The Journal of Systems and Software 93 (2014) 163–186

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Flexible resource monitoring of Java programs

Holger Eichelberger*, Klaus Schmid

Software Systems Engineering, University of Hildesheim, Marienburger Platz 22, 31141 Hildesheim, Germany

ARTICLE INFO

Article history:
Received 13 January 2014
Accepted 15 February 2014
Available online 27 February 2014

Keywords:
Resource monitoring
Software components
Performance engineering
Empirical analysis
Monitoring overhead
Java

ABSTRACT

Monitoring resource consumptions is fundamental in software engineering, e.g., in validation of quality requirements, performance engineering, or adaptive software systems. However, resource monitoring does not come for free as it typically leads to overhead in the observed program. Minimizing this overhead and increasing the reliability of the monitored data is a major goal in realizing resource monitoring tools. Typically, this is achieved by limiting capabilities, e.g., supported resources, granularity of the monitoring focus, or runtime access to results. Thus, in practice often several approaches must be combined to obtain relevant information.

We describe SPASS-meter, a novel resource monitoring approach for Java and Android Apps, which combines these conflicting capabilities with low overhead. SPASS-meter supports a large set of resources, flexible configuration of the monitoring scope even for user-defined semantic units (components), runtime analysis and online access to monitoring results in a platform-independent way. We discuss the concepts of SPASS-meter, its architecture, realization and validation, the latter in terms of case studies and an overhead analysis based on performance experiments with SPASS-meter, OpenCore and Kieker. SPASS-meter provides a detailed view of the runtime resource consumption at reasonable overhead of less than 3% processing power and 0.5% memory consumption in our experiments.

1. Introduction

Monitoring is a fundamental activity in software engineering as it is a perquisite to control and systematically improve the quality of the software development process and the developed software. Due to the ever increasing complexity of software, monitoring approaches must provide efficient, scalable, and flexible access to monitored properties. This is in particular true in the area of software engineering approaches relying on runtime monitoring, e.g., the engineering of adaptive software systems, as this requires efficient and immediate (online) access to observed information. However, fulfilling characteristics such as efficiency, scalability and flexibility for a monitoring approach typically also increases the costs of monitoring in terms of configuring a monitoring tool or its impact on the observed system (monitoring overhead).

Monitoring software properties can be performed at various points in time during the software lifecycle, in particular at development time or at runtime of the System Under Monitoring (SUM). Traditionally, *monitoring at development time* aims at detecting problems such as bottlenecks or (accidental) excessive resource consumption. For this purpose, profiling tools are applied most frequently (Snatzke, 2008). Typically, tools which aim at monitoring individual software properties within a SUM insert probes into the SUM in order to collect data which is neither accessible from the SUM nor from the execution environment. One example for such a property is the memory consumed by an individual function of the SUM. In order to make these properties accessible, profilers usually collect large amounts of data to support generic analysis and, thus, cause noticeable overheads on processing power and memory consumption. For example, in Okanovic and Vidakovic (2011), these overheads are characterized as "significant"; in Dmitriev (2003) overheads for the NetBeans profiler of up to factor 50 have been reported and in Eichelberger and Schmid (2012) we determined the overhead of one specific profiler as 32% of the processing power. For monitoring at development time such overheads are often not so problematic, as these monitoring activities are typically not time critical. In contrast, *monitoring at runtime* observes characteristics of the SUM while it is actually being executed in the field (Zanikolas and Sakellariou, 2005), e.g., to validate quality requirements, to analyze the quality of service, to observe the compliance of service level agreements or even to manipulate the SUM (Bubak et al., 2003). In the runtime case, overheads are very problematic as they affect the user's perception of the system's quality as well as the precision of the observed data, which, in turn, may affect the adequacy of subsequent reactions.

* Corresponding author. Tel.: +49 5121 883762; fax: +49 5121 883 769.
E-mail addresses: eichelberger@sse.uni-hildesheim.de (H. Eichelberger), schmid@sse.uni-hildesheim.de (K. Schmid).

http://dx.doi.org/10.1016/j.jss.2014.02.022
0164-1212/© 2014 Elsevier Inc. All rights reserved.

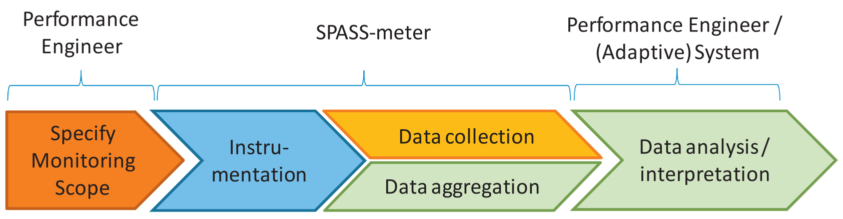

Fig. 1. Monitoring process.

