Workload characterization of JVM languages

Aibek Sarimbekov1, Lukas Stadler2, Lubomír Bulej1,4, Andreas Sewe3, Andrej Podzimek5, Yudi Zheng1 and Walter Binder1,*,†

1Università della Svizzera italiana (USI), Lugano, Switzerland
2Johannes Kepler Universität Linz, Linz, Austria
3Technische Universität Darmstadt, Darmstadt, Germany
4Charles University, Prague, Czech Republic

SUMMARY
Originally developed with a single language in mind, the JVM is now targeted by numerous programming languages—its automatic memory management, just-in-time compilation, and adaptive optimizations—making it an attractive execution platform. However, the garbage collector, the just-in-time compiler, and other optimizations and heuristics were designed primarily with the performance of Java programs in mind. Consequently, many of the languages targeting the JVM, and especially the dynamically typed languages, are suffering from performance problems that cannot be simply solved at the JVM side. In this article, we aim to contribute to the understanding of the character of the workloads imposed on the JVM by both dynamically typed and statically typed JVM languages. To this end, we introduce a new set of dynamic metrics for workload characterization, along with an easy-to-use toolchain to collect the metrics. We apply the toolchain to applications written in six JVM languages (Java, Scala, Clojure, Jython, JRuby, and JavaScript) and discuss the findings. Given the recently identified importance of inlining for the performance of Scala programs, we also analyze the inlining behavior of the HotSpot JVM when executing bytecode originating from different JVM languages. As a result, we identify several traits in the non-Java workloads that represent potential opportunities for optimization. © 2015 The Authors. Software: Practice and Experience Published by John Wiley & Sons Ltd.

Received 24 April 2014; Revised 30 April 2015; Accepted 8 May 2015

KEY WORDS: workload characterization; dynamic metrics; JVM languages; dynamic program analysis

1. INTRODUCTION

The complexity of modern applications has brought about a paradigm shift in software development: polyglot programming [1]. Different parts of an application call for different levels of performance and productivity, and polyglot programming encourages developers to use languages best suited for the various tasks at hand. Consequently, in some projects, core application parts can be written, for example, in a statically typed language, whereas data management can use a suitable domain-specific language, and the glue code that binds all the pieces together can very well be written in a dynamically typed language. Other projects may have different requirements or choose different languages, but support for polyglot programming makes these choices possible.

This paradigm shift has gained support in popular managed runtimes such as the .NET Common Language Runtime and the JVM, in which both offer automatic memory management,
high-performance just-in-time (JIT) compilation, and a rich class library. In case of the JVM, the introduction of scripting support (JSR-223) and support for dynamically typed languages (JSR-292) simplifies the development of dynamic language runtimes on top of the JVM and thus makes polyglot programming feasible. Consequently, literally hundreds of programming languages target the JVM. This not only avoids developing a new runtime from scratch, thereby re-inventing the wheel; but the language implementations also directly benefit from the JVM’s maturity, ubiquity, and performance. Today, programs written in popular dynamic languages such as JavaScript, Ruby, Python, or Clojure (a dialect of Lisp) can be run on the JVM, creating an ecosystem that boosts developer productivity.

Even though being targeted by so many languages, the JVM was originally designed with a single language in mind, namely Java, and was therefore tuned only with respect to characteristics of Java programs. Other languages targeting the JVM, and in particular dynamically typed languages, suffered from performance problems. To improve the performance of dynamic languages executing on the JVM, Java 7 introduced the `invokedynamic` bytecode, but the expected performance benefits did not (fully) materialize [2]. One of the reasons for reduced performance of dynamic languages is their inherently generic execution model. The developers of JVM language compilers are therefore advised to specialize the generic execution as much as possible, and not to rely solely on the JVM JIT when repurposing it for dynamically typed languages [3].

Making the JVM perform well with various statically and dynamically typed languages clearly requires significant effort, not only in optimizing the JVM but, more importantly, also in optimizing the bytecode-emitting language compilers. This in turn requires that the developers of both the JVM language compilers and the JVM itself understand the characteristics of the workloads imposed on the JVM by various languages or compilation strategies. Of particular interest is identification and understanding of corner cases that prevent or negate JVM optimizations (i.e., by identifying characteristic difficult-to-optimize bytecode patterns produced by non-Java compilers).

To aid in understanding the differences and commonalities between programs written in different languages, we present a set of new dynamic metrics for workload characterization across JVM languages and provide a toolchain for obtaining these metrics for any program executed by a standard JVM [4]. We have applied the toolchain to programs in Java and five JVM languages (Clojure, JavaScript, Python, Ruby, and Scala) and report on the results.

Specifically, we compare the character of the workloads imposed on the JVM by programs in these languages with that of workloads produced by functionally equivalent Java programs. We identify differences and commonalities among the selected JVM languages and point out optimization opportunities where applicable. We note that (based on our results) we cannot evaluate whether any actual optimizations would pay off—optimizing a program involves many trade-offs—and not all opportunities can be exploited. Nevertheless, we believe that the results of our investigation are worth the developer’s attention and that highlighting characteristic differences in workloads can provide directions for further study.

In general, this article consolidates our previous research on workload characterization [4–8]. To add an additional perspective to our study, we also present an analysis of the ability of the HotSpot JVM to perform a fundamental optimization, inlining, on the workloads written in different JVM languages. The analysis was triggered by the recently reported importance of inlining on the performance of Scala programs [8]. Because the JIT compiler decisions cannot be observed at the bytecode level (which is what our toolchain does), the data for this analysis were collected using a modified HotSpot JVM. The specific contribution of this article is therefore the first, previously unpublished, in-depth analysis of inlining decisions made by the HotSpot JVM not only for Scala but also for Clojure, JavaScript, Python, and Ruby. We believe that the results will be useful to a broader audience, because we have not found such a study in the literature.

The rest of the article is structured as follows: Section 2 introduces the dynamic metrics, their intended use, and the toolchain for collecting them. Section 4 provides details on the experimental setup and the benchmarks used to obtain the presented results. Section 5 presents and discusses the characterization results, while Section 6 investigates the HotSpot JVM inlining decisions for different JVM languages. Section 7 reviews related work, and Section 8 concludes the article.

© 2015 The Authors. *Software: Practice and Experience* Published by John Wiley & Sons Ltd. *Softw. Pract. Exper.* 2016; 46:1053–1089 DOI: 10.1002/spe
2. DYNAMIC METRICS

Similar to performance evaluation (benchmarking), workload characterization employs benchmarks (representing samples from the domain of applications) to induce workload on the observed system while collecting metrics that characterize different aspects of the behavior of the system. However, unlike benchmarking, which aims to determine *how well* a system performs at different tasks, workload characterization aims to determine *in what way* these tasks differ from each other, providing essential guidance, for example, for optimization effort. Ideally, the metrics characterizing JVM workloads should capture the differences between Java and non-Java workloads and—when correlated with JVM performance on a particular workload—they should provide developers of both JVM languages and the JVM itself with useful insights.

For example, a developer might hypothesize that a workload performed poorly because of heap pressure generated by increased usage of boxed primitive values, which are used relatively rarely in normal Java code but frequently in some other JVM languages such as in JRuby. JVM language developers could optimize their bytecode generator, for example, to try harder at using primitives in their unboxed form. A dynamic metric capturing the boxing behavior of a particular workload would allow these developers to quantify the effects of such optimizations. Meanwhile, JVM developers may benefit from the metrics in a different way. Because JVM optimizations are dynamic and adaptive, each optimization decision is guarded by a heuristic decision procedure applied to profile data collected at runtime. For example, the decision whether to inline a callee into a fast path depends on factors such as the hotness of that call site (evaluated by dynamic profiling) and the size of the callee. JVMs can therefore benefit from better heuristics that more accurately match real workloads, including non-Java workloads.

For maximum benefit, there must be an easy way for developers to compute these metrics over workloads of their choosing. However, no existing work has defined a comprehensive set of such metrics and provided the tools to compute them. Rather, existing approaches are fragmented across different infrastructures: many lack portability because of using a modified version of the JVM [9, 10], while others collect only architecture-dependent metrics [11]. In addition, at least one well-known metric suite implementation [12] runs with unnecessarily high-performance overhead. Ideally, metrics should be collected within reasonable time, because this enables the use of complex, real-world workloads and shortens the development cycles. Metrics should also be computed based on observation of the whole workload, which not all infrastructures allow. For example, existing metrics collected using AspectJ are suboptimal because they lack coverage of code from the Java class library [13–15].

Our approach bases all metrics on a unified infrastructure, which is JVM portable, offers non-prohibitive runtime overhead [16] with near-complete bytecode coverage, and computes a full suite of metrics ‘out of the box’. All the metrics are dynamic, meaning that they can be evaluated only by running a program with some input. The significance of dynamic metrics—in contrast to static metrics such as code size or instruction distribution—has been motivated elsewhere by Dufour *et al.* [12], who defined a list of 60 metrics considered useful for guiding optimization of Java programs. Our infrastructure can compute all of these metrics.

However, the workloads produced by the various JVM languages exhibit properties that vary significantly between Java and non-Java workloads, and require additional metrics for proper characterization. In this section, we present such metrics, summarized in Table I. Like those of Dufour *et al.*, the new metrics are defined at the bytecode level, which makes them JVM independent and platform independent; therefore, the toolchain to collect them can be implemented in a portable fashion. The new metrics highlight the differences arising from observed languages’ distinct semantics, implying that different optimizations will be required on the parts of the JVM developers and the language (front-end) developers. We have grouped the metrics according to the language-level concerns that motivate them: object access, object allocation, and code generation. We now review each group in turn.

© 2015 The Authors. *Software: Practice and Experience* Published by John Wiley & Sons Ltd.
Table I. Metrics that can be computed by our toolchain.

| Metric family          | Description of metrics                                                                 |
|------------------------|----------------------------------------------------------------------------------------|
| Field immutability     | Number of fields immutable, counted once per containing object                          |
|                        | Number of fields immutable (per class)                                                 |
| Field synchronization  | Number of objects synchronized on                                                      |
|                        | The average number of locking operations per object                                     |
|                        | The maximum nesting depth reached per lock                                             |
| Field sharing          | Number of objects partially read-shared between different threads                      |
|                        | Number of objects partially write-shared between different threads                     |
|                        | Number of objects fully read-shared between different threads                          |
|                        | Number of objects fully write-shared between different threads                         |
| Use of boxed types     | Number of boxed primitives allocated                                                   |
|                        | Number of boxed primitives requested (using valueOf; see text)                         |
| Object churn distance  | Distribution of object churn distances (see text)                                      |
| Implicit zeroing       | Number of primitive fields unnecessarily zeroed                                         |
|                        | Number of reference fields unnecessarily zeroed                                         |
| Use of identity hash codes | Execution counts of overridden  hashCode  methods                                      |
|                        | Execution counts of  System.  hashCode  methods                                         |
|                        | Execution counts of default  Object.  hashCode  method                                 |
| Object lifetimes       | Distribution of object survival times (see text)                                       |
| Object sizes           | Distribution of object sizes (see text)                                                |
| Instruction mix        | Execution counts for each distinct bytecode instruction (opcode)                       |
| Stack usage and recursion depth | Distribution of stack heights upon recursive calls                                   |
| Argument passing       | Distribution of floating-point arguments over all dynamic invocations (see text)       |
| Basic block hotness    | Contribution of the top 20% of basic blocks to the dynamic total number of basic block executions |
| Method hotness         | Contribution of the top 20% of methods to the dynamic total number of method executions |
| Call site polymorphism | Distribution of target method count over all dynamically dispatched calls              |
|                        | Number of dynamically dispatched call sites targeting a given number of methods        |
|                        | Number call sites using each of the four invoke instructions                           |
|                        | Number of calls made using each of the four invoke instructions                        |

2.1. Object-access concerns

Object-access concerns affect optimizations related to sharing of objects among threads. Accessing shared objects generally requires locking, unless immutable or lock-free data structures are used. Our metrics therefore focus on identifying effectively immutable objects, the kind of locks used, and the kind of access to shared objects.

