A Comprehensive Toolchain for Workload Characterization Across JVM Languages

Aibek Sarimbekov
University of Lugano
Lugano, Switzerland
aibek.sarimbekov@usi.ch

Stephen Kell
University of Lugano
Lugano, Switzerland
stephen.kell@usi.ch

Lubomír Bulej
Charles University
Prague, Czech Republic
lubomir.bulej@d3s.mff.cuni.cz

Andreas Sewe
Technische Universität Darmstadt
Darmstadt, Germany
sewe@st.informatik.tu-darmstadt.de

Yudi Zheng
University of Lugano
Lugano, Switzerland
yudi.zheng@usi.ch

Danilo Ansaloni
University of Lugano
Lugano, Switzerland
danilo.ansaloni@usi.ch

Walter Binder
University of Lugano
Lugano, Switzerland
walter.binder@usi.ch

ABSTRACT

The Java Virtual Machine (JVM) today hosts implementations of numerous languages. To achieve high performance, JVM implementations rely on heuristics in choosing compiler optimizations and adapting garbage collection behavior. Historically, these heuristics have been tuned to suit the dynamics of Java programs only. This leads to unnecessarily poor performance in case of non-Java languages, which often exhibit systematic differences in workload behavior. *Dynamic metrics* characterizing the workload help to identify and quantify useful optimizations, but so far, no cohesive suite of metrics has adequately covered properties that vary systematically between Java and non-Java workloads. We present a suite of such metrics, justifying our choice with reference to a range of guest languages. These metrics are implemented on a common portable infrastructure which ensures ease of deployment and customization.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Performance attributes; D.2.8 [**Metrics**]: Performance measures

General Terms

Languages, Measurement, Performance

Keywords

Workload characterization, dynamic program analysis, bytecode instrumentation, dynamic metrics

PASTE'13, June 20, 2013, Seattle, WA USA

1. INTRODUCTION

While originally designed for the Java language only, the Java Virtual Machine (JVM) nowadays is targeted by hundreds of languages ranging from Ada to Z-code. Some of the more popular languages include Clojure, Groovy, JRuby, Jython, Kotlin, and Scala. Moreover, numerous domain-specific languages (DSLs) also target the JVM—for instance, it is very easy to develop new DSLs in Scala. Java has thus become one among many languages that run on the JVM.

Since stable, high-performance and mature JVM implementations are available, the choice of the JVM as a target for newly launched languages is natural. Its automatic memory management and adaptive optimizations allow the developers of DSLs or newly-launched JVM languages[1] to focus on the language's high-level features.

Despite hosting so many languages, today's JVM implementations have primarily been tuned with respect to characteristics of Java programs only. Ideally, a JVM would handle all the JVM languages equally well, with respect to the performance achieved by just-in-time compilation, memory management and so on.

To guide developers towards this goal, we require the means of characterizing the full range of workloads on the JVM, including applications written in different JVM languages. Two classes of artifact are useful for this: benchmarks and metrics. The former draw representative samples from the space of application code, while the latter identify useful performance dimensions within their behaviour. Whereas benchmarking shows *how well* a system performs at different tasks, metrics show *in what way* these tasks differ from each other, providing essential guidance for optimization effort.

Recently, JVM benchmarks encompassing Scala code have been proposed [26, 28], and an informal collection of multi-

[1]We use the term "JVM language" to refer to any language having one or more implementations targeting the JVM.

language benchmarks has also emerged.[2] However, the available metrics are more limited. Existing work has defined various dynamic metrics for Java [9], but these are Java-focused and predate the proliferation of JVM languages. They do not cover language-dependent performance properties, such as the relative extents of call-site polymorphism or object immutability, which have observably different distributions in Java versus non-Java code.

Both JVM and language front-end implementers stand to gain useful insight from a carefully chosen set of cross-language metrics. 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 by using a modified version of the JVM [8, 15], while others collect only architecture-dependent metrics [30]. In addition, at least one well-known metric suite implementation [9] run with unnecessarily high performance overhead. Ideally, metrics should be collected within reasonable time, since 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 since they lack coverage of code from the Java class library [17]. Our approach bases all metrics on a unified infrastructure which is JVM-portable, offers non-prohibitive runtime overhead with near-complete bytecode coverage, and can compute a full suite of metrics "out of the box". This toolchain has already been successfully applied for characterizing both Java and Scala workloads. The empirical results of this work are found elsewhere [27, 28]. In summary, we present the following contributions:

- a suite of dynamic metrics focused on capturing diversity among JVM languages;
- a common infrastructure supporting computation of both these and previously defined metrics, with a unified approach to instrumentation and data collection;
- a *query-based* definition of a subset of our metrics, which is particularly amenable to customization and extension.

