Dependability Metrics Research Workshop

Proceedings

edited by
Felix C. Freiling, Irene Eusgeld, and Ralf Reussner

University of Mannheim, Department of Computer Science,

Technical Report TR-2009-002

Mannheim, May 4, 2009

Preface

Within computer science, the term "dependability" has been introduced as a general term to cover all critical quality aspects of computing systems. Following standard terminology, a system is dependable if trust can justifiably be placed in the service it delivers. In the early days of computer science, researchers thought that program correctness was the key to dependability meaning that a program always terminates and satisfies its postcondition if it is started in a state where its precondition holds. Today we know that many other factors influence the well-functioning of a computer system, for example hardware reliability, performance properties, or usability properties.

Justifying reliance in computer systems is based on some form of evidence about such systems. This in turn implies the existence of scientific techniques to derive such evidence from given systems or predict such evidence of systems. In a general sense, these techniques imply a form of measurement. The workshop "Dependability Metrics", which was held on November 10, 2008, at the University of Mannheim, dealt with all aspects of measuring dependability.

The workshop was a continuation event of a research seminar which was held October/November 2005 at Schloss Dagstuhl in Wadern/Germany. The seminar was sponsored by the German Computer Science Society (Gesellschaft für Informatik). The aim of this seminar was to bring together young researchers to work on interesting new foundational aspects of computer science and lay the setting for further development. The results of the seminar were recently published as an edited volume "Dependability Metrics" that appeared as number 4909 in Springer's Lecture Notes in Computer Science series. The workshop in Mannheim was also meant to celebrate the publication of this volume.

A dozen researchers, mainly from Germany, attended the workshop that featured five presentations. This technical report documents two research papers and three abstracts of these presentations.

The organizers wish to thank Sabine Braak and Jürgen Jaap for their help in organizing the event in Mannheim.

April 2009

Felix C. Freiling
Irene Eusgeld
Ralf Reussner

Contents

Industrial Control Systems (ICS): Modeling Methods for Reliability, Security and Vulnerabilities Assessment

Irene Eusgeld

ETH Zürich, Laboratory for Safety Analysis

Abstract ICS include supervisory control and data acquisition (SCADA) systems, distributed control systems (DCS), and other control system configurations such as skid-mounted Programmable Logic Controllers (PLC) as are often found in the industrial control sector. In contrast to traditional information processing systems logic executing in ICS has a direct effect on the physical world. These control systems are critical for the operation of complex infrastructures that are often highly interconnected and thus mutually dependent systems.
Numerous methodical approaches aim at modeling, analysis and simulation of single systems' behavior. However, modeling the interdependencies between different systems and describing their complex behavior by simulation is still an open issue. Although different modeling approaches from classic network theory to bio-inspired methods can be found in scientific literature a comprehensive method for modeling and simulation of interdependencies among complex systems has still not been established. An overall model is needed to provide security and reliability assessment taking into account various kinds of threats and failures. These metrics are essential for a vulnerability analysis. Vulnerability of a critical infrastructure is defined as the presence of flaws or weaknesses in its design, implementation, operation and/or management that render it susceptible to destruction or incapacitation by a threat, in spite of its capacity to absorb and recover ('resilience'). A significant challenge associated with this model may be to create 'what-if' scenarios for the analysis of interdependencies. Interdependencies affect the consequences of single or multiple failures or disruption in interconnected systems. The different types of interdependencies can induce feedback loops which have accelerating or retarding effects on a systems response as observed in system dynamics.
Threats to control systems can come from numerous sources, including hostile governments, terrorist groups, disgruntled employees, malicious intruders, complexities, accidents, natural disasters and malicious or accidental actions by insiders. The threats and failures can impact ICS themselves as well as underlying (controlled) systems. In previous work seven evaluation criteria have been defined and eight good praxis methods have been selected and are briefly described. Analysis of these techniques is undertaken and their suitability for modeling and simulation of interdependent critical infrastructures in general is hypothesized. With

respect to the evaluation code High Level Architecture (HLA), Agent-based Modeling (ABM) and Hybrid system Modeling cover the largest spectrum of characteristics.The question which of these three methods is the most appropriate one to model ICS within interdependencies should be discussed and clarified.

Does Input Data for Quantitative Security Assessment of Organizations Exist?

Steffen Weiß

University of Erlangen, Germany

Abstract Diverse approaches have been developed for security assessment of organizations. Probably the biggest difference is whether approaches are qualitative or quantitative. Even if results must be interpreted and thus help for management is limited, most approaches are qualitative because qualitative assessment is easier to perform. Quantitative assessment is more difficult to perform especially as adequate input data are currently not available in the required amount. Studying this problem in depth, it has been discovered that a central problem is providing input data that can be used for many organizations, so called general data. Having in mind a model for quantitative assessment of organization‘s security which was developed during previous work, it has been investigated whether such general data exists. During the talk, examples from real-world organizations will be given which show that this type of data exists in reality.

