Characteristics of Dynamic JVM Languages

Aibek Sarimbekov

University of Lugano

firstname.lastname@usi.ch

Andrej Podzimek

Charles University in Prague

podzimek@d3s.mff.cuni.cz

Lubomir Bulej

University of Lugano

firstname.lastname@usi.ch

Yudi Zheng

University of Lugano

firstname.lastname@usi.ch

Nathan Ricci

Tufts University

nricci01@eecs.tufts.edu

Walter Binder

University of Lugano

firstname.lastname@usi.ch

Abstract

The Java Virtual Machine (JVM) has become an execution platform targeted by many programming languages. However, unlike with Java, a statically-typed language, the performance of the JVM and its Just-In-Time (JIT) compiler with dynamically-typed languages lags behind purpose-built language-specific JIT compilers. In this paper, we aim to contribute to the understanding of the workloads imposed on the JVM by dynamic languages. We use various metrics to characterize the dynamic behavior of a variety of programs written in three dynamic languages (Clojure, Python, and Ruby) executing on the JVM. We identify the differences with respect to Java, and briefly discuss their implications.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Performance attributes; D.2.8 [*Software Engineering*]: Metrics—Performance measures

General Terms Languages, Measurement, Performance

Keywords workload characterization; dynamic metrics; Java; JRuby; Clojure; Jython

1. Introduction

The introduction of scripting support and support for dynamically-typed languages to the Java platform enables scripting in Java programs and simplifies the development of dynamic language runtimes. Consequently, developers of literally hundreds of programming languages target the Java Virtual Machine (JVM) as the host for their languages—both to avoid

developing a new runtime from scratch, and to benefit from the JVM's maturity, ubiquity, and performance. Today, programs written in popular dynamic[1] languages such as Ruby, Python, or Clojure (a dialect of Lisp) can be run on the JVM, creating an ecosystem that boosts developer productivity.

However, since the JVM was originally conceived for a statically-typed language, the performance of the JVM and its JIT compiler with dynamically-typed languages is often lacking, lagging behind purpose-built language-specific JIT compilers. Making the JVM perform well with various statically- and dynamically-typed languages clearly requires significant effort, not only in optimizing the JVM itself, but also, more importantly, in optimizing the bytecode-emitting language compiler, instead of just relying on the original JIT to gain performance [8]. This in turn requires that developers of both the language compilers and the JVM understand the characteristics of the JVM workloads produced by various languages or compilation strategies.

In the case of statically-typed languages such as Java and Scala, the execution charateristics have become rather well understood, thanks to established metrics, benchmarks, and studies [9, 11, 21–23]. In contrast, the execution characteristics of various dynamically-typed JVM languages have so far not been studied very extensively. We have previously presented a comprehensive toolchain [20] for workload characterization across JVM languages[2], which was successfully applied in studying the differences between Scala and Java workloads [21, 22]. Recently, our toolchain was applied in a study of execution characteristics of JVM languages by Li et al. [16], which included Clojure, JRuby, and Jython, as representatives of popular dynamic JVM languages.

In this paper, we report on the result of applying our toolchain to programs written in Java and three dynamic JVM languages—Clojure, Python, and Ruby. We adopt similar

---

[1] We use the terms scripting, dynamic, and dynamically-typed language interchangeably.

[2] We use the term *JVM language* to refer to any language that targets the JVM as execution platform.

methodology and base programs as Li et al. in their study [16], but present complementary metrics and results, seeking to improve understanding of the execution characteristics of the selected dynamic JVM languages.

Hence, the original scientific contribution of this paper is a workload characterization for selected Clojure, JRuby, and Jython programs, based on dynamic metrics such as call-site polymorphism, object lifetimes, object and class immutability, memory zeroing, and object hash code usage. We present and compare the results for functionally equivalent programs written using the dynamic JVM languages and Java, and briefly discuss the implications.

2. Experiment Design

2.1 Metrics

To analyze the dynamic program behavior, we collected various dynamic metrics[3] that influence performance or hint at optimization opportunities for programs executing on the JVM. In Section 3 we present details and briefly discuss results for the following metrics:

Call-site Polymorphism. Hints at opportunities for optimizations at polymorphic call-sites, e.g. inline caching [13] (based on the number of receiver types), or method inlining [10] (based on the number of target methods).

Field, Object, and Class Immutability. Enables load elimination [3] (replacing repeated accessess to immutable objects with an access to a compiler-generated temporary stored in a register), and identifies objects and side-effect-free data structures amenable to parallelization.

Object Lifetimes. Determines garbage-collector (GC) workload, and aids in design and evaluation of new GC algorithms, e.g. the lifetime-aware GC [15].

Unnecessary Zeroing. Hints at opportunities for eliminating unnecessary zeroing of memory for newly allocated objects, which comes with a performance penalty [25].

Identity Hash-code Usage. Hints at opportunities for reducing header size for objects that never need to store their identity hash code (often derived from their memory location upon first request [1]).

The metrics were collected using our workload characterization suite [20], which relies on bytecode instrumentation, and provides a near-complete bytecode coverage. We were thus able to collect metrics that cover both the application (and the dynamic language runtime) and the Java Class Library (including any proprietary JVM vendor-specific classes) on a standard JVM.

2.2 Workloads

Any attempts at characterizing or comparing dynamic JVM language workloads are inevitably hampered by the lack

Benchmark	Description	Input
binarytrees	Allocate and deallocate many binary trees	16
fannkuch-redux	Repeatedly access a tiny integer-sequence	10
fasta	Generate and write random DNA sequences	150,000
k-nucleotide	Repeatedly update hashtables and k-nucleotide strings	fasta output
meteor-contest	Search for solutions to a shape packing puzzle	2,098
mandelbrot	Generate a Mandelbrot set and write a portable bitmap	1,000
nbody	Perform an N-body 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
spectral-norm	Calculate an eigenvalue using the power method	500

Table 1. Benchmarks from the CLBG project (implemented in Java, Clojure, Ruby, and Python) selected for workload characterization.

of an established benchmark suite such as DaCapo [4], SPECjvm2008[4], or the Scala Benchmark suite [22]. Even though some language-specific benchmark suites (e.g. Py-Bench) exist, they are usually very low-level and do not allow for direct comparison among different languages.

