Automatable & Scalable Late Cycle Performance Analysis

Florian Mangold
Institut für Informatik
Universität München
Oettingenstr. 67
80538 München
Germany
mangold@pst.ifi.lmu.de

Moritz Hammer
Institut für Informatik
Universität München
Oettingenstr. 67
80538 München
Germany
hammer@pst.ifi.lmu.de

Harald Roelle
CT SE 1
Siemens AG
Otto-Hahn-Ring 6
81730 München
Germany
harald.roelle@siemens.com

ABSTRACT

Performance analysis of large, concurrent systems is a difficult problem that can hardly be approached with classical profiling. Performance issues might be caused by the interaction of modules and hardware components, making it difficult to find exact causes by considering single modules. By slowing down single modules artificially, dependencies of modules can be detected. Employing statistical means, such dependencies are detected in the covariance of runtime changes. We propose a way to detect the most meaningful dependencies in large-scale systems, allowing arbitrary scaling with respect to the granularity considered.

Categories and Subject Descriptors: C.4 [PERFORMANCE OF SYSTEMS]: Measurement techniques

General Terms: Measurement, Performance

Keywords: performance analysis, software performance engineering, profiling, latent component performance interdependencies

1. INTRODUCTION

The performance analysis of systems is a very important subject in software engineering. Full functional software with performance problems is considered as suboptimal. Among other things this situation creates costs and can result, in extreme cases, in inoperative software. Modern software systems consist of a huge amount of different components. These components are often developed externally, so that there is little information about their internal working. If there is a performance problem, the size of the software system prohibits effective profiling respectively an easy tuning of the system. If the profiled system is concurrent there are some additional effects, e.g., race conditions. Such effects are often not explainable through the run time behavior of the isolated components and therefore can not be grasped with classical profiling. We introduce a novel method to find and analyze performance dependencies in large, concurrent systems. We consider it to be a complementary technique to traditional profiling methods. For this, the system is executed several times and the performance of individual code fragments is varied at adequate locations. Generally there is no way to speed up parts of a system in an automated way, hence we slow them down artificially. In this work, the slowing down of code fragments is called "**prolongation**". With the measured variations in the runtime behavior of the system, we can draw conclusions on the inherent performance dependencies in the system. The granularity of the system units can be arbitrarily defined. The performance dependencies of a system for example can be analyzed on method, class or component level. Our approach to prolong software units is easy to implement and it is largely independent from the programming language used. We do not need to modify the code of the system. The measured data of the performance dependencies between modules can be further processed with statistical analyzing methods to extract the relevant information. Through this, our approach offers superior scalability because unmanageable amounts of information are aggregated to single factors.

WOSP/SIPEW'10, January 28–30, 2010, San Jose, California, USA.
ACM 978-1-60558-563-5/10/01.

2. PERFORMANCE ANALYSIS

At each phase in the software life cycle different information is available and thus, methods to achieve good performance in software systems differ. Woodside et al. [4] identify two general approaches to software performance engineering, a *model based* early approach and the common *measurement-based* approach late in the software life cycle. All approaches of Software Performance Engineering respectively Software Performance Analysis have some drawbacks. The early model-based approach integrates performance dependencies and interconnections, but to build a model experts are needed and sufficient knowledge must be available. So this approach is effective but also very costly and time consuming and therefore not generally accepted. Profiling and Tuning on module and unit level is insufficient because side effects and interconnections with other modules and hardware is not regarded. The late cycle measurement based approach only shows if the performance is inadequate but does not show the latent cause and performance dependencies of modules. The output consists only of absolute data, just likein the models of the early phase in performance analysis the co-operation is hidden. Our approach builds a bridge between the different approaches of software performance engineering. Notably it is an empirical examination in a running system, side effects with the operating system or similar interdependences can be identified. We see this point as an advantage because we incorporate the highly complex system environments in our analysis.

2.1 Performance Dependencies

We investigate software systems composed of *modules*. A *module* is the smallest considered unit from arbitrary granularity.

