Charles University

Faculty of Mathematics and Physics

# Habilitation Thesis

2018 Lubomír Bulej

Charles University

Faculty of Mathematics and Physics

# Habilitation Thesis

Lubomír Bulej

# Performance Awareness and Observability on Modern Platforms

*Computer Science, Software Systems*

Prague, Czech Republic 2018

# Contents

# Acknowledgment

This thesis presents selected results of research in performance evaluation, performance modeling, and dynamic program analysis. The research was carried out during my stay at the Department of Distributed and Dependable Systems of the Faculty of Mathematics and Physics of the Charles University in Prague, Czech Republic, and the Faculty of Informatics of Università della Svizzera italiana (USI) in Lugano, Switzerland.

The research results presented in the thesis are necessarily fruits of a collective, rather than individual effort, and the published papers are but a tip of an iceberg—the results are backed by software prototypes, use cases, and hundreds of days executing experiments and collecting and processing data. Due to complexity of modern software systems and reliance on thorough experimental evaluation, it would be impossible for an individual to complete such work alone while the topic is still relevant. Therefore, I would like to express my gratitude to my superiors and colleagues, with whom I had the pleasure and privilege of working in the last decade (or more), and who have made this work possible.

František Plášil, Petr Tůma, and Tomáš Bureš deserve special credit for (among other things) shouldering the heavy burden of establishing, leading, and sustaining our department. Thanks go to my collaborators—Vlastimil Babka, Martin Děcký, Ilias Gerostathopoulos, Petr Hnětynka, Viliam Holub, Vojtěch Horký, Pavel Ježek, Tomáš Kalibera, Jaroslav Keznikl, Jan Kofroň, Jaroslav Kotrč, Alena Koubková, Petr Kubát, Peter Libič, Lukáš Marek, Pavel Parízek, Andrej Podzimek, Petr Stefan, Ondřej Šerý, and Tomáš Trojánek, as well as to the current and former members of our department—Paolo Arcaini, Rima Al Ali, Martin Blicha, Jakub Daniel, Vlastimil Dort, David Hauzar, Pavel Jančík, Michal Kit, Filip Krijt, Michal Malohlava, Vladimír Matěna, Tomáš Pop, Viliam Šimko, Dominik Škoda, Antonín Steinhouser, Jiří Vinárek, and many others.

Walter Binder deserves a special credit for welcoming me to his research group at USI in Lugano and for the (now years long) fruitful cooperation. My thanks go to the current and former members of the Dynamic Analysis Group— Danilo Ansaloni, Omar Javed, Steven Kell, Andrea Rosà, Luca Salucci, Aibek Sarimbekov, Sebastiano Spicuglia, Haiyang Sun, Alex Villazón, and Yudi Zheng, as well as to our close collaborators—Lydia Y. Chen, Thomas Gross, Matthias Hauswirth, Yao Li, Achille Peternier, Zhengwei Qi, Nathan Ricci, Andreas Sewe, Lukas Stadler, and Thomas Würthinger.

I also gratefully acknowledge the international and national sources of funding which made this work possible. In particular, these are EU FP7 IP 605442 (ASHLEY), EU FP7 FET 257414 (ASCENS), EU FP7 STREP 215013 (Q-ImPrESS), Sciex-NMS$^{CH}$ 11.109 (DYNASTY), SNF 136225 (FAN), MŠMT INTER-EUREKA LTE117003 (ESTABLISH), GAČR P202/10/J042 (FERDINAND), GAČR 201/09/H057 (Res Informatica), and Charles University institutional funding (SVV).

# 1 Introduction

“Software is eating the world.” This now-famous adage [1] is a testament to the importance of software in a modern world. Software controls the engines in our cars, pilots planes, provides access to vast amounts of information, collects, mines, and exploits behavioral data from millions of users, stimulates and influences our social behavior, powers platforms that connect buyers and sellers (or consumers and advertisers), or provides citizens with access to state services.

Our reliance on software comes with expectations. The software systems we rely upon are often required to interact with a huge number of users, to perform their function without error, and to respond in a timely fashion. Failing that often prevents users or consumers from accessing the advertised services [17] or products [18], and in more serious circumstances, prevents citizens from accessing vital services [33, 58, 59].

Meeting these expectations is difficult, because software is notoriously difficult to develop. Not by accident, but rather as a consequence of what software is. Software ranks among the most complex human-engineered artifacts, yet contains very little redundancy—similarities are transformed into new concepts to avoid duplication. Software is infinitely malleable and has to be continuously adapted to conform to ever-changing requirements during its (very long) lifetime. Software is invisible and manifests itself only through behavior resulting from interactions among thousands of concepts captured in millions of lines of code at many different levels of abstraction. Software is discontinuous—a single erroneous statement may cause the system to fail. In other words, difficulty is inherent to software—there is no silver bullet [7, 48].

Many development processes for general-purpose software systems therefore focus primarily on managing complexity so as to deliver correctly functioning software on time [57, 4]. Other aspects of software design and construction, such as performance, which is an important aspect of the overall user experience, are considered to be secondary concerns that only need to be dealt with if they are found unsatisfactory. Even though in software-intensive systems failures are more likely to be caused by performance issues than by a faulty implementation of some features [60, 20], this approach to performance is actually a recommended best practice, succinctly captured as another adage known to most developers: “Premature optimization is the root of all evil.”

Performance of computer systems is difficult to predict, which also applies to impact of any code changes intended to improve performance. It is also commonly accepted that developers are mostly wrong about performance if they just follow their instincts or hunches. An experimental study by Horký et al. suggests that developers only "see" performance when they consciously decide to investigate it, but often introduce code patterns or code modifications based on performance assumptions that may be incorrect, because they are not based on actual performance observations [24]. Given these circumstances and the complexity of software, the idea of supporting performance-related design and code changes with evidence has a lot of merit.

But software performance evaluation is surprisingly difficult to do correctly [8, 42, 5]. It may intrude on the code and development work flow, requires deeper understanding of the execution platform to ensure that the performance measurement code actually observes and measures the right thing, and interpreting the results requires at least some familiarity with statistical hypothesis testing.

Considering that all this effort is not likely to provide immediate benefits, it is not surprising that the old adage (with the word "premature" left out) may have become an excuse for completely ignoring performance aspects of software, never mind the difference between (premature) optimization and frugal use of resources [25].

Where and when did it all go wrong? It appears that it has been going on for quite some time. In his 1974 paper [34], from which the adage originates, Donald E. Knuth was indeed cautioning against micro-optimizations, but also suggesting that good practice is to understand the system and its performance well enough to recognize which parts are critical, so that performance is not given up accidentally or through sloppiness:

> We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.

He also notes that "in established engineering disciplines, 12% performance improvement, easily obtained, is never considered marginal, so why would so many people pronounce it insignificant?" This would suggest that treating performance as a second-level concern (if at all) is not entirely new.

We can conjecture that because advances in computer hardware have been steadily supplying the software industry with "free" performance, dealing with performance in software design simply was not economical. As software grew in scale and complexity, performance was a problem being solved mainly by hardware. Performance for the sake of efficiency did not make economical sense either, because the energy costs of computing were not an issue (or at least a research topic) until recently. The established engineering disciplines did not have this luxury.

Another potentially contributing factor is that the complexity and performance of software is no longer dominated by its core algorithms and data structures, which a single person or a small team could understand completely. The complexity of large software-intensive systems is dominated by scale and architecture, and the behavior is a result of interactions among entities that number in hundreds and thousands. Even though software behavior is precisely defined in its source code, the scale, complexity, and often undocumented intent make the code difficult to comprehend. No single developer can even hope to read, let alone understand, all the code that makes up a software system, which is why developers have to rely (heavily) on abstraction, which has a tendency to hide details that are not related to structure and functional correctness.

How can we even identify the "critical 3%" in such systems, when those 3% can comprise thousands of lines of code spread around libraries and frameworks, passing through many layers of a modern software stack? Even profiling such a system (if at all possible) might not provide an actionable result, because the execution time will be distributed among hundreds of methods, neither of which will stand out in the profile.

In contrast, if we consider the development of real-time systems, where meeting real-time performance requirements is essential, we can observe that performance is a primary design concern which permeates the development process and the resulting system as a whole. Consequently, performance must be designed into the system and strictly controlled throughout its construction—it cannot be addressed locally or "added" as an afterthought.

However, adopting the real-time system development process for developing general-purpose systems is clearly not possible. The size, complexity, and the height of the software stack used to build general-purpose software-intensive systems typically dwarfs that of the special-purpose mission- or safety-critical real-time systems. The level of control that can be exerted over individual elements of real-time systems either does not scale, or is not possible at all, in addition to performance requirements being usually much less precise (if any), and not easily expressed in terms of latencies or deadlines.

The productivity of developers during development is also an important aspect. Where development of real-time systems limits flexibility to maintain control over performance (by avoiding any features of modern runtime platforms, such as just-in-time compilation or automatic memory management, unless their performance impact can be sufficiently controlled), development of general-purpose systems limits the control over performance to maintain development flexibility (by promoting the use of sophisticated frameworks and advanced runtime platforms to manage complexity and increase productivity).

Because introducing tight control over performance during development of general-purpose systems is impractical, we believe that we need to help developers to acquire certain level of performance awareness. This entails the ability to observe performance in a systematic way and to act upon these observations, as well as gaining an intuitive understanding of performance-related aspects of the underlying platform. We believe that increased performance awareness will improve our ability to construct performant software-intensive systems without relying on full control over all performance-relevant aspects of the system.

## 1.1 Problem Statement and Goals

In summary, performance is an important aspect of software systems, but unlike correctness, performance is largely "invisible" to developers because it is difficult to observe and reason about. Instead of having developers making design decisions based on intuition and incorrect assumptions about performance, the recommended software engineering practice is to deal with performance only if there is evidence that it is inadequate.

However, like good software design, performance is a result of a process—not an isolated feature. The process relies on developers being able to make good design decisions based on systematic observation and reasoning about performance, which is difficult due to the complexity of the underlying hardware and software execution platforms, and of the software system itself.

Consequently, the general goal of the research presented in this thesis is to make it easier to understand performance aspects of modern execution platforms, to make software running on those platforms more observable, and to make performance “visible” to developers so that it can be managed.

In particular, this thesis provides an overview of related contributions in the fields of computer system performance evaluation and dynamic program analysis. Specific topics include performance testing and performance awareness, performance aspects of modern platforms, construction of dynamic program analyses, and observability of modern managed platforms.

## 1.2 Structure of the Text

The thesis is structured as a commented collection of research papers. Chapter 1 provides a unifying context for the research presented, and Chapter 2 provides an overview of the papers included in the thesis. The overview is split into four topics covering two broader research areas, all in some way connected to observing, evaluating, and analyzing software systems. For each topic, the overview provides additional context for the papers included in the thesis and references related papers published by the author.

The papers making up the collection are included in their original form as Chapters 3–10. Each chapter starts with a full reference to the paper and cover page of the proceedings/journal where the paper has been published.

The conclusion focused primarily on future research directions is given in Chapter 11.

# 2 Overview of Selected Articles

The collection presented in this thesis comprises 8 articles, published in international scientific journals and in proceedings of international peer-reviewed conferences. The articles were selected to provide a representative (not exhaustive) overview of the author's research activities and achievements in the last 5 years.

Topically, the articles span two broader, but related, research areas. The first area comprises topics related to software performance, and represents work conducted mainly at the Department of Distributed and Dependable Systems of Charles University in Prague, Czech Republic. The second area comprises topics related to dynamic program analysis on managed platforms, and represents work conducted mainly at (or in cooperation with) the Faculty of Informatics of Università della Svizzera italiana in Lugano, Switzerland.

The recurring and unifying theme in the selected articles concerns observation, analysis and understanding of complex software systems executing on modern platforms. In all cases, the ultimate goal is to provide software developers and operators with methods and means to better understand the behavior of a software system at runtime, and thus contribute to making well-founded decisions during software development and operation.

