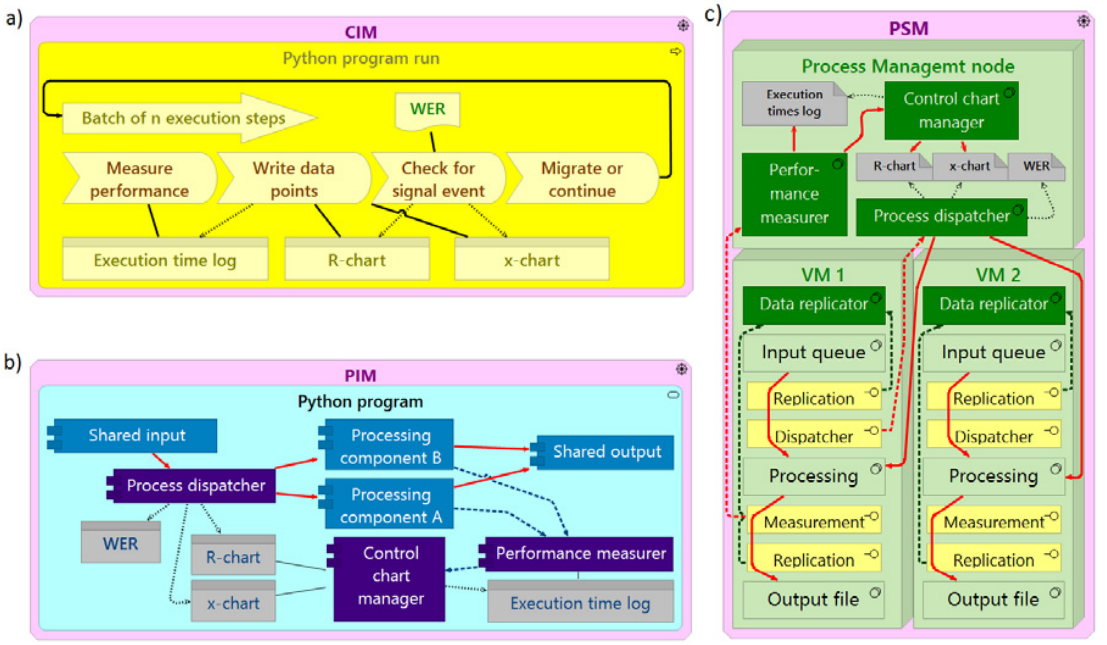

Fig. 1. (a) Computation independent model; (b) Platform independent model; (c) Platform specific model;

The CIM (Fig. 1a) is nothing else than the autoscaling process which is already described in section 2.4. A batch of n execution steps is performed, then performance is measured and data points are filled to the control charts. Then it is checked if the program should be migrated to another VM or continue with the processing in the same environment. The PIM (Fig. 1b) is more specific about this migration check: a process dispatcher component guides the input request to one of several competing processing components which create a shared output. The process dispatcher is guided by the Western Electric rules and the data points in the control charts. The control charts are generated by a control chart manager which gets data from an execution time log. This log file is fed by a performance measurer component which gets its data from the competing processing components. The processing components are ideal candidates where a join point should be placed. The measurement cross-cutting concern has to be woven into the program code of the processing components. The weavings are then modelled in the PSM (Fig. 1c) as dashed lines. There are yellow join points between the shared input queue and the processing unit as well as between the processing component and the output file.

The first series of join points direct the program flow to the process dispatcher and replicate input data to a second VM which allows for fast dispatching of the program flow to another VM. Thereby a data replicator unit has to be present on both VMs to ensure that input and output of the program are the same on both VMs. The process dispatcher is hosted on a management node where it reads data from the R- and $\bar{x}$-charts. It interferes with the program in case if the WER set is applicable.

The second series of join points capture process execution time and replicate the output data. The execution time is passed to a measurer unit on the process management node. Thereby the execution times of the last n process steps are consolidated and passed to the control chart manager which writes data points to the control charts. The control charts in turn indirectly control behavior of the process dispatcher, so the dispatching (and therefore the scaling of the application) is guided by control chart data – as it is required in the computation independent model (Fig. 1a).

