Mining Method Handle Graphs for Efficient Dynamic JVM Languages

Shijie Xu, David Bremner
IBM Centre for Advanced Studies (CAS Atlantic)
University of New Brunswick, Fredericton,
Canada E3B 5A3
{shijie.xu,bremner}@unb.ca

Daniel Heidinga
IBM Ottawa, 770 Palladium Dr
Ottawa, ON, Canada, K2V 1C8
Daniel_Heidinga@ca.ibm.com

ABSTRACT

The Java Virtual Machine (JVM) has been used as an execution platform by many dynamically-typed programming languages such as Ruby, Python, and Groovy. The main challenge to compile such dynamic JVM languages is choosing the most appropriate implementation of a method for various types of an object at runtime. To address this challenge, a new Java bytecode instruction, *invokedynamic*, has been introduced, allowing users to control the linkage between a call site and a method implementation. With this instruction, a method handle that refers to a method is linked to the call site and then potentially transforms the invocation to a real implementation. As a referenced method of a method handle might in turn refer to other method handles, multiple method handles constitute a *Method Handle Graph* (MHG).

In order to support more efficient dynamic JVM language implementations, we present methods to mine patterns in the method handle graph. We investigate two kinds of method handle patterns: the *transformation pattern* and the *instance pattern*. The transformation pattern refers to a composition of multiple method handle transformations, and the instance pattern refers to equivalent method handles in MHGs. Both patterns are mined by the presented *suffix tree* and *equivalency detector*, respectively, which are implemented as modules in the *Method Handle Mining System* (MHMS). Our experiments on the JRuby Micro-Indy benchmark reveal several findings: a) the frequency of different transformation patterns varies significantly, and the JRuby interpreter prefers a small number of transformation patterns, b) a large proportion of method handles, 28.83%, are equivalent, and most of these equivalent method handles can be eliminated to reduce consumed memory, and c) the distribution of equivalent sets for length-two method handle chains is also uneven. For example, only 7% of these sets have more than 30 equivalent method handle chains. We believe these insights are important steps towards further optimizations based on method handle graphs.

PPPJ '15, September 08 - 11, 2015, Melbourne, FL, USA

ISBN 978-1-4503-3712-0/15/09…$15.00

DOI: http://dx.doi.org/10.1145/2807426.2807440

Keywords

invokedynamic, JSR292, data mining, method handle, dynamically typed language

1. INTRODUCTION

Dynamically-typed languages[1] such as Python, Javascript, Ruby, and Groovy, are becoming increasingly prevalent, as they provide high programming flexibility, fast prototyping and agile interactive development. A highly efficient runtime is crucial to support such dynamic languages. To avoid developing a runtime for each language from scratch, there is a trend to port such dynamic languages to the Java virtual machine (JVM), which can take full advantage of the JVM's maturity and highly optimized JIT compiler.

However, as the JVM was originally designed only for staticly-typed languages, i.e., Java [16], there are many "pain points" [1, 2] to support dynamic languages on the JVM. The key reason is that variables in dynamic languages do not yield any type information at compilation time, while a Java bytecode instruction typically requires type information. For example, a Java instruction *iload_1* indicates that the operand is an *int* type, while the operand variable in *aload_2* is an *Object* type instead of a primitive type. Moreover, existing Java method invocation instructions, except *invokestatic*, also require availability of receiver types at compilation time. Although several solutions such as Java Reflection [12, 14] have been proposed, they are still inefficient due to the repeated type and security checks during method invocation.

To better support dynamic languages, Java Specification Request (JSR) 292, entitled *Supporting Dynamically Typed Languages on the Java Platform*, introduced a new Java bytecode instruction, *invokedynamic*, to allow user-defined linkage behavior. By using this new instruction, a method handle, which is a typed and direct reference to an underlying method or field with potential method transformations[2], can be linked to a program call site at runtime. The composition of method handles then constitutes a method handle graph (MHG), representing a number of method type transformations from a call site to the target method type. An illustrative example is shown in Figure 1, where a *GuardWithTesthandle* method handle, *0xFFF5BA48*, references

[1]In this paper, we use the terms dynamic, dynamic JVM and dynamically-typed language interchangeably.