Each article corresponds to one chapter of the thesis as follows:

**[Ch.3]** L. Bulej, T. Bureš, V. Horký, J. Kotrč, L. Marek, T. Trojánek, and P. Tůma: **"Unit Testing Performance with Stochastic Performance Logic"**. In *Automated Software Engineering* 24.1 (2017), pp. 139–187. ISSN: 0928-8910, 1573-7535. DOI: 10.1007/s10515-015-0188-0

**[Ch.4]** A. Podzimek, L. Bulej, L. Y. Chen, W. Binder, and P. Tůma: **"Robust Partial-Load Experiments with Showstopper"**. In *Future Generation Computer Systems* 64 (2016), pp. 15–38. ISSN: 0167-739X, 1872-7115. DOI: 10.1016/j.future.2016.04.020

**[Ch.5]** P. Libič, L. Bulej, V. Horký, and P. Tůma: **"On the Limits of Modeling Generational Garbage Collector Performance"**. In *Proc. 5th ACM/SPEC Intl. Conf. on Performance Engineering (ICPE)*. ACM, 2014, pp. 15–26. DOI: 10.1145/2568088.2568097

**[Ch.6]** Y. Zheng, L. Bulej, and W. Binder: **"An Empirical Study on Deoptimization in the Graal Compiler"**. In *Proc. 31st European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 30:1–30:30. DOI: 10.4230/LIPIcs.ECOOP.2017.30

**[Ch.7]** A. Sarimbekov, L. Stadler, L. Bulej, A. Sewe, A. Podzimek, Y. Zheng, and W. Binder: **“Workload Characterization of JVM Languages”**. In *Software: Practice and Experience* 46.8 (2016), pp. 1053–1089. ISSN: 0038-0644, 1097-024X. DOI: 10.1002/spe.2337

**[Ch.8]** D. Ansaloni, S. Kell, Y. Zheng, L. Bulej, W. Binder, and P. Tůma: **“Enabling Modularity and Reuse in Dynamic Program Analysis Tools for the Java Virtual Machine”**. In *Proc. 27th European Conference on Object-Oriented Programming (ECOOP)*. LNCS 7920. Springer, 2013, pp. 352–377. DOI: 10.1007/978-3-642-39038-8_15

**[Ch.9]** Y. Zheng, S. Kell, L. Bulej, H. Sun, and W. Binder: **“Comprehensive Multi-Platform Dynamic Program Analysis for Java and Android”**. In *IEEE Software* 33.4 (2016), pp. 55–63. ISSN: 0740-7459 1937-4194. DOI: 10.1109/MS.2015.151

**[Ch.10]** Y. Zheng, L. Bulej, and W. Binder: **“Accurate Profiling in the Presence of Dynamic Compilation”**. In *Proc. 30th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2015, pp. 433–450. DOI: 10.1145/2814270.2814281

The article in Chapter 10 received a *Distinguished Paper Award* as well as an endorsement from the Artifact Evaluation Committee for having submitted an easy-to-use, well-documented, consistent, and complete artifact at the OOPSLA 2015 conference, one of the top publication venues (CORE A*) in the area of programming languages and software systems. The journal article in Chapter 4 is an extended version of an article which received the *Best Paper Runner-Up Award* at the CCGRID 2015 conference, a premier venue (CORE A) in the fields of cluster, cloud and grid computing. The article in Chapter 5 received the *Best Research Paper Award* at the ICPE 2014 conference, a selective venue in the field of performance engineering.

The article in Chapter 9 has been published in the *IEEE Software* magazine. Even though the magazine is not strictly a scientific journal, it aims to be the best source of reliable and useful information for leading software practitioners and its technical articles are peer-reviewed to ensure they offer practical and reliable ideas and techniques to a broad audience of readers. This publication was included in this thesis to demonstrate a broader perspective and applicability of the research work. The article itself summarizes results from peer-reviewed articles published at the GPCE 2013 and MODULARITY 2015 conferences, and includes additional case studies.

In the following sections we provide a short overview of each of the topics and put the articles included in this thesis into a broader research context, with references to other articles that contribute to a particular topic.

## 2.1 Performance Testing and Performance Awareness

This topic represents the tip of a long-term research direction focused on automating discovery of performance problems in software systems. This research has been ongoing for almost 15 years, evolving from the original concept of *regression benchmarking* to the more general concepts of *performance testing* and *performance awareness*.

Inspired by our work on performance evaluation of CORBA middleware within the scope of a project with a major industrial partner (Borland International, Inc.), we proposed regression benchmarking as a method for detecting performance regressions during middleware development [15, 26, 16]. The general idea behind regression benchmarking is to execute a set of benchmarks with each version of a particular software system, and analyze the results to identify potential performance regressions between consecutive versions. While benchmarks have been commonly used to evaluate performance of software systems, the important distinction was the requirement for full automation of the whole process, including analysis of the results, which was intended to enable incorporating regression benchmarking into software development practice.

Our effort to automate regression benchmarking uncovered a great deal of obstacles and challenges, including among others the design of a fully automated and resilient environment for running benchmarks and performance tests [28, 30], detecting performance changes when faced with non-determinism and random biases in performance data [29, 27], or efficient use of resources when collecting performance data from benchmarking experiments [26, 31]. Other researchers have stressed the need for automated approaches to discovering performance regressions [21, 22], or attempted to tackle the problem of identifying the actual code changes that caused a performance regression [23].

Some of the challenges remain open to this day and new challenges have surfaced, reflecting changes in software development practice. Faced with increasing complexity of software systems, modern software development has embraced software testing to ensure that software works as expected. Software undergoes testing at different levels (with different goals and different stakeholders), and is commonly structured into a test pyramid [19] with a foundation made of many low-level unit tests which ensure that the basic software building blocks can be relied upon. An important secondary benefit of unit testing is that it makes it usually easier to adapt software in response to changing requirements, because developers are less afraid to make changes if they know that low-level requirements have been captured in unit test code. Even more importantly though, unit testing creates a demand for better design which is more amenable to changes—covering substantial portion of software with unit tests requires the software to be testable, which in turn requires a design that is more flexible and less coupled.

In this context, we can treat regression benchmarking as a form of software testing focusing on performance instead of functional correctness. However, while both kinds of testing fall under a common umbrella, there are significant differences that provide only limited room for analogy.

In both cases, the tests need to be written by developers, but the execution and evaluation of functional unit tests can be easily automated and integrated into development process, because the binary outcome of a functional unit test is determined by the test itself. This simplicity and the proven benefits—both primary and secondary—are the major factors in the widespread adoption of unit testing as a best practice in software development.

In contrast, performance tests are potentially more difficult to construct and execute, and their results are much more difficult to evaluate automatically, because the test condition is not evaluated by the test code. Instead, evaluating the outcome of performance tests requires analyzing the data collected during execution of the test workload. To avoid misleading results due to interference, the test workload needs to be run multiple times, necessitating probabilistic rather than deterministic evaluation. Yet automating statistical analysis and hypothesis testing on such data is difficult, because the underlying data distributions are typically unknown, often long-tailed and/or multi-modal, with conditional variance coming from multiple sources.

None of these challenges exist in functional unit testing, but they are inherent to performance testing, and need to be addressed before we can make performance testing similarly easy to deploy and adopt as functional testing. This brings us to the article included in Chapter 3, which presents a performance testing framework that addresses three major challenges associated with unit testing performance.

The very first challenge encountered when constructing a performance unit test is the specification of performance requirements, i.e., what performance is expected from the system. Any test condition very much depends on the test scope. High-level end-to-end performance requirements can be often naturally expressed using absolute time limits, but these are much less practical for low-level performance unit tests that typically deal with performance of individual methods. If the goal of performance unit testing is to detect small (10% or less) changes in performance, the timing bounds need to be very tight and independent of the underlying platform. This is very difficult to achieve with fixed time limits, because not only is it difficult to determine how fast a particular method should execute, but it is also difficult to scale the limits to account for tests executing on different platforms.

To address this challenge, our approach relies on Stochastic Performance Logic, a mathematical formalism for expressing performance requirements either in absolute terms, or as relations between performance of multiple methods. Test conditions in form of SPL formulas can be attached to individual methods, which allows the developer to express performance-related assertions such as “future modifications of method m() must not introduce performance degradation greater than 5% with respect to this version.”

The second challenge is related to the construction of performance tests. Because performance unit tests are essentially micro-benchmarks, their construction is prone to certain design pitfalls (typically giving the compiler the option of optimizing away the code that is to be exercised) that can produce misleading results. To mitigate the risk of flawed construction or test execution, our framework provides support for implementing performance tests which are automatically executed and evaluated—the developer only needs to provide code of the actual workload of interest.

The third challenge is related to test execution and data collection. While functional unit tests can be executed in parallel, doing the same with performance unit tests would produce invalid results due to interference between workloads. Even when executed on a dedicated machine, various aspects of the operating system (process memory layout) or the virtual machine (just-in-time compilation, garbage collection) can interfere with the measurement or cause a random bias in the results obtained from a particular test execution. To provide sufficiently representative measurements for automatic evaluation, the testing framework collects measurements from multiple test executions. Besides increasing test execution time, this also requires the test conditions to be evaluated probabilistically.

The SPL design reflects this requirement in that SPL formulas are interpreted using statistical hypothesis testing on the measured data. We provide multiple interpretations with different requirements on the measurement procedure, and allow the developer to control the trade-off between test sensitivity, measurement cost, and false alarm rate through adjustments to the test significance level.

In the context of performance testing, the SPL formalism can be also used to document performance of a particular method implementation, or to document performance expected from the environment or third party code. This leads us to the broader concept of *performance awareness*, which can be understood as the ability to evaluate and reason about performance of a software system and act in response to changes in its performance. While the performance testing framework presented in the article is intended to enable increased performance awareness among developers [9], we need not limit ourselves to human actors—performance awareness may be an ability desired in software systems.

We have explored this research direction within the scope of the ASCENS project [62, 61], which focused on developing a coherent, integrated set of methods and tools to build software for ensembles of autonomic components. In particular, we applied the SPL formalism to support performance awareness and adaptive deployment in component-based systems [12, 11, 10].

The article included in Chapter 3 combines and extends peer-reviewed material published at international conferences to provide a comprehensive presentation of our performance testing framework. Specifically, in addition to the basic definitions of the SPL formalism and the basic SPL interpretations initially published in [14], the article introduces new SPL interpretations that consider measurements from multiple runs, and extends experimental evaluation from [24].

## 2.2 Performance Aspects of Modern Platforms

In addition to capturing and testing performance assumptions, increasing performance awareness also relies on understanding of performance-related aspects of modern execution platforms. These include both virtualized execution environments found in data centers and virtual machine platforms with managed memory targeted by modern programming languages. Articles included in Chapters 4–7 deal with various aspects of modern platforms and provide basis for better understanding of their behavior with respect to performance.

Specific aspects include impact of partial load on performance and energy efficiency of different virtualization solutions (Chapter 4), performance impact of garbage collection (Chapter 5) and deoptimization (Chapter 6) on managed platforms, as well as the effects of workloads resulting from programs written in different languages targeting a single managed execution platform (Chapter 7).

## Performance and Power Efficiency in Virtualized Environment

Historically, the goal of performance evaluation experiments was to determine the peak performance a system can achieve at a particular task, with all resources at its disposal. However, computer systems rarely run at 100% utilization—they are often provisioned to handle bursts of workloads, at which time they work at full utilization, but most of the time they consume energy while idling or working at low utilization. That would not be a problem for a few servers, but large concentration of underutilized machines in data centers has made power efficiency an important aspects of large-scale computing.

To increase utilization of servers (and thus improve power efficiency), it is necessary to colocate multiple workloads on the same physical hardware, ideally in such a way that the peak utilization periods for different workloads do not overlap. While it was always possible to run multiple workloads on a single machine in the form of multiple processes, this was not really feasible in multi-tenant environments—such as data centers—due to security and many other logistical concerns. It was the virtualization and its widespread adoption which really enabled colocating workloads from multiple tenants on a single physical machine, thus increasing its utilization.