We begin by introducing our new metrics and their intended use cases.

2. DYNAMIC METRICS

All our metrics are *dynamic*, meaning that they can be evaluted only by running the program on some input. The significance of dynamic metrics, in contrast to static metrics such as code size, static instruction distribution, etc., has been motivated elsewhere by Dufour et al. [9], who defined a list of sixty metrics considered useful for guiding optimization of Java programs. Our infrastructure can compute all of these metrics. However, the diversity of JVM languages means that additional metrics are necessary to capture properties which vary significantly between Java and non-Java workloads. In this section we describe several new metrics which our toolchain computes. Like those of Dufour et al., our metrics are defined at the bytecode level, making them JVM-independent and allowing portable implementation.

Before introducing our metrics, it is worth summarizing in exactly what ways our metrics can be of use to developers.

2.1 Usage modes

Use in bytecode generators.

The values of our metrics are of particular interest to the developers of bytecode-emitting compilers and interpreters, because they allow these developers to quantify the effects of optimizations on their bytecode generators. 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 JRuby.[3] Developers could optimize their bytecode generator, for example, to try harder at using primitives in their unboxed form. A dynamic metric of boxing behaviour would allow these developers to quantify the effects of such optimizations. This usage mode is exactly that outlined by Dufour et al [9, §7]. In particular, we note that the illustrative application they give for their metrics is in various bytecode-level transformations.

Use in JVM development.

Meanwhile, JVM developers may also use our toolchain for guidance, but in a rather different way. JVM optimizations are *dynamic* and *adaptive*—we can think of each optimization decision as being guarded by a heuristic decision procedure applied to profile data collected at runtime. For example, the decision of 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 which more accurately match real workloads, including non-Java workloads.

However, it must be possible to evaluate these at runtime with low overhead. Our metrics are generally too expensive to be computed on-line, but many of our metrics are *query-based*, meaning the metric is computed in two stages: a detailed trace is collected, and the metric is described as a query (defined in the XQuery language[4]) computed over the collected data after termination. Since this trace also subsumes the typical profile information available during dynamic compilation, it can be used to search for correlations which will yield a better heuristic.

For example, suppose querying reveals that a feature of the callchain at an object's allocation site correlates well with future use of the object (such as whether it requires explicit zeroing; we describe such a metric in §2.2). In turn, this may suggest a particular optimization (such as inlining a specific version of the allocation path). Sampling the callchain can detect such allocation sites at runtime. In this way, our infrastructure reveals an optimization opportunity which can be exploited during dynamic compilation. We describe this query-based aspect of our toolchain in §3.2.

2.2 New dynamic metrics

We now describe the suite of dynamic metrics we propose. Table 1 provides a summary. We believe our metrics are

[2]The Computer Language Benchmarks Game: http://benchmarksgame.alioth.debian.org/. (All references to URIs refer to content as retrieved on 2013/4/12.)

[3]http://jruby.org/

[4]http://www.w3.org/TR/xquery

comprehensive with respect to the current selection of JVM languages, in that they cover the differences arising from these languages' distinct semantics. In turn, these differences imply that differing optimizations will be required on the part of JVM implementers and language (front-end) implementers. Therefore, we have grouped the metrics according to the language-level concerns which motivate them: the **object access** (affecting sharing- and immutability-related optimizations), the relative usage of **heap-allocated objects** versus primitives (affecting object allocation, initialization and identity optimizations), and the differing usages of **bytecode features** (affecting optimizations which depend on the use of virtual dispatch, the density of procedural abstraction, argument passing behaviours, and overall instruction mix).

2.2.1 Object access

Immutability.

In recent years, functional languages have gained much attention. Functional programs generally create immutable data structures, avoiding side effects, hence making them amenable to parallelization. Determining those immutable objects can help the developer to identify the possible places in the code that can be parallelized. Moreover, popular compiler optimization techniques benefit from immutable objects and data structures [19]. 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, since they are known not to change across method calls.

We distinguish between class and object immutability. Therefore, we define four metrics: 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.[5]

Lock usage and contention.