[2]These method transformations include but not limited to argument insertion, removal, conversion, and substitution [9, 10].

an if-else Java method *adapter* in Listing 1 and transforms the invocation to either B (*trueTarget*) or *falseTarget*.

```
T adapter(Object obj1, Object obj2) {
 if (A.invokeExact(obj1))
   return B.invokeExact(obj1, obj2);
 else
   return falseTarget.invokeExact(obj1,
       obj2);
}
```

Listing 1: Guard With Test Method handle

With the *invokedynamic* instruction, it is the responsibility of the language interpreter's implementor to build MHGs to transform the invocation at a call site to a real method implementation. Therefore, it is necessary to mine usage patterns in MHGs on the JVM level for detection of inefficient method handle usage as well as discovering other potential optimizations. In this paper, we mainly investigate two kinds of patterns: *transformation pattern* and *instance pattern*. The transformation pattern refers to the composition of method handle transformations, and the instance pattern refers to the equivalent MHG (An MHG is also represented by its root method handle, from which all method handles in the MHG are reachable). The benefit of analyzing these patterns is two-fold. First, these patterns can be feedback to the language implementers, leading to a more efficient use of MHG. Second, the mined patterns can be used to optimize dynamic languages at runtime. As a method handle invocation is normally compiled to the JVM instruction *invokevirtual*, which defers method linkage according to receiver's type at program runtime (i.e., late binding[3]), compiling these mined long and frequent method handle invocation chains into a single method handle is potentially significant to reduce the number of late bindings and benefit program performance.

Our main contributions are as follows. We first identify two method handle patterns, the transformation pattern and the instance pattern, and provide related mining approaches to detect such patterns from MHGs. We further build MHMS — an offline data mining framework for MHGs — to collect method handle traces and conduct data mining for both patterns, as a proof of concept. We also conduct experiments with the MHMS. Our results with the Micro-Indy benchmark for the Ruby language, one component of the Computer Language Benchmark Game (CLBG) benchmarks for different language implementations, reveal several findings as follows.

- The frequency of different transformation patterns varies significantly, and a small number of transformation patterns occur much more frequently than the others in the JRuby interpreter.
- A large proportion of method handles, an average of 28.83% in our test, are equivalent.
- The distribution of equivalent length-two method handle chains is also uneven. According to our experiment, only 7% of method handle chain sets have more than 30 equivalent chains, which implies an optimization on a small number of chains can maximize performance improvement.

[3]Late binding or dynamic binding is a widely used technology in object-orient programming languages where a method being called upon an object is looked up by its name at program runtime rather than static compilation. Compared to early binding (static binding), late binding has performance impacts on a program.

The rest of this paper is organized as follow. Section 2 introduces some background on the *invokedynamic* instruction, method handles. Section 3 presents how to mine useful patterns from the MHG. Section 4 provides an overview of our Method Handle Mining System (MHMS) framework, where the provided method handle mining approaches are implemented. Section 5 shows the mining results and findings on JRuby CLBG Micro-Indy benchmark. Section 6 discusses related work on method handles and program behaviors. Finally, Sections 7 and 8 conclude this paper.

2. BACKGROUND

2.1 Method Invocation Instruction in the JVM

A Java Virtual Machine (JVM) instruction contains an opcode, followed by zero or more operands. The data type (e.g., *int*, *double*, or complex class) of the performed operands is typically embedded in the JVM instruction [7, 8]. For example, the instruction *areturn* implies the returned object is an object reference, while the performed operand for the instruction *ireturn* is *int* type.

Methods in Java are also strongly typed, and a method invocation requires an exact matching of method name and type signatures between method definition and object references that are atop of JVM stack. Three of the existing JVM method invocation instructions, i.e., *invokevirtual*, *invokespecial*, and *invokeinterface* (*invokestatic* is an exception), treat the first argument in the stack as invocation receiver and statically cast it to the declared receiver's type. Due to variables' un-revealed type information during compilation, dynamically typed languages are not supported efficiently by the JVM and have many "pain points" in their Java implementations, e.g., failed inlining and polluted profile [19]. To address these "pain points", a new invocation instruction, *invokedynamic* was introduced in Java 7; this instruction aims to support non-Java languages on the JVM. The distinctive feature of this new instruction is that it supports a user-defined bootstrap method which links a method handle to the dynamic call site at program runtime.

