Towards Platform-Independent Component Measurement

Marcus MEYERHÖFER, Frank LAUTERWALD
Chair of Database Systems - Friedrich-Alexander University of Erlangen-Nuremberg
Martensstr. 3, 91058 Erlangen, Germany
{Marcus.Meyerhoefer|sifrlaut@i6.informatik.uni-erlangen.de}

Abstract

Deriving resource properties of software components is a prerequisite for the specification of non-functional properties allowing a developer to also consider these properties for the application to build. In this paper, we describe a platform-independent approach to measure Java components in terms of a selected set of basic constituents (so called atoms), which partition a program and are used to describe concrete runs of components. We combine those abstract measurements with data describing a concrete platform (so called system vector), which results in the estimation of the components' run-time on that previously unknown platform. Furthermore, we elaborate on how to choose a set of atoms and how to determine a system vector for an unknown platform. Our implementation uses bytecode instrumentation for the measurement of components and is able to support different sets of atoms.

1. INTRODUCTION

The application of component-based software engineering (CBSE) to industrial software development has risen with the propagation of mature component models in the last years, especially on the server side. Generally, this movement should be appreciated, because a widespread adoption of CBSE hopefully will facilitate the old dream of building software by using pre-existing, well-tested solutions which only have to be appropriately composed to yield the software to be built. But in practice, there are many obstacles on the way towards the component heaven, e.g. searching and finding the relevant components, different component models or components that do not fit because of not matching or just underspecified interfaces.

Besides those mainly functional issues the problem of non-functional properties (NFPs) [12] of components as well as assemblies of components is even more important, because if an application exhibits unacceptable performance, consumes too much resources or does not scale well, it might be of no use. Because of that, NFPs should be made an integral part in an component-oriented development process allowing a developer to choose among components not solely based on functional but on non-functional constraints (e.g. run-time or resource consumption) as well. One prerequisite for this to work is to have a mechanism to determine the non-functional properties of interest of components given—independent whether they were developed by third parties or in-house. This also paves the way to have non-functional component specifications which then can be used for the estimation of selected properties of an assembly before it is actually built. The main difficulty in the determination of such properties, in particular resource properties, is their inherent context dependency and therefore the limited usefulness in the usually unknown deployment environments a component will be used later on. One of the most interesting but also most complex properties is the run-time of a component which obviously is very context dependent (hardware, sometimes also software, e.g. the version of the VM in the case of Java). While it is clear that some kind of time measurement is a solution to characterize a component on a given and fixed platform, it is not clear what a platform- or even context-independent unit of CPU demand should be.

The contribution of this paper is to propose a novel approach to estimate the run-time of components executed on a Java Virtual Machine, aiming at platform-independence of the component measurements and using a mapping to yield concrete run-times on a new platform. It is based on the

assumption that every program can be somehow partitioned into a set of primitive constituents (called atoms) which can be used to describe concrete runs of a program. Combined with the information on the runtimes of the atoms chosen on a new, formerly unknown platform we are able to estimate the run-time of the components on that platform.

The remainder of this paper is organized as follows. Section two gives a short introduction to the technical background of our implementation. In Section three we describe our approach and discuss how to choose the atoms of measurement and how to determine their duration on a specific platform. Section four gives an overview of our implementation before we adress some related work in Section five. In Section six we conclude and discuss some open issues.

2. BACKGROUND

Before introducing our approach we will give a short overview of the techniques involved—the Java virtual machine as execution platform and its several interfaces which can be used for profiling and debugging of Java software.

The Java VM

Java programs are usually executed on a Java Virtual Machine (JVM) [2]. This implies an extra software layer between the application and the actual processor executing it. This layer imposes additional problems for performance prediction as the inner workings of the JVM are usually not exactly known. Modern implementations like Sun's HotSpot VM may choose to compile an application to native code or further optimize it. This VM features an interpreter as well as a compiler. While the program is executed by the interpreter, the JVM gathers information about performance critical parts of the code and compiles only the parts where the performance benefit of running the code natively outweighs the cost of compilation. Thus, it is difficult to estimate how long the execution of a given piece of code will take, as it is not known whether it will be executed in interpreted or in native mode. On the other hand, the existence of the JVM is convenient as it may provide various interfaces for gathering data about the application being executed, the most important ones beeing described below.