Even though virtualization potentially[1] solves many of the security and logistical problems, the sharing of the physical machines causes the colocated workloads to interact in numerous ways that influence their performance. The multitude of possible resource interactions are difficult to capture comprehensively, and the effects of the interactions are difficult to predict. Research into methods aiming at reducing performance interference and increasing power efficiency in workload colocation scenarios therefore requires certain amount of abstraction to make the problems tractable. Yet to avoid a huge gap between theoretical assumptions and real system behavior, it is also necessary to evaluate the impact of those abstractions through experimental evaluation.

However, experimental evaluation in this context involves colocated virtual machines with varying partial utilization levels, which is a bad match for the classic performance evaluation techniques. Moreover, evaluating resource interactions in realistic conditions is an expensive undertaking. Resource utilization depends on workload type in complex ways and even if we can control the workload, it is not always clear what workload intensity to use to achieve a particular resource utilization.

Similar to Section 2.1, the article included in Chapter 4 constitutes a comprehensive presentation of our work on experimental evaluation of partially utilized systems—it combines and extends previously published conference material, adding more depth to existing results, and presenting entirely new experiments.

Specifically, the article presents Showstopper, a tool that addresses some of these challenges and enables systematic experimental evaluation with varying partial utilization of shared resources. Given arbitrary concurrent workloads, Showstopper accurately controls the utilization of a selected resource (processor utilization in our case) by controlling the intensity of the workloads. This removes the need for implementing workload generation harnesses with configurable workload intensity, and makes it possible to examine system

[1]Assuming that the underlying hardware is fundamentally secure and not susceptible, e.g., to recent timing attacks (Meltdown, Spectre) on speculative features of modern CPUs.

behavior at a particular utilization directly. The desired resource utilization can be either constant, or follow a synthetic or recorded utilization trace. In addition to the basic concepts and high-level architecture of Showstopper presented in [44, 47], the article provides a detailed description of the architecture and a wider range of building blocks that comprise the modular control algorithm at the heart of Showstopper. Besides tool improvements in form of support for conducting measurements inside Linux LXC containers in addition to physical machines, the article also presents a quantitative evaluation of the accuracy of different configurations of the load-control algorithm.

To demonstrate how Showstopper can be used to explore the relationship between processor utilization and application throughput, the article presents the results of a study which illustrates the complex nature of this relationship, which is platform dependent, often non-linear, and in some cases even non-monotonic. In extension of prior work on transforming load traces to throughput measurements [44], the article introduces fast-forwarding of resource utilization traces to enable replaying traces captured on real systems at a faster pace. Subsequent evaluation of the effect of fast-forwarding on the accuracy of application throughput measurements shows that in most cases, fast-forwarding a trace does not have a significant impact on the average system throughput, which enables obtaining measurements at a fraction of the original trace duration.

Finally, the article presents the results of an extensive study of the impact of CPU pinning (restricting workloads to specific processors) on the performance and energy efficiency for pairs of colocated workloads. The experiments with different CPU pinning configurations at varying levels of background load expose a trade-off between workload isolation and overall system performance, but more importantly, show that the performance increase due to pinning configuration at certain background loads increases the overall energy efficiency of the system.

The study of the impact of CPU pinning on energy efficiency was originally presented in [45] and received the *Best Paper Runner-Up Award* at the International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2015), a premier (CORE A) venue in the field of cluster, grid, and cloud computing. The article complements the original results with a study of the overhead observed in the different colocation solutions (KVM virtual machines and LXC containers) at partial loads.

## Impact of Garbage Collector on Performance

Garbage collector (GC) is an essential component of managed runtime platforms and plays a major role in the overall system performance. Developing software to run on managed platforms with automatic memory management generally increases developers' productivity and lowers the rate of memory-related programming errors. In contrast though, performance engineers have difficulties capturing the overhead of garbage collection in their performance models, because it is decoupled from program code and is asynchronous with respect to program execution. As a result, garbage collection is often modeled as a constant factor for which a model needs to be calibrated.

Our research on garbage collector (GC) performance modeling resulted from our work in the scope of the Q-ImPrESS project, during which we focused on modeling quality attributes in service-oriented systems. Specifically, our task was to quantify how the sharing of various hardware and software resources among components making up such systems impacts their quality attributes [3]. Because enterprise systems often run on the Oracle Java Virtual Machine (JVM) platform, the garbage-collected heap was one of the resources the sharing of which had to be considered and investigated in addition to other resources, such as processors and their caches, system memory, and file systems.

We therefore investigated the issues related to modeling GC overhead in [37], focusing on whether the overhead warrants explicit attention in service performance modeling, whether the overhead can be captured as a calibrated model parameter, and whether it depends on external factors that could be captured in performance models. The results have shown that anomalous GC overhead can account for tens of percent of execution time, indicating that significant differences between modeled and observed performance can occur due to the GC overhead. In addition, besides memory requirements (GC overhead is sensitive to low-memory conditions), we identified the allocation speed and the lifetimes of allocated objects as prominent factors that need to be known if the GC overhead is to be modeled at all—even using a generic model that does not rely on too many implementation details.

There are many GC implementations, but none that would work best for all workloads in all scenarios. Platforms such as Oracle HotSpot JVM actually provide multiple GC implementations and allow the operator to choose a particular GC and tune it to the workload at hand. However, due to the complex nature of interactions between the application and the GC, this is a trial-and-error process for which there are only rough guidelines. The situation is even worse if we take into account developers, who are expected to treat the GC as a black box, because the implementation is complex and can change between platform implementations. Developers feel (or are made) responsible for performance, but lack the means to relate application-level performance and allocation behavior to GC performance.

The article included in Chapter 5 therefore investigates GC performance from the perspective of a developer with basic knowledge knowledge of GC principles (but not the internals of a particular GC implementation). The goal is to determine whether it is possible to relate application-level performance to GC performance and vice-versa based on inputs characterizing application's object allocation and retention behavior.

To this end, we define simplified models of a one-generation and two-generation GC—models that a developer could reasonably form based on available information—and compare the performance behavior of a real GC implementation with the simplified models implemented in a simulator. We evaluate the prediction accuracy of these models on a variety of workloads, and perform sensitivity analysis with respect to the input describing the application workload.

We show that given an almost-complete information about application behavior in form of allocation traces with object sizes, lifetimes, and reference updates, the simplified GC model can fairly accurately predict frequency of minor collections in a two-generation GC. The prediction quality for minor collections remains stable across workloads, and across inputs ranging from full traces to probabilistic distributions of object sizes and lifetimes.

However, the prediction of the frequency of full collections turns out to be mediocre (even when using an allocation trace with all information as simulator input), and gradually deteriorates when using less accurate (summarized) description of application's allocation behavior. Ultimately, the prediction quality reflects the ability of the GC model to evaluate the GC triggering conditions, the details of which are either implementation-specific or go beyond what we consider implementation-agnostic GC principles captured in the simplified models. We show that from the perspective of a developer, the GC implementation found in an industrially relevant real-world managed platform behaves in a very non-linear fashion, making it very difficult to relate workload changes to changes in GC performance.

While investigating the causes of such behavior, we point out drawbacks that plague garbage collection evaluation methods targeting GC implementations on real-world platforms such as the Oracle JVM. Specifically, when tracing application's object allocation and retention behavior using bytecode instrumentation techniques, the program instrumentation interferes with compiler analyses and optimizations such as inlining, code motion, and scalar replacement. In an unobserved program, the scalar replacement optimization causes certain objects (those that do not escape the scope of a compilation unit, i.e., a method with inlined callees) to be allocated on stack. This in turn avoids having to garbage-collect such objects from the heap later, reducing GC overhead.

By perturbing the compiler optimizations, the instrumentation used to observe application's allocation behavior causes all objects to be allocated on the heap, resulting in an inaccurate allocation trace. Given the non-linear nature of real-world GC, such a trace cannot be expected to allow making accurate predictions of GC behavior. We therefore show that our ability to model real-world GC performance hinges on our ability to obtain accurate allocation profiles, and that we need better approaches to observing application behavior on managed platform.

The article included in Chapter 5 received the *Best Research Paper Award* at the International Conference on Performance Engineering (ICPE 2014). The problem of perturbing compiler optimization when observing application behavior was addressed in further research and published in an article included in Chapter 10.

## Impact of Deoptimization on Performance

Another essential component of managed-memory execution platforms is a dynamic (just-in-time) compiler which these platforms rely on to achieve high performance. On these platforms, programs are initially executed and profiled by an interpreter, and frequently executed methods are compiled into machine code by a dynamic optimizing compiler (or a hierarchy of compilers) to speed up program execution. Because the effects of dynamic compilation accumulate over time, the goal is to speed up the program as soon as possible, but without slowing it down by the compilation work. By making a program run faster, the dynamic compiler frees up computational resources which can be used to perform more optimizations to achieve higher performance.

Unlike a classic compiler which compiles source to machine code statically and can generally perform only optimizations that are provably correct, the dynamic compiler can produce machine code based on assumptions about program behavior that can be checked at runtime. If a certain assumption about program behavior turns out to be wrong, the affected code can be recompiled to reflect the new behavior. This allows the dynamic compiler to safely perform aggressive and speculative optimizations which can help optimize away a significant portion of the abstraction associated with high-level object-oriented languages.

Ideally, speculative optimizations will always turn out to be right and provide performance gains that outweigh their cost in terms of compilation time. In reality, some speculations will fail and trigger *deoptimization*. Besides switching to interpreted (or otherwise less optimized) execution mode, deoptimization may also trigger recompilation of the affected code, thus wasting previous compilation work and adding to the overall cost of compilation. Even though deoptimization is at the heart of many techniques that are commonly used to make managed platforms fast, many qualitative and quantitative aspects of deoptimization have not been well studied in the literature. How often does it happen and for what reason? How much compilation effort is wasted? What trade-offs can be made and how does it affect performance?

To answer some of these questions, the article included in Chapter 6 presents a study of deoptimization behavior of code compiled by the Graal [2] dynamic compiler and the behavior of the VM runtime in response to the deoptimizations. Graal is a new compiler which can be optionally used within the Oracle HotSpot JVM, and which is known to perform more aggressive and speculative optimizations than the classic C2 server compiler currently used by the HotSpot JVM.

In the article, we characterize the deoptimization causes in the code produced by Graal for the DaCapo [6], ScalaBench [53], and Octane[3] benchmark suites, and show that only a small fraction (2%) of deoptimization sites is actually triggered, and that most of those (98%) invalidate the compiled code and reprofile it to avoid triggering the deoptimization when the code is compiled again later. We also evaluate the trade-offs made by Graal in its default deoptimization strategy, and show that by avoiding the conservative strategy provided by the HotSpot VM runtime, Graal gains better startup performance. Finally, we show that certain workload-specific level of tolerance to deoptimizations can provide performance benefits, and demonstrate the impact of different tolerances levels on the performance of selected workloads.

## Effects of Workloads Produced by JVM Languages

Different parts of a software system often call for different levels of performance and developer productivity. Ideally, the core application parts would benefit from the traits of a statically-typed language, data management would benefit from the expressiveness of a suitable domain-specific language, and the user interface and would benefit from the flexibility of a dynamically-typed language. However, such an approach was generally difficult to adopt in the past, and developers had to choose a single programming language for the entire project so that it fit the most common/important tasks, and they had to put

[2]The Graal Project, http://openjdk.java.net/projects/graal/
[3]Octane 2.0 JavaScript Benchmark, https://developers.google.com/octane/

up with inconvenience of the language for other tasks.

Modern managed-memory platforms such as the Oracle Java Virtual Machine (JVM) and the Microsoft .NET Common Language Runtime (CLR) have changed this. In addition to automatic memory management, high-performance just-in-time compilation, and a rich class library, these platforms also enable polyglot programming [43], which allows developers to use different languages best suited for different tasks. Consequently, the JVM and the CLR have become attractive as execution platforms targeted by designers of modern programming language compilers, hoping to benefit from their performance, maturity, and industrial adoption. This is especially true for dynamic programming languages, which trade the raw performance achievable with languages such as C or C++ for increased productivity, ease of maintenance, and other aspects.