The closest to a benchmark suite that can be used for a rough comparison of dynamic languages is the Computer Language Benchmarks Game (CLBG) project[5], which compares performance results for various benchmarks implemented in many different programming languages. Each benchmark has a prescribed algorithm and an idiomatic implementation in each supported language.

In general, the benchmarks cannot be considered representative of real-world applications, yet significant performance differencens between Python compilers we considered indicative by the developers of the Fiorano JIT compiler [8], and benchmarks from the CLBG project have been used in the recent study of JVM languages by Li et al. [16]. To provide complementary results for comparable workloads, we have decided to adopt the approach of Li et al., and based our study (mostly, but not completely) on 10 CLBG benchmarks, listed in Table 1 along with a brief description and inputs used.

We are aware of the general threat to validity due to the use of micro-benchmarks. However, we argue that in the context of our work, the threat is significantly mitigated by the fact that we study the work a JVM needs to perform when executing the benchmarks (rather than bare-metal performance).

Still, to avoid relying solely on micro-benchmarks, we complemented our workload selection with 4 real-world application benchmarks, listed in Table 2. These unfortunately lack the nice property of being idiomatic implementations

---

[3] They can only be obtained by running a program with a particular input.

[4] http://www.spec.org/jvm2008/,

[5] http://benchmarksgame.alioth.debian.org/

Application (language)	Description	Input
clojure-script (Clojure)	A compiler for Clojure that targets JavaScript	twitterbuzz source
eclipse (Java)	An integrated development envinronent (IDE)	DaCapo default
jython (Python)	An interpreter of Python running on the JVM	DaCapo default
opal (Ruby)	A Ruby to JavaScript compiler	meteor source

Table 2. Real-world applications selected as benchmarks to complement the CLBG benchmarks for workload characterization.

of the same task. The eclipse and jython benchmarks come from the DaCapo suite, while opal[6] and clojure-script[7] are open-source projects from GitHub.

2.3 Measurement Context

All metrics were collected with Java 1.6, Clojure 1.5.1, JRuby 1.7.3, and Jython 2.7 runtimes, yielding a total of 44 different language-benchmark combinations, all executed using the OpenJDK 1.6.0_27 JRE running on Ubuntu Linux 12.0.4.2. Due to 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[8] and platform independent.

3. Experimental Results

3.1 Call-site Polymorphism

Specialization is considered to significantly aid performance of dynamic languages targeting the JVM [8]. Hot polymorphic call-sites are good candidates for optimizations such as inline caching [13] and method inlining [10], which specialize code paths to frequent receiver types or target methods. 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 calls made at each polymorphic call-site during the execution of each workload. The plots in Figures 1–4 show the number of call sites binned according to the number of targeted methods (x-axis), with an extra bin for call-sites targeting 15 or more methods. The plots in Figures 5–8 then show the actual number of invocations performed at those call sites.

---

[6] http://opalrb.org

[7] https://github.com/clojure/clojurescript

[8] Different vendors may provide different implementation of the Java Class Library as well as other proprietary classes in their JVM.
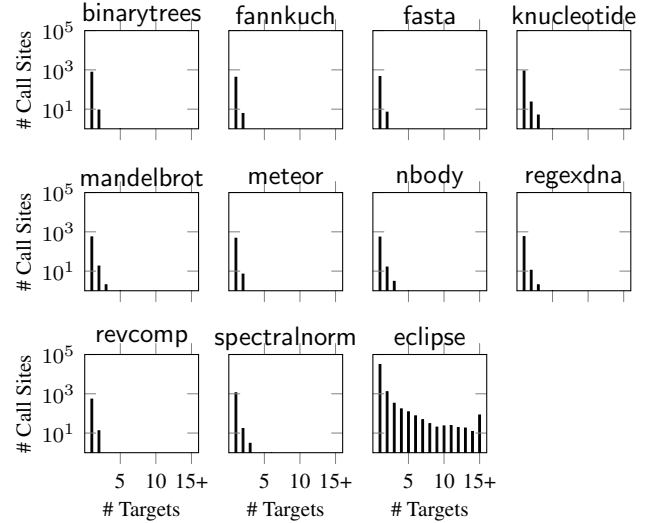

Figure 1. The number of dynamically-dispatched call sites targeting a given number of methods for the Java benchmarks.

We observe that polymorphic invocations in the CLBG benchmarks for Java do not target more than 6 methods. This is not suprising, given the microbenchmark nature of the CLBG workloads. The situation is vastly different—and more realistic—with the eclipse workload. Still, on average, 98.2% of the call sites (accounting for 90.8% of all method calls) only had a single target.

In the case of dynamic JVM languages, the microbenchmark nature of the CLBG workloads is much less pronounced (compared to the real application workload). This suggests that even the CLBG workloads do exhibit some of the traits representative of a particular dynamic language.

The results for Clojure workloads show that polymorphic invocations target 1 to 10 methods, with an average of 99.3% of the call sites (accounting for 91.2% of method calls) actually targeting a single method. The results for Jython workloads show that polymorphic invocations mostly target 1 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 method calls) target a single method.

Finally, the results for JRuby show little difference between the CLBG benchmarks and the real-world application, and consistently show a significant number of call-sites with 15 or more targets. Interestingly, the number of calls made at those sites is surprisingly high—comparable with the other sites. However, on average, 98.4% of the call sites (accounting for 88% of method calls) target a single method.