Since locking operations come at a cost, researchers have developed thin locks [3] and biased locks [20] 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 [4]. Ordinarily, objects shared among different threads potentially require some synchronization. However, the synchronization is redundant if we find that the object is immutable.

Our metrics here are counts of objects shared between different threads, with separate counts for read-only sharing (two or more readers; exactly one writer, i.e. the allocating thread) and write-sharing (two or more writers; any number of readers). As with immutability analysis we further distinguish between fully and partially shared objects, yielding four counts in total.

2.2.2 Allocation concerns

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- 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) is detrimental to performance, since it comes at a cost of very frequent garbage collection and inhibits parallelization if temporary objects require synchronization [29]. Dufour et al. [10] 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 [7].

Object churn distance is a metric defined recently elsewhere [27], and is depicted on Figure 1. For each object we keep track of its allocation and death calling contexts; the closest capturing calling context is derived from those two. The distance from the object's capturing context to its allocation or death context via the closest capturing context is its dynamic churn distance. This metric is of particular importance in dynamic languages where primitive types are boxed. These workloads are characterized by 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 [16], 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. It was shown by Yang et al. [31] that zeroing has large impact on performance.

This interests us because 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 their first read do not need to be explicitly zeroed. Our zeroing analysis records occurrences

[5] 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.

Metric family	Description of metrics
Argument passing	distribution of floating point arguments over all dynamic invocations (see text)
	distribution of reference 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
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.
Instruction mix	execution counts for each distinct bytecode instruction (opcode)
Method hotness	contribution of the top 20% of methods to the dynamic total number of method executions
Stack usage and recursion depth	distribution of stack heights upon recursive calls
Use of boxed types	number of boxed primitives allocated
	number of boxed primitives requested (using `valueOf`; see text)
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
Field synchronization	number of objects synchronized on
	the average number of locking operations per object
	the maximum nesting depth reached per lock
Field immutability	number of fields immutable, counted once per containing object
	number of fields immutable (per class)
	number of objects immutable (all fields immutable).
	number of classes immutable (all fields immutable for all objects)
Implicit zeroing	number of primitive fields unnecessarily zeroed
	number of reference fields unnecessarily zeroed
Use of identity hashcodes	execution counts of overridden `hashCode` methods.
	execution counts of `System.identityHashCode` methods.
	execution counts of default `Object.hashCode` method.
Object churn distance	distribution of object churn distances (see text)
Object lifetimes	distribution of object survival times (see text)
Object sizes	distribution of object sizes (see text)

Table 1: Metrics that can be computed by our toolchain.

of this pattern. The metric is a count of unnecessary zeroing of primitive and reference fields.[6]

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 [2], and explicitly store the hash code only in the case when the object is moved *and* its identity hash code has previously been issued. An extra header slot is lazily added to the object in this case.

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. Systematic variation between Java and Scala workloads has been identified in our previous work [27]. Some heuristic is necessary in order to decide between the eager and lazy approaches. We define three metrics over binned invocation counts: frequency of objects receiving overridden `hashCode` invocations; frequency of objects receiving `System.identityHashCode` invocations; frequency of objects receiving the default `Object.hashCode` invocation (either by lack of override, or use of `super`).

[6]Our count of unnecessary zeroing currently assumes that all fields are zeroed. However, a future JVM might apply some optimization—perhaps motivated by our metric!—to selectively skip zeroing. To give accurate results in these cases, our metric would have to be reimplemented to account for this, e.g., by instrumenting the allocation path in the optimized zero-skipping case.

Object lifetimes and sizes.

Some languages allocate more, smaller and/or shorter-lived objects than others. Object lifetime analysis is of particular importance for garbage collector (GC) implementers. New GC algorithms are designed and evaluated by simulation based on object lifetime traces. An example of such an algorithm is lifetime-aware GC [14], 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, since larger objects that live longer incur greater overhead than small, short-lived ones.

Our lifetime metric counts objects binned according to their survival time measured in cumulative bytes allocated ($\leq$1MiB, sim. 2MiB, 4MiB and 100MiB, and a separate bin for objects not surviving beyond nursery collection). The size metric collects a binned distribution of object sizes (including the header).

2.2.3 *Code generation concerns*

Instruction mix.