Immutability. In recent years, functional languages have gained much attention. In general, functional programs tend to use immutable data structures to avoid side effects, which make such programs amenable to parallelization. Finding objects that are effectively immutable can help a developer to identify code locations where using immutable types could simplify parallelization. In addition, popular compiler optimization techniques benefit from immutable objects and data structures [17]. One example of such an optimization is load elimination, which replaces repeated memory accesses to the immutable objects with access to a compiler-generated temporary (likely to be stored in a register). However, this optimization is defeated in the presence of method calls or synchronization. Immutable objects avoid this problem, because they are known not to change across method calls.
To characterize immutability, we define four metrics, distinguishing between class and object immutability: number of instance fields that are per-object immutable, number of objects that are immutable (i.e., all fields immutable), number of fields that are immutable in all allocated objects of the defining class, and number of classes for which all allocated instances are immutable‡.

**Lock usage and sharing.** Because locking operations come at a cost, researchers have developed thin locks [18] and biased locks [19] to minimize the runtime overhead and memory cost. Thin locks are used in the situation where most locks are never subject to contention by multiple threads. Moreover, if most of the locks are only acquired by a single thread, biased locks are used. To apply synchronization optimizations, one has to identify the common-case nature of locking operations in the application. We count the number of objects synchronized on, and the average number of locking operations per object, and the maximum nesting depth reached per (recursive) lock.

**Unnecessary synchronization.** Immutability and sharing analyses can be used in combination to aid in removal of unnecessary synchronization [20]. Ordinarily, objects shared among different threads potentially require some synchronization. However, the synchronization is redundant if we find that the object is immutable. The metrics capture the number of objects shared between different threads, with separate counts for read-only sharing (two or more readers; exactly one writer, that is, the allocating thread) and write sharing (two or more writers; any number of readers). As in the case of the immutability analysis, we distinguish between fully and partially shared objects, yielding four distinct metrics in total.

### 2.2. Object allocation concerns

In a managed runtime, developers rely on a garbage collector (GC) to reclaim unused memory. While such a programming model greatly simplifies development, abusing it may result in undesirable pressure on the GC, causing significant performance degradation. This is of paramount importance for developers of JVM language compilers, because their decisions regarding minute details in the implementation of various language constructs may greatly influence the character of the workload imposed on the JVM.

**Use of boxed types.** Different languages make differing use of boxed primitives. For example, all primitive values in JRuby are boxed. However, boxing is expensive because it creates additional heap pressure and can defeat optimization passes usually applied to stack-allocated and register-allocated primitive values. Different optimization techniques can be used to reduce performance overhead incurred by boxed values. Therefore, a metric characterizing the extent of boxing in the workload is very useful. Our two metrics here are the counts of boxed primitives allocated and boxed primitives requested (by calls to `valueOf()` methods on `Integer`, `Byte`, and so on).

**Object churn.** Creation of many temporary objects (i.e., object churn), which may or not be boxed types, is detrimental to performance, because it comes at a cost of very frequent garbage collection and inhibits parallelization if temporary objects require synchronization [21]. Dufour et al. [22] showed that object churn is the main source of performance overhead in framework-intensive Java applications. Identifying places where object churn happens leverages performance understanding and is the basis of escape analysis [23].

Object churn distance is a metric defined recently by Sewe et al. [6] and has been shown to be different for Java and Scala workloads.

We illustrate the concepts behind the metric in Figure 1. For each object, we keep track of the calling contexts where an object had been allocated and where it died, that is, stopped being referenced.

---

‡Many of our metrics are collected as raw numbers but could be more usefully represented as fractions. Although we do not state this explicitly from hereon, in all such cases, the relevant total is available for use as a divisor. As such, both fractions and raw numbers are available.
Figure 1. The churn distance of an object is computed as the distance between its allocation (✱) and death (✝) calling contexts via their closest capturing context.

The dynamic churn distance is then the distance between the allocation and the death calling contexts via the closest capturing (common parent) context. This metric is of particular importance in dynamic languages where primitive types are boxed—these workloads exhibit lower average churn distances. We group objects by their churn distances and count the frequency for each group.

Impact of zeroing. According to the JVM specification [24], every primitive and reference type has to be initialized to a zero value—0 in case of primitive types, false in case of a boolean type, and null in case of a reference type. Yang et al. [25] have shown that zeroing has a large impact on performance. However, different languages have different rules concerning the initialization of fields, and different programming styles lead to greater or lesser extents of explicit initialization. For example, more declarative languages are less likely to rely on constructor-based piecewise imperative initialization of objects than conventional Java code.

A zero initialization analysis can help compiler developers to see whether implicit zeroing is actually necessary—fields that are written before they are first read do not need to be explicitly zeroed. Our zeroing analysis records occurrences of this pattern. The metric is a count of unnecessary zeroing of primitive and reference fields.

Identity hash codes. The JVM requires that every object has a hash code. If the object does not override the hashCode() method, then identityHashCode() is used instead. Implementation of the latter varies between JVM implementations, but commonly, a computed identity hash code is stored in the header of each object. This incurs costs in memory and cache usage. The overhead can be eliminated by using header compression techniques that define the default hash code of an object to be its address [26]. The hash code is explicitly stored only if an object has been moved by the GC and its identity hash code has been exposed—in such case, an extra header slot is lazily added to the object.

In workloads where the identity hash code is rarely used, this extra slot will rarely be allocated, yielding lower memory consumption with little runtime cost. In other workloads, eagerly allocating the header space for the hash code will yield better performance. Consequently, some heuristic is necessary to decide between the two approaches. We define three metrics over binned invocation counts: frequency of objects receiving overridden hashCode invocations, frequency of objects receiving System.identityHashCode invocations, and frequency of objects receiving the default Object.hashCode invocation (either by lack of override or by use of super).

Object lifetimes and sizes. Some languages allocate more, smaller, and/or shorter-lived objects than others. Object lifetime analysis is of particular importance to GC developers. New GC
algorithms are designed and evaluated by simulation based on object lifetime traces. An example of such an algorithm is lifetime-aware GC [27], in which the allocator lays out objects based on their death-time predictions. At each collection, only objects that are expected to die are scavenged. An object’s lifetime together with its size provide an estimate of the GC cost, because larger objects that live longer incur greater overhead than small, short-lived ones.

Our lifetime metric measures an object’s lifetime in the total amount of allocated memory in bytes, and the size represents the amount of space occupied by an object on the heap, that is, the size includes object header. This information lends itself to further analysis, for example, determining the distribution of object lifetimes either globally or for objects in a certain size range, and vice versa.

2.3. Code generation concerns

To take advantage of the infrastructure provided by a JVM, that is, its JIT compiler and the GC, a JVM language should be compiled into Java bytecode—while interpreted JVM languages exist, the optimization performed by the JVM only applies to the interpreter and not to the code it is executing. For compiled JVM languages, the resulting bytecode executed by the JVM plays a major role in the resulting performance. To aid in compiler construction, the last set of metrics characterizes properties that affect dynamic optimizations, which in turn depend on the use of virtual dispatch, the density of procedural abstraction, argument-passing behaviors, and the overall instruction mix.

Instruction mix. An instruction mix metric can identify the nature of the application—whether it is floating-point intensive or pointer intensive. This is relevant because, for example, some languages are more commonly used for numerical computations. This metric can be used for checking the diversity of the benchmarks in a benchmark suite, thus verifying that the benchmark suite indeed covers different application domains. Moreover, this metric can lead to possible dynamic optimizations. For instance, array bounds check removal for array intensive applications can help further optimizations like code motion and loop transformations.

To classify applications based on the instruction mix they execute, the bytecodes executed by the JVM are split into groups that are specific to particular application types. In contrast to Dufour et al. [12], who grouped over 200 individual bytecodes manually, we use PCA [28] to reduce the dimensionality of the data and to obtain a high-level view of the instruction mix in which the groupings of bytecode instructions are tailored to the workload.

Stack usage and recursion depth. This is an important metric for the developers of dynamic languages supporting the functional programming paradigm, for example, Clojure. Programs in functional languages often use recursion to perform iteration; therefore, it is very important for compiler developers to detect such situations and perform tail call elimination whenever possible to ensure that iterations implemented using recursion execute in constant space.

To provide insight on stack usage and recursion depth, our metric captures distributions of stack heights for all method invocations, for ‘potentially recursive’ invocations (virtual invocation that may dispatch to the same method), and for ‘truly recursive’ invocations (which actually do dispatch to the same method).

Argument passing. Information on parameters passed to methods can be used by JVM developers to choose an optimal calling convention in JIT-compiled code, making use of the registers available on the target architecture. Some architectures require particular types of arguments to be passed differently, for example, using special floating-point registers. We therefore distinguish three kinds of arguments: integer primitive values, references, and floating-point values. For each call, we count the number of arguments and record the kind of each argument. We then produce histograms capturing the distribution of the total number of arguments or the number of arguments of specific kind. The histograms contain a fixed number of bins corresponding to zero to four arguments and ≥ 5 arguments. The data lend itself to further analysis, for example, the distribution of floating-point arguments in four-argument methods.
Basic block hotness. Hotness metrics are fundamental, because any JVM with a JIT compiler prioritizes code optimization based on its hotness (i.e., a compiler primarily optimizes the most frequently executed code). While hotness is traditionally identified at the granularity of methods, certain dynamic compilers use trace-based approaches that rely on identifying sequences of hot basic blocks (possibly crossing method boundaries), such as PyPy [29] or Mozilla's TraceMonkey Javascript implementation§. While the trend now is to move back to coarser tracing approaches, having a more fine-grained hotness metric is still useful.

Having both method and basic block hotness data can indicate the relative gains from different compiler optimizations (say, inlining versus loop unrolling). Our metrics report to what extent the most executed 20% of all (distinct) methods in the code contribute to overall dynamic bytecode execution and likewise for basic blocks.

3. TOOLCHAIN DESCRIPTION

In the following, we describe the toolchain for collecting the presented dynamic metrics. Our toolchain consists of several distinct tools with a common infrastructure, which is designed for ease of use and extension.

3.1. Deployment and use

The primary goal of our infrastructure is to avoid imposing unnecessary overheads on developers wanting to make use of dynamic metrics. These include learning and setting up multiple new runtime environments and/or instrumentation tools. To avoid such overheads, all our tools are implemented using DiSL [30], a domain-specific language for instrumentation. DiSL provides full bytecode coverage, meaning execution within the Java class library is covered—this is essential for the accuracy of our metrics. Each metric can be computed for a given workload application using a single script invocation. The execution produces a trace, whose contents vary according to the metric being computed. A separate postprocessor script uses the trace to calculate the metric’s value. This separation is useful because in some cases, multiple metrics can be computed from the same trace; several of our metrics exploit this, as we explain shortly (Section 3.2).

Because all instrumentation is carried out using the same high-level domain-specific language (DiSL), our implementations are amenable to customization with relatively low familiarization overhead. We envisage they can usefully be tweaked and extended for specific needs, such as dumping the trace in a different format or adding a custom online analysis. A subset of our metrics are query-based, and these offer an additional level of customizability, because custom queries can be written in the high-level XQuery language.

The tools in our toolchain exhibit acceptable runtime overhead. Among the most heavyweight of our tools is JP2, which produces calling-context trees; this incurs an overhead factor of roughly 100 [16]. However, this cost is amortized in that many different metrics are computed (as queries) over its output. Object lifetime analysis also relies on heavyweight instrumentation. However, other tools instrument considerably fewer events—for example, hash-code analysis instruments only a few method entries—and incur correspondingly less overhead. We note that our instrumentation-based approach generally outperforms like-for-like metric implementations using the older JVM profiling interface, including those described by Dufour et al. [12].