However, the optimizations found in these modern runtimes have been mostly tuned with a single language in mind, which is especially true for the JVM and Java. By today's standards, Java (running on the HotSpot JVM) is considered fast, especially in comparison to dynamic object-oriented languages. For other languages, even though running on the JVM provides some immediate benefits, the sought-after benefit of Java-like performance does not materialize automatically by just running on the JVM. Improving performance of dynamic JVM languages requires significant effort, which can be aided by understanding the differences between Java and non-Java workloads running on the JVM.

This can be facilitated by characterization of the workloads originating from other languages than Java executing on the JVM. Workload characterization uses various metrics to capture various qualitative and quantitative aspects of a workload, which can be then used to gain insight into performance-related behavior of the combination of a particular workload and a particular execution platform. Workload characterization can be performed using different tools, but existing approaches often provide only incomplete information or suffer from limited compatibility with standard JVMs. Completeness and accuracy of the profiles are essential for tasks such as workload characterization, and compatibility with standard production JVMs is important to ensure that complex workloads can be executed.

To aid in understanding the differences between programs written in different languages executing on the JVM, and to avoid the aforementioned limitations, the article included in Chapter 7 introduces a new set of dynamic metrics that are sensitive to differences in the workloads resulting from bytecode produced by different JVM languages, along with an easy-to-use toolchain to collect the metrics on a standard JVM. We apply the toolchain to applications written in six JVM languages (Java, Scala, Clojure, Jython, JRuby, and JavaScript) and discuss the findings. Given the importance of inlining for performance (inlining increases the scope for many other optimizations), we also analyze the inlining behavior of the HotSpot JVM when executing bytecode originating from different JVM languages. We identify several traits in the non-Java workloads that represent potential opportunities for optimization.

The article included in Chapter 7 combines and extends previously published peer-reviewed articles [49, 50] to provide a comprehensive presentation of our workload characterization approach for JVM languages. To ensure compatibility with standard production JVM, the tool chain relies on our DiSL instrumentation framework and the ability to observe program behavior on modern platforms, which we discuss in Sections 2.3 and 2.4.

## 2.3 Construction of Dynamic Program Analysis Tools

This section marks a transition from the general area of software performance to the area of dynamic program analysis. In particular, this section deals with construction of dynamic program analysis (DPA) tools for use with programs executing on modern managed platforms such as the Java Virtual Machine.

DPA tools are programs that observe the behavior of another program (the base program) while it is executing, and report on the properties of that particular execution. This provides developers and software engineers with insight into the dynamics and behavior of software systems at runtime, which allows them to better understand, debug, optimize, or refactor such systems. Due to the scale and complexity of modern software, the insight provided by DPA tools is difficult to obtain by reading the source code, and is often beyond reach of static program analysis tools. However, constructing DPA tools is unduly difficult, error-prone, and requires developers with considerable expertise. Our contributions in this area deal with some of the challenges associated with construction of DPA tools, and the article included in Chapter 8 represents one such contribution.

### Instrumentation Framework for Dynamic Analysis

The first challenge encountered by DPA tool developers is the fact that to observe the execution of a program, its code needs to be rewritten so that it captures occurrences of relevant events during program execution. This can be done in different ways in different contexts, but in the case of managed platforms such as the Java Virtual Machine, most DPA tools use *bytecode instrumentation* to modify the base program's code. Despite the existence of various libraries that raise the level of abstraction and relieve developers from handling the lowest-level details, implementing the program-rewriting part of a DPA tool is error-prone, requires significant developer expertise, and results in code that is verbose, complex, and difficult to maintain. Even though higher-level code transformation frameworks exist, they were designed for more general code transformation and optimization tasks, which does not necessarily help with development of observation-only code transformations.

To address this challenge, we have developed DiSL [41, 40, 63], a domain-specific language and framework designed specifically for instrumentation-based dynamic program analysis. DiSL is unique among the existing instrumentation frameworks because it succeeds in reconciling high-level abstractions, flexibility, and efficiency. The allow expressing instrumentations in a concise matter, DiSL adopts the high-level pointcut/advice model found in aspect-oriented programming (AOP), rather than relying on low-level bytecode manipulation constructs. This programming model is easily adopted by developers, and leads to compact and readable code. To retain the flexibility of low-level bytecode manipulation libraries, the DiSL framework provides an open join-point model that allows instrumenting any bytecode sequence, coupled with techniques that extend the instrumentation coverage to any method with bytecode representation. DiSL also provides specialized features that enable implementation of efficient instrumentations, without incurring the overhead caused by using very high-level, but costly, features in AOP frameworks such as AspectJ. This allows achieving performance and efficiency that is on par with tools implemented using low-level bytecode manipulation libraries.

DiSL relieves DPA tool developers from having to deal with too many low-level details associated with program instrumentation, allowing them to instead focus on the dynamic analysis itself. By lowering the barrier to creating a new DPA tool, DiSL enables rapid prototyping of novel dynamic analyses. The software-engineering benefits of DiSL were demonstrated in a small-scale controlled experiment [52], the DiSL framework served as the basis for a toolchain used in two extensive JVM workload characterization studies [51, 35] (c.f. Section 2.2 and Chapter 7), and was used to collect application allocation behavior traces in our research on modeling Java garbage collector performance (c.f. Section 2.2 and Chapter 5). DiSL was accepted by the Technology Council of the OW2 Consortium as an incubator project, and is available[4] as open-source software. DiSL was also accepted into the SPEC Research Group repository[5] of peer-reviewed tools for quantitative system evaluation and analysis.

## Modular Composition of Dynamic Analyses

The second challenge associated with construction of DPA tools is that high-level requirements must be translated into code reacting to low-level execution events. For example, when creating a simple context-sensitive memory profiler which counts the allocated bytes in each method and provides a total for individual call chains, the developer needs to express the analysis in terms of method entry and method exit events, as well as a number of object allocation events corresponding to various low-level object allocation mechanisms. In addition, the developer is responsible for creating instrumentation code that will be inserted into the base program code to reify those events for the analysis.

In general, instrumentation frameworks (including DiSL) only provide mechanisms to intercept various control-flow events. Everything else, including abstraction and aggregation over the captured events, is the responsibility of the developer. This means that even for dynamic analyses that consume a common subset of events to implement similar abstractions, the authors of different tools have to repeat the (considerable) effort necessary for bridging the gap between the high-level tool requirements and the low-level instrumentation. Arguably, the prevailing approach based on *control-flow interposition* is too low-level and does not allow dealing with recurring concerns found in dynamic analyses at a higher level of abstraction.

To address this problem, the article included in Chapter 8 proposes an alternative approach based on *state-oriented* decomposition, which enables more succinct description of different dynamic analyses. The key idea is to decompose the high-level analysis' requirements in terms of structures holding the accumulated state of an analysis, and the semantics with which these structures evolve in response to consumed events.

[4]http://disl.ow2.org/
[5]http://research.spec.org/projects/tools.html

For example, the state of most profiling tools will consist of structures that represent event counts, often associated with structures representing a context for the counters. The state of each of those structures evolves differently. While counters are simply incremented, another structure keeps track of the current context, and yet another structure maps the current context to the context-specific counter instance.

The state-oriented decomposition allows extracting the commonality found in different dynamic analyses into a library of reusable components—generic data structures and state transformers. A suitably described combination of such library components can be then used to satisfy recurring dynamic analysis requirements.

The FRANC framework for composition of dynamic analyses—the second major contribution of the article included in Chapter 8—demonstrates the feasibility of the state-oriented decomposition approach. The API provided by FRANC lifts dynamic analysis construction from the level of instrumentation to an event-based publish-subscribe system with convenient reusable abstractions for data collection and aggregation. The key aspects of the API are three interfaces which facilitate the state-oriented decomposition: *instrumentations*, which reify base-program events; *mappers*, which route events to the subset of analysis state to be updated by subsequent events; and *updaters*, which evolve the analysis's output-producing state in response to events. We demonstrate that this factoring can be used to express a wide variety of distinct dynamic analyses and that analysis tools constructed using FRANC offer performance that is generally competitive with tools developed using lower-level frameworks.

## 2.4 Observability on Modern Managed Platforms

The final section of this overview concerns observability of programs executing on modern managed platforms. Dynamic program analysis critically depends on the ability to observe the execution of a program. However, modern managed platforms such as the Oracle JVM or the Google Dalvik VM (Android) do not make this task particularly easy—after all, the goal of these virtual machines is to execute programs at peak performance.

This does not necessarily mean that a modern virtual machine is a complete black box—only some of them, e.g., the Dalvik VM, which was designed for a resource constrained environment. In contrast, the Oracle JVM provides a Java Platform Debugger Architecture (JPDA)—a set of interfaces intended for use by debuggers in development environments running on desktop systems. These interfaces are adequate for debugging purposes, where neither expressiveness nor performance is expected, but their utility for dynamic program analysis is limited precisely by these two factors.

Apart from modifying the host system itself, which is very difficult and results in a non-portable solution, many DPA tools rely on bytecode instrumentation to observe program execution events relevant to a particular analysis. Most DPA tools use a library or a framework to manipulate program bytecode. The JPDA interfaces—specifically the JVM Tooling Interface (JVMTI)—are then used mainly to enable support for on-demand instrumentation at program load time. Our DiSL and FRANC frameworks discussed in Section 2.3 are examples of such frameworks.

However, developing instrumentation is merely the first step in developing a DPA tool.

Depending on the instrumentation framework used, it may be more or less difficult, but developing a high quality DPA tool is still a challenging task. One particular challenge lies in dealing with mutually antagonistic requirements: *isolation*, i.e., not influencing program execution through observation, *coverage*, i.e., observing all relevant events during execution, and *performance*, i.e., minimizing slowdown due to the analysis. The conflicting requirements lead to a classic dilemma, forcing the developer to choose at most two of the three desired properties. And even then, achieving the selected properties requires significant expertise and effort. Like in any physical system, observation is bound to cause certain kinds of perturbation, and the JVM has been shown to be riddled with traps for the unwary [32]. Our research in this area focuses on enabling construction of DPA tools with high level of isolation and coverage. The articles included in Chapters 9 and 10 represent some of our contributions to the state of the art in this area.

## The ShadowVM Analysis Framework

Most DPA tools for the JVM use bytecode instrumentation to capture events related to program execution that are not available through the JVMTI interface. This is the recommended approach. However, even though the JVMTI documentation endorses a separate-process design for the analysis to avoid interfering with the normal execution of the observed program, most tools rely on *internal observation*, a design in which the observed program and the analysis execute in the same address space. A likely explanation of this design choice is that simplicity and performance were the deciding factors—the recommended approach involving a separate process is more complex and incurs additional overhead. These factors certainly played an important role in the single-process design of our DiSL instrumentation framework.

However, the approach based solely on internal observation inevitably leads to problems when we want to expand analysis coverage and observe activity in all executed code, including system libraries (Java Class Library). These libraries provide (often the only) means to perform operations that dynamic analyses need, such as input and output, accessing reflective metadata, or keeping weak references to program objects. When these libraries are instrumented, the library-internal resources become shared between the observed program and the analysis in an uncoordinated fashion. Consequently, even simple instrumentation scenarios can suffer from state corruption, deadlocks, and memory exhaustion [32].

The issues associated with internal observation make the development of high-quality dynamic analyses difficult, and our research in this area addresses some of them. One of the contributions is *ShadowVM* [38], a dynamic program analysis framework which adopts an event-based programming model and realizes dynamic program analysis as a distributed event-processing system.

The base program executes in the *observed VM* and produces events which are consumed by the analysis executing in the *shadow VM*. In the observed VM, instrumentation inserted into the base program acts as an *event producer* and emits events required by the analysis. The event producer programming model adopts the programming model of DiSL, which allows expressing instrumentation using high-level aspect-oriented abstractions. In addition, the framework generates special events that mark the disposal of resources (objects, threads, or the VM itself) and provide analyses with triggers that allow them to clean up internal state or output results.