An instruction mix metric can describe 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 computation than others. 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

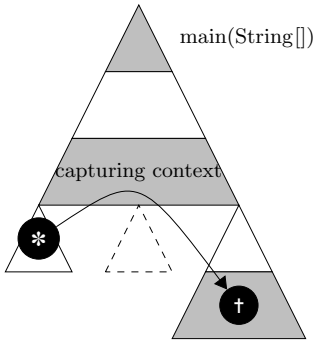

Figure 1: The churn distance of an object is computed as the distance between its allocation (✻) and death (†) calling contexts via their closest capturing context. This metric has previously been shown to exhibit variation between Java and Scala workloads [27].

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.

In contrast to Dufour et al. [9] where the authors grouped bytecodes manually, we instead use principal component analysis (PCA) [18]. It offers a high-level view of the instruction mix in which the selected groupings of bytecode instructions are tailored to the workload. PCA allows to drastically reduce the data's dimensionality, thus allowing better comprehension of the results.

Stack usage and recursion depth.

This an important metric for the developers of dynamic languages supporting the functional programming paradigm such as Clojure. Functional languages often leverage recursion to perform loops. Therefore it is very important for compiler developers of those languages to perform tail call elimination, such that executed method will not allocate any new stack frames, making recursive calls to be executed in constant space.

Our metric collects the distribution of stack heights upon each of three cases of method calls: all method calls, "potentially recursive" calls (virtual calls which *can* dispatch to the same method), and "true recursive" calls (which actually *do* dispatch in this way, whether virtually or by `final`).

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 partition arguments into three kinds—integer primitive values, references and floating-point values—and count each separately for each call. Our metrics bin all method invocations by their total argument count, then for each bin, compute a 5-vector counting the number of those arguments that are floating-point (zero to four and $\geq$5; elements beyond the total argument count are always zero), and similarly for reference arguments.

Basic-block hotness.

Hotness metrics are fundamental, since any JVM with a just-in-time compiler optimizes the code based on its hotness (i.e., the code that is most frequently executed). While hotness is traditionally identified at the granularity of methods, some modern dynamic compilers instead use trace-based approaches which rely on identifying sequences of hot basic blocks (possibly crossing method boundaries). These are particularly popular among contemporary dynamic language implementations such as PyPy [5] or Mozilla's TraceMonkey Javascript implementation.[7] Therefore, a finer-grained hotness metric is 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 which 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

Our toolchain consists of several distinct tools with a common infrastructure which is designed for ease of use and extension. In this section we describe this infrastructure.

3.1 Deployment and use

A 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[8], 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. 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 (§3.2).

Since all instrumentation is done using the same high-level domain-specific language, namely 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, since 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 [25]. 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, hashcode analysis instruments only a few method entries—and incur correspondingly less

[7] https://developer.mozilla.org/en-US/docs/SpiderMonkey/Internals/Tracing_JIT
[8] http://disl.ow2.org/

overhead. We note that our instrumentation-based approach generally outperforms like-for-like metric implementations using the older JVMPI[9] interface, including those described by Dufour et al. [9].

Metrics such as field immutability, zeroing, field sharing, and use of identity hashcodes are collected via custom tools that use DiSL to perform bytecode instrumentation. In each case, the instrumentation maintains shadow state for each object. Depending on the analysis different events are intercepted and different information is stored in a shadow state. For example, to measure 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 of which is a state machine with states `virgin` (i.e., not read or not written to), `immutable` (i.e., read or was written to inside the dynamic extent of its owner object's constructor), or `mutable` (otherwise). Figure 2 depicts the corresponding `FieldState` class.

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

Object lifetime analysis uses a custom tool implementing the Merlin algorithm [13]. Our tool is very similar to the ElephantTracks (ET) tool built around this algorithm [22]. However, our implementation has the advantage that its instrumentation part is expressed cleanly using DiSL, uniformly with the rest of our toolchain. In contrast, the original ElephantTracks implementation primarily uses explicitly-coded instrumentation (using the ASM library[10]) and JVMTI callbacks. (Our implementation uses JVMTI only for heap traversal required for reachability computations.)

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 implemented over a new implementation of the JP2 system. JP2 [24, 25] is a calling-context profiler which produces execution traces in the form of an annotated calling-context tree (CCT). For this work we have reimplemented JP2 using the DiSL instrumentation framework, for uniformity with other tools in our toolchain. Each node in a CCT corresponds to a particular callchain and keeps the dynamic metrics, such as number of method invocations and number of executed bytecodes. JP2 is call-site aware, meaning 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- and intra-procedural analysis and reports dynamic execution counts for each basic-block of code in methods.