4. Experimental validation of the effectiveness of the tool

The experiment to validate the effectiveness of the tool is a test with two samples: the first sample is the execution times of all program steps when no autoscaling mechanism is applied. The second sample is the same set of execution times when autoscaling is applied. Thereby we run the Python program in the first scenario and then in the second scenario. The program measures execution time of each step and writes it to a log file. After several steps the program halts. Then we collect all execution times and compute their mean and variance. We compare both sample means and check with statistical tests if the difference in means is valid and not just a random result. A two-sample “t-test” may reveal if the mean execution time of the first sample differs from the mean execution time of the second sample.[4] In case if both samples have unequal variance, the t-test is not applicable.[6] Then a non-parametric “Wilcoxon-rank-test”[6] can be applied as an alternative.

4.1. Design of experiment

The experiment to validate effectiveness of the autoscaling algorithm is a two sample test: first we run the Python program several times without autoscaling mechanism, then we run it with autoscaling enabled. For each run we measure total program execution time. Thereby we create two samples which are then compared to each other. For each sample the Python program is executed 30 times, so the sample size m is 30. The sample size is kept small, so the t-test is applicable.[4] Another important reason for the relatively small sample size is that test runs have to take long time in order to make the autoscaling mechanism employable. Autoscaling is only used if the Python program gets slow, which requires a time consuming test program.

The test program is a calculation program that receives some input and computes some numbers as output. Each input is a random number r between 9’000 and 100’000. The program calculates the r-th Fibonacci number. This program is chosen, because efficient calculation of Fibonacci numbers involves caching of intermediate results[12] which can lead the VM to use memory and system bus capacities. Input numbers are put on a shared queue to allow to migrate program inputs to another VM. In order to make the program slow down a machine, we chose a high

number of inputs which have to be calculated. 1'000 inputs are produced per test run, which results in 9 - 100 Mio. arithmetic operations per program run, because each input requires 9'000-100'000 computation steps.

Autoscaling involves evaluating of program step executions times. This evaluation is performed every time a number has been processed. After 10 numbers have been calculated, the program checks if it should migrate to another VM. The number of steps required for an autoscaling action is 10, because this number is small enough to detect performance changes quickly, but large enough to allow for a complete migration of input buffers to another VM.

The time is measured at the beginning of the calculation of all 1'000 numbers and after all numbers have been computed. The difference is the total program execution time. The total program execution time is measured 30 times with enabled autoscaling function and 30 times without autoscaling mechanism. Then both measurements are compared to each other.

5. Discussion of experiment results

Table 1 shows mean execution time of the program with and without the autoscaling mechanism. The mean program execution time can be reduced to 49.48 s from 70.60 s which means a reduction of 29.9 %. The results have been checked with a t-test for validity. The t-value of 25.225 is clearly outside of the 95 % confidence interval between 19.410 and 22.835. Therefore we reject the null hypothesis that the true difference in means of both samples equals 0. Thereby we get a false positive with a probability equal to the p-value (which is lower than $2.2 \cdot 10^{-16}$).

Table 1. Results of t-test for both programs.

	without autoscaling	with autoscaling
Sample size n	30	30
Mean execution time (sec)	70.60 s	49.48 s
t-value (2-sample test)	-	25.225
p-value	-	$< 2.2 \cdot 10^{-16}$
95% confidence interval (of t-value)	-	19.410 22.835

Although the effectiveness of the autoscaling mechanism is validated by this result, it is not so clear why autoscaling results in faster program execution: execution times have to be measured continually, the process dispatcher must be contacted every 10 program execution steps and the program context is often shifted. This involves a lot of network traffic and data transfer to coordinate the program flow. We assume that this coordination effort takes less time than the savings in computation time when the Fibonacci numbers are computed on the faster machine. The autoscaling mechanism compensates the additional network communication with an increase in computation speed by switching to another VM. If large network lags are involved, it is likely that this speed advantage might disappear.

6. Conclusion

Providing elasticity to computer programs is a main benefit of cloud computing, especially because cloud computing can be seen (from the consumer perspective) as utility service which should offer high availability and performance. Although many cloud providers offer elasticity which is based on resource consumption, end users may want their cloud environment to provide better performance. In this paper we showed an elastic scaling mechanism which is capable to scale cloud applications to their actual performance by employing statistical process control methods and the Western Electric rules.