2.2 Method Handle

A method handle is a typed, direct reference to an underlying method, a constructor, or a field with potential method type transformations [6, 9, 10]. These transformations include method argument insertion, removal, and substitution. A summary of existing transformations is shown in Table 1.

```
1 MethodHandle cat = lookup().
      findVirtual(String.class, "concat"
      , methodType(String.class, String.
      class));
2 assertEquals("xy", (String) cat.
      invokeExact("x", "y"));
3 MethodType bigType = cat.type().
      insertParameterTypes(0, int.class,
       String.class);
4 MethodHandle d0 = dropArguments(cat,
      0, bigType.parameterList().subList
      (0,2));
```

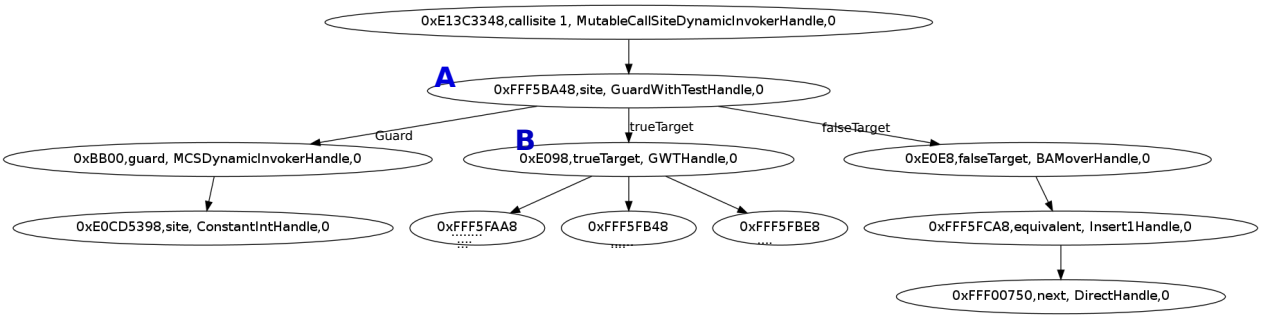


Figure 1: Method Handle Sample

Adapters	Description
convertArguments	pairwise cast, (un)box, pad/truncate
dropArguments	ignore N consecutive arguments
insertArguments	insert N arguments at given location
permuteArguments	reorder (also, drop and/or duplicate)
collectArguments	collect N trailing (enter varargs)
spreadArguments	spread N trailing (exit varargs)
filterArguments	apply filter to arguments
guardWithTest	wrapper if-else routine
foldArguments	call target with arguments which contain pre-processed result
catchException	catch exception if target throws exception

Table 1: The Method Handle Adapters [19]

```
5  assertEquals(bigType, d0.type());
6  assertEquals("yz", (String) d0.
       invokeExact(123, "x", "y", "z"));
```

Listing 2: Method Handle sample

There are two kinds of method handles, *direct* and *non-direct* method handles. A direct method handle is a method handle that refers to a method or field directly without any transformation. A direct method handle does not have any other method handle field members, and an invocation of a direct method handle is equivalent to the target method invocation. In contrast to a direct method handle, a non-direct method handle is a method handle that has at least one method handle field member. Normally, a non-direct method handle involves method type transformations. For example, the direct method handle *cat* in Listing 2 (from [11]) refers to the function *concat* of the *String* class, and its method type is *(String)String*, indicating the method only accepts one *String* parameter and returns *String* value, while the non-direct method handle *d0* adapts an invocation method type *(String, String, String)String* to the *cat*.

3. METHOD HANDLE DATA MINING

3.1 Method Handle Graph

A Method Handle Graph (MHG) is composed of multiple method handle instances, and each method handle (specifically, non-direct method handle) represents a single method invocation transformation. Therefore, an MHG expresses a sequence of method type transformations for an invocation at a call site. Figure 1 shows a sample MHG created by the JRuby interpreter. In this graph, both A and B are *GuardWithTestHandle* (*GWTHandle* is short for *GuardWithTestHandle*), and chain $A \rightarrow B$ transforms the invocation at A by the if-else routine twice.