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

[9] http://docs.oracle.com/javase/1.5.0/docs/guide/jvmpi/
[10] http://asm.ow2.org/

```
public class FieldState {
  private State currentState = State.VIRGIN;
  private enum State {
          VIRGIN, IMMUTABLE, MUTABLE };
  private boolean defaultInit = false;
  public synchronized void onRead() {
    switch (currentState) {
      case VIRGIN:
        defaultInit = true;
        currentState = State.IMMUTABLE;
        break;
  } }
  public synchronized void onWrite(
                        boolean isInConstructor) {
    switch(currentState) {
      case VIRGIN:
      case IMMUTABLE:
        currentState = isInConstructor ?
                  State.IMMUTABLE : State.MUTABLE;
        break;
  } }
  /* ... */
}
```

Figure 2: The field immutability state machine.

Figure 3 depicts a three-step process of computing dynamic metrics with JP2. First, the application is instrumented for profiling; second, the collected profile is dumped in an XML-based format for later offline analysis; finally, the desired metrics are computed offline. Dumping in XML format allows using off-the-shelf tools for metrics computation. We use XQuery for formulating metrics as queries.

3.3 Instrumentation

Some of the information needed for our metrics' computation is not stored in a CCT, but depends on static properties of class files. For this, we use another facility of JP2, which can dump a list of all classes loaded during execution. These classes are converted to an XML representation to allow querying alongside the CCT data [25]. Many of our queries make use of the ability to cross-reference between CCT and class data.

Figure 4 shows an example of a query for identifying methods with 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 with typical profilers. The algorithm is straightforward: return the methods of the application, sorted in decreasing order of the total execution counts over all their contained basic blocks.

A key benefit of the query-based design is that custom queries can be used to formulate previously unanticipated metrics. For example, 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 [1].

The separation between dumps and queries avoids potential problems with nondeterminism. Multiple different metrics can be computed without the need for repeated application runs, hence avoiding any risk of divergent behaviour across such runs.

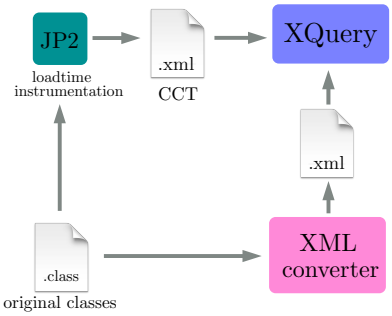

Figure 3: Query-based metrics are implemented on top of JP2 [25].

```
for $method in functx:distinct-nodes(
  for $bb in $methods/dcg:basicBlock
    order by $bb/dcg:executionCount/xs:
             long(.) descending
    return $bb/..)
```

Figure 4: Example of a query for identifying methods with hottest basic blocks. `dcg` refers to "dynamic call graph", referring to the calling context tree. Other identifiers are self-explanatory.

4. RELATED WORK

Our work is inspired by the *J metric suite developed by Dufour et al. [9]. This defines five families of dynamic metrics that characterize Java applications with respect to program size, data structures, concurrency and synchronization, and polymorphism. All these metrics can be obtained using our toolchain; we expand on them by introducing metrics which usefully characterize non-Java workloads. Moreover, our infrastructure offers several benefits. *J's implementation relies on the JVMPI[11], a deprecated profiling interface for the JVM. JVMPI also exhibits huge performance overhead, and has certain limitations which prevent the authors from collecting memory-related metrics. In contrast, our implementation uses bytecode instrumentation, offering substantial performance improvements and executing on contemporary production JVMs.

Shiv et al. [30] compare the SPECjvm98[12] and SPECjvm2008[13] benchmark suites. The authors present quantitative evaluation using different JVM- and architecture-dependent metrics. They look at the effectiveness of Java runtime systems including just-in-time compilation, dynamic optimizations, synchronizations, and object allocations. They also report results for the SPECjAppServer2004[14] and the SPECjbb2005[15] benchmarks. This approach yields metrics whose values strongly depend on the chosen architecture, which are of limited utility in our envisaged usage scenarios. We avoid this problem by computing JVM- and machine-independent metrics.