We explained how SPC and the Western Electric rules can help to detect performance flaws in an application by observing variation in program execution times and deducing trends if execution times tend to fall into critical areas.

Our autoscaling mechanism applies the SPC to a Python program that runs on a virtual machine in the cloud. By injection of continual program execution time measurements in the program code control charts are created. The control charts are then used by a process dispatcher to direct the program flow from a slow performing VM to a better performing one if performance decreases are detected. The architecture of the autoscaling mechanism requires a "process management node" which applies the WER-based process dispatching to the VMs that execute the Python program.

We made an experiment to show that the Python program executes faster when the autoscaling function is applied to it. Thereby we executed a performance-demanding Python program 30 times on the slow VM and 30 times on the slow and fast VMs which use the process dispatcher for autoscaling. We measured the program execution times in both situations and found that mean execution time can be reduced to 70.1 % of its original value (70.60 s) by applying the autoscaling mechanism. The effectiveness of the tool is validated - though in the context of a small OpenStack environment with little network traffic involved.

Future work should consider larger deployments as well. Another interesting topic could be to use other trend detecting tools than Western Electric rules for performance-based autoscaling of cloud applications. Thereby one could think of "ARIMA models"[5] which analyze performance measurements as an autoregressive time series.

Acknowledgements

The author participates in the "eXperimental Infrastructures for the Future Internet" (XIFI) project which is part of the "Future Internet Private Public Partnership" (FI-PPP) run by the European Commission. The XIFI integration project supports advanced experiments on the FI-PPP core platform in order to leverage existing public investments in advanced infrastructures. In particular, XIFI is establishing a marketplace for test infrastructures and Future Internet services to cope with large trial deployments involving end users. This is being achieved through a core federation of test infrastructures, and by coordinating efforts with ongoing FI infrastructures and pilots assisted by investments in pan-European network infrastructures such as GÉANT.

References

1. Buyya, R., Yeo, C. S., & Venugopal, S. (2008). Market-oriented cloud computing: Vision, hype and reality for delivering IT services as computing utilities. *IEEE International Conference on High Performance Computing and Communications, HPCC 2008*, (S. 5-13). doi:10.1109/HPCC.2008.172
2. Champ, C. W., & Woodall, W. H. (1987). Exact results for Shewhart control charts with supplementary runs rules. *Technometrics, 4*, S. 393-399.
3. Goh, T. N., & Xie, M. (2003). Statistical control of a six sigma process. *Quality Engineering, 15*(4), S. 587-592.
4. Haynes, W. (2013). Student's t-Test. In *Encyclopedia of Systems Biology* (S. 2023-2025). New York: Springer New York.
5. Ho, S. L., & Xie, M. (1998). The use of ARIMA models for reliability forecasting and analysis. *Computers & industrial engineering, 35*(1), S. 213-216.
6. Hollander, M., & Wolfe, D. A. (1973). *Nonparametric Statistical Methods.* New York: John Wiley & Sons.
7. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. M., & Irwin, J. (1997). ECOOP'97—Object-oriented programming. *Aspect-oriented programming* (S. 220-242). Berlin: Springer Berlin Heidelberg.
8. Kleppe, A. G., Warmer, J. B., & Bast, W. (2003). *Kleppe, A. G., Warmer, J. B., & Bast, W. (2003). MDA explained: the model driven architecture: practice and promise.* Addison-Wesley Professional.
9. Koutras, M. V., Bersimis, S., & Maravelakis, P. E. (2007). Statistical process control using Shewhart control charts with supplementary runs rules. *Methodology and Computing in Applied Probability, 9*(2), S. 207-224.
10. MacGregor, J. F., & Kourti, T. (1995). Statistical process control of multivariate processes. *Control Engineering Practice, 3*(3), S. 403-414.
11. Montgomery, D. C., & Woodall, H. W. (1999). Research issues and ideas in statistical process control. *Journal of Quality Technology, 4*, S. 376-387.
12. Skiena, S. S. (2010). *The Algorithm Design Manual* (2nd ed.). London: Springer London.
13. Western, E. (1956). *Statistical Quality Control Handbook.* Indianapolis, Ind.: Western Electric Corporation.
14. Yang, J. H., & Yang, M. S. (2005). A control chart pattern recognition system using a statistical correlation coefficient method. *Computers & Industrial Engineering, 48*(2), S. 205-221.