3.2 Field, Object, and Class Immutability

To study 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
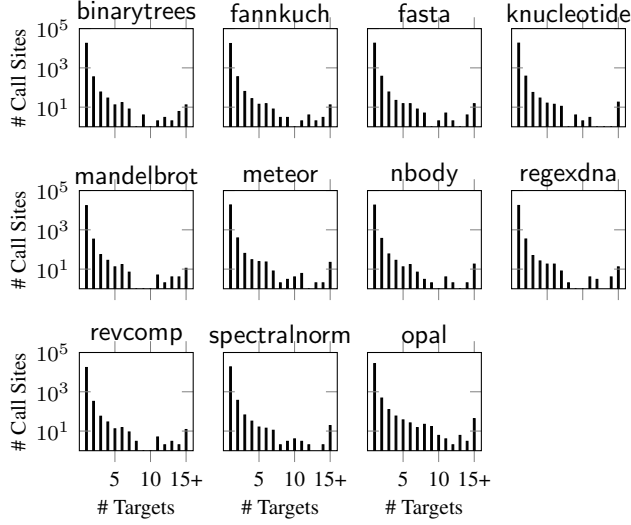
Figure 2. The number of dynamically-dispatched call sites targeting a given number of methods for the JRuby benchmarks.
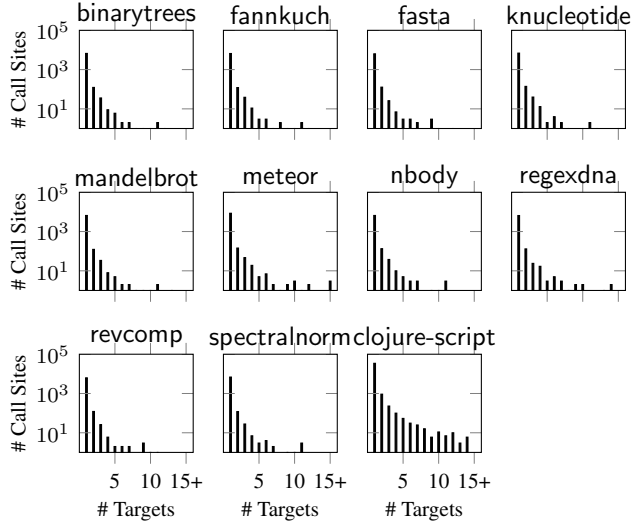

Figure 4. The number of dynamically-dispatched call sites targeting a given number of methods for the Jython benchmarks.

Figure 3. The number of dynamically-dispatched call sites targeting a given number of methods for the Clojure benchmarks.

Figure 5. The number of dynamically-dispatched calls made at call sites with a given number of targets for the Java benchmarks.

constructor. This notion is dynamic in the sense that it may hold only for a particular program execution or for a specific program input [20, 21].

Extending the notion to objects and classes, we distinguish (1) *immutable fields*, assigned at most once during the entire program execution, (2) *immutable objects*, consisting only of immutable fields, and (3) *immutable classes*, for which only immutable objects were observed.

The results shown in Figure 9 indicate that there is a significant fraction of immutable fields (as per our definition) in most of the studied workloads, without significant differ-
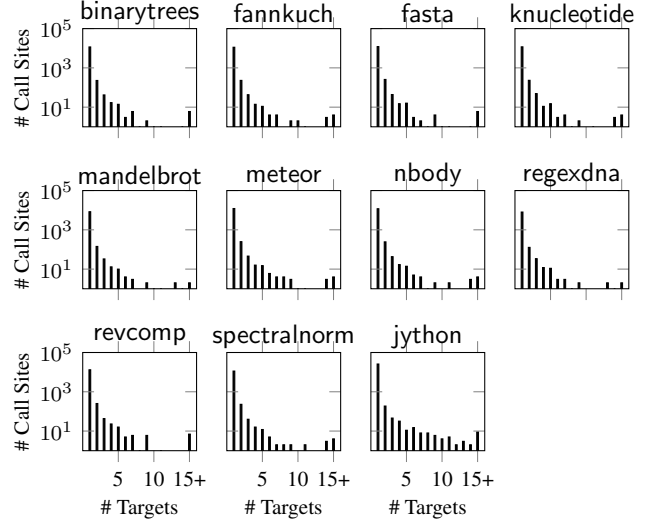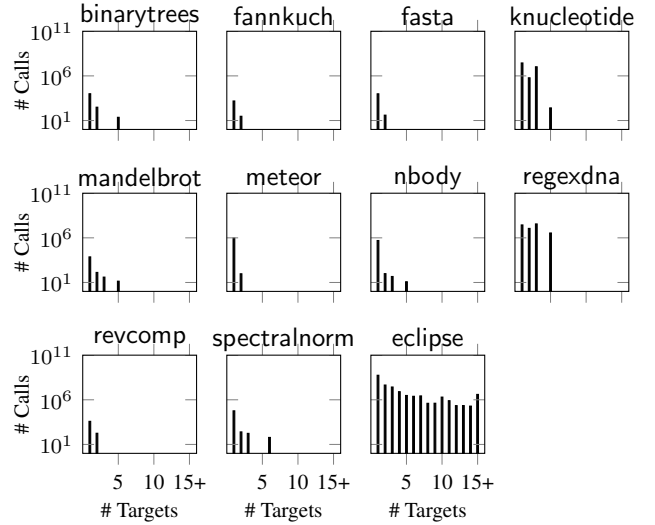
ences between the CLBG and real-world benchmarks. Except in the Java binarytrees CLBG workload, we observed more than 50% of immutable fields in all benchmarks, with Jython having the highest average number of immutable fields.