Daly et al. [8] analyze the Java Grande benchmark suite [6] using JVM-independent metrics. The authors consider static and dynamic instruction mix and use five different Java-to-bytecode compilers to quantify the impact of the choice of a compiler on the dynamic bytecode frequency. To compute the metrics, the authors use a modified version of the Kaffe Java Virtual Machine[16]. Gregg et al. [11] also use a modified Kaffe to characterize workloads by their number of native methods activations. Similar approaches are used elsewhere [12, 15]. The choice of a non-standard JVM imposes additional overhead on developers willing to use the tools to compute dynamic metrics. In contrast, our tools are based on bytecode instrumentation and can be run on any production JVM used by the developer.

Several works aim at characterizing JavaScript applications. Ratanaworabhan et al. [21] compares JavaScript benchmarks with real web applications using various static and dynamic platform-independent metrics, including instruction mix, method hotness, and the number of executed instructions. Although our infrastructure can compute all the dynamic metrics used in the comparison, it cannot be used to reproduce the measurements, because it targets the JVM, whereas the authors have conducted their experiments directly in a web browser.

Richards et al. [23] computes several dynamic metrics for JavaScript code. Both the metrics and the infrastructure are specialized for JavaScript, and the problem motivating our work, namely tuning a multi-language infrastructure, does not arise in this scenario. However, many metrics in this work target features specific to JavaScript and other highly dynamic languages—such as prototype-based object creation, field additions and so on. These features can be supported on the JVM, but only in simulated form (e.g., by generating new classes at run time). It would be interesting to extend our toolchain to subsume these metrics, using an awareness of these simulation approaches.

5. CONCLUSIONS AND FUTURE WORK

Despite hosting many different languages, today's JVM implementations are still unable to handle all JVM languages equally well. To achieve this goal, both language and JVM developers need to explore the space of possible optimizations and code generation techniques with non-Java languages in mind. Consequently, they need suitable benchmarks to obtain samples of representative application behavior, and metrics to characterize workloads produced by programs originating in non-Java languages.

To aid in their exploration, we have presented a unified infrastructure for characterization of workloads executing in the JVM. Our infrastructure is portable, offers non-prohibitive overhead with near-complete bytecode coverage, and provides a full suite of dynamic metrics, including several new metrics which have observably different distributions in Java versus non-Java code. This paper will be accompanied by an open-source implementation of the toolchain which can be used out-of-the-box for collecting various dynamic metrics. We therefore believe that our toolchain will attract users from the ranks of JVM and language developers, and that it will help to shed light on performance regressions affecting various language–VM pairings.

[11] http://docs.oracle.com/javase/1.5.0/docs/guide/jvmpi/index.html, deprecated in Java 6.
[12] http://www.spec.org/jvm98
[13] http://www.spec.org/jvm2008
[14] http://www.spec.org/jAppServer2004
[15] http://www.spec.org/jbb2005
[16] http://www.kaffe.org

The availability of an established and widely recognized benchmarking suite—something akin to the DaCapo suite, but for non-Java languages—remains an open issue. Such a suite is needed to capture relevant real-world workloads for analysis by JVM developers—with the exception of Scala, benchmark applications for other languages targeting the JVM come mostly from the Programming Languages Shootout Project. The obvious problem of such applications is that they are rather small, and not representative of complex, real-world workloads. Besides refining and extending the set of metrics provided by our suite, an obvious continuation of our work may therefore include the design of such a benchmarking suite.

The metrics presented here are not bound to the JVM and can be generalized for the case of .NET and Common Language Runtime (CLR), although the instrumentation technique would be slightly different. Developing a version of the toolchain for the CLR is considered to be a subject for future work.

Acknowledgments

The research presented in this paper has been supported by the Swiss National Science Foundation (project CRSII2_136225), by a Sino-Swiss Science and Technology Cooperation (SSSTC) Institutional Partnership (project IP04–092010), by the European Commission (Seventh Framework Programme grant 287746), and by the Czech Science Foundation (project GACR P202/10/J042).

References

[1] G. Ausiello, C. Demetrescu, I. Finocchi, and D. Firmani. k-calling context profiling. In *Proc. Intl. Conf. on Object oriented programming: systems, languages and applications*. ACM, 2012.

[2] D. F. Bacon, S. J. Fink, and D. Grove. Space- and time-efficient implementation of the Java object model. In *Proc. 16th European Conf. on Object-Oriented Programming*. Springer-Verlag, 2002.