JVMPI and JVMTI

In order to measure the duration and execution frequencies of atoms automatically (i.e. without manually analyzing the bytecode or writing special tests to measure the duration of each atom), it is convenient to be able to gather data about what code is executed, when and how often. The following interfaces can be used for this task:

With the publication of version 1.2 of the Java software development kit in 1998 the JVMP interface (JVMPI) [3],[11] especially designed for profiling purposes was added to the standard and has since been widely used for the creation of profilers. However, the JVMPI interface never changed it status and has been rated “experimental” since its introduction. JVMPI incorporates an event-driven mechanism for the development of profiling software into the Java Virtual Machine (VM): there are more than 30 events defined, such as loading of a class into the VM, object allocation or object destruction. JVMPI is mainly based on events but additionally endows the programmer with the ability to perform arbitrary instrumentation of the Java bytecode inside the profiler by handling an event that notifies the profiler about the loading of the bytecode for one class.

With the new version 5.0 (codename “Tiger”) of the Java SDK, which was released in autumn 2004, a new interface for profiling tools was introduced: the Java Virtual Machine Tool Interface (JVMTI) [4]. It combines the JVMPI and the Java Virtual Machine Debugger Interface (JVMDI)—a former standard mainly aimed at debugging Java software. The most important change of the new specification is that the event-driven mechanism of JVMPI are being superseded by a new profiling paradigm: the new architecture now relies on bytecode engineering. JVMPI and JVMDI will still be available in v5.0 but are marked as “deprecated” and will be removed sometime in the future.

The core part of a profiler for the Java platform—irrespective whether based on the old JVMPI or the new JVMTI—is implemented as a platform-specific library that is loaded by the VM into the address space of the profilee. This so called profiler agent has to register for the events of interest and will be notified by the VM whenever such events occur allowing the agent to request additional event-describing data.

3. DESCRIPTION OF APPROACH

In [6] we have described an infrastructure for performance measurement of Enterprise Java Beans. Using the interceptor concept of the application server JBoss [5], the infrastructure facilitates the transparent response time measurement of EJB components [1]. However, while these measurements are a valuable source for the non-functional specification of a component, the results are only valid on the specific platform the measurements were taken on. To overcome this limitation, we intend to use a platform-independent measurement unit; combined with additional information of a new platform the "abstract" measurement will be mapped to a resulting concrete time measurement. Note however, that our approach is not limited to EJB components on a Java VM but can be used for any Java program that satisfies our assumptions stated below.

Our approach to estimate the runtime of programs executed on a Java Virtual Machine (JVM) is based on the assumption, that every program can somehow be partitioned into a set of primitive constituents. We shall call these constituents atoms. While there may be multiple possibilities to split a program into atoms, a good choice is one that ensures that the following conditions are met:

- It is possible to determine the duration of each atom as well as the execution frequency of each atom within a certain run of a given program.
- The duration of an atom is the same for each execution of the atom. This means that the atoms are not influenced by the context in which they are executed.
- Every program can be partitioned into the set of atoms.

The programs we look at behave deterministically; i.e. when a program is run multiple times with fixed parameters pa_i the execution frequencies of the program's atoms are the same for every run. This of course excludes the category of interactive programs.

If these conditions are met, it is possible to construct an Application Vector P for every program and a System Vector S for every computer system C where $P = (p_1 \quad p_2 \quad p_3 \quad \ldots \quad p_n)^T$ holds the execution frequencies of the n atoms a program may consist of, while $S = (s_1 \quad s_2 \quad s_3 \quad \ldots \quad s_n)^T$ holds the duration of these atoms on the system C.

Assuming that the execution time of atoms is context-independent, we can compute the execution time of a program as the dot product $P \bullet S$ of those vectors. Note that this does not imply any special choice of atoms as long as the above assumptions hold true. Predicting the performance of an Java application or Java component using our approach consists of the following tasks:

– Choose a suitable set of atoms (this has to be done only once).
– Determine the system vector (this has to be done once only for each system to be predicted).
– Determine the application vector (the application for example could refer to a "regular" Java application or to a method of an EJB component).