Metrics such as field immutability, zeroing, field sharing, and using of identity hash codes are collected via custom tools that use DiSL to perform bytecode instrumentation. In each case, the instrumentation maintains shadow state [31] for each object. Depending on the analysis, different events are intercepted, and different information is stored in a shadow state. For example, to characterize immutability, our shadow object keeps track of all field accesses to the underlying object, according to a state machine. Each shadow object records the class name, object allocation site, and an array of field states. Each field can be in one of the following three states: (i) virgin, if a field was neither read from nor written to; (ii) immutable, if a field was only read or written to inside

§https://developer.mozilla.org/en-US/docs/SpiderMonkey/Internals/Tracing_JIT.
the dynamic extent of its owner object’s constructor; and (iii) *mutable*, if a field was ever written to outside the dynamic extent of a constructor. The transitions between the states are illustrated in Figure 2.

A suitably modified version of this shadow object approach is used in field sharing, field synchronization, and hash code analyses (storing counters for thread accesses, counters for monitor ownership, and counters for executions of `Object.hashCode()` and `System.identityHashCode()` methods, respectively).

### 3.2. Query-based metrics

Many of our metrics are defined as queries over trace data. Specifically, these are metrics concerning instruction mix, call-site polymorphism, stack usage and recursion depth, argument passing, method and basic block hotness, and use of boxed types. All these metrics are obtained using JP2 [16, 32], which has been reimplemented using the DiSL instrumentation framework to fit well into our framework.

JP2 is a calling-context profiler that produces execution traces in the form of an annotated calling-context tree (CCT). Each node in a CCT corresponds to a particular callchain and keeps the dynamic metrics, such as the number of method invocations and the number of executed bytecodes. JP2 is call-site aware, meaning that different call sites in the same method are distinguished even if their target method is the same. Unlike many other profilers, JP2 performs both inter-procedural and intra-procedural analyses and reports dynamic execution counts for each basic block of code in methods.

JP2 provides a complete profile for the entire execution, including coverage of the Java class library and limited coverage of the native code. Although native methods do not have any bytecode representation, JP2 uses JVM tool interface’s native method prefixing feature to insert bytecode wrappers for each native method. Control flow within native methods is covered only from the points where these call back into Java code or other prefixed natives.

Some of the information needed for calculating our metrics is not stored in the CCT profile but depends on static properties of the application classes. For this, we use another feature of JP2, which allows storing the bytecode of all classes loaded during execution. These classes are then converted to an XML representation to allow querying alongside the CCT data [16]—many of our queries make use of the ability to cross-reference between the CCT and the class data.

Obtaining dynamic metrics using JP2 is a three-step process. In the first step, the application is executed under the JP2 profiler. As the JVM loads the application classes, JP2 instruments them for profiling and stores the original classes. During application execution, the inserted profiling
Figure 3. An excerpt from an XQuery script for identifying methods with the hottest basic blocks. The ‘dynamic call graph’ (dcg) refers to the calling-context tree with profiling data.

declare variable $dcg := doc($dcg-xml)/dcg:dynamicCallGraph;

declare variable $benchmark-methods := $dcg/dcg:method[dcg:executionCount gt 0];
declare variable $basic-blocks := {
    for $b in $benchmark-methods/dcg:basicBlock
    order by xs:long($b/dcg:executionCount) descending
    return $b
};

declare variable $basic-block-sizes := {
    for $b in $basic-blocks
    return number($b/@endInstruction) - number($b/@startInstruction) + 1
};

declare variable $basic-block-sizes-relative := {
    let $sizes-total := sum($basic-block-sizes)
    for $i in $basic-block-sizes
    return $i / $sizes-total
};

declare variable $basic-block-executions := {
    for $i in 1 to count($basic-blocks)
    return number($i/dcg:executionCount) * $basic-block-sizes[$i]
};

declare variable $basic-block-executions-relative := {
    let $executions-total := sum($basic-block-executions)
    for $i in $basic-block-executions
    return $i / $executions-total
};

Figure 3. An excerpt from an XQuery script for identifying methods with the hottest basic blocks. The ‘dynamic call graph’ (dcg) refers to the calling-context tree with profiling data.

code collects the CCT data, and when the application terminates, JP2 stores the resulting profile in an XML file. In the second step, the bytecode of the classes loaded by the application is used to produce an XML file with static class information. Finally, in the third step, the desired metrics are calculated using the CCT profile and the static class information as input. Using XML allows using off-the-shelf tools such as XQuery for calculating the metrics defined as queries over the CCT and class information documents.

Figure 3 shows an excerpt from an XQuery script used for identifying methods with the hottest basic blocks. It can be useful for finding methods with rich intra-procedural control flow but with low method execution counts that cannot be spotted by common profilers. The part of the script shown in the example is responsible for calculating the relevant basic block metrics. The algorithm is straightforward: from all methods executed at least once, collect all basic blocks in descending order of their execution count. Then for each basic block, calculate its absolute size (number of instructions), relative size (fraction of the total number of instructions), the number of executed instructions (size multiplied by the number of basic block executions), and the relative number of executed instructions (fraction of the total number of executed instructions). The result is then output (not shown) to enable identification of the methods with the hottest basic blocks—each basic block has a unique identifier that can be traced back to the containing method.

A key benefit of the query-based design is that custom queries can be used to formulate previously unanticipated metrics. For example, the dumped CCTs contain enough information to recover a $k$-calling-context forest, which offers an alternative ($k$-bounded) level of context sensitivity offering advantages in certain scenarios [33].

The separation between the profile data and the queries avoids potential problems with nondeterminism. Different metrics can be computed without the need for repeated application runs, hence avoiding any risk of divergent behavior across such runs. However, this comes at a cost of having only one application run. Collecting data from multiple runs is possible, but given the overhead of the toolchain and the amount of data that needs to be processed, we do not expect this to be a common use case.
4. EXPERIMENTAL SETUP

To obtain information necessary to answer the research question stated earlier, we used our toolchain to collect the presented dynamic metrics. We analyzed the collected data, looking for significant differences between the pure Java workloads and the workloads induced by JVM languages, which may hint at optimization opportunities for programs executing on the JVM. Here, we discuss the workloads and the measurement context of the experiments.

4.1. Workloads

The lack of an established benchmark suite—something akin to the DaCapo [34] or SPECjvm2008 [35] suites, but for JVM languages other than Java—is a widely recognized issue. A new benchmark suite has recently been proposed for Scala [5], but there is no such suite for most dynamic languages, including those we consider here.

The closest to a benchmark suite for dynamic languages is the Computer Language Benchmarks Game (CLBG) project [36], which collects and compares performance results for various benchmarks implemented in many different programming languages, including Java, Clojure, Python, Scala, JavaScript, and Ruby, which we chose for the comparison. For each benchmark in the suite, there is a prescribed algorithm, which is then implemented in an idiomatic way in each language. Considering their size and focus on algorithms, the CLBG benchmarks clearly fall into the category of microbenchmarks and only represent a certain aspect of real-world applications. The authors of the CLBG project are well aware of this and explicitly warn against jumping to conclusions regarding performance of real-world applications based on the benchmark results.

However, for lack of any better multi-language benchmark suite, the CLBG project remains a popular source of rough estimates of raw performance achievable by many programming languages. Recently, benchmarks from the CLBG project have been used in a study on performance differences between Python compilers [3], while Li et al. [37] published an exploratory study characterizing workloads for five JVM languages using a selection of benchmarks from the CLBG project, complemented by several larger application benchmarks.

The CLBG benchmarks not only have an obvious utility for cross-language comparisons but also present a potential threat to validity of any study. Considering this threat in the context of our study, we argue that the threat is already significantly diminished by the fact that we do not intend to use the benchmarks for evaluation or comparison of raw performance of the JVM languages. Instead, we are interested in the character of the corresponding workload resulting from the bytecode generated by a JVM language compiler. We expect that even for small benchmarks implemented using dynamic JVM languages, the generated bytecode implementing the semantics of the corresponding JVM language will differ significantly from typical Java bytecode. The generated code will also rely on some kind of runtime library, thus inflating the amount of JVM language-specific code executed by the JVM.

In choosing the workloads for our study, we therefore decided to adopt the approach of Li et al., which further mitigates the potential threat to validity by including several real-world applications in the workload selection. We use the same CLBG benchmarks as Li et al. (with the exception of the meteor-contest benchmark), thus providing complementary results for the same workloads. We list the used CLBG benchmarks in Table II, along with a brief description and the inputs used.

With respect to real-world application workloads, our selection differs from that of Li et al. for various reasons—mainly related to our familiarity with the applications and their suitability for automated experiment environment. Our final workload selection contains three real-world applications (listed in Table III) for each of the JVM languages considered. The avrora, eclipse, fop, and jython benchmarks come from the DaCapo suite [34], while appart, factorie, and scalac come from the Scala benchmarking suite [5]. The deltablue, raytrace, and Richards benchmarks come

---

1) jython from the Dacapo benchmarking suite is the only non-Java benchmark, which runs Python microbenchmarks.
2) http://benchmarksgame.alioth.debian.org/dont-jump-to-conclusions.php.
Table II. Benchmarks from the CLBG project (implemented in Java, Clojure, Ruby, Python, Scala, and JavaScript) selected for workload characterization.

| Benchmark     | Description                                   | Input          |
|---------------|-----------------------------------------------|----------------|
| binarytrees   | Allocate and deallocate many binary trees     | 16             |
| fannkuch      | Repeatedly access a tiny integer-sequence     | 10             |
| fasta         | Generate and write random DNA sequences       | 150,000        |
| knucleotide   | Repeatedly update hashtables and knucleotide strings | fasta output   |
| mandelbrot    | Generate a Mandelbrot set and write a portable bitmap | 1000           |
| nbody         | Perform an nbody simulation of the Jovian planets | 500,000       |
| regexdna      | Match DNA 8-mers and substitute nucleotides for IUB code | fasta output   |
| revcomp       | Read DNA sequences and write their reverse complement | fasta output   |
| spectralnorm  | Calculate an eigenvalue using the power method | 500            |

IUB: International Union of Biochemistry.

Table III. Real-world applications selected as benchmarks to complement the CLBG benchmarks for workload characterization.

| Application     | Description                                   | Input          |
|-----------------|-----------------------------------------------|----------------|
| Clojure          |                                               |                |
| clojure-script  | A compiler for Clojure that targets JavaScript | twitterbuzz source |
| clj-pdf         | A library for generating PDFs from Clojure    | default example |
| frak            | A library for transforming strings into regular expressions | default example |
| Java            |                                               |                |
| avrora          | A simulator of programs running on a grid of AVR microcontrollers | DaCapo default |
| eclipse         | An integrated development environment         | DaCapo default  |
| fop             | An output-independent print formatter         | DaCapo default  |
| JavaScript       |                                               |                |
| deltablue       | A one-way constraint solver                   | Octane default  |
| raytracer       | A ray tracer benchmark                        | Octane default  |
| Richards        | An operating system kernel simulation benchmark | Octane default  |
| Ruby            |                                               |                |
| opal            | A Ruby to JavaScript compiler                 | meteor source  |
| rubyflux        | A Ruby to Java compiler                       | meteor source  |
| voodoo          | Image manipulation library for Ruby           | simple image   |
| Jython          |                                               |                |
| jython          | An interpreter of Python running on the JVM   | DaCapo default  |
| minilight       | A minimal global illumination renderer         | default example |
| scavis          | A scientific computation environment          | Matrix multiplication |
| Scala           |                                               |                |
| apparat         | Framework to optimize ABC, SWC, and SWF files | Scalabench default |
| factorie        | Toolkit for deployable probabilistic modeling | Scalabench default |
| scalac          | Compiler for the Scala 2 language             | Scalabench default |

CLBG: Computer Language Benchmarks Game.

from the Octane benchmarking suite [38]. The remaining benchmarks, clj-pdf [39], frak [40], minilight [41], scavis [42], rubyflux [43], voodoo [44], opal [45], and clojure-script [46], are open-source projects from GitHub. These workloads lack the nice property of being idiomatic implementations of the same task but serve as a sanity check for the results obtained using the CLBG benchmarks.