Adaptive Capacity Management for the Resource-Efficient Operation of Component-Based Software Systems[⋆]

André van Hoorn

Graduate School TrustSoft
University of Oldenburg
D-26111 Oldenburg, Germany

Abstract Overprovisioning capacity management for application service provision causes underutilized computing resources during low or medium workload periods. This paper gives an overview of our work in progress aiming for improving the resource efficiency in operating large component-based software systems that are exposed to highly varying workloads. Based on continuously updated architectural runtime models of the application and its deployment environment, the number of allocated computing resources as well as the deployment of the software components are automatically adapted with respect to current demands and specified performance requirements.

1 Introduction

Today's enterprise applications are complex, business-critical software systems. An important extra-functional characteristic of these systems is performance, consisting of timing behavior and resource utilization [6]. Especially requirements on timing behavior metrics such as throughput or end-to-end response time are part of the so-called Service Level Agreements (SLAs) the provider and the client of a service agreed on. The SLAs constitute a contractual specification regarding the Quality of Service (QoS) that must be satisfied by the application service provider.

Particularly interactive software systems which are accessible through the Internet are exposed to highly varying and bursty workloads, e.g., in terms of the number of concurrent users or the usage profiles [1, 5, 9]. The timing behavior of such systems is significantly influenced by the workload conditions due to resource contention caused by concurrent demands. Over the last years, capacity management for application service provision was performed in a rather static and overprovisioning way, i.e., deploying software components to a fixed infrastructure of application and database servers which satisfy the needs for anticipated peak workload conditions. Future infrastructure demands are satisfied

[⋆] This work is supported by the German Research Foundation (DFG), grant GRK 1076/1.

in a spirit of "kill-it-with-iron": adding additional resources to the infrastructure or replacing existing resources by more powerful ones. The shortcoming of this approach is that during medium or low workload periods, the allocated resources may be heavily underutilized causing unnecessarily high operating costs due to power consumption or infrastructure leases.

We are working on an automatic approach for adaptive runtime capacity management, overviewed in Section 2, which allows component-based software systems [10] to be operated more efficiently. Efficiency shall be improved by allocating only as much computing resources at a time as required for satisfying the specified SLAs. We consider a set of architecture-level adaptation operations based on which the software system is reconfigured at runtime in a more fine-grained way than for example classic load-balancing or virtualization approaches do.

2 Overview of the Approach

Section 2.1 describes the adaptation operations based on which the software system is reconfigured at runtime. Continuously updated architectural models are used to evaluate the performance of the architecture and that of possible adaptation alternatives. The required information to be captured in the models is outlined in Section 2.2. Section 2.3 gives an overview of the analysis activities.

2.1 Adaptation Operations

We consider the following three architecture-level adaptation operations:

(1) **Node Allocation & Deallocation.** A server node is allocated or deallocated, respectively. In case of an allocation, this includes the installation of an execution environment, e.g., a JBoss runtime environment for Java EE components, but it does not involve any (un)deployment operation of software components. Intuitively, the goal of the allocation is providing additional computing resources and the goal of the deallocation is saving operating costs caused by power consumption or usage fees.

(2) **Software Component Migration.** A software component is undeployed from one execution context and deployed into another. The goals of this fine-grained application-level operation are both to avoid the allocation of additional server nodes or respectively to allow the deallocation of already allocated nodes by executing adaptation operation (1).

(3) **Component-level Load-(un)balancing.** This application-level operation consists of the duplication of a software component and its deployment into another execution context (as well as the reverse direction). Future requests to the component are distributed between the available component instances. The goals of this application-level operation are the same as the goals of operation (2).

A middleware layer is responsible for executing the adaptation operations in a way that is transparent to the software system to be adapted. Operation (1) is the most expensive operation in terms of the time required for executing it. In lab experiments, we measured that software component redeployments similar to the adaptation operations (2) and (3) can be executed within milliseconds [4].

2.2 Architectural Models

Relevant static and dynamic aspects of the software architecture are captured in architectural models of the software system. During runtime, these models are updated through measurements, reflect the architectural changes caused by executed adaptation operations, and are used for the continuous analysis (Section 2.3). The following list gives an overview of the information to be captured in the models:

- Components (interfaces and internal performance-relevant behavior)
- Assembly (connection of the components through their interfaces)
- Deployment environment (resources and their performance characteristics)
- Component deployment (mapping of components to execution contexts)
- SLAs and internal performance requirements (component interfaces)
- Adaptation
  - Components to which the adaptation operations are applicable
  - Conditions or rules when to perform an adaptation (analysis)

The performance-relevant modeling of the software architecture will be based on the state of the art in modeling for software performance prediction by annotating architecture models with performance aspects which can be transformed into solvable performance analysis models like queueing networks [2]. Examples for architecture-level software performance modeling notations are the UML SPT/MARTE profiles [7,8] or the Palladio Component Model [3] for performance prediction of component-based software systems.