The estimation of a program's runtime from a known program and system vector is considered so straightforward that it is not discussed any further here. Therefore, the rest of this section will focus on the three tasks given above, while we try to achieve the following goals:

– We aim at complete automation, i.e. computing the application vector for a certain program should not depend on support by the program's author (e.g. in the form of special comments) or on manually analyzing the generated bytecode. Likewise, computing the machine vector should not require any user interaction other than running one script or program.

– We try to separate the prediction from the choice of atoms as far as possible. This allows for easily testing other sets of atoms to check their feasibility for performance prediction.

CHOOSING A SET OF ATOMS

When choosing a set of atoms, we have to consider the necessity that—in order to derive a system vector—it must be possible to measure the duration of those atoms on a specific platform. Therefore, we basically distinguish between atoms for which the duration can be directly measured and atoms for which this is not possible, e.g. because no sufficiently accurate timer is available or because there is no way to determine the beginning and end of an atom's execution. It is however possible to determine the duration for atoms without direct measurement, as long as their execution frequency can be determined. An outline of how this can be done will be described below. The intricacy of this concept is that it requires measurements for at least as many benchmarks as there are different atoms. This in practice turns out to impose a serious restriction on the number of atoms.

The following sections describe our considerations when choosing a set of atoms.

Bytecodes

Using bytecode instructions executed by the JVM as atoms seems a natural choice. Our implementation is able to measure the execution frequency for each instruction within the run of a program. They are however generally thought of as ill-suited (e.g. [7], [8]) because their execution time is not constant enough to derive accurate predictions. This is thought to be especially true in the presence of a JIT compiler. As the duration of individual bytecode instructions is too small, we cannot measure it directly but have to revert to our approach for indirect measurement which is described below.

In spite of all possible problems, we chose individual bytecode instructions as our main set of atoms, because the usage of basic blocks is currently not feasible (see below).

Sequences of bytecodes

We considered basic blocks[1] as atoms in the hope that their execution time will be less prone to JIT optimizations and thus provide more accurate predictions. There are some similarities to individual bytecode instructions: basic blocks, too, are too short to measure their duration directly but our implementation can determine their execution frequency and thus we can—theoretically—compute their duration. Unfortunately, the number of possible basic blocks is too large in practice. As we need at least as many benchmarks for calibration as there are atoms, we are currently unable to use basic blocks as atoms due to the lack of benchmarks. Unfortunately, the creation of such benchmarks can hardly be automized, because semantical knowlegde is necessary to satisfy the pre- and postconditions for the execution of a basic block.

Other sets of atoms

We also considered using other atoms as basis for performance prediction. Of those, well-chosen methods of system classes—as also used by Zhang/Seltzer (see [7])—seem most promising. We plan to augment our current set of atoms with such methods. We expect native methods to be of greatest value in our context because non-native methods can already be accounted for by looking at their executed bytecodes. Note however, the choice of a set of atoms does not affect our next steps and therefore allows the choice of atoms to be optimized later on when further insight is available.

[1] A basic block is a sequence of bytecode instructions that will either be executed completely or not at all. Therefore, only the first instruction of such a sequence may be a label and only the lasts one may be a branch instruction.

DETERMINING THE APPLICATION VECTOR

The application vector (that is, the result of the platform neutral measurement of a component method or Java application in general) is generated by determining how often each atom is executed during the execution of a given application/component. In our environment, this is achieved by a profiler that instruments all classes upon load. For each basic block we insert a counter as well as code to increment this counter each time the basic block is executed. These counters can then be used to determine the execution count of each basic block or to compute the execution count of individual bytecode instructions.

DETERMINING THE SYSTEM VECTOR

As stated above, we distinguish between atoms that have directly measurable execution time and atoms whose execution time can not be directly measured. Obviously, it is trivial to determine the system vector for atoms with directly measurable execution time: the execution time for each atom is measured and the results form the system vector.