4.2. Measurement context

All metrics were collected with Java 1.6, Clojure 1.5.1, JRuby 1.7.3, Rhino 1.7, Scala 2.10.3, and Jython 2.7 runtimes, yielding a total of 72 different language benchmark combinations, all executed using the OpenJDK 1.6.0_27 Java runtime environment running on Ubuntu Linux 12.0.4.2. Because of the high number of combinations and extensive duration of the experiments, we have not included
different language runtimes among independent variables. Similarly, we have not varied the execution platform, because the metrics are defined at the bytecode level and can be considered largely JVM** and platform independent.

5. EXPERIMENTAL RESULTS

While we have collected the full range of dynamic metrics for different workloads, in this article, we only discuss metrics that show striking difference between static and dynamic languages: call-site polymorphism; field, object, and class immutability; object lifetimes; unnecessary zeroing; and identity hash-code usage. Thanks to our toolchain, we were able to collect metrics that cover both the application (and the corresponding language runtime) and the Java class library (including any proprietary JVM vendor-specific classes) on a standard JVM. All the collected metrics are dynamic (i.e., metrics describe characteristics of a running application).

5.1. Call-site polymorphism

Hot polymorphic call sites are good candidates for optimizations such as inline caching [47] and method inlining [48], which specialize code paths to frequent receiver types or target methods. In the case of dynamic languages targeting the JVM, specialization is considered to be one of the most beneficial performance optimizations [3]. In our study, we collected metrics that are indicative specifically for method inlining, which removes costly method invocations and increases the effective scope of subsequent optimizations.

The results consist of two sets of histograms for each language, derived from the number of target methods and the number of virtual method invocations made at each (potentially) polymorphic call site during the execution of each workload. The plots in Figures 4–6 show the number of call sites grouped by the number of targeted methods (x-axis), with an extra group for call sites targeting 15 or more methods. The plots in Figures 5–7 then show the actual number of virtual method invocations performed at those call sites.

We observe that polymorphic invocations in the CLBG benchmarks target no more than six methods in the case of Java and no more than 11 methods in the case of Scala (with the majority below six). This is not surprising, given the microbenchmark nature of the CLBG workloads. The situation is vastly different—and more realistic—with the real-world applications. Nevertheless, most of the call sites only had a single target—98.2% in the case of Java (accounting for 90.8% of all virtual method invocations) and 97% in the case of Scala (accounting for 80% of all virtual method invocations).

The microbenchmark nature of the CLBG workloads is much less pronounced (compared with the real application workloads) in the case of dynamic JVM languages. This suggests that even the CLBG workloads do exhibit some of the traits representative of a particular dynamic language and that the infrastructure and library code is exercised even when executing a microbenchmark.

The results for the Clojure workloads show that polymorphic invocations target one to 10 methods, with an average of 99.3% of the call sites (accounting for 91.2% of all virtual method invocations) actually targeting a single method. The results for the Jython workloads show that polymorphic invocations mostly target one to 10 methods, with a small number of sites targeting 15 or more methods. Invocations at such sites are surprisingly frequent, but still, 98.7% of the call sites (accounting for 91.7% of all virtual method invocations) target a single method.

Interestingly, JavaScript stands out from the rest of the dynamic languages, and the results look similar to those of Scala—the number of targets in both languages is similar, but JavaScript workloads perform more calls at call sites with lower number of targets. Also, the difference between microbenchmarks and real-world applications is not as pronounced as in the case of Scala. On average, 98% of the call sites (accounting for 86% of all virtual method invocations) target a single method.

**Different vendors may provide different implementation of the Java Class Library as well as other proprietary classes in their JVM.
Figure 4. The number of dynamically dispatched call sites targeting a given number of methods for the Computer Language Benchmarks Game in Clojure, Java, JavaScript, JRuby, Jython, and Scala.
Figure 5. The number of dynamically dispatched calls made at call sites with a given number of targets for the Computer Language Benchmarks Game for Clojure, Java, JavaScript, JRuby, Jython, and Scala.
Finally, the results for JRuby show that there is only a slight difference between the CLBG benchmarks and the real-world applications. For both kinds of workloads, there is a significant number of call sites with 15 or more targets. Interestingly, the number of calls made at these call sites is surprisingly high—comparable with the other call sites. A cursory inspection revealed that these are methods of the JRuby runtime. However, on average, 98.4% of the call sites (accounting for 88% of all virtual method invocations) target a single method.
5.2. Field, object, and class immutability

In our study of the dynamic behavior of JVM workloads, we use an extended notion of immutability instead of the ‘classic’ definition: an object field is considered immutable if it is never written to outside the dynamic extent of that object’s constructor. This notion is dynamic in the sense that it holds only for a particular program execution or for a specific program input [4, 6].

Extending this notion to objects and classes, we distinguish (i) per-object immutable fields, assigned at most once during the entire program execution; (ii) immutable objects, consisting only of immutable fields; and (iii) immutable classes, for which only immutable objects were observed.

The results shown in Figure 8 indicate that there is a significant fraction of immutable fields (as per our definition) in most of the studied workloads, without significant differences between the CLBG and real-world benchmarks. Except in the Java binarytrees CLBG workload, we observed more than 50% of immutable fields in all benchmarks, with Clojure having the highest average number of immutable fields. The binarytrees benchmark allocates and deallocates many binary trees, constructing them in a bottom-up fashion and traversing them in a top-down fashion. Most of the time, the benchmark only accesses the three fields found in the representation of a tree node—the references to the left and right child nodes and the (integer) value. The tree node class is a pure data class, with its fields initialized outside the constructor. Therefore, the amount of immutable fields for the Java version of the binarytrees benchmark is considerably lower compared with other languages. An interesting observation is also that the fractions of immutable reference instance fields for the JavaScript benchmarks are considerably lower compared with other languages. This happens mainly for benchmarks that perform numeric computations, which suggest that the JavaScript runtime creates a huge number of value-type instances with primitive value fields to represent the numbers. Consequently, the proportion of the reference instance fields becomes very small overall. This is also supported by cross-checking with the results in Figure 9 (discussed next).

At the granularity of objects, the results in Figure 9 show varying immutability ratios across different workloads. Apart from few exceptions, the ratios are consistently high, especially for the

![Image of Figure 8](image-url)
Figure 9. Fractions of immutable objects.

Figure 10. Fractions of immutable classes.
dynamic languages (mostly over 50%), with Clojure, JavaScript, and JRuby scoring almost 100% on five workloads (with four workloads common to Clojure and JRuby). This can be attributed to the large amount of boxing and auxiliary objects created by the language runtimes [37]. In the case of Java, the reason for basically no immutable object instances in the binarytrees benchmark is the same as for the low fraction of immutable reference fields discussed earlier. In the case of Clojure, the spectralnorm benchmark exhibits a surprisingly low amount of immutable objects compared with other benchmarks. This is due to extensive usage of the java.lang.reflect.Method.copy() method (92% of all method calls), which creates a copy of the Method class for each reflective invocation of the static sqrt() method in the java.lang.Math class. One of the fields of the Method class is mutated after initialization, resulting in an excessive amount of mutable objects unrelated to what the benchmark actually does.

Finally, at the granularity of classes, the results in Figure 10 show consistent fractions of immutable classes across different workloads (except for Jython), with significant differences between the languages. These systematic differences can be attributed both to different coding styles typical for the respective languages and to the number of helper classes produced by a particular dynamic language runtime environment.

5.3. Identity hash-code usage

The JVM requires every object to have a hash code. The default implementation of the hashCode method in the Object class uses the System.identityHashCode method to ensure that every object satisfies this requirement. The computed hash code is usually stored in the object header, which increases memory and cache usage—JVMs therefore tend to use an object’s address as its implicit identity hash code and store it explicitly only upon first request (to make it persistent in the presence of a copying GC that moves objects around).

Figure 11. Fraction of objects hashed using an overridden hashCode method (■), the identityHashCode method (□), or both (■), along with an average number of hash operations per object.
That said, performance may be improved by allocating the extra header slot either eagerly or lazily, depending on the usage of identity hash codes in a workload. Because systematic variations in hash-code usage were identified between Java and Scala workloads [6], we also analyzed the usage of hash codes for the workloads in our study.

The results shown in Figure 11 suggest that the identity hash code is never requested for a vast majority of objects. Despite a comparatively frequent usage of hash code in the Java and Scala workloads, the use of identity hash code (which requires storing the hash code in the object header) remains well below 2% for most workloads. The workloads implemented using the dynamic languages appear to be using hash code very little, with the exception of the knucleotide benchmark in the cases of Jython and JavaScript. Clojure shows the highest use of hash code among dynamic languages, in contrast to JRuby and JavaScript. Among the real-world benchmarks, only the eclipse and clojure-script benchmarks had objects on which both an overridden hashCode and the identityHashCode methods were invoked.

The increased use of hash codes in the knucleotide benchmark is due to it typically being implemented using hash tables, which are the primary users of the hashCode method. Both the Java and Scala versions of the knucleotide benchmark override the hashCode method. While the Java version uses the standard HashMap class, resulting in the hashCode method being used on 31.5% of objects (with an average of 5.8 hash operations per object), the Scala version uses a Scala-specific hash map implementation, resulting in lower number of hash operations. In the cases of JavaScript and Jython, the hashCode method is overridden by the language runtime, which uses the standard HashMap implementation to implement associative data structures.

Because dynamic languages appear to produce many short-lived objects (cf. Section 5.5), the results suggest that object header compression with lazy identity hash-code slot allocation is an adequate heuristic for the dynamic language runtimes, especially for JRuby and JavaScript.
5.4. **Unnecessary zeroing**

The Java language specification requires that all fields have a default value of `null`, `false`, or `0`, unless explicitly initialized. The explicit zeroing may cause unnecessary overhead [25], especially in the case of fields assigned (without being first read) within the dynamic extent of a constructor—the uninitialized values of such fields can be observed neither by the program nor the GC. Our analysis detects and reports such fields.

A JVM could potentially try to optimize away some of the initialization overhead, for example, by not zeroing unused fields where appropriate, but it would have to ensure that the (uninitialized) field values are never exposed to the GC. While such an optimization can make the results of our analysis less accurate, trying to detect and correctly handle the effect of this optimization in our analysis would be difficult and possibly require making it JVM specific, which is what we want to avoid. We therefore do not take these potential optimizations into account and consider zeroing of an unused field to be mandatory.

Figure 12 shows the amount of unnecessary zeroing (according to our metric) performed by the workloads in our study. For the CLBG benchmarks, Clojure exhibits the highest average percentage of unnecessary zeroing (85.9%), followed by JavaScript (77.8%), Scala (71.74%), Jython (66.4%), JRuby (46.04%), and Java (49.03%). Interestingly, this language ordering appears to partially correlate with the ordering imposed by the percentage of immutable instance fields (shown in Figure 8) with average values of 91.5% for Clojure, 90.8% for JavaScript, 89.2% for JRuby, 86.8% for Jython, 83.9% for Scala, and 73.4% for Java. Our results therefore suggest that the amount of unnecessary zeroing increases with the amount of immutable instance fields. While this may not come as a surprise, it shows there is potential for optimization regarding immutable fields or immutable classes. Using immutable classes in multi-threaded programs is already a recommended practice to avoid locking issues, and eliminating the needless work can make the use of immutable data types less costly.

5.5. **Object lifetimes**

Object sizes and lifetimes characterize a program’s memory management ‘habits’ and largely determine the GC workload because of the program’s execution. To approximate and analyze the GC behavior, we used ElephantTracks [49] to collect object allocation, field update, and object death traces, and run them through a GC simulator configured for a generational collection scheme with a 4-Mib nursery and 4-Gib old generation. None of the microbenchmarks allocated enough memory to trigger a full heap (old generation) collection.

The results are summarized in Table IV. The *mark* column contains the number of times the GC marked an object live, the *cons* column contains the number of allocated objects, and the *nursery survival* column contains the fraction of allocated objects that survive a nursery collection.