The analysis code deployed in the shadow VM acts as an *event consumer* and performs computations and analysis state updates in response to events from the observed VM. The analysis developer supplies the instrumentation to generate these events and controls the payload they carry. The payload can include primitive types and object references; the latter are exposed to the analysis as *shadow objects*. These preserve the identity of the objects from the observed VM, and expose reflective metadata mirroring the class hierarchy of the base program. Shadow objects do not mirror the contents of the original objects, but allow attaching and accessing arbitrary analysis state.

Our evaluation of the ShadowVM framework has shown that dynamic analyses can be written using convenient high-level languages and APIs, retaining the feel of a bytecode instrumentation system but achieving higher levels of isolation and coverage. By minimizing interference with the observed program, analyses can observe code in core classes, from the earliest stages of VM execution. Despite the addition of a process separation, the performance of ShadowVM is acceptable for many use cases.

In addition to the original publication at GPCE 2013 [39], this work has been summarized in an *IEEE Software* article included in Chapter 9.

## Dynamic Program Analysis on Android

Even though the Java platform is very popular in the enterprise world, the world of mobile devices has been mostly dominated by the Android platform. This creates demand for software development tools targeting Android, including program analysis tools that provide insight into application behavior, e.g., to allow auditing an application's use of permissions granted by the user. However, the range of approaches used by various security analysis tools (OS and VM modifications, various instrumentation techniques, use of CPU emulator) shows that Android lacks the means to develop DPA tools using a high-level programming model, without resorting to complex, platform-specific implementation.

Android poses a specific challenge to classic approaches to dynamic program analysis based on instrumentation and internal observation in that Android applications (although written in Java) are split into multiple components (with multiple entry points) executing in multiple virtual machines. While with conventional applications it was often reasonable (apart from problems with interference) to colocate the analysis with the executing application, doing the same with Android applications would result in distributing the analysis state across multiple application components. This might not matter with some analyses, but in general, partitioning the analysis state would needlessly complicate the analysis code.

In this context, the ShadowVM approach with the analysis split between an event producing instrumentation colocated with application code, and an event consuming analysis code executing in a dedicated virtual machine is a good match for the specifics of the Android platform. However, the Dalvik Virtual Machine (DVM), which executes most of the application code on Android, lacks certain features that have enabled implementation of the ShadowVM framework on the JVM—most notably a tool interface similar to JVMTI.

To enable development of DPA tools for the DVM, we modified the DVM to provide essential DPA tooling support. This concerns mainly (a) handling of essential events such as class loading and initialization, (b) tracking of virtual machine, thread, and object lifecycle, and (c) exportable object identities, application event notifications, and component communication tracking.

The modifications to the ShadowVM framework and programming model include mainly support for multi-process applications and inter-process communication. An analysis observing an Android application needs to handle events from multiple VM instances. This is enabled by associating the observed events and object identities with a *VM context* provided to the analysis with each delivered event. To enable observation of multi-process applications and their interactions with the wider system, the ShadowVM framework on Android emits special communication events (in addition to the lifecycle events) to which an analysis can subscribe, and which capture the low-level IPC operations that Android applications use for communication and control transfer.

Because Android applications execute in a resource-constrained environment (from the perspective of contemporary server and desktop systems), we execute the analysis VM on a remote system. This avoids competing for resources with the analyzed application and allows using any Java features in the analysis code. The ShadowVM programming model, along with seamless support for both the JVM and the DVM, also enables development of cross-platform analyses—code executing in the analysis VM can react to event notifications from any platform-specific front-end. Evaluation of the framework on several case studies has shown that thanks to the high-level programming model, the framework reduces development effort for many analysis tools, while providing isolation and comprehensive bytecode coverage.

In addition to the original publication at MODULARITY 2015 [56], this work has been summarized in an *IEEE Software* article included in Chapter 9. The ShadowVM framework has been also demonstrated at the conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH 2015) [54], and at the Asian Symposium on Programming Languages and Systems (APLAS 2015) [55].

## Avoiding Perturbations to Compiler Operation

Modern managed platforms rely on tiered compilation and an optimizing dynamic compiler to achieve high performance. Programs executing on these platforms are usually first interpreted (or compiled by a baseline compiler), and frequently executed methods are later compiled by the optimizing dynamic compiler. State-of-the-art dynamic compilers perform optimizations based on profiling information gathered during program execution, and many optimization result in machine code that does not even perform certain operations (method invocations, heap allocations, lock acquisition) present at the level of bytecode.

Many DPA tools for such platforms rely on bytecode instrumentation (often in combination with suitable debugging interfaces) to observe the execution of an application—inserting bytecode that triggers analysis execution into the observed application bytecode. Because dynamic compilation is transparent, the DPA tools are not aware of the compilation, let alone the optimizations performed by the compiler. Similarly, the compiler is not aware of the inserted instrumentation code—it will compile and attempt to optimize the instrumented methods just like any other methods.

This can lead to inaccurate analysis results, because the events reported by the instrumentation code may be correspond to actual operations in the compiled code. For example, if the compiler removes instructions that are being observed but does not remove the associated instrumentation code, the (removed) operations will still be reported to the analysis. Alternatively, because optimization is driven by heuristics, the compiler may decide not to perform some optimizations—the inserted code may have increased the method code size, or may have introduced additional data or control flow dependencies. In this case, the analysis may observe operations that are present in the compiled code even though they would have been removed if only the original (uninstrumented) application code was being compiled.

We have now discussed two different kinds of perturbation caused by observing application execution using bytecode instrumentation. The first—caused by sharing of library-internal resources between an analysis and the observed application—may result in state corruption, deadlocks, or VM crashes, especially for high-coverage analyses. The second—caused by mere presence of the instrumentation code in the application code—may perturb the operation of the dynamic compiler and cause an analysis to report inaccurate or incorrect results.

In general, when using instrumentation to observe program execution, we cannot completely eliminate the observer effect—the perturbation of low-level dynamic metrics (such as hardware or operating system performance counters) cannot be avoided. However, the two kinds of perturbation discussed in this section can be avoided. The ShadowVM framework presented earlier avoids the first kind of perturbation by isolating the analysis and the observed application from each other, thus eliminating sharing of library-internal state. An approach to avoiding perturbation of dynamic compilation is the key contribution of the article included in Chapter 10.

The root of the problem lies in the inability of the dynamic compiler to distinguish between the inserted instrumentation code and the base program code, and due to the inability of the inserted code to adapt to the optimizations performed by the dynamic compiler. The key idea of our approach is to make the compiler aware of the two kinds of code, and treat them differently. For the base program code, the goal is to let the dynamic compiler process it in the usual fashion, making optimization decisions and performing optimizations as if the inserted code was not there. For the inserted code, the goal is to preserve its purpose and semantics by adapting it in response to the optimizations performed by the dynamic compiler on the base program code.

Even though our approach has been implemented in a particular state-of-the-art dynamic compiler [6], it has been formulated for a method-based dynamic compiler using a graph-based intermediate representation (IR) in the Static Single Assignment (SSA)

[6]The Graal Project, http://openjdk.java.net/projects/graal/

form, with optimizations implemented as IR graph transformations. When the dynamic compiler builds the IR of the method being compiled, we identify the boundaries between the base program code and the inserted code, and unlink the inserted code from the base program IR, creating inserted code subgraphs (ICGs) associated with base program nodes. We then let the dynamic compiler work on the base program IR while tracking the operations it performs on the IR graph nodes. Whenever the compiler performs an operation on a node with an associated ICG, we perform a reconciling operation on the corresponding ICG to preserve its semantics throughout the transformations performed by the compiler. When the compiler finishes optimizing the base program IR, we splice the ICGs back into the base program IR—before it is lowered to machine-code level.

An important aspect of our approach is that it allows the inserted code to query and adapt to the dynamic compiler's decisions. The queries are represented by invocations of special methods, *query intrinsics*, which are recognized and handled by the compiler similarly to normal intrinsics. The query intrinsics represent an API that provides an instrumentation developer with the means to determine both compile-time and runtime compiler decisions, and allows creating an instrumentation that adapts accordingly.

Additional contributions comprise case studies and tools, including an object allocation and lifetime profiler, a callsite polymorphism profiler, and a compiler testing framework. These demonstrate that our approach enables collection of accurate information that faithfully represents the execution of a base program without profiling, and that the approach is applicable in different scenarios.

This work also provides a solution to an issue identified as a major cause of inaccuracy of object allocation profiles used for prediction of garbage collection cycles, discussed in Chapter 5.

The article included in Chapter 10 received the *Distinguished Paper Award* at the international conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015), as well as an endorsement from the Artifact Evaluation Committee for having submitted an easy-to-use, well-documented, consistent, and complete artifact for evaluation.

# 3
# Unit Testing Performance with Stochastic Performance Logic

L. Bulej, T. Bureš, V. Horký, J. Kotrč, L. Marek, T. Trojánek, and P. Tůma: **“Unit Testing Performance with Stochastic Performance Logic”**. In *Automated Software Engineering* 24.1 (2017), pp. 139–187. ISSN: 0928-8910, 1573-7535. DOI: 10.1007/s10515-015-0188-0

Extended version of L. Bulej, T. Bureš, J. Keznikl, A. Koubková, A. Podzimek, and P. Tůma: **“Capturing Performance Assumptions Using Stochastic Performance Logic”**. In *Proc. 3rd ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 2012, pp. 311–322. DOI: 10.1145/2188286.2188345,

and also V. Horký, F. Haas, J. Kotrč, M. Lacina, and P. Tůma: **“Performance Regression Unit Testing: A Case Study”**. In *Proc. 10th European Performance Engineering Workshop*. EPEW. Springer, 2013, pp. 149–163. DOI: 10.1007/978-3-642-40725-3_12.

# 4 Robust Partial-Load Experiments with Showstopper

A. Podzimek, L. Bulej, L. Y. Chen, W. Binder, and P. Tůma: **“Robust Partial-Load Experiments with Showstopper”**. In *Future Generation Computer Systems* 64 (2016), pp. 15–38. ISSN: 0167-739X, 1872-7115. DOI: 10.1016/j.future.2016.04.020

Extended version of A. Podzimek, L. Bulej, L. Y. Chen, W. Binder, and P. Tůma: **“Analyzing the Impact of CPU Pinning and Partial CPU Loads on Performance and Energy Efficiency”**. In *Proc. 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2015, pp. 1–10. DOI: 10.1109/CCGrid.2015.164.