In order to obtain the duration for atoms whose duration cannot be directly measured, we propose the following:

Let $P_i = (p_{i1} \quad p_{i2} \quad ... \quad p_{in})^T$, $1 \le i \le m$ be the application vectors for m different benchmark programs, describing each program in terms of the partioning into the chosen atoms A_i, $1 \le i \le n$. Let t_i , $1 \le i \le m$ be the duration of these programs and $T = (t_1 \quad t_2 \quad ... \quad t_m)^T$. Let the duration of atom A_i be s_i and $S = (s_1 \quad s_2 \quad ... \quad s_n)^T$. Then we write the m equations $t_j = P_j \bullet S = p_{j1} * s_1 + p_{j2} * s_2 + ... + p_{jn} * s_n$, $1 \le j \le m$ in the following form:

$$\underbrace{\begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ p_{m1} & p_{m2} & \cdots & p_{mn} \end{pmatrix}}_{application\ vectors} \underbrace{\begin{pmatrix} s_1 \\ s_2 \\ \cdots \\ s_n \end{pmatrix}}_{system\ vector} = \begin{pmatrix} t_1 \\ t_2 \\ \cdots \\ t_m \end{pmatrix}$$

In general, this equation cannot be exactly solved. However, this is a standard problem in mathematics called „linear least squares problem“. An optimal solution is one that minimizes the residual $P \bullet S - T$. Such a solution can be calculated by numerical methods, given that $m \ge n$ and the matrix P has full rank.

4. IMPLEMENTATION

Figure 1 gives an overview of our implementation of a measurement framework. The components to be measured are hosted on an application server and have a special interceptor attached to them [6],[5]. A profiling agent is used to instrument all Java classes as they are loaded. The instrumentation comprises the addition of execution frequency counters for each basic block and has to be done by a second VM, in order not to infere with the setup of the primary VM (e.g. cyclic dependencies between classes required for instrumentation and classes to be instrumented). The interceptors are responsible for starting and stopping the measurements and trigger the transfer of the results to the parser frontend. The frontend does a preprocessing step (e.g. matching of identical blocks, splitting basic blocks into individual byteblocks) and then stores the data to the central repository. The analysis frontend uses this data as well as the runtime information of benchmark programs to generate a system vector and performs prediction.