The most striking difference is the number of objects allocated by the dynamic language CLBG benchmarks compared with their Java and Scala counterparts—in all of them, the Java and Scala workloads allocate at least one order of magnitude less objects and in some cases, even several orders less. The results for the statically typed languages, such as Java and Scala, are not too surprising—given the microbenchmark nature of the CLBG workloads—but they indicate how inherently costly the dynamic language features are in terms of increased GC workload.

The plots in Figure 13 show the evolution of the object survival rate plotted against logical time expressed as cumulative memory allocated by a benchmark. In the cases of Java and Scala, the fannkuch, fasta, mandelbrot, nbody, revcomp, and spectralnorm benchmarks are not shown in the plots, because they allocate less than 1 Mib. In the case of Java, this also applies to the regexmdma benchmark. The results show that the workloads written in the dynamic languages allocate much more objects than their counterparts written in the static languages. However, most of the objects die young, suggesting that they are mainly temporaries resulting from features specific to dynamic languages.
Table IV. Garbage collector workload.

| Benchmark  | Clojure | JRuby | Clojure | JRuby |
|-----------|---------|-------|---------|-------|
| Mark Cons | Nursery survival (%) | Mark Cons | Nursery survival (%) |
| Mark Cons | Nursery survival (%) | Mark Cons | Nursery survival (%) |
| binarytrees | 4021096 | 147910977 | 0.03 | 2.72 | binarytrees | 30303133 | 118527001 | 0.26 | 7.40 |
| fannkuch | 38900 | 317062 | 0.12 | 12.27 | fannkuch | 263722 | 10041379 | 0.03 | 2.63 |
| fasta | 122560 | 2721195 | 0.05 | 4.50 | fasta | 513005 | 27557549 | 0.02 | 1.86 |
| knucleotide | 1158380 | 18436145 | 0.06 | 6.28 | knucleotide | 1788684 | 19762437 | 0.09 | 9.05 |
| mandelbrot | 134617 | 2822088 | 0.05 | 4.77 | mandelbrot | 81977235 | 200409954 | 0.41 | 1.81 |
| regexdna | 197257 | 43769 | 0.34 | 33.65 | regexdna | 148158 | 399044 | 0.37 | 37.13 |
| revcomp | 197257 | 43769 | 0.34 | 33.65 | revcomp | 109951 | 635720 | 0.17 | 5.70 |
| spectralnorm | 563338 | 5592427 | 0.10 | 10.07 | spectralnorm | 9262955 | 11332054 | 0.08 | 1.93 |
| clojure-script | 8658091 | 2965612 | 0.28 | 28.24 | clojure-script | 382540 | 2632867 | 0.15 | 14.53 |
| Java | | | | |
| Mark Cons | Nursery survival (%) | Mark Cons | Nursery survival (%) |
| Mark Cons | Nursery survival (%) | Mark Cons | Nursery survival (%) |
| binarytrees | 1136479 | 29581095 | 0.04 | 3.84 | binarytrees | 73870068 | 297905683 | 0.25 | 3.22 |
| fannkuch | 0 | 2226 | 0.00 | 0.00 | fannkuch | 44953062 | 80129997 | 0.03 | 2.87 |
| fasta | 0 | 2329 | 0.00 | 0.00 | fasta | 340799 | 311487 | 0.11 | 30.82 |
| knucleotide | 962320 | 967492 | 0.99 | 99.47 | knucleotide | 12538292 | 40411959 | 0.31 | 31.03 |
| mandelbrot | 0 | 3699 | 0.00 | 0.00 | mandelbrot | 2301641 | 80129997 | 0.03 | 2.87 |
| regexdna | 1295 | 2848 | 0.48 | 48.00 | regexdna | 58047534 | 227296897 | 0.26 | 3.78 |
| revcomp | 1158 | 2848 | 0.41 | 40.66 | revcomp | 183874 | 1012778 | 0.18 | 18.16 |
| spectralnorm | 59684 | 2075466 | 0.03 | 2.88 | spectralnorm | 6131554 | 140908487 | 0.04 | 3.29 |
| avora | 17176557 | 66569509 | 0.25 | 30.82 | avora | 30596 | 43752983 | 0.24 | 24.35 |
| clojure-script | 582276 | 1715457 | 0.33 | 33.00 | clojure-script | 138121 | 2413103 | 0.05 | 5.70 |

6. INLINING IN THE JVM

Method inlining improves performance of programs by removing the overhead of method calls. By expanding the scope for analysis and optimizations in the caller, inlining increases the potential for intra-procedural optimizations and specialization of the code being inlined. In a JVM, inlining is performed by the JIT compiler. Given the importance of inlining, we investigate the ability of the HotSpot JVM to perform inlining with bytecode originating from different JVM languages.

The number of inlining opportunities is influenced by two factors: the structure of the application that is running and the bytecode that the language compiler generates for the application. Some languages encourage expressing algorithms as a large number of fine-grained units of code, while other languages usually have larger methods. While some compilers generate code that uses many small helper methods, and therefore has many inlining opportunities, other compilers will perform a certain amount of inlining before creating bytecode††.

In this section, we first present an overview of the inlining decisions made by the compiler—a result of studying the source code of the server JIT compiler in the HotSpot JVM. Then, having modified the compiler to collect traces of actual inlining decisions while the JVM was

††These effects can work against each other: while Scala code usually consists of many small methods, the Scala compiler will mitigate some of the effects by performing inlining.

*The value is missing because the tool for collecting object lifetimes did not terminate with this benchmark.

© 2015 The Authors. Software: Practice and Experience Published by John Wiley & Sons Ltd. DOI: 10.1002/spe
executing the workloads from our study, we present metrics calculated using these traces. The metrics provide insights into differences between JVM languages that affect inlining and include the fraction of inlined and not inlined methods, the fraction of deoptimized methods, the distribution of inlining levels, and the distribution of top 10 reasons for which inlining did and did not happen.

6.1. Inlining decisions

While inlining can facilitate dramatic performance increases, it also results in significant duplication of code, which may limit or negate the potential performance benefits:

- The size of the compiled methods will increase significantly, increasing memory requirements.
- CPU instruction caches will be polluted with multiple versions of the same method, taking up space that may be needed by other hot methods, increasing cache contention.
- Compilation time increases because of the larger compilation scope, delaying the deployment of the optimized code.
A JIT compiler therefore needs to constantly make decisions whether to inline or not to inline a particular method, balancing the costs and benefits of inlining at each call site. Because it is not possible to determine the optimal set of inlined call sites, compilers employ complex heuristics to make the inlining decisions. The heuristic used by the HotSpot server compiler, extracted from the compiler source code in form of a decision tree, is shown in Figure 14. The decision tree is basically an expert system applied to static and dynamic information about the code being executed, which tells the compiler whether to inline or not to inline a method. When considering a specific call site, the decision procedure can be represented as a traversal of the decision tree, starting from the root and branching depending on the conditions until a leaf node representing a decision is reached. In the HotSpot JVM, the decision procedure is implemented in conditional code scattered among multiple source files and a number of methods. We use the decision-tree abstraction to present the decision procedure as a whole, rather than fragments corresponding to individual methods.

The compiler first checks whether the target method is a compiler intrinsic—a special method that is always inlined. The server compiler differentiates between system intrinsics and method-handle intrinsics. In the next step, the compiler checks the usage of strict floating-point arithmetics. If either the caller or the callee requires it but the other does not, the call cannot be inlined.

If the call site is polymorphic and the call needs dispatching, the compiler consults the dynamic profiling information to determine whether there is a prominent target for that call site, that is, whether the call at this particular call site has exactly one or two receivers, or whether there is a major (more frequent) receiver when there are more than two potential targets. If there is no prominent target for this particular call site, the call cannot be inlined and has to undergo virtual method dispatch. Otherwise, the call site is considered for inlining but is subjected to additional checks to avoid inlining in situations in which the potential performance benefits may not materialize or may be negated by the adverse effects of code duplication.

If the call site was not encountered often enough during profiling, that is, the call site is considered cold, or if the size of the target method exceeds a certain limit, the decision will be not to inline. These two checks will be skipped if the call site is forcibly inlined or if the call site received many thrown exceptions. Afterwards, simple accessor methods will be always inlined, while for other methods, all of the following conditions must hold for them to be inlined:

- The size of the caller method (in terms of the number of bytecode instructions) must be below a certain limit so that it does not qualify as a ‘giant method’.
- The call site needs to have been reached during profiling.
- Inlining must not be disabled.
- The current number of nested inlining scopes must be below a certain limit.
- The current number of nested recursive inlining scopes must be below a certain limit.
- The sum of sizes (number of bytecode instructions) of all inlined methods must be below a certain limit.

To capture the inlining decisions taken during execution, we modified the compiler to collect the inlining decisions for each call site. In the following two sections, we summarize the main reasons for inlining and not inlining methods for each JVM language and workload in our study. Each decision basically represents a path from the root of the decision tree to one of its leaves. To explain why a particular decision was made, we encode this path using a sequence of mnemonic codes that capture the outcome of the conditionals along the path. Each mnemonic code starts with a capital letter so that multiple codes in a sequence can be easily distinguished. Table V summarizes the individual mnemonic codes and needs to be consulted when interpreting the results presenting inlining decisions. For example, a combination of mnemonic codes reading ‘SSmFTsA’ encodes a positive inlining decision because of the inlinee being an accessor method (A), in addition to being a synthetic (S), static monomorphic (Sm), and frequently called (F) method of trivial size (Ts).

### 6.2. Top reasons for inlining methods

We first consider the positive inlining decisions. Figure 15 shows the distribution of reasons that account for 90% of methods that were inlined. Each color denotes a distinct decision, comprising a...
Figure 14. The inlining decision procedure of the HotSpot just-in-time compiler.
Table V. Codes corresponding to reasons for decisions to inline or not to inline methods.

| Code | Description                   | Code | Description                        |
|------|-------------------------------|------|------------------------------------|
| Cc   | Cold call site                | Ctb  | Inlinee already compiled to big method |
| Pb   | Profiled bimorphic            | Ctm  | Inlinee already compiled to medium method |
| S    | Synthetic method              | Ie   | Inlinee was executed infrequently  |
| Sm   | Static monomorphic            | Mrl  | Maximum recursive inlining level reached |
| Ts   | Trivial size                  | Ml   | Maximum inlining level reached      |
|      |                               | N    | Inlinee is native method            |
|      | Reasons for not inlining      |      |                                    |
| A    | Inlinee is accessor method    | Nmm  | No major megamorphic receiver       |
| F    | Inlining frequently called method | Ne   | Inlinee was never executed         |
| Mhi  | Ininlinee is method-handle intrinsic | Tbc  | Inlinee too big for cold call site  |
| Nf   | Inlining non-frequent method  | Tbh  | Inlinee too big for hot call site   |
| Si   | Inlinee is system intrinsic  | Uc   | Unreached call site                 |

Figure 15. Breakdown of reasons for inlining that account for 90% of inlined methods.

combination of reasons. For the sake of readability, we filtered out reasons that account for less than 2%, which turned out to be especially pronounced in the case of Clojure. To interpret the results, we cross-reference the legend in Figure 15 with Table V.

We observe that in the case of Java, the results for the CLBG workloads are so varied and irregular that it is impossible to come up with a meaningful interpretation. This is because the CLBG benchmarks are too small for JVM-native language. The value for the revcomp benchmark is missing, because it only contains two compiled methods: one of which was deoptimized and the other was never inlined. The situation improves significantly with the real-world workloads, where most inlining applies to frequently called (hot) static methods (both normal and trivial sized).

In the case of Scala, also a statically typed language, the results for the CLBG workloads appear to suffer from the same problem as in Java—the benchmarks being too small to really exercise the language runtime. Still, they are partially consistent with the results for the real-world workloads. In contrast to Java, we note much more trivial-sized static methods being inlined.
In the case of Clojure, we observe inlining of a large number of static accessor-style methods and system intrinsics. The CLBG workloads appear similar to the real-world workload, except for higher number of inlined system intrinsic methods.