At the granularity of objects, the results in Figure 10 show varying immutability ratios across different workloads. Apart from few exceptions, the ratios are consistently high, especially for the dynamic languages (mostly over 50%), with Clojure and JRuby scoring almost 100% on five workloads (with four common to both). This can be attributed to the
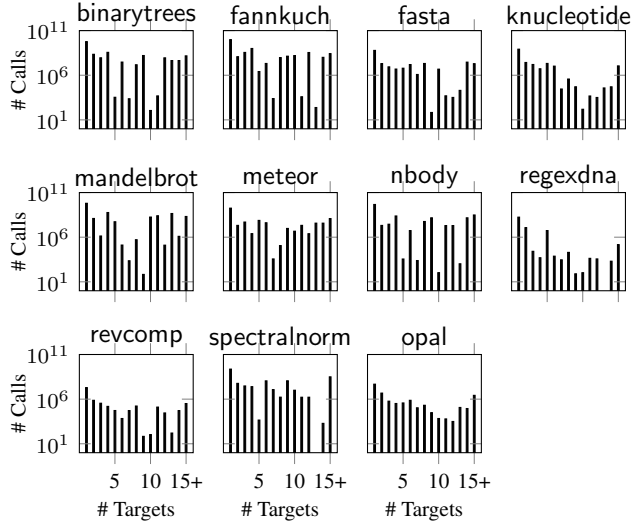
Figure 6. The number of dynamically-dispatched calls made at call sites with a given number of targets for the JRuby benchmarks.
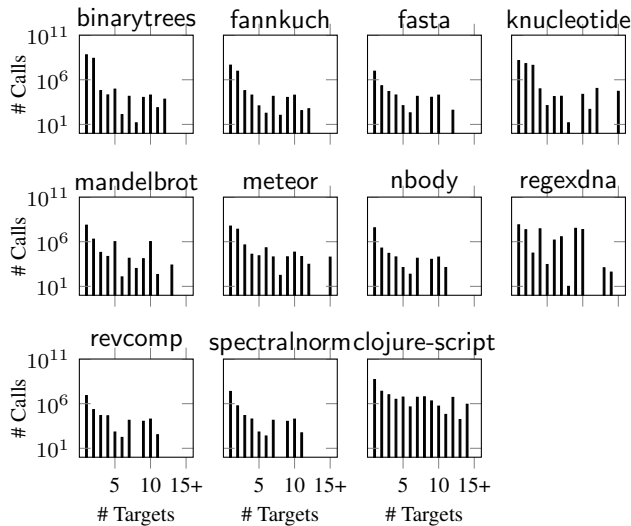

Figure 7. The number of dynamically-dispatched calls made at call sites with a given number of targets for the Clojure benchmarks.
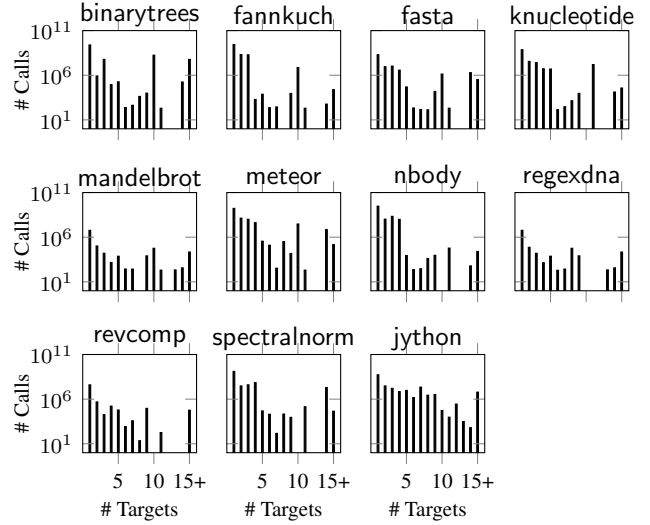

Figure 8. The number of dynamically-dispatched calls made at call sites with a given number of targets for the Jython benchmarks.
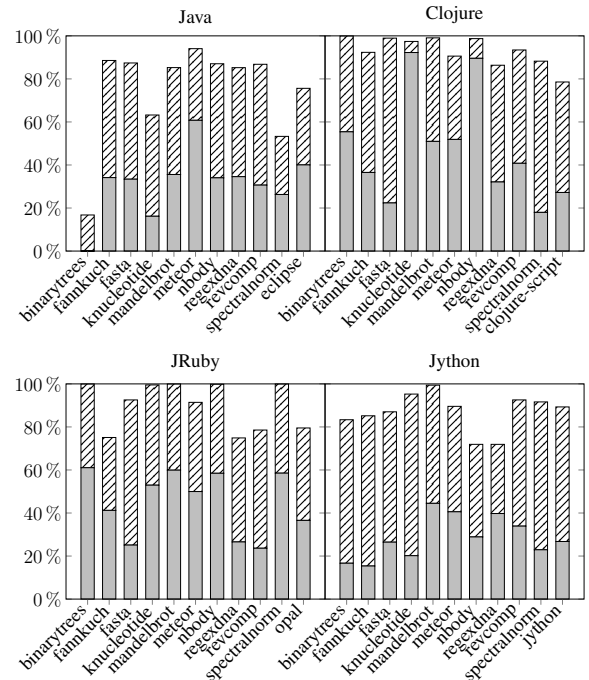

Figure 9. Fraction of primitive (▦) and reference (▨) *instance* fields that are per-object immutable

large amount of boxing and auxiliary objects created by the language runtimes [16].

Finally, at the granularity of classes, the results in Figure 11 show rather consistent results across different workloads (except for Jython), with significant differences between the languages. On average[9], the ratio of immutable classes ranges from 23.5% for JRuby, through 31.7% and 58.8% for Jython and Java, respectively, to 77.5% for Clojure. These systematic differences can be attributed both to different coding styles typical for the particular languages,

and to the number of helper classes produced by a particular dynamic language runtime environment.