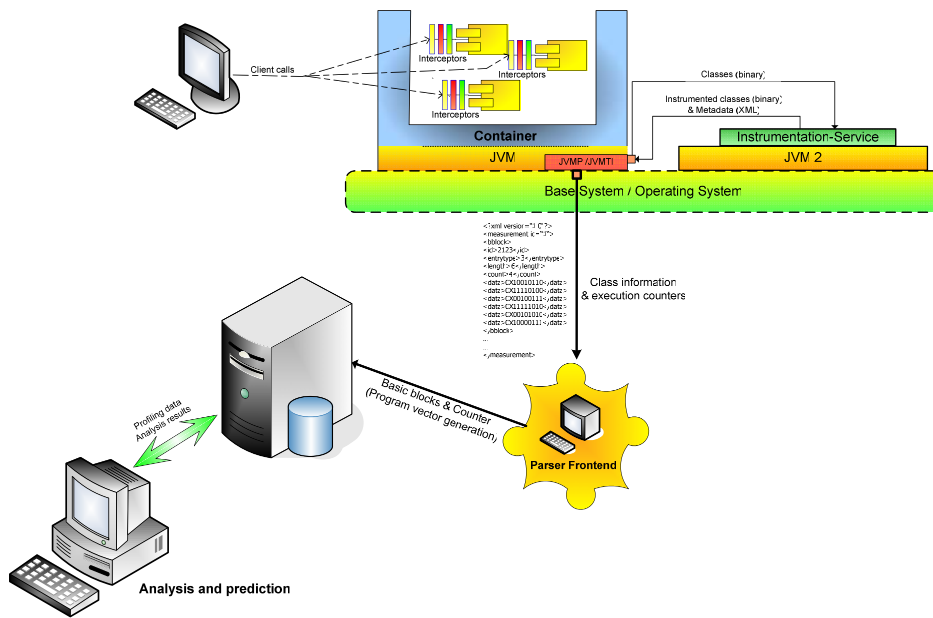

Figure 1 - Architectural overview of the measurement framework

5. RELATED WORK

Peter Wong [8] proposes a concept to predict the performance of Java programs. Sequences of bytecode instructions (similar to our basic blocks), called Sequential Bytecode Blocks (SBB) are used as atoms. The duration of each SBB is measured directly by executing it a large number of times. As a SBB is usually dependent on the context (e.g. it may expect certain variables to be on top of the stack), it is necessary to write special initialization code for each block. This has to be done manually which requires a lot of work, especially for programs consisting of a large number of SBBs. By varying the number of iterations and applying heuristics about when the JVM compiles a piece of code, it is possible to estimate two execution times for each SBB, representing the duration in interpreted and native mode respectively. The execution frequencies are determined by automatic static source code analysis, possibly augmented by manual refinement.

Puschner/Bernat [9] use a similar approach to compute the worst-case execution-time of portable programs, using Java programs as an example. They assume that timing information for an instruction depends only on the local context, e.g. the basic block to which the instruction belongs. They do not, however, state how this timing information is to be computed but just assume that this can be easily done. Another property of their approach is the need for the developer of a software component to annotate his program with path information. This path information is used to compute the execution frequency for each atom once all parameters are known.

In contrast, we view software components as black boxes that may not even be available in source code form and therefore do not expect that the components' developers have taken any specific action to support the prediction of execution times.

Based on the framework described by Puschner/Bernat, Kirner, Puschner and Wenzel [10] try to remedy the need for user annotations and propose a measurement-based approach to derive frequency and timing information. However, their approach still requires either user annotations or restrictions on the allowed code structure.

Zhang/Seltzer [7] also use the concept of a system and an application vector to predict the

execution time of an application on a given machine. It is stated, that optimal results require the JVM vendor to provide the system vector. Different sets of atoms are considered and a set of about 30 methods of system classes are chosen (those atoms are highlevel compared to ours; moreover, they do not have the partitioning property). Microbenchmarks are used to directly measure the execution time of each of these methods. While we currently use a different set of atoms, we plan to eventually add methods of system classes to our set of atoms, especially as the results obtained by Zhang/Seltzer are quite promising. We expect their set of atoms to be less than optimal for numerically-intensive applications that make little use of system classes. We currently target exactly these applications, making our current choice of atoms more suitable at the moment. However, a combination of the two sets of atoms should allow for superior predictive power for different kinds of applications.

Firus et. al. [13] argue for the need of parametric contracts for software components, as the performance of a component can hardly be specified as a single value because of it´s dependencies on external calls and the environment it is deployed in. Therefore they enhance service effect automata by annotating them with random variables to model the time consumption of services offered as well as services called. However, platform independence is not yet achieved as the discrete distributions they use have to be derived somehow and the example in the paper shows a response time distribution which seems to be platform specific. Opposed to that, we try to describe each single service of a component by a platform independent "application vector"; the prediction of assemblies is outside of the scope of this paper. However, the approach of [13] might be combined with our measurements as we will have to deal distributions as well if we describe component services independent of parameter values the services have been invoked with.

6. CONCLUSIONS AND FURTHER WORK

In this paper we have introduced a novel approach for platform-independent measurement of CPU resource usage. We focus on the Java platform and use standard interfaces of the virtual machine to transparently instrument classes in order to measure their runtime behaviour. The basic idea is to measure in terms of some basic units; as shown in Section five and discussed in Section three, there are different possibilities how to choose these units. We have identified three conditions such atoms should satisfy and have described three different sets of atoms. In contrast to other approaches we do not require a developer to attribute his code for performance prediction and have shown how to numerically calculate the duration of atoms. Once such a system vector for a previously unknown platform has been determined, we are able to use it for predicting the runtime of components/applications with a known application vector. Currently, we are completing our prototype; it already incorporates measuring application vectors and durations but lacks support for establishing a system vector as of now.