[3] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. In *Proc. ACM SIGPLAN Conf. on Programming language design and implementation*. ACM, 1998.

[4] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Proc. Conf. on Object-oriented programming: systems, languages, and applications*. ACM, 1999.

[5] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proc. W. on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ACM, 2009.

[6] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A methodology for benchmarking Java grande applications. In *Proc. ACM Java Grande Conf.* ACM, 1999.

[7] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Trans. Programing Languages and Systems*, 25(6), Nov. 2003.

[8] C. Daly, J. Horgan, J. Power, and J. Waldron. Platform independent dynamic Java virtual machine analysis: the Java grande forum benchmark suite. In *Joint ACM Java Grande / ISCOPE Conf.* ACM, 2001.

[9] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *Proc. Conf. on Object-oriented Programing, Systems, Languages, and Applications*. ACM, 2003.

[10] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *Proc. 16th ACM SIGSOFT Intl. Symposium on Foundations of software engineering*. ACM, 2008.

[11] D. Gregg, J. Power, and J. Waldron. A method-level comparison of the Java grande and SPECjvm98 benchmark suites: Research articles. *Concurrency and Computation: Practice and Experience*, 17, 2005.

[12] N. M. Hanish and W. Cohen. Hardware support for profiling Java programs. In *Wksp. on Hardware Support for Objects And Microarchitectures for Java*, 1999.

[13] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Error-free garbage collection traces: how to cheat and not get caught. In *Proc. ACM SIGMETRICS Intl. Conf. on Measurement and modeling of computer systems*. ACM, 2002.

[14] R. Jones and C. Ryder. Garbage collection should be lifetime aware. In *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, 2006.

[15] G. Lashari and S. Srinivas. Characterizing Java application performance. In *Proc. 17th Intl. Symposium on Parallel and Distributed Processing*. IEEE, 2003.

[16] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman, 2nd edition, 1999.

[17] D. J. Pearce, M. Webster, R. Berry, and P. H. J. Kelly. Profiling with AspectJ. *Software: Practice and Experience*, 37(7):747–777, 2007.

[18] K. Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2:559–572, 1901.

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

[20] F. Pizlo, D. Frampton, and A. L. Hosking. Fine-grained adaptive biased locking. In *Proc. 9th Intl. Conf. on Principles and Practice of Programming in Java*. ACM, 2011.

[21] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. JSMether: comparing the behavior of JavaScript benchmarks with real web applications. In *Proc. USENIX Conf. on Web application development*. USENIX Association, 2010.

[22] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant Tracks: generating program traces with object death records. In *Proc. 9th Intl. Conf. on Principles and Practice of Programming in Java*. ACM, 2011.

[23] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proc. Conf. on Programming Language Design and Implementation*. ACM, 2010.

[24] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, and M. Mezini. JP2: Call-site aware calling context profiling for the Java Virtual Machine. *Science of Computer Programming*, 2012.

[25] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, M. Schoeberl, and M. Mezini. Portable and accurate collection of calling-context-sensitive bytecode metrics for the Java Virtual Machine. In *Proc. of the 9th Intl. Conf. on Principles and Practice of Programming in Java*. ACM, 2011.

[26] A. Sewe. *Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine*. PhD thesis, Technische Universität Darmstadt, 2013.

[27] A. Sewe, M. Mezini, A. Sarimbekov, D. Ansaloni, W. Binder, N. Ricci, and S. Z. Guyer. new Scala() instance of Java: a comparison of the memory behaviour of Java and Scala programs. In *Proc. Intl. symposium on Memory Management*. ACM, 2012.

[28] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder. Da capo con Scala: design and analysis of a Scala benchmark suite for the Java virtual machine. In *Proc. Conf. on Object-oriented Programming, Systems, Languages and Applications*. ACM, 2011.

[29] A. Shankar, M. Arnold, and R. Bodik. Jolt: lightweight dynamic analysis and removal of object churn. In *Proc. Conf. on Object-oriented Programming, Systems, Languages and Applications*, 2008.

[30] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. SPECjvm2008 performance characterization. In *Proc. of the 2009 SPEC Benchmark Wksp. on Computer Performance Evaluation and Benchmarking*, pages 17–35, Berlin, Heidelberg, 2009. Springer-Verlag.

[31] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley. Why nothing matters: the impact of zeroing. In *Proc. Conf. on Object-oriented Programming, Systems, Languages, and Applications*. ACM, 2011.