3.3 Object Lifetimes

Object sizes and lifetimes characterize program memory management "habits" and largely determine the GC workload. To approximate and analyze it, we used ElephantTracks [19]
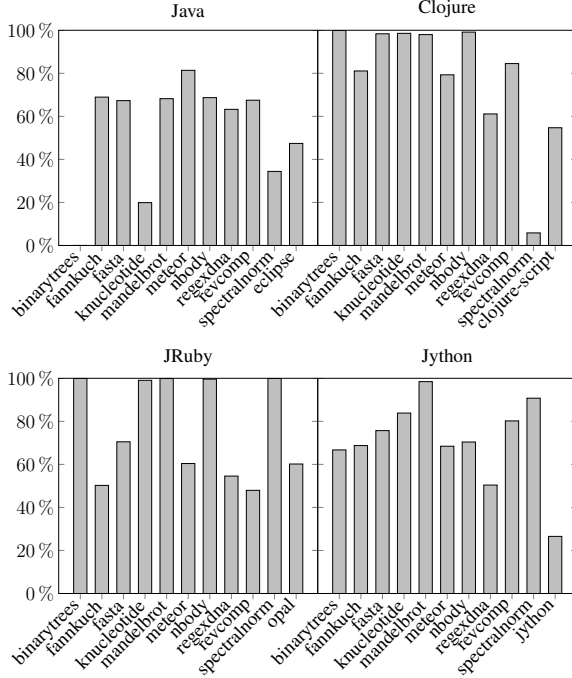
---

[9] Calculated from the total number of classes.

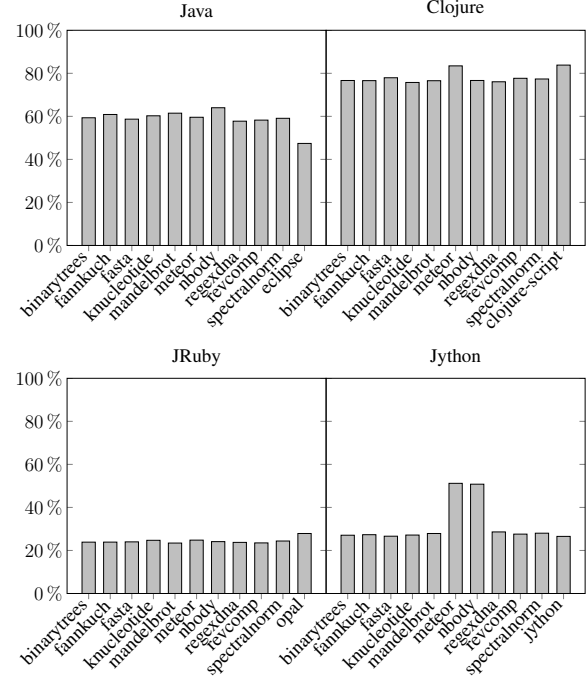**Figure 10.** Fraction of immutable objects

Figure 11. Fraction of immutable classes

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.[10]

The results are summarized in Table 3, where *mark* is the number of times the GC marked an object live, *cons* is the number of allocated objects, and *nursery survival* is 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 to their Java counterparts—in all of them, Java allocates at least one order or magnitude less objects, and in some cases several orders less. Again, given the microbenchmark nature of the CLBG workloads, the results for Java are not too surprising, but they indicate how inherently costly the dynamic language features are in terms of increased GC workload.

The plots in Figures 12, 13, 14, and 15 show the evolution of object survival rate plotted against logical time expressed as cumulative memory allocated by a benchmark.[11]

The results show that even though the dynamic languages allocate many objects, most of them die young, suggesting that they are mainly temporaries resulting from features specific to dynamic languages.
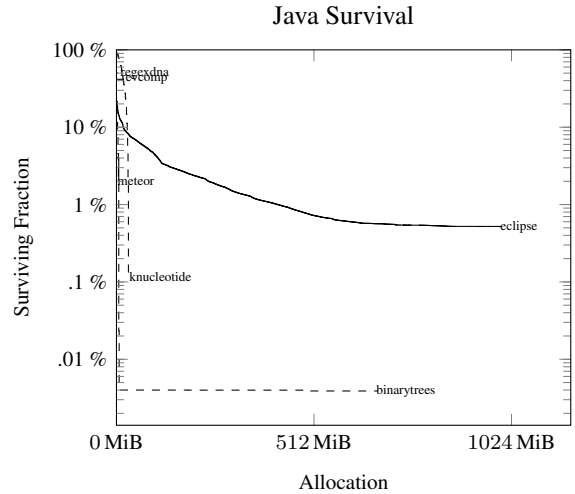

Figure 12. Fraction of objects surviving more than a given amount of allocation in the Java benchmarks

3.4 Unnecessary Zeroing

The Java language specification requires that all fields have a default value of `null`, `false` or `0`, unless explicitly initialized. Our analysis detects fields assigned within the dynamic extent of a constructor without being read before. For such fields, explicit zeroing causes unnecessary overhead [25], because their uninitialized value cannot be observed by the program or the garbage collector. A real JVM may optimize the initialization overhead away more aggressively, e.g., by excluding unused fields where appropriate. However, unini-

---

[10] None of the microbenchmarks allocated enough memory to trigger a full heap (old generation) collection.

[11] The Java fannkuch-redux, fasta, mandelbrot, nbody, and spectralnorm benchmarks are not shown, because they allocate less than 1 MiB.