There are still some issues that require further investigation, mostly how to deal with just-in-time compilation (JIT) of modern VMs. JIT will lead to inaccurate results as the duration of an instruction may vary too much. For this reason, [7] as well as [8] claim that bytecode instructions are a challenging choice of atoms. We hope to overcome this problem by using basic blocks as atoms. To do so, we will need to analyze how many different basic blocks need to be considered for real world applications. We will also have to find a way to easily write microbenchmarks to determine the duration for all these basic blocks. To decrease the number of basic blocks, which we assume to be prohibitively large, we will examine if certain groups of different blocks can be considered equal although they are different in a strict sense[2].

[2] E.g. the following basic blocks, written in pseudo-assembler might be considered equal as they only differ in one instruction and one could assume that the instructions in which they differ are equally fast: load a/ load b / add / store c – load a / load b / sub / store c

Additionally, the influence of JIT-compilers might be tackled by splitting each atom A into two new ones A_{int} and A_{nat} that represent interpreted and native execution respectively. Using a special JVMPI-event, it is possible to determine when a method has been compiled to native code. We can then compute execution frequencies P_{int} and P_{nat} for A_{int} and A_{nat}: P_{int} is the number of executions before the compiled method is loaded and P_{nat} is $P_{total} - P_{\text{int}}$. This solution does, however, introduce a dependency on the JVM: The number of executions before a certain method is compiled is only measured during determination of the application vector i.e. only on one machine and JVM. When determining the system vector on a different machine, the JVM may choose to compile the method at a different time (or not at all).

REFERENCES

[1] Sun Microsystems, "J2EE 1.3 Platform Specifications", July 2001.

[2] Sun Microsystems, "The Java HotSpot Performance Engine Architecture", April 1999, http://java.sun.com/products/hotspot/whitepaper.html

[3] Sun Microsystems, "Java Virtual Machine Profiler Interface", 1998, http://java.sun.com/j2se/1.4.1/docs/guide/jvmpi/jvmpi.html

[4] Sun Microsystems, "The JVM Tool Interface (JVMTI)", 2004, http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html

[5] Fleury M., Reverbel F.,"The JBoss Extensible Server", In: Endler, M., Schmidt, D. (ed.), International Middleware Conference, Rio de Janeiro, Brazil, 2003.

[6] Meyerhöfer M., Neumann C., "TESTEJB – A Measurement Framework for EBJs", in Ivica Crnkovic (Hrsg.): Lecture Notes in Computer Science, International Symposium on Component-based Software Engineering (CBSE7), Edinburgh, Scottland, 2004.

[7] Zhang X., Seltzer M., "HBench:Java: an application-specific benchmarking framework for Java virtual machines", In: Proceedings of the ACM 2000 conference on Java Grande, pp 62 – 70, 2000.

[8] Wong P., "Bytecode Monitoring of Java Programs", Master thesis, University of Warwick, July 2003.

[9] Puschner P., Bernat G., "WCET Analysis of Reusable Portable Code", In: Proceedings of the 13th Euromicro International Conference on Real-Time Systems, pp. 45-52, June 2001.

[10] Kirner R., Puschner P., Wenzel I., "Measurement-Based Worst-Case Execution Time Analysis using Automatic Test-Data Generation", In: Proc. 4th Euromicro International Workshop on WCET Analysis, pp. 67-70, June 2004.

[11] Viswanathan D., Liang S., "Java Virtual Machine Profiler Interface", IBM Systems Journal, Vol. 39 (1), pp. 82-95, 2000.

[12] Brahnmath, G., Raje, R. R., Olson, A., Bryant, B., Auguston, M., Burt, C., "A quality of service catalog for software components", In: Proc. Southeastern Software Engineering Conf., Huntsville, Alabama, pp. 513-520, 2002.

[13] Firus, V., Becker, S., Happe, J., "Parametric Performance Contracts for QML-specified Software Components", *In:* Proceedings of Formal Foundation of Embedded Software and Component-Based Software Architectures (FESCA), Edinburgh, Scotland, 2005.