2.3 Runtime Analysis

The continuous runtime analysis constitutes the core part of the approach. We identified four main analysis activities to be executed. All these activities rely on the runtime model of the software architecture which needs to be continuously updated through measurements.

(1) **Performance evaluation**. In this activity, the performance of the current system configuration is evaluated. This includes whether or not performance requirements (especially the SLAs) are satisfied and to what degree the resources are utilized.
(2) **Workload analysis and estimation**. The result of this activity is an estimation of the near-future workload derived from trends in past workload measurements.

(3) **Performance prediction**. In this activity, the performance of the current system configuration is predicted based on the performance evaluation and the workload estimation in activities (1) and (2). Performance analysis models derived from the architectural models are used for prediction.
(4) **Adaptation analysis**. The effect of possible adaptations on the performance is evaluated using similar techniques as they were used during the performance prediction activity. The result is a selection of adaptation operations to be executed.

3 Conclusions and Future Work

This paper provided on overview of our work in progress on an approach aimed for improving the resource efficiency in operating large component-based software systems which are exposed to highly varying workloads. Based on continuous analyses, the configuration of the software system in terms of the allocated computing resources and the deployment of components to execution contexts, is adapted using three architecture-level adaptation operations.

Since the work is still in an early phase, a lot of future work remains: (1) the adaptation operations will be formally specified and implemented as a proof-of-concept; (2) a suitable notation for the architectural models needs to be identified; and particularly, (3) the required runtime analyses must be developed in detail. We plan to perform an evaluation by simulation in the first place before setting up a case study with a realistic application in the lab.

Acknowledgment

The author thanks the participants of the 2008 DMetrics Workshop, as well as the members of the Software Engineering Group and the Graduate School TrustSoft at the University of Oldenburg for giving valuable early feedback on this work.

References

1. Martin F. Arlitt, Diwakar Krishnamurthy, and Jerry Rolia. Characterizing the scalability of a large web-based shopping system. *ACM Transactions on Internet Technology*, 1(1):44–69, 2001.
2. Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.
3. Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio Component Model for Model-Driven Performance Prediction. *Journal of Systems and Software*, 82(1):3–22, 2009. Special Issue: Software Performance - Modeling and Analysis.
4. Sven Bunge. Transparent redeployment in component-based software systems (in German), December 2008. Diploma Thesis, University of Oldenburg.

5. Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. Workload analysis and demand prediction of enterprise data center applications. In *Proceedings of the 2007 IEEE International Symposium on Workload Characterization (IISWC-2007)*, September 2007.
6. Heiko Koziolek. Introduction to performance metrics. In Irene Eusgeld, Felix Freiling, and Ralf Reussner, editors, *Proceedings of the 2005 Dependability Metrics Workshop (DMetrics)*, volume 4909 of *LNCS*, pages 199–203. Springer, 2008.
7. Object Management Group. UML Profile for Schedulability, Performance, and Time. `http://www.omg.org/cgi-bin/doc?formal/2005-01-02`, January 2005.
8. Object Management Group. UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE), Beta 1. OMG adopted specification ptc/07-08-04. `http://www.omg.org/cgi-bin/apps/doc?ptc/07-08-04.pdf`, August 2007.
9. Christopher Stewart, Terence Kelly, and Alex Zhang. Exploiting nonstationarity for performance prediction. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*, pages 31–44. ACM, 2007.
10. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 2. edition, 2002.

A Copula-based Approach for Dependability Analyses of Fault-tolerant Systems With Interdependent Basic Events

Max Walter[1], Sebastian Esch[2], and Philipp Limbourg[2]

[1] Lehrstuhl für Rechnertechnik und Rechnerorganisation, Technische Universität München, Germany
[2] Universität Duisburg-Essen, Germany

Abstract Traditional combinational techniques for the dependability evaluation of fault-tolerant systems rely on the assumption that there are no dependencies in the failure and repair behavior of the components the system consists of.
In practice, however, the components of a system are usually not independent. Prominent examples for inter-component dependencies include failures with a common cause (e.g. due to spatial closeness or a shared design), failure propagation, limited repair resources, failures induced by repair, and overload due to failure.
We have developed a novel approach, based on copulas, for dealing with arbitrary inter-component dependencies. The system is modeled by a traditional reliability block diagram (RBD) describing the redundancy structure of the system. There are no limitations regarding the structure of this diagram. In particular, non-series-parallel diagrams are supported and components can be attributed to more than one of the RBD's edges. In addition to the RBD, an undirected dependency graph is used to identify pairwise dependent components. An edge from component A to component B with label c in this graph means that the components A and B are correlated with factor c. c is a real number between -1 and 1. A value of zero means that A and B are independent (such edges are usually omitted). Positive values indicate dependent components. A value of 1 indicates a strong dependency: A is failed if and only if B is failed. Anti-dependencies can be modeled by using negative values. For instance, a value of -1 indicates a strong anti-dependency: A is failed if and only if B is not failed.
For evaluation, the model is first divided into independent sub-models by a BDD-like approach. The Boolean terms of the reduced sub-models are then converted in such a way that only conjunction terms of positive literals remain. These conjunction terms can be numerically evaluated by calculating the respective Gaussian copula using the coefficients defined by the dependency graph.
The method was implemented in the tool COBAREA (COpula-BAsed REliability and Availability modeling environment) and two examples were created to illustrate the proposed approach.