Benchmark	Mark	Cons	$\frac{\text{Mark}}{\text{Cons}}$	Nursery Survival
Clojure				
binarytrees	4 021 096	147 910 977	0.03	2.72 %
fannkuchredux	38 900	317 062	0.12	12.27 %
fasta	122 560	2 721 195	0.05	4.50 %
knucleotide	1 158 380	18 436 145	0.06	6.28 %
mandelbrot	134 617	2 822 088	0.05	4.77 %
meteor	509 781	10 268 791	0.05	4.96 %
nbody	180 333	5 320 075	0.03	3.39 %
regexdna	197 257	586 176	0.34	33.65 %
revcomp	60 604	437 639	0.14	13.85 %
spectralnorm	563 338	5 592 427	0.10	10.07 %
clojure-script	8 658 091	30 656 512	.28	28.24 %
Java				
binarytrees	1 136 479	29 581 095	0.04	3.84 %
fannkuchredux	0	2226	0.00	0.00 %
fasta	0	2329	0.00	0.00 %
knucleotide	962 320	967 492	0.99	99.47 %
mandelbrot	0	3699	0.00	0.00 %
meteor	10 467	262 816	0.04	3.98 %
nbody	0	2726	0.00	0.00 %
regexdna	1295	2698	0.48	48.00 %
revcomp	1158	2848	0.41	40.66 %
spectralnorm	0	6263	0.00	0.00 %
eclipse	171 766 557	66 569 509	2.58	30.82 %
JRuby				
binarytrees	30 303 133	118 527 001	0.26	7.40 %
fannkuchredux	263 722	10 041 379	0.03	2.63 %
fasta	513 005	27 557 549	0.02	1.86 %
knucleotide	1 788 684	19 762 437	0.09	9.05 %
mandelbrot	81 977 235	200 409 954	0.41	1.81 %
meteor	5 026 466	131 510 575	0.04	1.64 %
nbody	44 632 312	154 314 308	0.29	1.90 %
regexdna	148 158	399 044	0.37	37.13 %
revcomp	109 951	635 720	0.17	17.30 %
spectralnorm	9 262 955	113 332 054	0.08	1.93 %
opal	382 540	2 632 867	0.15	14.53 %
Jython				
binarytrees	73 870 086	297 905 683	0.25	3.22 %
fasta	340 799	3 111 487	0.11	10.95 %
knucleotide	12 538 292	40 411 395	0.31	31.03 %
mandelbrot	2 301 641	80 129 997	0.03	2.87 %
meteor	12 465 700	165 578 823	0.08	2.52 %
nbody	13 095 284	161 984 760	0.08	2.83 %
regexdna	58 047 534	227 296 897	0.26	3.78 %
revcomp	183 874	1 012 778	0.18	18.16 %
spectralnorm	6 131 554	140 908 487	0.04	3.29 %
jython	10 654 369	43 752 983	0.24	24.35 %

Table 3. Garbage collector workload

tialized values of unused fields must not be exposed to the garbage collector, which is strongly related to a particular JVM implementation. Striving to keep our metrics JVM-independent, we do not take these additional optimizations into account, i.e., zeroing of a field not accessed by the program is considered necessary.

Figure 16 shows the amount of unnecessary zeroing (according to our metric) happening in the workloads from the different languages. For the CLBG benchmarks, Clojure exhibits the highest average percentage of unnecessary zeroing (86.8 %), followed by Jython (64.2 %), JRuby (40.8 %) and
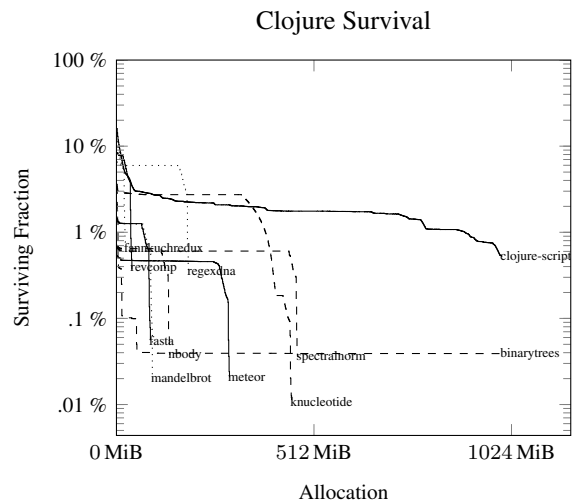

Figure 13. Fraction of objects surviving more than a given amount of allocation in the Clojure benchmarks
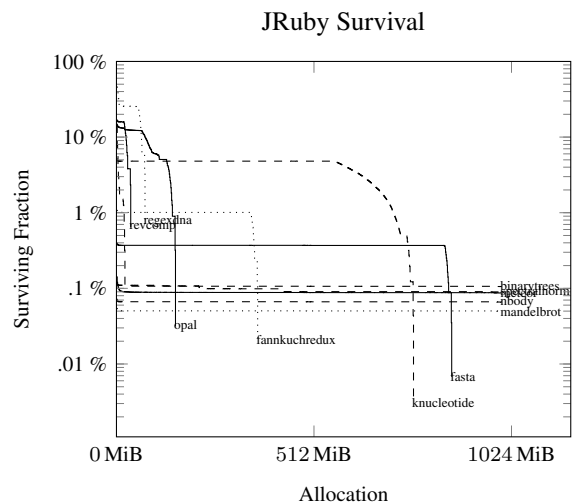

Figure 14. Fraction of objects surviving more than a given amount of allocation in the JRuby benchmarks

Java (39.8 %). Interestingly, this language ordering appears to correlate with the ordering imposed by the percentage of immutable instance fields (shown in Figure 9) with average values of 94.5 %, 91.2 %, 86.8 %, and 74.8 %, respectively. Our results therefore suggest that the more immutable instance fields exist, the more unnecessary zeroing takes place.