Monitoring overhead can be mastered by focusing on certain monitoring capabilities, such as limiting the number or types of supported resources or restricting the supported forms of (online) analysis. Thus, the challenge is to deal with these trade-offs in order to enable deep insights at low overhead. The corresponding research question in this article is:

Can we realize an efficient, scalable and flexible online monitoring approach to observe a broad number of physical resources while causing only a small runtime overhead?

The core idea of our approach is to enable the performance engineer to focus on parts of interest of the SUM, in particular on semantic units such as individual components or services. Accordingly, the insertion of probes and the subsequent (online) data analysis can be configured for those parts of interest and perform more efficiently. In particular, this enables comparisons of the parts of interest among each other as well as to the entire SUM, whereby SUM-level monitoring is typically provided by the operation system in an efficient manner. The relevant parts of the SUM are expressed in terms of the *monitoring scope*, e.g., as the packages, classes or methods the monitoring activities should be concentrated on. As an example, let us assume that the performance engineer is interested in the resource consumption (here CPU and memory usage) of the default and an alternative storage component. Then, monitoring shall observe these particular semantic units of the SUM and the subsequent data analysis should support the comparative statements (here including SUM-level information) such as: *The default storage component causes 23% of the total heap allocation with peaks impacting the load of the entire device, while the alternative storage component requires less heap allocation and no memory peaks*. Consequently, the monitoring scope focuses on the externally visible interfaces of both components and implies dependent parts, e.g., due to method calls, but also the resource consumption of the entire SUM.

The specification of the monitoring scope by the performance engineer is the first step in the monitoring process illustrated in Fig. 1. Using the scope configuration as input, in the instrumentation step, our monitoring tool SPASS-meter[1] augments the SUM with monitoring probes, i.e., additional code to obtain information on individual resource consumptions. Probe insertion may be done statically, i.e., before starting the SUM, or, alternatively, at certain points in time during runtime of the SUM such as load or initialization time. The data collection step is responsible for receiving relevant resource consumption data from the probes. Data aggregation determines the resource consumption of individual SUM parts. In SPASS-meter, data collection is interleaved with data aggregation to reduce memory overhead, i.e., to avoid storing a plethora of data needed to cope with dynamism and polymorphism in object-oriented systems. Further data analysis and interpretation is left to the performance engineer or additional systems that may rely on live data or post-mortem summaries provided by SPASS-meter.

In this article, we will focus on monitoring the resource consumption of Java programs as SPASS-meter aims at Java SUMs. Moreover, our approach and its implementation are designed to support also other SUM types.

In summary, the core contributions for answering the research question in this article are:

C1: **Flexible definition of the monitoring scope** expressing the particular interest in a SUM. Individual methods or functions but also entire classes or modules may be defined as members of logical monitoring groups. During analysis, the resource consumption caused by a monitoring group is aggregated for the group members and can be compared to other monitoring groups or to the data aggregated for the entire SUM, the virtual machine or the operating system.
C2: **A large and configurable set of accountable resources**, namely CPU time, response time as consumed system time, memory allocation and memory use, network and file usage in terms of transferred bytes and load as fraction of the CPU utilization. Basically, the relevant resources may be configured for the entire SUM but also for individual monitoring groups.
C3: **Multi-level data aggregation** to relate the resource consumption of individual monitoring groups with that of the total SUM, the runtime environment or the entire system.
C4: **Independence from the runtime environment**, i.e., the approach is portable as it does not require explicit modifications to the runtime environment and can even be applied in resource-restricted environments such as Android devices.
C5: **Low monitoring overhead**. Definition of the monitoring scope as well as configuration of the considered resources helps avoiding superfluous monitoring operations and, thus, supports in reducing the monitoring overhead.

The remainder of this article is organized as follows: In Section 2, we will discuss related work and provide a literature-based benchmark of related monitoring approaches. In Section 3, we will focus on the flexible configuration of the monitoring scope (C1). In Section 4, we will discuss the various types of resources supported by SPASS-meter as well as the fundamental techniques and strategies to realize monitoring probes for these resources (C2). In Section 5, we will detail the various levels of data aggregation, including user-specified monitoring groups as well as predefined system-specific levels (C3). In Section 6, we focus on the efficient realization of the concepts introduced before, i.e., the decisions leading to the architecture of SPASS-meter, its independence from the runtime environment (C4), as well as specific implementation tradeoffs. In Section 7, we will validate the individual contributions and, thus, our answer to the general research question in terms of case studies and performance experiments to determine the monitoring

[1] SPASS is the acronym for Simplifying the develoPment of Adaptive Software Systems and SPASS-meter is one of the foundational building bricks of our work on that topic. In German, the term “Spass” means “fun” and points to the tons of fun we had while realizing this tool.