In the case of JavaScript, the results for the CLBG and the real-world workloads are consistent, which suggests that a significant amount of JavaScript runtime code becomes executed even with microbenchmarks. Most of the inlined methods are static (both normal and trivial sized), with inlining performed surprisingly even for significant number of cold call sites and methods. A small number (but more than with the other languages) of inlined methods were actually virtual but profiled monomorphic or bimorphic.

In the case of JRuby, we can again observe a certain similarity between the CLBG and the real-world workloads, indicating that significant amount of framework code becomes executed even with the microbenchmarks. Interestingly, the CLBG workloads exhibit inlining of a significant number of synthetic method-handle intrinsics compared with the real-world workloads, which we attribute to the smaller size of the CLBG workloads. The real-world workloads exhibit a striking similarity among themselves, despite representing rather different tasks (compiler versus image manipulation).

The results for Jython are similar to JavaScript, which is interesting, because unlike JavaScript, Jython is an interpreter. In contrast to JavaScript, there are more inlined system intrinsic and static accessor methods. As in the case of JavaScript, the results for the CLBG and the real-world workloads are rather similar, indicating that significant amount of framework code becomes executed even with microbenchmarks.

### 6.3. Top reasons for not inlining methods

We now consider the negative inlining decisions. Figure 16 shows the distribution of reasons that account for 90% of methods that were not inlined. To interpret the results, we again cross-reference the legend in Figure 16 with Table V.

The results for Java again confirm the fact that the CLBG benchmarks are too small for a JVM-native language. The value for the fannkuch is missing, because there has been only one deoptimized method, while in the case of the spectralnorm benchmark, there were four methods in total that were

![Figure 16. Breakdown of reasons for not inlining that account for 90% of non-inlined methods.](image-url)
inlined. The real-world workloads all show a moderate amount of cold call sites where the inlinee was too big to be inlined. In the case of the avror benchmark, many potential inlineses were excluded because they were never executed before. The other two real-world workloads show many methods that were excluded because they have already been compiled to medium-sized or big methods.

The results for Scala again exhibit certain similarity to Java, because of the results for the CLBG workloads significantly deviating from the results for the real-world workloads. The prevailing reason for not inlining methods in the CLBG workloads was that the methods were too big for a cold call site. The reasons become more diverse but somewhat more consistent for the real-world workloads. In addition to methods being too big to inline, we can observe the appearance of polymorphic call sites without a major receiver.

In the case of Clojure, many methods were too big or cold, in addition to being considered at cold call sites. In most workloads, there were polymorphic call sites without a major receiver as well as a significant number of unreached call sites. The latter is caused by the compiler generating code to handle special cases that never occurred at runtime. There are no striking differences between the CLBG and real-world workloads.

In the case of JavaScript, methods were not inlined mainly because they were too big and considered at cold or even unreached call sites. Overall, the results appear qualitatively similar to Closure, except for the different proportions. In contrast to Clojure, there were not many unreached call sites at which hot methods were considered for inlining. The results for the CLBG workloads do not deviate significantly from the results for the real-world workloads.

The results for JRuby are very consistent across all workloads. The prominent reason for not inlining was again the methods being too big for a cold call site and a significant number of unreached call sites.

The results for Jython are qualitatively similar to JRuby, again with different proportions between variants of similar reasons (methods being too big to inline), but still quite consistent across all workloads. Compared with JRuby, there is an increased proportion of hot methods considered at unreached call sites.

6.4. Inlining depths

The inlining depth metric represents the number of methods inlined at a given level, and we expect it to reflect the additional levels of abstraction introduced by dynamic language implementations and the VM facilities they use (e.g., invokedynamic).

Figure 17 shows the analysis results for the CLBG workloads. We observe that JRuby has the deepest inlining structure—this is a symptom of JRuby using invokedynamic and the specific invokedynamic implementation that adds additional inlining levels. A single call from one JRuby method to another regularly incurs 5–10 inlining levels.

Jython and Clojure use deeply nested static helper functions throughout the generated code, which results in much deeply nested inlining. The amount of inlining for JavaScript is lower, but the inlining depth still reaches 10 levels for many of the CLBG workloads.

Scala only adds one or two levels of indirection when compared with the Java code. Scala was designed with execution on the JVM in mind from the start, and its language structure can be efficiently represented in Java bytecode. In general, Scala introduces additional layers of abstraction when compared with Java so that inlining should have a larger influence on the performance of Scala code than it has on Java code. This assumption is supported by the observation made in [50].

The results for Java only show at most four levels of inlining. Because Java does not need additional abstractions when running on a JVM, the inlining depth actually approximates the level of inlining naturally present in the algorithm implemented by a particular CLBG benchmark.

The situation is vastly different in the case of real-world workloads, with analysis results shown in Figure 18. In the cases of Java and (especially) Scala, these workloads appear to be much more complex than the CLBG workloads. Interestingly, in the case of JRuby, the real-world workloads appear to be actually less complex than the CLBG workloads. This suggests that the inlining depths reported in the figure are more indicative of the workload character than the language implementation.
Figure 17. The number of methods inlined at a given level for the Computer Language Benchmarks Game.

6.5. Fraction of inlined methods

The results in Figure 19 show the fraction of inlined methods. Again, we can observe that the results for Java and (to a certain extent) Scala show significant variance, suggesting that the CLBG workloads for these two languages are rather small and not representative enough. The results for Java are more indicative of the workload algorithm structure, while the results for other languages show primarily the inlining behavior of the language runtime.

The use of the `invokedynamic` bytecode instruction in JRuby leads to many successful inlinings, which correlates with the high inlining depths presented earlier. The results for JavaScript show a rather high proportion of unsuccessful inlining attempts, which hints at the generated code calling large helper methods—this is supported by the breakdown of reasons for not inlining methods in
Section 6.3. The results for the real-world workloads in Scala show the largest fraction of successful inlinings, which again correlates with the high inlining depths observed earlier.

6.6. Fraction of deoptimized methods

To achieve high performance, the JVM sometimes performs aggressive optimizations that are based on rather optimistic assumptions [51]. This includes static assumptions about the system’s state (e.g.,

© 2015 The Authors. Software: Practice and Experience Published by John Wiley & Sons Ltd.
Softw. Pract. Exper. 2016; 46:1053–1089 DOI: 10.1002/spe
the hierarchy of loaded classes) and dynamic assumptions about the behavior of the application (e.g., unreached branches within the application code). When the assumptions that a JIT compiler is made when compiling a method do not or cease to hold, the method will be invalidated and its compiled version discarded. The fraction of deoptimized methods metric shows how good the JVM is in making assumptions about the code it executes.

The results of this analysis are shown in Table VI. The significant variations in the results for Java and Scala again suggest that the CLBG workloads fail to stress the language runtime of these languages. The results for JRuby, Jython, and Clojure are very consistent. With the CLBG workloads, Clojure exhibits an increased number of deoptimized methods compared with the other two languages, but the difference is less pronounced with the real-world workloads. In the case of JavaScript, there are benchmarks with no deoptimization whatsoever, but there are also many cases where a large fraction of methods becomes deoptimized. This suggests that the assumptions made for the JavaScript workloads often do not hold during the lifetime of the application.

Table VI. The numbers of optimized and deoptimized methods.

| Benchmark  | Optimized | Deoptimized | Fraction (%) | Benchmark  | Optimized | Deoptimized | Fraction (%) |
|------------|-----------|-------------|--------------|------------|-----------|-------------|--------------|
|            | Clojure   |             |              | Java       |           |             |              |
| binarytrees| 108       | 18          | 16.67        | binarytrees| 5         | 0           | 0.00         |
| fannkuch   | 82        | 12          | 14.63        | fannkuch   | 1         | 1           | 100.00       |
| fasta      | 75        | 11          | 14.67        | fasta      | 4         | 0           | 0.00         |
| knucleotide| 162       | 37          | 22.84        | knucleotide| 19        | 1           | 5.26         |
| mandelbrot | 110       | 15          | 13.64        | mandelbrot | 2         | 0           | 0.00         |
| nbody      | 67        | 11          | 16.42        | nbody      | 2         | 0           | 0.00         |
| regexdna   | 96        | 20          | 20.83        | regexdna   | 16        | 6           | 37.50        |
| recomp     | 88        | 11          | 12.50        | recomp     | 2         | 1           | 50.00        |
| spectralnorm| 95       | 12          | 12.63        | spectralnorm| 4        | 0           | 0.00         |
| clj         | 215       | 33          | 15.35        | avrora     | 278       | 19          | 6.83         |
| frak        | 270       | 59          | 21.85        | fop        | 340       | 28          | 8.24         |
| clojure-script| 2021   | 347         | 17.17        | eclipse    | 2226      | 496         | 22.28        |
|            | JRuby     |             |              | Jython     |           |             |              |
| binarytrees| 320       | 31          | 9.69         | binarytrees| 229       | 27          | 11.79        |
| fannkuch   | 408       | 26          | 6.37         | fannkuch   | 185       | 15          | 8.11         |
| fasta      | 488       | 32          | 6.56         | fasta      | 272       | 24          | 8.82         |
| knucleotide| 423       | 50          | 11.82        | knucleotide| 211       | 27          | 12.8         |
| mandelbrot | 374       | 24          | 6.42         | mandelbrot | 168       | 12          | 7.14         |
| nbody      | 606       | 29          | 4.79         | nbody      | 159       | 8           | 5.03         |
| regexdna   | 214       | 31          | 14.49        | regexdna   | 142       | 11          | 7.75         |
| recomp     | 319       | 22          | 6.90         | recomp     | 196       | 4           | 2.04         |
| spectralnorm| 415      | 33          | 7.95         | spectralnorm| 166      | 12          | 7.23         |
| opal       | 562       | 81          | 14.41        | jython     | 1128      | 159         | 14.1         |
| rubyflux   | 374       | 53          | 14.17        | minilight  | 545       | 32          | 5.87         |
| voodoo     | 208       | 33          | 15.87        | scavis     | 130       | 29          | 22.31        |
|            | Scala     |             |              | JavaScript |           |             |              |
| binarytrees| 4         | 1           | 7.14         | binarytrees| 72        | 17          | 23.61        |
| fannkuch   | 6         | 0           | 0.00         | fannkuch   | 27        | 1           | 3.70         |
| fasta      | 11        | 0           | 0.00         | fasta      | 153       | 36          | 23.53        |
| knucleotide| 56        | 0           | 0.00         | knucleotide| 157       | 48          | 30.57        |
| mandelbrot | 8         | 0           | 0.00         | mandelbrot | 77        | 0           | 0.00         |
| nbody      | 9         | 0           | 0.00         | nbody      | 49        | 0           | 0.00         |
| regexdna   | 38        | 12          | 31.58        | regexdna   | 94        | 15          | 15.96        |
| recomp     | 9         | 1           | 11.11        | recomp     | 135       | 14          | 10.37        |
| spectralnorm| 8        | 1           | 12.50        | spectralnorm| 35        | 1           | 2.86         |
| apparat    | 836       | 94          | 11.24        | deltablue  | 160       | 18          | 11.25        |
| factorie   | 502       | 44          | 8.76         | raytrace   | 150       | 33          | 22.00        |
| scalac     | 1538      | 257         | 16.71        | richards   | 96        | 24          | 25.00        |
6.7. Discussion

There is a noticeable trend in the results for dynamic languages: the microbenchmarks from CLBG apparently manage to exercise a significant amount of the language runtime code, resulting in workloads that do not differ much from real-world workloads. However, the situation is vastly different with the statically typed languages, that is, Java and Scala. The microbenchmark nature of the CLBG workloads is clearly evident in all the metrics, especially when compared with the results for the real-world workloads. This means that conclusions concerning statically typed JVM languages can be misleading when based solely on observations of microbenchmark behavior. This is consistent with general benchmarking practices. The interesting result is that this is not necessarily the case with dynamic JVM languages, which include a significant amount of language runtime and framework code in their execution.

The primary reasons for inlining methods vary between the languages but remain rather consistent for workloads in the same language. The majority of inlined methods is either static, where the inlining decision boils down to method size and other properties, or intrinsic. Inlining of virtual methods is much less common and was more visible only in the case of JavaScript, where the JVM managed to inline some methods that are profiled monomorphic, bimorphic, and megamorphic with a major receiver.