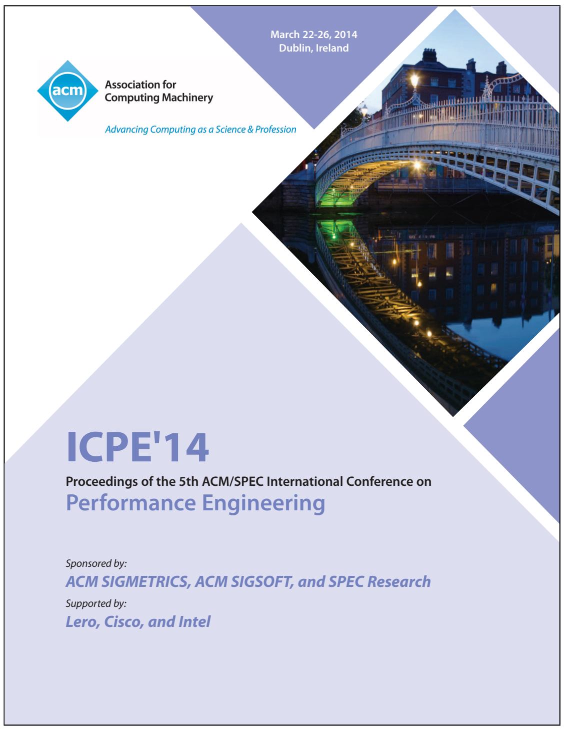
Best Paper Runner-Up Award

5

# On the Limits of Modeling Generational Garbage Collector Performance

P. Libič, L. Bulej, V. Horký, and P. Tůma: **“On the Limits of Modeling Generational Garbage Collector Performance”**. In *Proc. 5th ACM/SPEC Intl. Conf. on Performance Engineering (ICPE)*. ACM, 2014, pp. 15–26. DOI: 10.1145/2568088.2568097

Best Research Paper Award



March 22-26, 2014
Dublin, Ireland

Association for Computing Machinery

*Advancing Computing as a Science & Profession*

ICPE'14

Proceedings of the 5th ACM/SPEC International Conference on

Performance Engineering

*Sponsored by:*

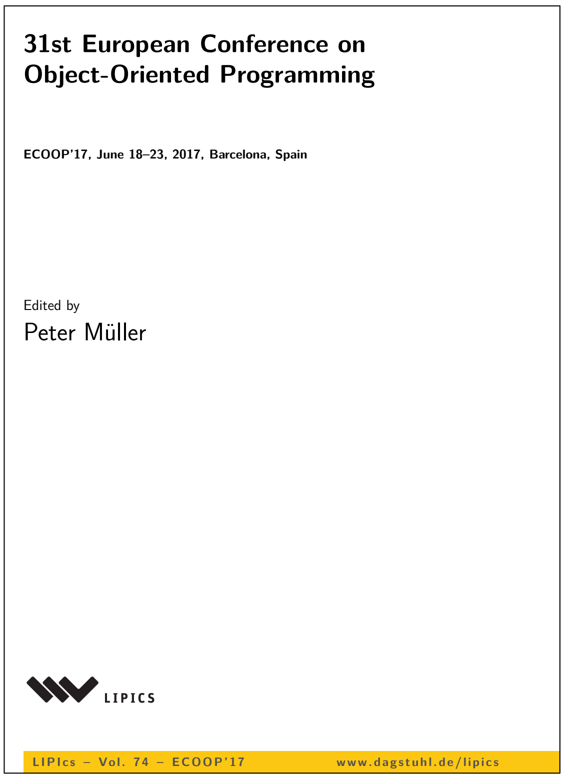
***ACM SIGMETRICS, ACM SIGSOFT, and SPEC Research***

*Supported by:*

***Lero, Cisco, and Intel***

# 6
# An Empirical Study on Deoptimization in the Graal Compiler

Y. Zheng, L. Bulej, and W. Binder: **“An Empirical Study on Deoptimization in the Graal Compiler”**. In *Proc. 31st European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 30:1–30:30. DOI: 10.4230/LIPIcs.ECOOP.2017.30



31st European Conference on
Object-Oriented Programming

ECOOP'17, June 18–23, 2017, Barcelona, Spain

Edited by
Peter Müller

LIPICS

LIPIcs – Vol. 74 – ECOOP'17 www.dagstuhl.de/lipics

# 7 Workload Characterization of JVM Languages

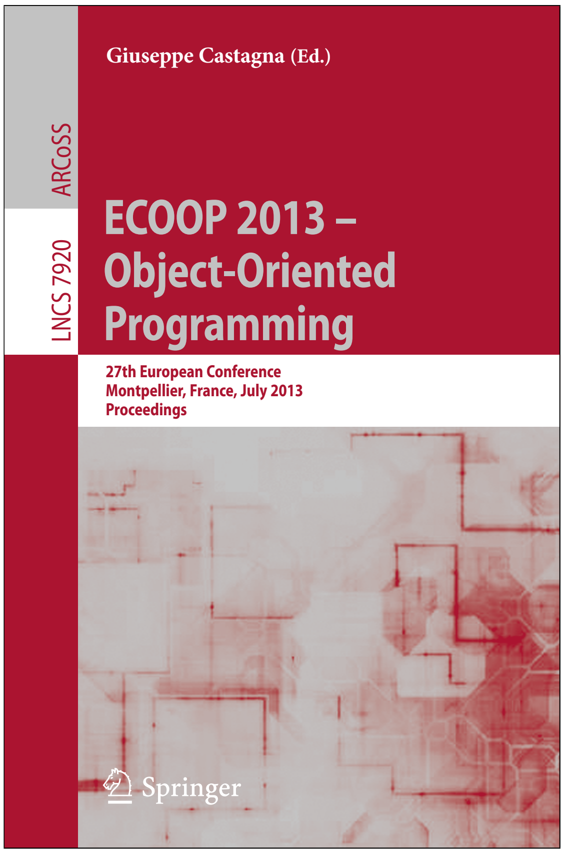
A. Sarimbekov, L. Stadler, L. Bulej, A. Sewe, A. Podzimek, Y. Zheng, and W. Binder: **“Workload Characterization of JVM Languages”**. In *Software: Practice and Experience* 46.8 (2016), pp. 1053–1089. ISSN: 0038-0644, 1097-024X. DOI: 10.1002/spe.2337

Extended version of A. Sarimbekov, A. Podzimek, L. Bulej, Y. Zheng, N. Ricci, and W. Binder: **“Characteristics of Dynamic JVM Languages”**. In *Proc. 7th ACM Workshop on Virtual Machines and Intermediate Languages (VMIL)*. ACM, 2013, pp. 11–20. DOI: 10.1145/2542142.2542144,

and also A. Sarimbekov, A. Sewe, S. Kell, Y. Zheng, W. Binder, L. Bulej, and D. Ansaloni: **“A Comprehensive Toolchain for Workload Characterization Across JVM Languages”**. In *Proc. 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. ACM, 2013, pp. 9–16. DOI: 10.1145/2462029.2462033.

# 8
# Enabling Modularity and Re-use in Dynamic Program Analysis Tools for the Java Virtual Machine

D. Ansaloni, S. Kell, Y. Zheng, L. Bulej, W. Binder, and P. Tůma: **"Enabling Modularity and Reuse in Dynamic Program Analysis Tools for the Java Virtual Machine"**. In *Proc. 27th European Conference on Object-Oriented Programming (ECOOP)*. LNCS 7920. Springer, 2013, pp. 352–377. DOI: 10.1007/978-3-642-39038-8_15



# 9
# Comprehensive Multiplatfrom Dynamic Program Analysis for Java and Android

Y. Zheng, S. Kell, L. Bulej, H. Sun, and W. Binder: **“Comprehensive Multi-Platform Dynamic Program Analysis for Java and Android”**. In *IEEE Software* 33.4 (2016), pp. 55–63. ISSN: 0740-7459 1937-4194. DOI: 10.1109/MS.2015.151

Summary of L. Marek, S. Kell, Y. Zheng, L. Bulej, W. Binder, P. Tůma, D. Ansaloni, A. Sarimbekov, and A. Sewe: **“ShadowVM: Robust and Comprehensive Dynamic Program Analysis for the Java Platform”**. In *Proc. 12th International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2013, pp. 105–114. DOI: 10.1145/2517208.2517219,

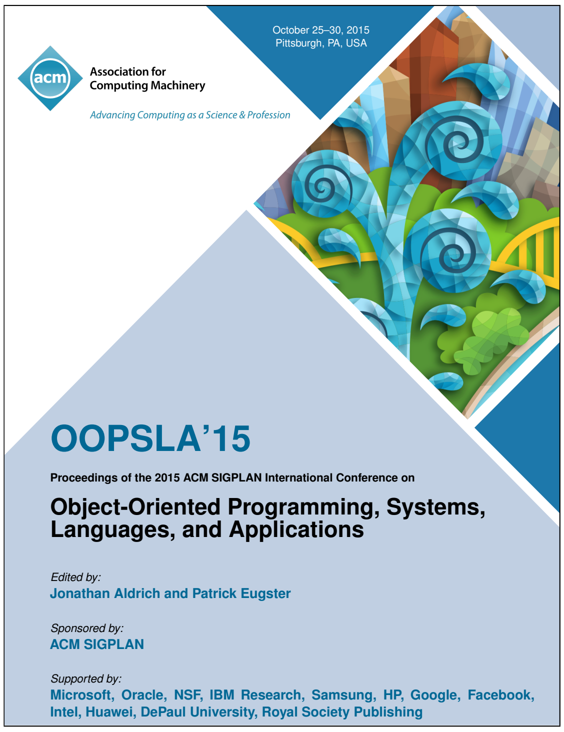
and also of H. Sun, Y. Zheng, L. Bulej, A. Villazón, Z. Qi, P. Tůma, and W. Binder: **“A Programming Model and Framework for Comprehensive Dynamic Analysis on Android”**. In *Proc. 14th International Conference on Modularity (AOSD/MODULARITY)*. ACM, 2015, pp. 133–145. DOI: 10.1145/2724525.2724566.

# 10

# Accurate Profiling in the Presence of Dynamic Compilation

Y. Zheng, L. Bulej, and W. Binder: **“Accurate Profiling in the Presence of Dynamic Compilation”**. In *Proc. 30th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2015, pp. 433–450. DOI: 10.1145/2814270.2814281

Distinguished Paper Award
Evaluated Artifact



October 25–30, 2015
Pittsburgh, PA, USA

Association for Computing Machinery

*Advancing Computing as a Science & Profession*

OOPSLA’15

Proceedings of the 2015 ACM SIGPLAN International Conference on

Object-Oriented Programming, Systems, Languages, and Applications

*Edited by:*
Jonathan Aldrich and Patrick Eugster

*Sponsored by:*
ACM SIGPLAN

*Supported by:*
Microsoft, Oracle, NSF, IBM Research, Samsung, HP, Google, Facebook, Intel, Huawei, DePaul University, Royal Society Publishing

# 11 Conclusion and Future Research

The papers included in this thesis provide an overview of contributions to making performance a first-class concern in development of general-purpose software systems. This relies both on understanding performance-related aspects of the underlying execution platform, but also on the ability to observe and reason about performance-related behavior of a software system.

One obstacle to making performance a first-class concern is related to the economy of software development. Historically, developers relied on advances in hardware to provide more performance to their software. But keeping up with Moore's Law is becoming a significant technical and, more importantly, economical challenge even for the largest processor manufacturers. Software developers therefore cannot expect much more "free" performance from advances in general-purpose processors. In addition, energy consumption has become a visible factor in the cost of computing, providing another incentive for creating more efficient software.

While we believe that the general circumstances are now more in favor of paying attention to software performance during development, adopting a systematic approach to dealing with performance during development is still rather difficult. Instead of a conclusion we therefore offer a view to potential research directions to address some of the technical and methodological challenges.

## 11.1 Performance Testing

In modern software development, testing has made functional aspects software quality visible to developers. Testing drives design and makes refactoring practically viable, enabling software designs to evolve with changing requirements, ultimately improving software quality. If unit testing made software quality and correctness visible by making them testable, we need to make performance visible by making performance requirements and assumptions testable as well.

However, if performance testing is to be at the heart of performance awareness, then efficiency and manageability are the key attributes needed for integration into the existing development processes. We need the developers to adopt the approach and support it by writing performance tests. In return, they will expect careful attention to the time they need to invest in the process, to results being available in a timely fashion, and generally to high accuracy (no false positives) and efficiency over completeness. To this end, progress is required on a number of open issues.

**Limits on expressible constraints.** The necessary condition to make a performance requirement testable is to express it in a machine-readable form that is sufficiently convenient to produce for human developers. Our contribution to this topic includes Stochastic Performance Logic (SPL) [13, 14], which allows expressing such requirements using relative performance of existing code as a reference. However, our approach to formalizing performance requirements using SPL is currently limited to constraints on mean execution time. Other desirable metrics include, e.g., scalability and (algorithmic) complexity, and recently also power consumption, which directly translates to costs (cloud) or battery life (mobile devices). We need to investigate how to extend the SPL formalism to support alternative metrics and statistics other than mean.

**Construction of relevant workloads.** Performance requirements and assumptions need to be tested on workloads that are relevant to system performance. The most relevant workloads are those from real deployment, but to ensure that performance awareness permeates the whole development process, we also need performance requirements or assumptions that are testable even in early development stages. Currently, workloads have to be provided by the developers, which is one of the most time-consuming part of writing a performance test. To make providing relevant workloads less time-consuming, we need methods and tools for capturing workloads from real deployment and for deriving workloads from unit tests or test traces.