3.5 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 the JVM 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
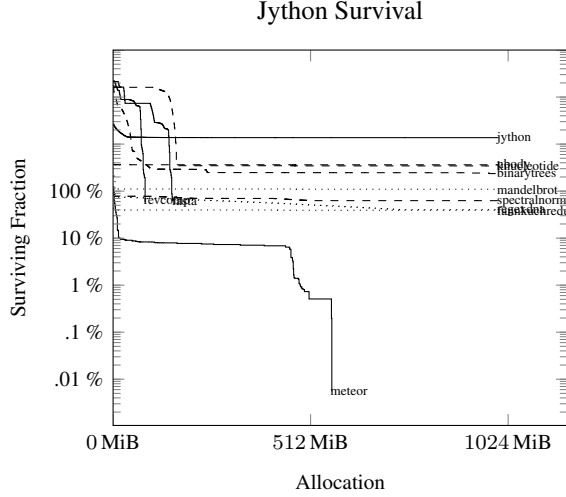
17

Figure 15. Fraction of objects surviving more than a given amount of allocation in the Jython benchmarks
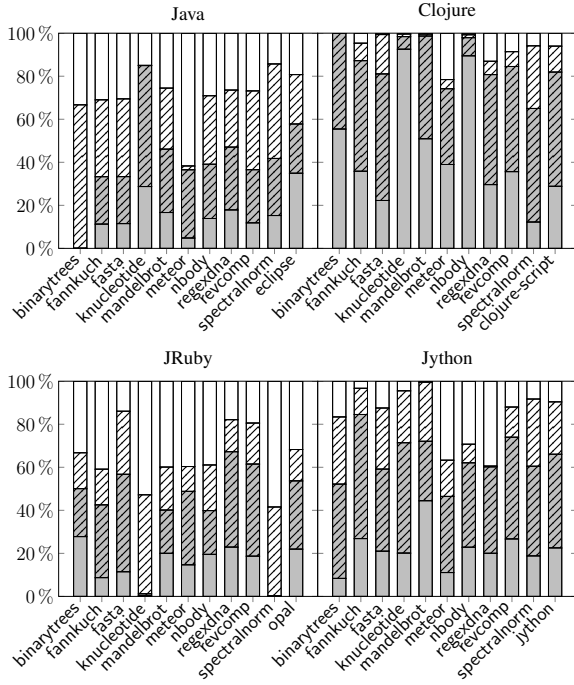

Figure 16. Unnecessary (■, ▨) and necessary (□, ▨) zeroing of primitive (■, □) and reference (▨, ▨) instance fields.

store it explicitly only upon first request (to make it persistent in presence of a copying GC).

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. Since systematic variations in hash code usage were identified between Java and Scala workloads [21], we also analyzed hash code usage of the workloads in our study.
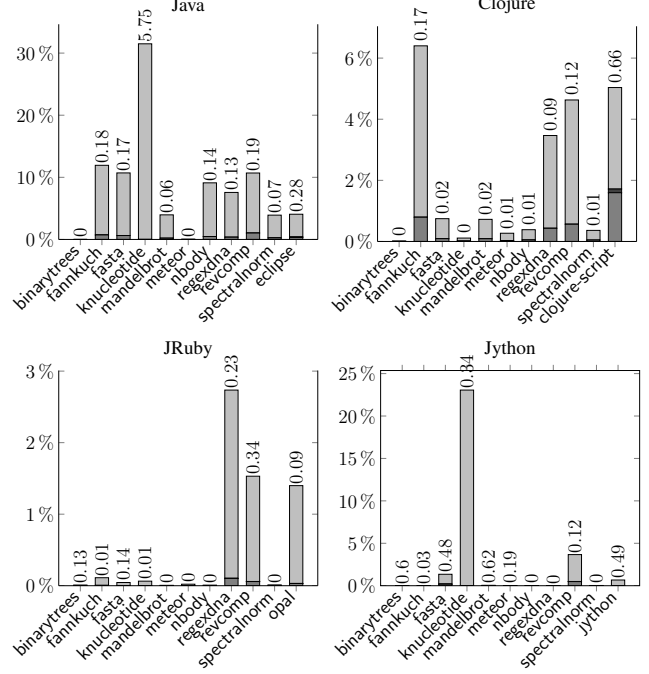

Figure 17. Fraction of objects hashed using an overriden hashCode method (■), the identityHashCode method (■), or both (■), along with an average number of hash operations per object.

The results shown in Figure 17 suggest that the identity hash code is never requested for a vast majority of objects. Despite a comparatively frequent use of hash code in the Java workloads, the use of identity hash code remains well below 0.1% for most workloads. Increased usage of both hash code and identity hash code can be observed in some of the Clojure workloads, specifically in the clojure-script (1.7%), fannkuch (0.79%), revcomp (0.56%), and regexdna (0.43%) benchmarks.

Since dynamic languages appear to produce many short-lived objects (c.f. Section 3.3), the results suggest that object header compression with lazy identity hash code slot allocation is an adequate heuristic for the dynamic language runtimes.

4. Related Work

The research presented in this paper can be seen as a continuation of our previous work [20–22] and complementary to that of Li et al. [16], who 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 similar approach regarding workload selection, but opted for non-interactive real-world applica-

tions to complement our CLBG benchmark mix, 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 [14] and Bull et al. [6] 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. [18] 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 micro-benchmarks.

Daly et al. [9] examine optimization oportunities at the Java-to-bytecode compiler level. The authors analyze the Java Grande benchmark suite [7] 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.

Dufour et al. [11] define a list of sixty 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.

Several other works also use dynamic metrics that hint at different optimization opportunities for JVM languages. Yermolovich et al. [26] propose to run an interpreter of a dynamic language on top of an optimizing trace-based VM, thus avoiding the need to create a custom JIT compiler for a dynamic language. A similar approach is proposed by Gal et al. [12], where the authors create a tracing JIT compiler for the JavaScript VM running in the Firefox web browser. Zaleski et al. [27] 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. Dynamic instruction frequencies and basic block hotness are the fundamental metrics that are used in tracing-based compilation approaches. These metrics are also used in the work by Williams et al. [24], where the authors propose specializing native code to program properties found at JIT-compilation time, thus improving the overall program execution time.