Definition 1. A module m is called *direct performance dependent* on module m' if a variation of module m' causes an alteration of the performance of m. A module m is called *performance dependent* if there exists a module m' on which it is performance dependent. Otherwise module m is called *performance independent.*

A module m is performance dependent on all modules $m_1, \ldots, m_3$ it calls synchronously. The runtime of m is the total time spent on its own code and that of the modules it invokes. Obviously, a performance variation of m_i $(1 < i < n)$ will cause a performance alteration of m. A module can also be performance dependent on another module without explicitly calling it, for example by using a shared resource concurrently. A variation of one of the modules (for example releasing the resource earlier) can then change the performance of other modules. A module is performance independent if its performance is not affected by any other module. Performance dependence is of interest because it identifies the capability for performance improvement. As the performance of one performance dependent module m is dependent on another module m', an alteration of m' can change the performance of m. This is very interesting in concurrent systems where concurrent resource use can result in performance problems that are difficult to analyze using classical profiling that follows the method invocation while neglecting inter-thread issues.

2.2 Prolongation

The means we propose for detecting arbitrary performance dependencies is to artificially slow down modules while measuring the behavior of other modules. This is accomplished in two phases:

1. The runtime of each module must be measured. For example the time between call and return of a method invocation is measured and logged.

2. The performance of individual modules must be varied (prolonged) without causing functional changes of the code. This is, for example, accomplished by adding waiting time right after method invocation.

Given means to execute this two phases, a single *prolongation run* is conducted by prolonging a single module while leaving the others unmodified. The overall system's performance dependencies are determined by conducting a prolongation run for each module in turn. The runtime of each individual modules is measured and logged during each run. The runtime of modules are considered to be the variables. The prolongation then introduces artificial variance in the performance of the modules. By detecting covariant variables, performance dependencies can be detected. If a module m is performance dependent on another module m', prolonging m' will result in a slowdown of m.

2.3 Factor analysis

It is practically impossible to detect, from the logged data alone, the dependencies and latent interconnections of all the modules by hand. Due to the large amount of modules, the complexity of the data is incomprehensible. In order to make the structure of the data more visible, we apply factor analysis [1] to examine the data. Factor analysis is performing a dimension reduction of the multidimensional (multivariate) data whereby the underlying causes and effects (factors) are detected. This is not a priori knowledge for the developer. Code is re-used, systems are distributedly developed.

An elegant and readily implemented method is to instrument Java code with AspectJ [2, 3] for realizing the prolongation and the logging functionality. If the granularity of modules is chosen to be method invocation, pointcuts can be used for introducing the prolongation; if a larger granulatiry is desired, we still instrument at the level of methods but aggregate the data during logging. In a C project we used C preprocessing macros to wrap functions.

The instrumented system executed in different prolongation runs, with each run prolonging a single module. From our experience it is advisable that each relevant module is prolonged with at least three different time intervals, but interference by the operating system and the effects of non-deterministic thread scheduling might make an even higher number desirable for large-scale applications. Usually, this makes it necessary to resort to a granularity higher than individual methods in order to keep the number of required prolongation runs to a tracktable number.

Our approach is useful as it supports the user in detecting possible potential for improvement that is not directly linked to problematic modules. Classical profiling will produce the modules that consume most of the runtime. Often, this is sufficient as a starting point for performance optimization. Sometimes, however, the degraded system performance cannot be explained by a single module alone, or it is not possible to optimize the problematic module. It is then advisable to consider the modules found in the same factor, as the problematic module is performance dependent on them; if their performance can be improved (or their resource usage optimized), the problematic module will also exhibit a performance improvement.

3. CONCLUSIONS

In this work we presented a novel approach for a performance analysis of large-scale, concurrent systems. It helps to explain performance problems introduced by concurrency which remain unexplained by classical, invocation-based profiling methods. Furthermore our approach is almost arbitrarily scaleable. To handle the resulting amount of data we suggest factor analysis, a statistical tool to discover latent factors from the interconnection of the modules. Our experience suggests that this approach, by its convenience and its explanatory potential, is suitable for existing enterprise software systems.

4. REFERENCES

[1] Backhaus, Klaus et al. *Multivariate Analysemethoden.* Springer, 1996.

[2] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with aspectj. *Commun. ACM*, 44(10):59–65, 2001.

[3] The Eclipse Foundation. The AspectJ Project. `http://www.eclipse.org/aspectj/`, 2007.

[4] M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 171–187, Washington, DC, USA, 2007. IEEE Computer Society.