**Timely feedback on test results.** By formally describing performance requirements, developers can capture testable performance assumptions about their or third-party code. This can increase awareness of performance that can be expected from various elements of the system, ultimately increasing confidence and correctness of day-to-day decisions that impact performance. However, the feedback on such requirements needs to be readily available and easily accessible to developers, which is at odds with the inherently time-consuming process of collecting and evaluating performance data.

Unlike unit tests, performance tests require more hardware resources to evaluate, because the tests need to be executed repeatedly to account for multiple sources of variance in the measured data. In a realistic setting, a continuous testing infrastructure has (much) less than 24 hours to provide test results, and if there are too many performance tests, the infrastructure will be unable to run them all. We need to investigate methods that enable automatic prioritization and scheduling of the most relevant tests first—based on the logical structure of the expression capturing the performance assumption, on the relationship between (changed) code locations and performance tests, or the correlation of performance observations from different tests.

**Accuracy and automation.** Determining the outcome of performance tests requires processing significant amounts of measurement data, where detecting small, isolated changes in performance is difficult, because the required sensitivity is typically below the noise level. Current methods try to increase sensitivity by collecting more measurement data, which takes more time and in the end we usually learn that variation is high, and therefore performance changes are not detected. However, with multiple commits per day and each commit having the potential to reduce performance by a small amount, the performance testing infrastructure should be able to detect accumulated performance degradation. We need to investigate methods that correlate performance observations with other indicators, e.g., hardware performance counters, to distinguish essential performance changes from noise and thus improve test evaluation results or potentially reduce the number of experiments that need to be conducted to evaluate a test.

## 11.2 Dynamic Program Analysis

While performance testing is intended to make performance of a software system “visible” to developers, dynamic program analysis is intended to allow developers to observe and gain insight into what a program is doing with respect to the underlying hardware and software platform.

In dealing with software complexity, developers rely on modern programming languages and application frameworks which introduce additional layers of abstraction to hide some of the complexity. These abstractions need to be mapped efficiently to the underlying hardware and software platforms, and if there is a mismatch between what the software does and what the platform can do efficiently, performance suffers. Dynamic program analysis allows to see through layers of abstractions to get insight into what kind of workload a program imposes on the underlying platform and how the platform handles it. Together with timing information, program behavior can be correlated with program or platform performance, contributing to performance awareness.

We have made several contributions to dynamic program analysis that make it easier to develop and apply dynamic analysis tools, but further progress is necessary in several research directions.

**Observability on modern platforms.** The ability to observe program behavior is crucial for dynamic program analysis and modern runtime platforms make it increasingly difficult to observe what the software does. This applies in particular to memory-managed platforms executing higher-level hardware-independent bytecode, such as the Oracle Java Virtual Machine, the Google Dalvik Virtual Machine, or the Microsoft Common Language Runtime. These virtual machines are generally optimized for performance, with program observation being a secondary concern (if at all), which makes it difficult (or impossible) to observe certain behavior. We need to focus on making those platforms effectively (and efficiently) observable. Specific challenges include efficient reconstruction of event ordering and correlation of low-level performance metrics associated with machine code executed by the processor with high-level program bytecode executed by the virtual machine.

**Analysis construction.** Developing analysis tools requires expertise both in developing instrumentation for the given platform, as well as in developing analysis algorithms processing the events coming from the observed programs. This makes construction of analysis tools difficult, because the knowledge of the underlying platform needed to be able to successfully instrument a program is substantial. Existing frameworks mainly assist with the task of instrumentation, but provide limited or no support at all in the way of analysis composition, forcing analysis authors to re-address recurring requirements. We need to focus on the software engineering aspects of dynamic program analysis, specifically on simplifying analysis composition and on composition of hybrid analyses, where parts of an analysis may execute in different address spaces.

**Analysis of concurrent programs.** As new hardware platforms appear and software platforms evolve, new kinds of program behavior become relevant and need to be studied. This decade is marked with the shift from single-core towards multi-core processors, and software follows suit to make use of the additional processor cores. Multi-threaded programming is notoriously difficult and error prone, not only with respect to correctness, but also with respect to performance. High-level abstractions (and managed-memory platforms) typically hide details such as data layout, locking, or the use of memory barriers, but these details may contribute to a misalignment between what the software does and what the underlying hardware or software platform can do efficiently. We need to focus on identifying behavior of concurrent programs that is detrimental to performance on modern multi-core and non-uniform memory access (NUMA) hardware, and distributed computational platforms.

# References

[1] M. Andreessen: **“Why Software Is Eating The World”**. In *The Wall Street Journal* (2011).

[2] D. Ansaloni, S. Kell, Y. Zheng, L. Bulej, W. Binder, and P. Tůma: **“Enabling Modularity and Reuse in Dynamic Program Analysis Tools for the Java Virtual Machine”**. In *Proc. 27th European Conference on Object-Oriented Programming (ECOOP)*. LNCS 7920. Springer, 2013, pp. 352–377. DOI: 10.1007/978-3-642-39038-8_15.

[3] V. Babka, L. Bulej, M. Děcký, J. Kraft, P. Libič, L. Marek, C. Seceleanu, and P. Tůma: **“Q-ImPrESS Project Deliverable D3.3: Resource Usage Modeling”**. Tech. rep. D3.3. Q-ImPrESS Consortium, 2009, p. 210.

[4] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas: **“Manifesto for Agile Software Development”**. In (2001).

[5] S. M. Blackburn, A. Diwan, M. Hauswirth, P. F. Sweeney, J. N. Amaral, T. Brecht, L. Bulej, C. Click, L. Eeckhout, S. Fischmeister, D. Frampton, L. J. Hendren, M. Hind, A. L. Hosking, R. E. Jones, T. Kalibera, N. Keynes, N. Nystrom, and A. Zeller: **“The Truth, The Whole Truth, and Nothing But the Truth: A Pragmatic Guide to Assessing Empirical Evaluations”**. In *ACM Transactions on Programming Languages and Systems* 38.4 (2016), 15:1–15:20. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/2983574.

[6] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann: **“The DaCapo Benchmarks: Java Benchmarking Development and Analysis”**. In *Proc. 21st ACM SIGPLAN Intl. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 2006, pp. 169–190. DOI: 10.1145/1167473.1167488.

[7] F. P. J. Brooks: **“No Silver Bullet Essence and Accidents of Software Engineering”**. In *Computer* 20.4 (1987), pp. 10–19. ISSN: 0018-9162. DOI: 10.1109/MC.1987.1663532.

[8] A. Buble, L. Bulej, and P. Tůma: **“CORBA Benchmarking: A Course with Hidden Obstacles”**. In *Proc. 17th International Parallel and Distributed Processing Symposium*. Intl. W. on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS). 2003, pp. 1–6. DOI: 10.1109/IPDPS.2003.1213501.

[9] L. Bulej: **“Performance Testing in Software Development: Getting the Developers on Board (Invited Talk Abstract)”**. In *Companion Proc. 7th ACM/SPEC International Conference on Performance Engineering*. 5th Intl. W. on Large-Scale Testing (LT). ACM, 2016, pp. 9–9. DOI: 10.1145/2859889.2880448.

[10] L. Bulej, T. Bureš, I. Gerostathopoulos, V. Horký, J. Keznikl, L. Marek, M. Tschaikowski, M. Tribastone, and P. Tůma: **“Supporting Performance Awareness in Autonomous Ensembles”**. In *Software Engineering for Collective Autonomic Systems*. LNCS 8998. Springer, 2015, pp. 291–322. DOI: 10.1007/978-3-319-16310-9_8.

[11] L. Bulej, T. Bureš, V. Horký, and J. Keznikl: **“Adaptive Deployment in Ad-Hoc Systems Using Emergent Component Ensembles (Vision Paper)”**. In *Proc. 4th ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 2013, pp. 343–346. DOI: 10.1145/2479871.2479922.

[12] L. Bulej, T. Bureš, V. Horký, J. Keznikl, and P. Tůma: **“Performance Awareness in Component Systems (Vision Paper)”**. In *Proc. 36th IEEE Annual Computer Software and Applications Conference Workshops*. 4th IEEE Intl. W. on Component-Based Design of Resource-Constrained Systems (CORCS). 2012, pp. 514–519. DOI: 10.1109/COMPSACW.2012.96.

[13] L. Bulej, T. Bureš, V. Horký, J. Kotrč, L. Marek, T. Trojánek, and P. Tůma: **“Unit Testing Performance with Stochastic Performance Logic”**. In *Automated Software Engineering* 24.1 (2017), pp. 139–187. ISSN: 0928-8910, 1573-7535. DOI: 10.1007/s10515-015-0188-0.

[14] L. Bulej, T. Bureš, J. Keznikl, A. Koubková, A. Podzimek, and P. Tůma: **“Capturing Performance Assumptions Using Stochastic Performance Logic”**. In *Proc. 3rd ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 2012, pp. 311–322. DOI: 10.1145/2188286.2188345.

[15] L. Bulej, T. Kalibera, and P. Tůma: **“Regression Benchmarking with Simple Middleware Benchmarks”**. In *Proc. 23rd IEEE International Performance, Computing, and Communications Conference*. Intl. W. on Middleware Performance (IWMP). IEEE, 2004, pp. 771–776. DOI: 10.1109/PCCC.2004.1395179.

[16] L. Bulej, T. Kalibera, and P. Tůma: **“Repeated Results Analysis for Middleware Regression Benchmarking”**. In *Performance Evaluation* 60.1–4 (2005), pp. 345–358. ISSN: 0166-5316. DOI: 10.1016/j.peva.2004.10.013.

[17] J. Cheng: **“Steve Jobs: MobileMe ”Not up to Apple’s Standards””**. *Ars Technica*, 2008. URL: http://arstechnica.com/apple/2008/08/steve-jobs-mobileme-not-up-to-apples-standards/ (visited on 09/12/2016).

[18] P. Cohen: **“Firefox 3.0 Released, Servers Overwhelmed”**. *Macworld*, 2008. URL: http://www.macworld.com/article/1134018/firefox.html (visited on 09/12/2016).

[19] M. Cohn: **“Succeeding with Agile: Software Development Using Scrum”**. Addison-Wesley Professional, 2009. ISBN: 978-0-321-57936-2.

[20] J. Dean and L. A. Barroso: **“The Tail at Scale”**. In *Commun. ACM* 56.2 (2013), pp. 74–80. ISSN: 0001-0782. DOI: 10.1145/2408776.2408794.

[21] K. Foo, Z. M. Jiang, B. Adams, A. Hassan, Y. Zou, and P. Flora: **“Mining Performance Regression Testing Repositories for Automated Performance Analysis”**. In *Proc. QSIC 2010*. 2010, pp. 32–41. DOI: 10.1109/QSIC.2010.35.

[22] S. Ghaith, M. Wang, P. Perry, and J. Murphy: **“Profile-Based, Load-Independent Anomaly Detection and Analysis in Performance Regression Testing of Software Systems”**. In *Proc. 17th European Conference on Software Maintenance and Reengineering*. 17th European Conference on Software Maintenance and Reengineering. 2013, pp. 379–383. DOI: 10.1109/CSMR.2013.54.

[23] C. Heger, J. Happe, and R. Farahbod: **“Automated Root Cause Isolation of Performance Regressions During Software Development”**. In *Proc. ICPE 2013*. ACM/SPEC International Conference on Performance Engineering (ICPE). ACM, 2013, pp. 27–38. DOI: 10.1145/2479871.2479879.

[24] V. Horký, F. Haas, J. Kotrč, M. Lacina, and P. Tůma: **“Performance Regression Unit Testing: A Case Study”**. In *Proc. 10th European Performance Engineering Workshop*. EPEW. Springer, 2013, pp. 149–163. DOI: 10.1007/978-3-642-40725-3_12.

[25] R. Hyde: **“The Fallacy of Premature Optimization”**. In *Ubiquity* 2009 (February 2009). ISSN: 1530-2180. DOI: 10.1145/1513450.1513451.

[26] T. Kalibera, L. Bulej, and P. Tůma: **“Automated Detection of Performance Regressions: The Mono Experience”**. In *Proc. 20th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. MASCOTS. IEEE Computer Society, 2005, pp. 183–190. DOI: 10.1109/MASCOT.2005.18.