FOBIC: A Platform-Independent Performance Metric based on Dynamic Java Bytecode Counts

Michael Kuperberg

Chair for Software Design and Quality, University of Karlsruhe, Germany
`mkuper@ipd.uka.de`

Abstract For bytecode-based applications, runtime instruction counts can be used as a dynamic performance metric. However, in practice, the definition of the metric varies and the tools to measure it are either incomplete or require expensive JVM instrumentation for instruction-level counting. Also, the assumptions underlying the metric definition are not discussed. While it is clear that different instruction types have different execution durations so they must be counted separately, existing approaches often do not identify method invocations, or do not consider the parameters of bytecode instructions and method invocations, despite their substantial impact on the performance. Also, the impact of just-in-time compilation and other runtime virtual machine optimisations on this metric are an open question. In this report, we give a precise definition of a platform-independent performance metric, which is based on runtime counting of executed bytecode instructions and method invocations. We discuss the computation of the metric with the ByCounter tool, which instruments only the application bytecode and not the JVM, and which can be used without modifications on any JVM.

1 Introduction

The runtime behaviour of applications has functional aspects such as correctness, but also extra-functional aspects, such as performance. Classic performance metrics are response time of a method or a service, utilisation of a resource, or throughput of a system [6], but they are platform-specific as they are measured on a concrete platform (i.e. hardware, operating system etc.). These performance metrics can be expressed with parametrisation over the usage profile [7], i.e. over the parameters of the considered method, the workload and the number of users for a system, etc.

For performance prediction, platform-independent performance metrics are preferred, as they allow to disentangle the influence of the execution platform from the influence of the application itself. That is, platform-independent performance metrics should be *parameterised* over the execution platform. The expected benefit is that fewer measurements are needed: for u usage profiles on p platforms, instead of $u \cdot p$ measurements (each usage profile on each platform), just $u + p$ measurements are needed: a platform-independent metric value for each of the u profiles, and p measurements to compute the parametrisation of the platform.

The conventional wisdom in defining platform-independent performance metrics is to look at the *building blocks* of the considered application, and not at the externally observed execution metrics based on time or related to time. During execution, these building blocks use platform resources (e.g. CPU, memory, caches, network connections, hard disks, locks, semaphores, etc.). This usage is subject to resource scheduling and contention, which add waiting times to the actual processing times. In particular, it is not sufficient to count the CPU cycles spent executing the application service, as it is neither the only nor the major factor contributing to the performance of the service.

However, the detailed consideration of platform resources is too complex and requires very detailed knowledge and modeling of the underlying mechanisms, especially scheduling. To take the scheduling effects into account, simulations must be performed, which require substantial time and effort. Therefore, abstractions that simplify the analysis of building blocks' performance have been developed [9]. In this report, we do not consider how the building blocks are mapped to (platform-specific) resource usage, but focus on the obtainment of the metric values w.r.t. building blocks.

The building blocks can be chosen from different software development phases: they can be source code elements, elements of the executable as created by the compiler (e.g. bytecode instructions), or the elements of the executable form as they are indeed executed by the execution platform (e.g. machine code created by just-in-time compilers immediately before the execution or during it).

Source code is not well-suitable for platform-independent metrics as different compilers produce very different executable code, and this can render the metric's values incomparable for programs compiled with different compilers. Additional factors that complicate working with source code (for object-oriented languages) are operator overloading and polymorpism: obtaining performance metrics from source code would thus require knowledge and reasoning capabilities that equal those of a real compiler.

Therefore, the executable form of an application is preferred as the basis for platform-independent performance metrics. For applications that are compiled to machine code, machine code instructions can be considered as building blocks. This means that the "platform-independent" metric only applies to platforms that can execute a given machine code format. To overcome the limited portability of machine code, virtual machines have been invented, which form a layer between the application and the operating system.

A virtual machine can execute applications that have been compiled to an intermediate executable code, for example the Java bytecode which can be executed by the Java Virtual Machine (JVM). Different (source code) languages can be compiled to a given bytecode language: for example, Groovy [5] and Scala [14] compile to Java bytecode. In the remainder of this report, we focus on the Java bytecode, but our ideas and notes generally apply to other bytecode languages, e.g. to the CIL (Common Intermediate Language) of the .NET platform.