The primary reasons for not inlining methods were mostly method sizes in conjunction with cold or even unreached call sites generated by dynamic language compilers. Qualitatively, the results for many dynamic languages were similar, with different proportions between various decisions. Each language exhibited a specific set of negative inlining decisions, but these tended to be in minority compared with negative decisions related to method sizes.

The results of the inlining depth analysis suggest that the dynamic languages employ additional level of abstractions to executed the workload code, with JRuby having the deepest inlining structure. Compared with other dynamic languages, JavaScript appears to be somewhat inlining unfriendly and unpredictable for the JIT compiler, as evidenced by the cases with high ratio of deoptimized methods.

7. RELATED WORK

Our primary goal is to equip developers of both the JVM languages and the JVM itself with tools that enable analyzing workloads produced by languages targeting the JVM. Our workload characterization framework uses whole-program instrumentation, which suffers from large overhead compared with sampling techniques [52, 53], but produces more accurate and detailed data about the observed application. Given the fact that developers would run the toolchain once per workload under study, we believe this approach ultimately pays off. In the rest of this section, we discuss our approach in the context of related work in the area of workload characterization of different programming languages.

Li et al. [37] recently published an exploratory study characterizing workloads for five JVM languages using both CLBG project and real-world application benchmarks. In their study, Li et al. collected metrics for Java, Scala, Clojure, JRuby, and Jython programs, characterizing N-gram coverage, method size, stack depths, method and basic block hotness, object lifetimes and size, and use of boxed primitives. In our study, we adopted a similar approach regarding workload selection, but opted for non-interactive real-world applications to complement our mix of CLBG benchmarks, and collected complementary metrics to characterize similar workloads from a different perspective.

Numerous works exist on the topic of workload characterization and programming languages comparison. Hundt [54] and Bull et al. [55] use idiomatic implementations of the same algorithm for performance comparisons. While the former implemented a loop recognition algorithm in Java, Scala, Go, and C++, the latter reimplemented the Java Grande benchmarking suite in C and Fortran. We follow a similar approach by using idiomatic implementations from the CLBG project, but our goals are different, as we aim to contribute to the understanding of JVM workloads produced by dynamic JVM languages.
Ratanaworabhan et al. [56] compare JavaScript benchmarks with real web applications and compute different static and dynamic platform-independent metrics, such as instruction mix, method hotness, and the number of executed instructions. The authors conclude that existing JavaScript benchmarks are not representative of real web sites, and the conclusions reached from measuring only the benchmarks can be misleading. We therefore complemented our study with real-world application benchmarks to avoid solely relying on microbenchmarks.

Daly et al. [9] examine optimization opportunities at the Java-to-bytecode compiler level. The authors analyze the Java Grande benchmark suite [57] using JVM-independent metrics. The authors consider static and dynamic instruction mixes and identify differences in optimizations performed by five different Java-to-bytecode compilers. Our work is similar in that we also search for optimization opportunities but at a lower level—our metrics capture workload characteristics that influence JIT compilation to machine code.

Dufour et al. [12] define a list of 60 dynamic metrics that characterize Java applications with respect to program size, data structures, concurrency and synchronization, and polymorphism. In their work, the authors present a case study that shows the usefulness of the defined metrics for guiding optimizations of Java programs. While the data collected by our toolchain can be used to compute most of the metrics defined by Dufour et al., our toolchain currently supports only a subset of these metrics. These are defined as queries over profile data and static class information, and for complete support of the metrics defined by Dufour et al., new query-based metrics would have to be introduced.

Some of the metrics supported by our toolchain have been used in other works, which support their relevance with respect to development and optimization of dynamic language runtimes. For example, dynamic instruction frequencies and basic block hotness metrics are fundamental in tracing-based compilation approaches. These metrics are also used in the work by Williams et al. [58], where the authors propose specializing native code to program properties found at JIT compilation time, thus improving the overall program execution time. The call-site polymorphism metric is used by Mostafa et al. [59] for possible caching of method call targets and inlining. It is also used in the work by Brunthaler et al. [60] to speedup the execution of dynamic languages by caching the call types. Barany et al. [61] analyze the usage of boxed types in order to eliminate extensive boxing behavior and instead allocate temporary objects on the heap.

Several other works make use of dynamic metrics for hinting at different optimization opportunities for JVM languages. Yermolovich et al. [62] propose to run an interpreter of a dynamic language on top of an optimizing trace-based virtual machine (VM), thus avoiding the need to create a custom JIT compiler for a dynamic language. A similar approach is proposed by Gal et al. [63], where the authors create a tracing JIT compiler for the JavaScript VM running in the Firefox web browser. Zaleski et al. [64] followed the goal of building an interpreter that could be extended to a tracing VM. The authors use tracing to achieve inlining, indirect jump elimination, and other optimizations for Java. In contrast to these works, our approach is JVM implementation independent, and the toolchain does not require any modification to a (standard) JVM. More focused analyses may benefit from modifying the JVM at the cost of portability, but this requires deep understanding of the particular JVM implementation internals. Using our toolchain, developers can first identify candidates for in-depth analyses, before investing effort into modifying the JVM.

8. CONCLUSIONS AND FUTURE WORK

Many programming languages are now targeting the JVM to take advantage of its high performance and maturity. 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 the JVM implementations have been mostly tuned with Java in mind; therefore, 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.
Workload characterization is a general approach that enables systematic analysis of system behavior in response to specific traits in the workloads the system is expected to handle. In the context of this article, workload characterization is used to aid in understanding the differences between workloads produced by Java and non-Java programs executing on the JVM. The work presented in this article contributes to the area of workload characterization on the JVM platform by providing new metrics that are sensitive to differences in the workloads resulting from bytecode produced by different JVM language compilers and by providing an easy-to-use toolchain that allows collecting these metrics on a standard JVM.

To contribute to an initial characterization of JVM language workloads, we applied our toolchain to nine functionally equivalent programs written in Java, Scala, Closure, Jython, JavaScript, and JRuby. Because of the lack of a proper benchmarking suite for the dynamic languages, we opted, like Li et al. [37] before us, to use the functionally equivalent benchmarks from the CLBG project, augmented with 18 different (three per language) real-world applications written in those languages. The findings result from a total of 72 experimental runs, weeks of experiment time, and analysis of hundreds of gigabytes of collected data. The study presented in this article thus demonstrates a paradigm for exploratory comparative analysis of programming languages via an application-based approach.

As expected, the microbenchmark nature of the CLBG workloads was apparent in the case of the statically typed languages, that is, Java and Scala, with the character of the CLBG workloads significantly differing from that of the real-world application benchmarks. In the case of the dynamic languages, the character of the workloads resulting from the CLBG benchmarks was actually very similar to that of the real-world applications. While caution is always appropriate when using microbenchmarks, their use in certain situations can be justified. This is important, because a simple program producing workload with the characteristics of a more complex program is more useful from the perspective of a developer, because the behavior of the program is easier to understand and the program is much easier to work with.

Because of the sheer amount of data, we were not able to perform an in-depth analysis of every result of every experiment. Instead, we were seeking general findings that may invite further research and more focused analyses. The general findings of our study can be summarized as follows:

**Call-site polymorphism.** Despite high number of polymorphic call sites targeting multiple methods, a very high percentage of method invocations actually happens at sites that only target a single method.

**Field, object, and class Immutability.** The dynamic languages use a significant amount of immutable (see Section 5.2 for the extended notion) classes and objects.

**Object lifetimes.** Compared with static languages, the dynamic language workloads allocate significantly more objects, but most of them are short-lived. This suggests that the dynamic languages create many temporaries, often resulting from unnecessary boxing and unboxing of primitive types.

**Unnecessary zeroing.** The dynamic languages (especially Clojure and Jython) exhibit a significant amount of unnecessary zeroing. This correlates with the significant amount of short-lived immutable objects allocated by the respective dynamic language workloads.

**Identity hash-code usage.** All the workloads use the identity hash code very scarcely, suggesting that object header compression with lazy handling of identity hash-code storage is an appropriate heuristic for reducing object memory and cache footprint.

To provide an additional perspective, and as an example of a more focused analysis, we modified the HotSpot JIT compiler to provide information on inlining decisions made during workload execution. We gathered additional metrics related to the ability of a JIT compiler to inline code produced by different JVM language compilers. Again, because of the sheer amount of data, we were mainly seeking general trends.

In the real-world application workloads, Scala code proved to be very amenable to inlining, with a high fraction of inlined methods and low fraction of deoptimized methods. The results for Java are similar, if slightly worse on the workloads considered in this study. The results for the dynamic languages with the exception of JavaScript were comparable, especially in the case of real-world
workloads. In the case of JavaScript, the JIT compiler was able to perform significantly less inlining. In addition, the assumptions about code produced by the JavaScript engine were more often violated, resulting in significantly higher proportion of deoptimized methods.

In general, the corpus of JVM language programs for which workload characterization has been performed is still small. Our work and the work of Li et al. provide complementary metrics for the functionally equivalent workloads, extending the corpus only where it concerns the real-world applications. At the moment, it would be naïve to expect far-reaching conclusions to be made based on the limited corpus, and we would be extremely lucky to discover a gaping difference in the workload characters that could be immediately resolved by some simple optimization. This does not invalidate our work, but merely acknowledges that given the complexity and sophistication of contemporary JIT compilers, all the simple things (and much more) have been already carried out. Save for a paradigm shift in compiler implementation, such as the Oracle Graal Compiler, we believe that improvements in the performance of dynamic JVM languages with contemporary JIT compilers will most likely come from many small incremental improvements. Our metrics, tools, and findings can help provide directions for specialized analyses leading to those improvements.

ACKNOWLEDGEMENTS
The research presented here was conducted while Lubomír Bulej was with the Università della Svizzera italiana in Lugano, Switzerland. It was supported by the Swiss National Science Foundation (project CRSII2_136225), by the European Commission (contract ACP2-GA-2013-605442), by the Sino-Swiss Science and Technology Cooperation (SSSTC), and by the Oracle (ERO project 1332).

REFERENCES
1. Ousterhout JK. Scripting: higher-level programming for the 21st century. Computer 1998; 31(3):23–30.
2. Siek J, Bhuradwaj S, Baker J. invokedynamic and Jython. Presentation at 2011 JVM Language Summit, retrieved 2013. Available from: http://wiki.jvmlangsummit.com/images/8/8d/Indy_and_Jython-Shashank_Bharadwaj.pdf [last accessed December 2013].
3. Castanos J, Edelsohn D, Ishizaki K, Nagpurkar P, Nakatani T, Ogasawara T, Peng W. On the benefits and pitfalls of extending a statically typed language JIT compiler for dynamic scripting languages. Proceedings of the 27th International ACM Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA ‘12, Tucson, Arizona, ACM, 2012; 195–212.
4. Sarimbekov A, Kell S, Bulej L, Sewe A, Zheng Y, Ansaloni D, Binder W. A comprehensive toolchain for workload characterization across JVM languages. Proceedings of the 11th ACM Workshop on Program Analysis for Software Tools and Engineering, PASTE ‘13, Seattle, Washington, ACM, 2013; 9–16.
5. Sewe A, Mezini M, Sarimbekov A, Binder W. Da Capo con Scala: design and analysis of a Scala benchmark suite for the Java virtual machine. Proceedings of the 26th ACM Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA ‘11, Portland, Oregon, ACM, 2011; 657–676.
6. Sewe A, Mezini M, Sarimbekov A, Ansaloni D, Binder W, Ricci N, Gayer SZ. New Scala( )stance of Java: a comparison of the memory behaviour of Java and Scala programs. Proceedings of the International Symposium on Memory Management, ISMM ’12, Beijing, China, 2012; 97–108.
7. Sarimbekov A, Podzimek A, Bulej L, Zheng Y, Ricci N, Binder W. Characteristics of dynamic JVM languages. Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages, VMIL ’13, Indianapolis, Indiana, ACM, 2013; 11–20.
8. Andreas S. Design and analysis of a Scala benchmark suite for the Java virtual machine. Ph.D. Thesis, Technische Universität Darmstadt, 2013.
9. Daly C, Horgan J, Power J, Waldron J. Platform independent dynamic Java virtual machine analysis: the Java Grande forum benchmark suite. Proceedings of the Conference on Java Grande, JGI ’01, Palo Alto, California, ACM, 2001; 106–115.
10. Lashari G, Srinivas S. Characterizing Java application performance. Proceedings of the 17th International Parallel and Distributed Processing Symposium, IPDPS ’03, Nice, France, IEEE, 2003; 22–26.
11. Shiv K, Chow K, Wang Y, Petrochenko D. SPECjvm2008 performance characterization. Proceedings of the SPEC workshop on Computer Performance Evaluation and Benchmarking, Austin, Texas, Springer-Verlag, 2009; 17–35.
12. Dufour B, Driesen K, Hendren L, Verbrugge C. Dynamic metrics for Java. Proceedings of the 18th ACM Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA ’03, Anaheim, California, ACM, 2003; 149–168.
13. Pearce DJ, Webster M, Berry R, Kelly PHJ. Profiling with AspectJ. Software: Practice and Experience 2007; 37(7):747–777.