[27] T. Kalibera, L. Bulej, and P. Tůma: **“Benchmark Precision and Random Initial State”**. In *Proc. International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*. SPECTS. SCS, 2005, pp. 853–862. ISBN: 978-1-62276-350-4.

[28] T. Kalibera, L. Bulej, and P. Tůma: **“Generic Environment for Full Automation of Benchmarking”**. In *Proc. 1st International Workshop on Software Quality (SOQUA)*. GI, 2004, pp. 125–132. ISBN: 3-88579-387-3.

[29] T. Kalibera, L. Bulej, and P. Tůma: **“Quality Assurance in Performance: Evaluating Mono Benchmark Results”**. In *Quality of Software Architectures and Software Quality: Proc. 2nd International Workshop on Software Quality (SOQUA)*. LNCS 3712. Springer, 2005, pp. 271–288. DOI: 10.1007/11558569_20.

[30] T. Kalibera, J. Lehotský, D. Majda, B. Repček, M. Tomčányi, A. Tomeček, P. Tůma, and J. Urban: **“Automated Benchmarking and Analysis Tool”**. In *Proc. 1st Intl. Conf. on Performance Evaluation Methodolgies and Tools*. VALUETOOLS. ACM, 2006. DOI: 10.1145/1190095.1190101.

[31] T. Kalibera and P. Tůma: **“Precise Regression Benchmarking with Random Effects: Improving Mono Benchmark Results”**. In *Proc. 3rd European Performance Engineering Workshop*. EPEW. Springer, 2006, pp. 63–77. DOI: 10.1007/11777830_5.

[32] S. Kell, D. Ansaloni, W. Binder, and L. Marek: **“The JVM Is Not Observable Enough (and What to Do About It)”**. In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages*. ACM, 2012, pp. 33–38. DOI: 10.1145/2414740.2414747.

[33] K. Kennedy: **“Government Did Not Test Health Care Site as Needed”**. *USA TODAY*, 2013. URL: http://www.usatoday.com/story/news/nation/2013/10/24/cms-update-healthcaregov-bandwidth/3179545/ (visited on 09/12/2016).

[34] D. E. Knuth: **“Structured Programming with Go to Statements”**. In *ACM Comput. Surv.* 6.4 (1974), pp. 261–301. ISSN: 0360-0300. DOI: 10.1145/356635.356640.

[35] W. H. Li, D. R. White, and J. Singer: **“JVM-Hosted Languages: They Talk the Talk, but Do They Walk the Walk?”** In *Proc. PPPJ 2013*. ACM, 2013, pp. 101–112. DOI: 10.1145/2500828.2500838.

[36] P. Libič, L. Bulej, V. Horký, and P. Tůma: **“On the Limits of Modeling Generational Garbage Collector Performance”**. In *Proc. 5th ACM/SPEC Intl. Conf. on Performance Engineering (ICPE)*. ACM, 2014, pp. 15–26. DOI: 10.1145/2568088.2568097.

[37] P. Libič, P. Tůma, and L. Bulej: **“Issues in Performance Modeling of Applications with Garbage Collection”**. In *Proc. 1st Intl. W. on Quality of Service-Oriented Software Systems (QUASOSS)*. ACM, 2009, pp. 3–10. DOI: 10.1145/1596473.1596477.

[38] L. Marek, S. Kell, Y. Zheng, L. Bulej, W. Binder, P. Tůma, D. Ansaloni, A. Sarimbekov, and A. Sewe: **“ShadowVM: Robust and Comprehensive Dynamic Program Analysis for the Java Platform”**. In *Proc. 12th International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2013, pp. 105–114. DOI: 10.1145/2517208.2517219.

[39] L. Marek, S. Kell, Y. Zheng, L. Bulej, W. Binder, P. Tůma, D. Ansaloni, A. Sarimbekov, and A. Sewe: **“ShadowVM: Robust and Comprehensive Dynamic Program Analysis for the Java Platform”**. In *Proc. 12th International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2013, pp. 105–114. DOI: 10.1145/2517208.2517219.

[40] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi: **"DiSL: A Domain-Specific Language for Bytecode Instrumentation"**. In *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development*. ACM, 2012, pp. 239–250. DOI: 10.1145/2162049.2162077.

[41] L. Marek, Y. Zheng, D. Ansaloni, L. Bulej, A. Sarimbekov, W. Binder, and P. Tůma: **"Introduction to Dynamic Program Analysis with DiSL"**. In *Science of Computer Programming* 98, Part 1 (2015), pp. 100–115. ISSN: 0167-6423, 1872-7964. DOI: 10.1016/j.scico.2014.01.003.

[42] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney: **"Producing Wrong Data Without Doing Anything Obviously Wrong!"** In *ACM SIGPLAN Notices* 44.3 (2009), pp. 265–276. ISSN: 0362-1340. DOI: 10.1145/1508284.1508275.

[43] J. K. Ousterhout: **"Scripting: Higher Level Programming for the 21st Century"**. In *Computer* 31.3 (1998), pp. 23–30. ISSN: 0018-9162. DOI: 10.1109/2.660187.

[44] A. Podzimek and L. Y. Chen: **"Transforming System Load to Throughput for Consolidated Applications"**. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. 2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems. 2013, pp. 288–292. DOI: 10.1109/MASCOTS.2013.37.

[45] A. Podzimek, L. Bulej, L. Y. Chen, W. Binder, and P. Tůma: **"Analyzing the Impact of CPU Pinning and Partial CPU Loads on Performance and Energy Efficiency"**. In *Proc. 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2015, pp. 1–10. DOI: 10.1109/CCGrid.2015.164.

[46] A. Podzimek, L. Bulej, L. Y. Chen, W. Binder, and P. Tůma: **"Robust Partial-Load Experiments with Showstopper"**. In *Future Generation Computer Systems* 64 (2016), pp. 15–38. ISSN: 0167-739X, 1872-7115. DOI: 10.1016/j.future.2016.04.020.

[47] A. Podzimek, L. Y. Chen, L. Bulej, W. Binder, and P. Tůma: **"Showstopper: The Partial CPU Load Tool (Tool Paper)"**. In *Proc. 22nd IEEE International Symposium on Modelling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2014, pp. 510–513. DOI: 10.1109/MASCOTS.2014.75.

[48] V. Rajlich: **"Software Engineering: The Current Practice"**. Innovations in Software Engineering and Software Development. Chapman and Hall/CRC, 2011. ISBN: 978-1-4398-4122-8.

[49] A. Sarimbekov, A. Podzimek, L. Bulej, Y. Zheng, N. Ricci, and W. Binder: **"Characteristics of Dynamic JVM Languages"**. In *Proc. 7th ACM Workshop on Virtual Machines and Intermediate Languages (VMIL)*. ACM, 2013, pp. 11–20. DOI: 10.1145/2542142.2542144.

[50] A. Sarimbekov, A. Sewe, S. Kell, Y. Zheng, W. Binder, L. Bulej, and D. Ansaloni: **“A Comprehensive Toolchain for Workload Characterization Across JVM Languages”**. In *Proc. 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. ACM, 2013, pp. 9–16. DOI: 10.1145/2462029.2462033.

[51] A. Sarimbekov, L. Stadler, L. Bulej, A. Sewe, A. Podzimek, Y. Zheng, and W. Binder: **“Workload Characterization of JVM Languages”**. In *Software: Practice and Experience* 46.8 (2016), pp. 1053–1089. ISSN: 0038-0644, 1097-024X. DOI: 10.1002/spe.2337.

[52] A. Sarimbekov, Y. Zheng, D. Ansaloni, L. Bulej, L. Marek, W. Binder, P. Tůma, and Z. Qi: **“Dynamic Program Analysis - Reconciling Developer Productivity and Tool Performance”**. In *Science of Computer Programming* 95, Part 3 (2014), pp. 344–358. ISSN: 0167-6423. DOI: 10.1016/j.scico.2014.03.014.

[53] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder: **“Da Capo Con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine”**. In *Proc. 26th ACM SIGPLAN Intl. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 2011, pp. 657–676. DOI: 10.1145/2048066.2048118.

[54] H. Sun, Y. Zheng, L. Bulej, W. Binder, and S. Kell: **“Custom Full-Coverage Dynamic Program Analysis for Android (Demo Paper)”**. In *Companion Proc. 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)*. ACM, 2015, pp. 7–8. DOI: 10.1145/2814189.2814190.

[55] H. Sun, Y. Zheng, L. Bulej, S. Kell, and W. Binder: **“Analyzing Distributed Multi-Platform Java and Android Applications with ShadowVM (Demo Paper)”**. In *Proc. 13th Asian Symposium on Programming Languages and Systems (APLAS)*. LNCS 9458. Springer, 2015, pp. 356–365. DOI: 10.1007/978-3-319-26529-2_19.

[56] H. Sun, Y. Zheng, L. Bulej, A. Villazón, Z. Qi, P. Tůma, and W. Binder: **“A Programming Model and Framework for Comprehensive Dynamic Analysis on Android”**. In *Proc. 14th International Conference on Modularity (AOSD/MODULARITY)*. ACM, 2015, pp. 133–145. DOI: 10.1145/2724525.2724566.

[57] H. Takeuchi and I. Nonaka: **“The New New Product Development Game”**. In *Harvard Business Review* (January 1986). ISSN: 0017-8012.

[58] C. Tran: **“Website Crashes as Australians Attempt to Complete Their Census Online”**. *Daily Mail Online*, 2016. URL: http://www.dailymail.co.uk/news/article-3731150/Census-website-crashes-thousands-Australians-attempt-complete-form-online.html (visited on 09/12/2016).

[59] J. Venturová, A. Horáček, P. Ježek, and M. Kolařík: **“Nový Centrální Registr Vozidel Zkolaboval Po Pár Minutách Provozu”**. *iDNES.cz*, 2012. URL: http://zpravy.idnes.cz/centralni-registr-vozidel-nefunguje-dsp-/domaci.aspx?c=A120709_085450_domaci_jpl (visited on 09/13/2016).

[60] E. J. Weyuker and F. I. Vokolos: **“Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study”**. In *IEEE Transactions on Software Engineering* 26.12 (2000), pp. 1147–1156. ISSN: 0098-5589. DOI: 10.1109/32.888628.

[61] M. Wirsing, M. Hölzl, N. Koch, and P. Mayer, eds.: **“Software Engineering for Collective Autonomic Systems”**. LNCS. Springer, 2015. DOI: 10.1007/978-3-319-16310-9.

[62] M. Wirsing, M. Hölzl, M. Tribastone, and F. Zambonelli: **“ASCENS: Engineering Autonomic Service-Component Ensembles”**. In *Proceedings of FMCO 2011 (Revised Selected Papers), Turin, Italy*. Springer, 2011, pp. 1–24. DOI: 10.1007/978-3-642-35887-6_1.

[63] Y. Zheng, D. Ansaloni, L. Marek, A. Sewe, W. Binder, A. Villazón, P. Tůma, Z. Qi, and M. Mezini: **“Turbo DiSL: Partial Evaluation for High-Level Bytecode Instrumentation”**. In *Intl. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*. TOOLS. Springer, 2012, pp. 353–368. DOI: 10.1007/978-3-642-30561-0_24.

[64] Y. Zheng, L. Bulej, and W. Binder: **“Accurate Profiling in the Presence of Dynamic Compilation”**. In *Proc. 30th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2015, pp. 433–450. DOI: 10.1145/2814270.2814281.

[65] Y. Zheng, L. Bulej, and W. Binder: **“An Empirical Study on Deoptimization in the Graal Compiler”**. In *Proc. 31st European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 30:1–30:30. DOI: 10.4230/LIPIcs.ECOOP.2017.30.

[66] Y. Zheng, S. Kell, L. Bulej, H. Sun, and W. Binder: **“Comprehensive Multi-Platform Dynamic Program Analysis for Java and Android”**. In *IEEE Software* 33.4 (2016), pp. 55–63. ISSN: 0740-7459 1937-4194. DOI: 10.1109/MS.2015.151.