The report is structured as follows: in Section 2, we define a dynamic platform-independet performance metric, called FOBIC (Family Of Bytecode Instruction Counts). In Section 3, we describe how to compute FOBIC using the ByCounter tool. Section 4 presents a small overview of the impact of Just-In-Time compilation on FOBIC. Related work is presented in Section 5. Finally, we list our assumptions and limitations in Section 6 and conclude the report in Section 7.

2 FOBIC: a Dynamic Performance Metric for Java Bytecode

The Java bytecode is a stack-based language, and it contains both low-level "elementary" instructions (e.g. `IADD` for additing two primitive integer values) and high-level constructs, such as method invocations. To invoke Java methods from Java bytecode, four instructions are used including those of the Java API: `INVOKEINTERFACE`, `INVOKESPECIAL`, `INVOKESTATIC` and `INVOKEVIRTUAL` (hereafter called `INVOKE*`). The signature of the invoked method appears as the parameter of the `INVOKE*` instruction, while the parameters of the invoked method are prepared on the stack before method invocation. Invoked methods have a large quantitative contribution to the performance of Java bytecode, i.e. the parameters of the `INVOKE*` instructions play a crucial role.

Not only different methods, but also different bytecode instruction types have different execution durations, so they also must be counted separately and individually. Through this, bytecode instruction counting leads to a *family of metrics* (FOM), not to a single metric, because each instruction count has its own unit (which corresponds to the instruction's name/opcode), and each method has its own count (i.e. a metric) as well. As each Java application can define its own methods and its own packages, the size of the FOM is not bounded, and the FOM can be expanded.

Some researchers have suggested basic blocks as bytecode-oriented metrics [11] . Basic blocks are opcode sequences that are not interrupted by control flow statements. Basic blocks can be of different length, and their number is significantly higher than the number of single bytecode instructions (there are potentially $255^4+255^3+255^2+255 > 4 \cdot 10^9$ basic blocks up to and including those with length 4). In this report, we only consider basic blocks of length 1, i.e. individual bytecode instructions and individual method invocations.

We define our metric as an expandable family of metrics, called FOBIC (Family Of Bytecode Instruction Counts), which also contains method invocation counts despite its name. Each bytecode instruction type (opcode) has an own metric in FOBIC, and each method signature has its own metric in FOBIC as well.

To make instances of FOBIC comparable, their elements must be comparable. In particular, two elements (metrics) with the same metric name, one would expect the names to refer to the same method (or instruction). But the package-oriented namespaces in Java are globally non-unique, which means that two (fully-qualified) method signatures from two different origins may be equal, but refer to different physical method *implementations*. In such a case, it is impossible to distinguish between them in the FOBIS instances, and it is the obligation of the user to make sure the elements of compared FOBIC instances refer to the same entities.

2.1 Treatment of Calling Trees and Decomposing Method Implementations

Invoked methods include both calls to the Java API, and calls to methods defined in non-API classes. Furthermore, native methods (i.e. methods defined and implemented outside of Java, e.g. in a shared native library) are supported by the JVM. If an invoked method is part of the Java API, its implementation can be different across operating systems, as it may call platform-specific native methods (e.g. for file system access). Hence, decomposing a method into the elements of its implementation would destroy the platform-independent property of FOBIC.

Thus, to avoid platform-dependent counts, invocations of API and *all* other methods must initially be counted atomically, i.e. as they appear in application's bytecode, without decomposing them into the elements of their implementation. This results in a "flat" view which does not decompose the invoked method into the elements of its implementation. This "flat" view summarises the execution of the analysed bytecode in a platform-independent way. If needed, instruction counts for the *invoked* methods can be obtained using the same approach, too. Using this additional information, counts for the entire (expanded) *calling tree* of the analysed method can be computed, and such stepwise approach promotes reuse of counting results.

2.2 Parameters

For bytecode-based performance prediction, parameters of invoked methods, but also parameters of non-`INVOKE*` bytecode instructions can be significant, because they influence the execution speed of the instruction [8]. The latter parameters and their locations are described in the JVM specification [13]; for example, the `MULTIANEWARRAY` instruction is followed by the array element type and the array dimensionality directly in the bytecode, while the sizes of the individual array dimensions have to be prepared on the stack.

Hence, in order to describe the runtime behaviour of programs as precisely as possible, each metric in the family of metrics must be able to account for the parameters of the instruction/method it counts. This leads to to need to normalise the parameter values: e.g. for the `newarray` instruction, the type of the array and its size must be considered for conversion to a multiple of the norm (unit) of the metric. For a `newarray` of type `double` and size 10, we can specify that it is equal to $2 \cdot 10 = 20$ `newarray` calls with type `int` and size 1 (as a `double` occupies two bytes and `int` just one). But even this simple normalisation is not error-prone: on 64-bit systems, a single `int` may occupy 64 bits (i.e. 2 bytes - as much as a `double`). Also, the JVM usually allocates a minimum array size which may be larger than the given parameter: allocating an array with 5 `int`s may in reality lead to allocation of 10 slots in the array.