© 2015 The Authors. Software: Practice and Experience Published by John Wiley & Sons Ltd.
14. Bodden E, Havelund K. Aspect-oriented race detection in Java. *IEEE Transactions on Software Engineering* 2010; 36(4):509–527.

15. Chen K, Chen J-B. Aspect-based instrumentation for locating memory leaks in Java programs. *Proceedings of the 31st International Computer Software and Applications Conference, COMPSAC ’07, Beijing, China, IEEE, 2007; 23–28.*

16. Sarimbekov A, Sewe A, Binder W, Moret P, Schoeberl M, Mezini M. Portable and accurate collection of calling-context-sensitive bytecode metrics for the Java virtual machine. *Proceedings of the 9th conference on Principles and Practice of Programming in Java, PPP ’11, Kongens Lyngby, Denmark, ACM, 2011; 11–20.*

17. Pechtchanski I, Sarkar V. Immutability specification and its applications. *Concurrency and Computation: Practice and Experience* 2005; 17(5–6):639–662. Special Issue Java Grande/ISCPOE 2002.

18. Bacon DF, Konuru R, Murthy C, Serrano M. Thin locks: featherweight synchronization for Java. *Proceedings of the ACM Conference on Programming Language Design and Implementation, PLDI ’98, Montreal, Canada, ACM, 1998; 258–268.*

19. Pizlo F, Frampston D, Hosking AL. Fine-grained adaptive biased locking. *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, PPP ’11, ACM, New York, NY, USA, 2011; 171–181.*

20. Bogda J, Hölzle U. Removing unnecessary synchronization in Java. *Proceedings of the 14th ACM Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA ’99, Denver, Colorado, ACM, 1999; 35–46.*

21. Shankar A, Arnold M, Bodik R. Jolt: lightweight dynamic analysis and removal of object churn. *Proceedings of the 23rd ACM Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA ’08, ACM, New York, NY, USA, 2008; 127–142.*

22. Dufour B, Ryder BG, Sevitsky G. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. *Proceedings of the 16th International Symposium on Foundations of Software Engineering, FSE ’08, ACM, Atlanta, Georgia, 2008; 59–70.*

23. Choi J-D, Gupta M, Serrano MJ, Sreedhar VC, Midkiff SP. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems* 2003; 25(6):876–910.

24. Tim L, Frank Y. *Java Virtual Machine Specification* (2nd edn). Addison-Wesley Longman Publishing Co., Inc., 1999.

25. Xi Y, Blackburn SM, Frampston D, Sartor JB, McKinley KS. Why nothing matters: the impact of zeroing. *Proceedings of the 26th ACM Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA ’11, Portland, Oregon, ACM, New York, NY, USA, 2011; 307–324.*

26. Bacon DF, Fink SJ, Grove D. Space- and time-efficient implementation of the Java object model. *Proceedings of the 16th European Conference on Object-Oriented Programming, ECOOP ’02, Springer-Verlag, Malaga, Spain, 2002; 111–132.*

27. Jones R, Ryder C. Garbage collection should be lifetime aware. *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, Nantes, France, 2006; 182–196.*

28. Pearson K. On lines and planes of closest fit to systems of points in space. *Philosopshical Magazine* 1901; 2:559–572.

29. Bolz CF, Cuni A, Fijalkowski M, Rigo A. Tracing the meta-level: PyPy’s tracing JIT compiler. *Proceedings of the 4th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOLPS ’09, Genoa, Italy, 2009; 18–25.*

30. Marek L, Villazón A, Zheng Y, Ansaloni D, Binder W, Qi Z. DiSL: a domain-specific language for bytecode instrumentation. *Proceedings of the 11th International Conference on Aspect-oriented Software Development, AOSD ’12, Potsdam, Germany, ACM, 2012; 239–250.*

31. Nethercote N, Seward J. How to shadow every byte of memory used by a program. *Proceedings of the 3rd Conference on Virtual Execution Environments, VEE ’07, San Diego, California, ACM, 2007; 65–74.*

32. Sarimbekov A, Sewe A, Binder W, Moret P, Mezini M. JP2: call-site aware calling context profiling for the Java virtual machine. *Science of Computer Programming* 2014; 79:146–157.

33. Ausiello G, Demetrescu C, Finocchi I, Firmani D. k-Calling context profiling. *Proceedings of the 27th international ACM conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA ’12, Tucson, Arizona, ACM, 2012; 867–878.*

34. Blackburn SM, Garner R, Hoffmann C, Khan AM, McKin-ley KS, Bentzur R, Diwan A, Feinberg D, Frampston D, Guyer SZ, Hirzel M, Hosking A, Jump M, Lee H, Moss JEB, Moss B, Phansalkar A, Stefanović D, VanDrunen T, von Dincklage D, Wiedermann B. The DaCapo benchmarks: Java benchmarking development and analysis. *Proceedings of the 21st ACM Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA ’06, Portland, Oregon, ACM, 2006; 169–190.*

35. The Standard Performance Evaluation Corporation. SPECjvm2008 benchmarks, 2008. Available from: http://www.spec.org/jvm2008/ [last accessed December 2013].

36. CLBG. The Computer Language Benchmarks Game. Available from: http://benchmarksgame.alioth.debian.org/ [last accessed December 2013].

37. Wing HL, Jeremy S, David W. JVM-hosted Languages: they talk the talk, but do they walk the walk? *Proceedings of the 10th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, PPPJ ’13, Stuttgart, Germany, ACM, 2013; 101–112.*

38. octane-benchmark. The JavaScript benchmark suite for the modern web. Available from: https://developers.google.com/octane/ [last accessed December 2013].

© 2015 The Authors. *Software: Practice and Experience* 
Published by John Wiley & Sons Ltd. 
DOI: 10.1002/spe
39. clj-pdf. A library for generating PDFs from Clojure. Available from: https://github.com/yogthos/clj-pdf [last accessed December 2013].
40. frak. Regular expressions generator in Clojure. Available from: https://github.com/noprompt/frak [last accessed December 2013].
41. minilight. Minimal global illumination renderer. Available from: https://github.com/brunoney/minilight-python [last accessed December 2013].
42. Scavis. Scientific computation and visualization environment. Available from: http://jwork.org/scavis/overview [last accessed December 2013].
43. rubyflux. Ruby to Java compiler. Available from: https://github.com/headius/rubyflux [last accessed December 2013].
44. voodoo. Image manipulation library for JRuby. Available from: https://github.com/jruby/image_voodoo [last accessed December 2013].
45. opal. Ruby to JavaScript compiler. Available from: https://github.com/opal/opal [last accessed December 2013].
46. clojure-script. Clojure to JavaScript compiler. Available from: https://github.com/clojure/clojurescript [last accessed December 2013].
47. Hölzle U, Chambers C, Ungar D. Optimizing dynamically typed object-oriented languages with polymorphic inline caches. *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP ’91, Springer-Verlag, Geneva, Switzerland, 1991; 21–38.
48. Detlefs D, Agesen O.Inlining of virtual methods. *Proceedings of the 13th European Conference on Object-oriented Programming*, ECOOP ’99, Springer-Verlag, Lisbon, Portugal, 1999; 258–278.
49. Ricci NP, Geyer SZ, Moss JEB. Elephant tracks: generating program traces with object death records. *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPI ’11, Kongens Lyngby, Denmark, ACM, 2011; 139–142.
50. Stadler L, Dubosq G, Mössenböck H, Wüthrich T, Simon D. An experimental study of the influence of dynamic compiler optimizations on Scala performance. *Proceedings of the 4th Workshop on Scala*, SCALA ’13, Montpellier, France, ACM, 2013; 9:1–9:8.
51. Wimmer C, Truffle TW. A self-optimizing runtime system. *Proceedings of the 3rd Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH ’12, Tucson, Arizona, ACM, 2012; 13–14.
52. Binder W. Portable, efficient, and accurate sampling profiling for Java-based middleware. *Proceedings of the 5th Workshop on Software Engineering and Middleware*, SEM ’05, Lisbon, Portugal, ACM, 2005; 46–53.
53. Whaley J. A portable sampling-based profiler for Java virtual machines. *Proceedings of the ACM Conference on Java Grande*, JAVA ’00, San Francisco, California, ACM, 2000; 78–87.
54. Hundt R. Loop recognition in C++/Java/Go/Scala. *Technical Report*, Google, 2010.
55. Bull JM, Smith LA, Pottage L, Freeman R. Benchmarking Java against C and Fortran for scientific applications. *Proceedings of the Conference on Java Grande*, JGI ’01, Palo Alto, California, ACM, 2001; 97–105.
56. Ratanaworabhan P, Livshits B, Zorn BG. JSMether: comparing the behavior of JavaScript benchmarks with real web applications. *Proceedings of the Conference on Web Application Development*, WebApps’10, Boston, Massachusetts, USENIX, 2010; 27–38.
57. Bull JM, Smith LA, Westhead MD, Henty DS, Davey RA. A methodology for benchmarking Java Grande applications. *Proceedings of the Conference on Java Grande*, JAVA ’99, Palo Alto, California, ACM, 1999; 81–88.
58. Williams K, McCandless J, Gregg D. Dynamic interpretation for dynamic scripting languages. *Proceedings of the 8th Symposium on Code Generation and Optimization*, CGO ’10, Toronto, Ontario, ACM, 2010; 278–287.
59. Mostafa N, Krintz C, Cascaval C, Edelsohn D, Nagprurkar P, Peng W. Understanding the potential of interpreter-based optimizations for Python. *Technical Report*, UCSB, 2010.
60. Brunthaler S. Inline caching meets quickening. *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP ’10, Springer-Verlag, Maribor, Slovenia, 2010; 429–451.
61. Barany G. Static and dynamic method unboxing for Python. *Proceedings Software Engineering (Workshops)*, Vol. 215 of LNI, GI, Aachen, Germany, 2013; 43–57.
62. Yermolovich A, Wimmer C, Franz M. Optimization of dynamic languages using hierarchical layering of virtual machines. *Proceedings of the 5th Dynamic Languages Symposium*, DLS ’09, Orlando, Florida, ACM, 2009; 79–88.
63. Gal A, Eich B, Shaver M, Anderson D, Mandelin D, Haghighat MR, Kaplan B, Hoare G, Zbarsky B, Orendorff J, Ruderman J, Smith EW, Reitmaier R, Bebenita M, Chang M, Franz M. Trace-based just-in-time type specialization for dynamic languages. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, PLDI ’09, Dublin, Ireland, ACM, 2009; 465–478.
64. Zaleski M, Brown AD, Stoodley K. YETI: a gradually extensible trace interpreter. *Proceedings of the 3rd Conference on Virtual Execution Environments*, VEE ’07, San Diego, California, ACM, 2007; 83–93.