Call site polymorhism metric is used by Mostafa et al. [17] for possible caching of method call targets and inlining. It is also used in the work by Brunthaler et al. [5] to speedup the execution of dynamic languages by caching the call types.

Barany et al. [2] analyze the usage of boxed types in order to eliminate extensive boxing behaviour and instead allocate temporary objects on the heap.

5. Conclusions

The introduction of scripting support and support for dynamically-typed languages to the Java platform made the JVM and its runtime library an attractive target for the developers of new dynamic programming languages. Those are typically more expressive—trading raw performance of the statically-typed languages for increased developer productivity—and by targeting the JVM, their authors hope to gain the performance and maturity of the Java platform, while enjoying the benefits of the dynamic languages. However, the optimizations found in JVM implementations have been mostly tuned with Java in mind, therefore the sought-after benefits do not automatically come just from running on the JVM.

In this paper, we performed workload characterization for 44 different workloads produced by benchmarks and applications written in Java, Clojure, Python, and Ruby. Using our workload characterization suite [20], we collected and analyzed hundreds of gigabytes of data resulting from weeks of running experiments with the aim to contribute to the understanding of the characteristics of workloads produced by dynamic languages executing on the JVM. Due to the lack of a proper benchmarking suite for the dynamic languages, we opted, like Li et al. [16] before us, to use the benchmarks from the CLBG project augmented with several real-world applications as the workloads for our study.

Here we summarize the findings of our study:

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 3.2 for the extended notion) classes and objects.

Object Lifetimes. Compared to Java, the dynamic language workloads allocate significantly more objects, but most of them do not live for long, which suggests at 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.

We acknowledge that the corpus of dynamic JVM language programs analyzed so far does not allow for far-reaching conclusions, and that our findings contribute mainly to an initial characterization of dynamic language workloads on the JVM. In future work, we plan to focus our experiments on phenomena characterized by individual metrics and specialized in-depth analyses.

Acknowledgments

References

[1] D. F. Bacon, S. J. Fink, and D. Grove. Space- and time-efficient implementation of the Java object model. In *Proc. ECOOP*, pages 111–132, 2002.

[2] G. Barany. Static and dynamic method unboxing for Python. In *Proc. Software Engineering (Workshops)*, volume 215 of *LNI*, pages 43–57. GI, 2013.

[3] R. Barik and V. Sarkar. Interprocedural load elimination for dynamic optimization of parallel programs. In *Proc. PACT*, pages 41–52, 2009.

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. OOPSLA*, pages 169–190, 2006.

[5] S. Brunthaler. Inline caching meets quickening. In *Proc. ECOOP*, pages 429–451, 2010.

[6] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for scientific applications. In *Proc. Java Grande*, JGI '01, pages 97–105, 2001.

[7] 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. Java Grande*, pages 81–88, 1999.

[8] J. Castanos, D. Edelsohn, K. Ishizaki, P. Nagpurkar, T. Nakatani, T. Ogasawara, and P. Wu. On the benefits and pitfalls of extending a statically typed language JIT compiler for dynamic scripting languages. In *Proc. OOPSLA*, pages 195–212, 2012.

[9] C. Daly, J. Horgan, J. Power, and J. Waldron. Platform independent dynamic Java virtual machine analysis: the Java Grande forum benchmark suite. In *Proc. Java Grande*, pages 106–115, 2001.

[10] D. Detlefs and O. Agesen. Inlining of virtual methods. In *Proc. ECOOP*, pages 258–278, 1999.

[11] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *Proc. OOPSLA*, pages 149–168, 2003.

[12] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proc. PLDI*, pages 465–478, 2009.

[13] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proc. ECOOP*, pages 21–38, 1991.

[14] R. Hundt. Loop Recognition in C++/Java/Go/Scala. Technical report, Google, 2010.

[15] R. Jones and C. Ryder. Garbage collection should be lifetime aware. In *Proc. ICOOOLPS*, 2006.

[16] W. H. Li, J. Singer, and D. White. JVM-Hosted Languages: They talk the talk, but do they walk the walk? In *Proc. PPPJ*, 2013.

[17] N. Mostafa, C. Krintz, C. Cascaval, D. Edelsohn, P. Nagprurkar, and P. Wu. Understanding the potential of interpreter-based optimizations for Python. Technical report, UCSB, 2010.

[18] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. JSMether: comparing the behavior of JavaScript benchmarks with real web applications. In *Proc. WebApps*, pages 27–38, 2010.

[19] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant tracks: generating program traces with object death records. In *Proc. PPPJ*, pages 139–142. ACM, 2011.

[20] A. Sarimbekov, S. Kell, L. Bulej, A. Sewe, Y. Zheng, D. Ansaloni, and W. Binder. A comprehensive toolchain for workload characterization across JVM languages. In *Proc. PASTE*, pages 9–16, 2013.

[21] 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. ISMM*, pages 97–108, 2012.

[22] 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. OOPSLA*, pages 657–676, 2011.

[23] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. SPECjvm2008 performance characterization. In *Proc. SPEC W. on Computer Performance Evaluation and Benchmarking*, pages 17–35, 2009.

[24] K. Williams, J. McCandless, and D. Gregg. Dynamic interpretation for dynamic scripting languages. In *Proc. CGO*, pages 278–287, 2010.

[25] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley. Why nothing matters: the impact of zeroing. In *Proc. OOPSLA*, pages 307–324, 2011.

[26] A. Yermolovich, C. Wimmer, and M. Franz. Optimization of dynamic languages using hierarchical layering of virtual machines. In *Proc. DLS*, pages 79–88, 2009.

[27] M. Zaleski, A. D. Brown, and K. Stoodley. YETI: a graduallY Extensible Trace Interpreter. In *Proc. VEE*, pages 83–93, 2007.