Thus, in general, the normalisation is non-linear. There are other challenges as well: bytecode instructions or methods can have parameters of arbitrary object types, and it is not obvious how the object types can be normalised to match the metric unit. It remains to be studied how precise the normalisation must be to fulfill a given precision of the bytecode-based performance prediction.

In the next section, we provide a short overview of ByCounter [10], our approach to obtain Java bytecode frequencies that form the FOBIC metric. ByCounter accounts for method invocations, parameters of methods and instructions.

3 Computing the FOBIC Metric

As we have pointed out in [10], to obtain all these runtime counts, static analysis (i.e. without executing the application) could be used, but it would have to be augmented to evaluate runtime effects of control flow constructs like loops or branches. Even if control flow consideration is attempted with advanced techniques such as symbolic execution, additional effort is required for handling infinite symbolic execution trees [12, pp. 27-31]. Hence, it is often faster and easier to use dynamic (i.e. runtime) analysis for counting executed instructions and invoked methods.

However, dynamic counting of executed Java bytecode instructions is not offered by Java profilers or conventional Java Virtual Machines (JVMs). The `traceInstructions` method in Java platform API class `java.lang.Runtime` that should enable/disable the *tracing* of bytecode instructions did not work for none of the platforms or JVM vendors and versions that we have studied (which included Windows and Linux machines running Sun and Bea JVMs in versions 1.6, 1.5 and 1.4). Same was true for the method `traceMethodCalls` in the same class. Even if would have worked, additional effort would be needed to aggregate the traced instructions/method into counts.

As discussed in [8], existing program behaviour analysis frameworks for Java applications (such as JRAF [2]) do not differentiate between bytecode instruction types, do not identify method invocations performed from bytecode, or do not work at the level of bytecode instructions at all. These frameworks frequently rely on the instrumentation of the JVM, however, such instrumentation requires substantial effort and must be reimplemented for different JVMs.

A portable approach for lightweight portable runtime counting of Java bytecode instructions and method invocations is called ByCounter [10] and it works by instrumenting the application bytecode instead of instrumenting the JVM. Through this, ByCounter can be used with any JVM, and the instrumented application can be executed by any JVM, making the ByCounter approach truly portable. Furthermore, ByCounter does not alter existing method signatures in instrumented classes nor does it require wrappers, so the instrumentation does not lead to any structural changes in the existing application.

To make performance characterisation through bytecode counts more precise, ByCounter provides basic parameter *recording* (e.g. for the array-creating instructions), and it also offers extension hooks for the recording mechanism. ByCounter does *not* perform normalisation, but the recorded parameters allow to perform the normalisation offline, i.e. after the execution of the instrumented method. To do so, ByCounter users can write their normalisation algorithms which operate on the recorded data.

As it is not always technically possible or rational to perform persistent parameter recording by simply saving the parameter value(s). For example, `Object`-typed parameters may not be serialisable. In such a case, a *characterisation* of the parameter object instance should be recorded: for a (custom) data structure, its size could be a suitable characterisation; with the `toString()` method as the "fallback" characterisation. To allow users to provide their own characterisations for Java classes of their choice, ByCounter offers suitable extension hooks.

In [10], ByCounter was evaluated on two different Java virtual machines using applications that are subsets of three Java benchmarks. For these applications, the evaluation showed that despite accounting of single bytecode instructions, the ByCounter overhead during the counting phase at runtime is reasonably low (between ca. 1% and 85% in all cases except one outlier; no parameter recording was performed), while instrumenting the bytecode requires less than 0.3 seconds in all studied cases.

The case study also revealed that "dead code" (i.e. reachable code that can be skipped because it has no side effects, as confirmed by purity analysis) needs special attention w.r.t. bytecode instruction counts. Uninstrumented "dead code" is skipped by those virtual machines that detect its effectlessness, while ByCounter-driven instrumentation forces the JVM to execute the counting facilities inserted by ByCounter. Hence, ByCounter counts metrics for bytecode sections that would be skipped otherwise. To prevent this, a bytecode "cleaner" should be run on bytecode to remove "dead code", or ByCounter could be combined with a purity analysis tool to instrument only non-dead code.

One significant open question is the role of other runtime optimisations performed by the Java Virtual Machine, particularly Just-In-Time compilation (JIT) of bytecode into machine code. Our discussion of JIT in the next section treats detection and control (both explicit and implicit) of timepoint and range of JIT compilation.

4 Impact of Just-In-Time Compilation on FOBIC

JIT can compile bytecode into machine code, but the timepoint and range of JIT compilation are decided by the JVM. In particular, JIT is usually applied to "hot" methods, i.e. only to methods that execute frequently and consume a substantial ratio of the execution time. The detection of "hot" methods takes some time, and is transparent to the executed application. The JIT compilation is carried out in parallel to program execution.

The JIT is important to bytecode-based performance metrics because JIT often speeds up the execution of a method by up to an order of magnitude. Our experience shows that if the duration of a JIT-compiled method is divided by the number of bytecode instructions in the method's original bytecode implementation, the average JIT analogon of an instruction executes in less than a CPU cycle for some methods (of course, these methods have been carefully checked to contain no "dead code").

For **explicit control** of JIT compilation, the Sun JVM provides a command-line flag (`-Xint`) to advise the JVM to run in interpreted mode, where the JIT is not performed[1]. A *non-Sun* JVM may not provide this or other means to *explicitly* control the JIT.

Furthermore, it is not clear when JIT is run and to which method it applies. Collecting such information requires a special JVM, or applying manual changes to the JVM source code. However, specifically for the Sun HotSport JVM, there exist options that provide more control over and insight into the JIT compiler. The list of option is well-hidden in the `-XX` parameters of the Sun JVM which are not listed in command line help, and which are not documented in the official *distributed* docs - they are found online at the Sun HotSpot JVM webpages[2], but Eugene Kuleshov[3] provides a longer list. One can find options such as `LogCompilation`, `PrintCompilation`, `CITime`, `UseCompiler` etc., but the logged results are hard to access (e.g. when they are printed to the command line), and are not well-suited for collecting metrics. It remains to be studied whether JMX agents can be written to extract such information, or whether redirecting "standard output" can be used as a workaround.

For **implicit control** of JIT compilation, the programmers can write additional code to perform a "warmup" of the methods that *should* be subject to JIT compilation. However, the number of calls needed to "warm up" the method depends on the JVM, and it is questionnable whether the warmup phase is realistic.

For **detection** of JIT compilatio, the command-line options described above are currently the only explicit option known to the author. To detect JIT compilation implicitly, one could try to measure the duration of each invocation of each candidate method duration, and conclude that the method has been JIT-compiled once its duration has (significantly) fallen. However, this approach is questionable due to resolution and invocation cost of timer methods, and due to the overhead it introduces (be it manually inserted measurements or profiler-based assessments).

Summarising the facts listed in this section, we can see that (a lot of) additional work must be done to reflect the impact of JIT compilation onto our

[1] The -X options can be listed by typing `java -X` on the command line; they are non-standard and can be changed by the JVM vendor in future versions

[2] http://java.sun.com/javase/technologies/hotspot/vmoptions.jsp

[3] http://www.md.pp.ru/ eu/jdk6options.html

bytecode metrics. To the best of our knowledge, there exists no approach capable of *predicting* the exact scope, duration, outcome and impact of JIT compilation. In our own work, we have addressed the quantitative performance impact of JIT empirically [9].

5 Related Work

Bytecode instruction counts can be considered as a dynamic bytecode metric. In [3], a collection of other metrics for Java bytecode is presented, but that collection does not include execution counts for individual bytecode instructions and method invocations. Herder and Dujmovic [4] have performed frequency analysis of Java bytecodes but disregard method invocations and parameters. O'Donoghue et al. [15] have performed frequency analysis on bytecode bigrams.

Radhakrishnan et al. [16] analysed a suite of Java benchmarks (SPEC JVM98) from a bytecode perspective to analyze bytecode locality, instruction mix and dynamic method sizes. Stephenson et al. [17] use bytecode sequences to optimise virtual machines.

A detailed study of work related to obtaining dynamic instruction counts can be found in [8].

6 Assumptions and Limitations

We assume that it is possible to pass the final class bytecode that will be executed to ByCounter for instrumentation. For applications where bytecode is generated on the fly and not by the Java compiler (for example in Java EE application servers), additional provisions must be taken. We also assume that the bytecode to instrument conforms to the JVM specification, even if it has been protected using obfuscation.

The obtained instruction counts depend on the input parameters that have been provided to the instrumented method, for example due to control flow constructs that depend on these parameters. Currently, this dependency cannot be expressed by ByCounter because neither control flow constructs are recognised by it, nor states of variables/fields during method execution are inspected.

7 Conclusions

This report discussed the details of defining and obtaining a platform-independent performance metric, based on runtime counts of executed bytecode instrucions and method invocations. The metric is called FOBIC, which stands for Family Of Bytecode Instruction Counts, although FOBIC also includes individual counts of method invocations. The report discussed the importance of method invocations from bytecode, the role of instruction parameters and method parameters, and the effects observed for "dead code", i.e. reachable code that may be skipped by the virtual machine because it has no observable effects.

This report also describes how to compute the metric using ByCounter, an approach that transparently instruments the application bytecode and not the JVM, thus simplifying the entire counting process and making the approach portable accross JVMs. The instrumentation added by ByCounter is lightweight, leading to low runtime costs of counting. To minimise disruptions, ByCounter instrumentation preserves the signatures of all methods and constructors, and it also preserves the application architecture. For reporting of counting results, ByCounter offers two alternatives: either using structured log files or using a result collector framework (the latter can aggregate counting results accross methods and classes).

Currently, the metric computed by ByCounter is being integrated into Palladio [1], which is an approach to predict the performance of component-based software architectures. For bytecode-based performance prediction [9], the metric computed by ByCounter has been combined with platform-specific instruction timings (i.e. execution durations) to predict platform-specific response time.

This report has shown that bytecode instruction counting leads to a family of metrics, not a single metric, because each instruction type and each method signature need an individual count. The unit of a family member is then the instruction name (called opcode in Java bytecode), and the method signature, respectively. Further research is needed to normalise the calls of the instruction with different parameters to a fixed-parameter "unit" of the corresponding metric; normalisation is also needed for method invocations and method parameters.

While we have described the effect of Just-In-Time compilation on the computation of bytecode-based metrics and the problems in controlling and detecting Just-In-Time compilation, significant amount of work remains to be done to account for Just-In-Time compilation in ByCounter and in the FOBIC metric we have defined.

Acknowledgements

The author would like to thank Martin Krogmann for his help with implementing ByCounter and Ralf Reussner and Klaus Krogmann for insightful discussions on its design and implementation.

References

1. Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio Component Model for Model-driven Performance Prediction. *Journal of Systems and Software*, 82:3–22, 2009.
2. Walter Binder and Jarle Hulaas. Using Bytecode Instruction Counting as Portable CPU Consumption Metric. *Electr. Notes Theor. Comput. Sci.*, 153(2):57–77, 2006.
3. Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic Metrics for Java. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 149–168, New York, NY, USA, 2003. ACM.
4. Carl Herder and Jozo J. Dujmovic. Frequency Analysis and Timing of Java Bytecodes. Technical report, Computer Science Department, San Francisco State University, 2000. Technical Report SFSU-CS-TR-00.02.

5. D. König, A. Glover, P. King, G. Laforge, and J. Skeet. *Groovy in Action*. Manning, 2007.
6. Heiko Koziolek. *Dependability Metrics*, volume 4909 of *LNCS*, chapter Introduction to Performance Metrics, pages 199–203. Springer, 2008.
7. Heiko Koziolek. *Parameter Dependencies for Reusable Performance Specifications of Software Components*. PhD thesis, University of Oldenburg, 2008.
8. Michael Kuperberg and Steffen Becker. Predicting Software Component Performance: On the Relevance of Parameters for Benchmarking Bytecode and APIs. In Ralf Reussner, Clemens Czyperski, and Wolfgang Weck, editors, *Proceedings of the 12th International Workshop on Component Oriented Programming (WCOP 2007)*, July 2007.
9. Michael Kuperberg, Klaus Krogmann, and Ralf Reussner. Performance Prediction for Black-Box Components using Reengineered Parametric Behaviour Models. In *Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE 2008), Karlsruhe, Germany, 14th-17th October 2008*, volume 5282 of *LNCS*, pages 48–63. Springer, October 2008.
10. Michael Kuperberg, Martin Krogmann, and Ralf Reussner. ByCounter: Portable Runtime Counting of Bytecode Instructions and Method Invocations. In *Proceedings of the 3rd International Workshop on Bytecode Semantics, Verification, Analysis and Transformation, Budapest, Hungary, 5th April 2008 (ETAPS 2008, 11th European Joint Conferences on Theory and Practice of Software)*, 2008.
11. Jonathan Lambert and James F. Power. An Analysis of Basic Blocks within SPECjvm98 Applications. Technical Report NUIM-CS-TR-2005-15, Department of Computer Science, National University of Ireland, Maynooth, Co. Kidare, Ireland, 2005.
12. Jooyong Lee. *Program Validation by Symbolic and Reverse Execution*. PhD thesis, BRICS Ph.D. School, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2006.
13. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.
14. M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. *LAMP-EPFL*, 2004.
15. Diarmuid O'Donoghue, Aine Leddy, James Power, and John Waldron. Bigram Analysis of Java Bytecode Sequences. In *PPPJ '02/IRE '02: Proceedings of the inaugural conference on the Principles and Practice of programming, 2002 and Proceedings of the second workshop on Intermediate representation engineering for virtual machines, 2002*, pages 187–192. National University of Ireland, 2002.
16. R. Radhakrishnan, J. Rubio, and L.K. John. Characterization of Java Applications at Bytecode and Ultra-SPARC Machine Code Levels. In *Proceedings of IEEE International Conference on Computer Design*, pages 281–284, 1999.
17. Ben Stephenson and Wade Holst. Multicodes: Optimizing Virtual Machines Using Bytecode Sequences. In Ron Crocker and Guy L. Steele Jr., editors, *OOPSLA Companion*, pages 328–329. ACM, 2003.

Author Index