In this paper, an MHG is abstracted by a number of nodes and edges. A node is defined as a four-member tuple: (*nodeID*, *classType*, *fieldName*, other), and a directed edge between two method handles indicates there is a reference dependence between the source and the target method handle. The *nodeID* is the method handle's memory address. The *classType* is the transformation name, i.e., class name, of this method handle instance. The *fieldName* is the name of the source method handle's field that references this method handle, and its value is also associated with the directed edge from the source method handle to the current method handle. *fieldName* is invalid if a method handle has more than one parent. Normally, the root method handle of a graph is the method handle that is linked to the call site at runtime, and its *fieldName* value is set as "call site". In this paper, a root method handle is defined as the one which can reach all method handle nodes in the MHG.

3.2 Method Handle Graph Patterns

Since a method handle is a reference to a Java method or field, a single method handle inherently has two characteristics: method type *type* and transformation name *TN*. The method type of a method handle represents arguments and the return type accepted and expected for the method handle. For example, the *type* of method handle *cat* in Listing 2 is *(String)String*, indicating invocation of *cat* requires a *String* argument and will return a *String* object. The transformation name *TN* is the name of either the method handle's class type or the API that creates this method handle. For example, a transformation name of a method handle created by API *guardWithTest* is *GuardWithTestHandle*, and the transformation name of the root method handle in Figure 1 is *MutableCallSiteDynamicInvokerHandle*.

Correspondingly, we define two kinds of patterns for a method handle graph: *transformation patterns* and *instance patterns*. These are discussed in Section 3.3 and Section 3.4, respectively.

3.3 Mining Frequent Transformation Patterns

Given an MHG, a *transformation chain* refers to a transformation sequence from the root method handle to a direct

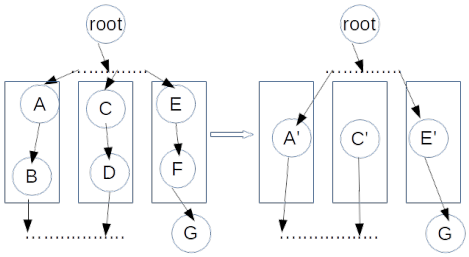


Figure 2: Transformation Examples

method handle, and *frequent transformation patterns* are defined as the method handle transformation sub-chains that appear in an MHG (or MHGs) with a frequency no less than a user-specified threshold. For example, two of the transformation chains in Figure 1 are
MutableCallSiteDynamicInvoker → *GuardWithTestHandle* → *MCSDynamicInvokeHandle* → *ConstantIntHandle*,
and
MutableCallSiteDynamicInvoker → *GuardWithTestHandle* → *MoverHandle* →*Insert1Handle* → *DirectHandle*.

Mining frequent transformation patterns can provide opportunities for method handle invocation optimizations. As shown in Figure 2, there are three method handle sub-chains, $A \rightarrow B$, $C \rightarrow D$, and $E \rightarrow F$. If A, C, and E have the same transformation name, while B, D, and F also have the same transformation name, then these three transformation sub-chains are equal, and they might be a frequent transformation pattern. Accordingly, an intuition is to make intensive optimization on this frequent transformation chain by merging the chain into a single but more efficient transformation (e.g., inline cache), so that a number of late bindings during method invocation can be reduced (three sub-chains are changed to A', C', and E' as shown on the right of Figure 2).

Mining frequent transformation patterns consists of two steps: *graph conversion* and *suffix tree mining*. In the graph conversion step, an MHG is converted to a transformation chain set, which allows duplicate transformation chains. The duplicates in a set might be very common for a large MHG, because the transformation chains of different method handle chains could be the same even though these method handle chains are unique. In practice, duplicate transformation chains in a set are simplified by a tuple, (*chain*, *weight*), where the *weight* is the *chain*'s frequency.

The second step is *suffix tree mining*, which aims to detect the most frequent transformation chains. The procedure to find the most frequent transformation sub-chains is as follows. First, a unique chain tuple is created for a pair of two arbitrary unique chains in the set, and a weight is associated with this new tuple, indicating its frequency. For example, given two transform chains ($chain_i$, $weight_i$) and ($chain_j$, $weight_j$), the corresponding unique chain tuple is

$$(chain_i, chain_j, weight_j \times weight_i)$$

which indicates that the permutation ($chain_i, chain_j$) occurs $weight_j \times weight_i$ times. Second, a suffix tree is built for the chain tuple and the top $2N$ longest transformation sub-chains are recorded as ($subTransForm_k$, o_k), in which o_k is the occurrence of this transformation sub-chain in both $chain_i$ and $chain_j$, and N is the number of most frequent sub-chains to be mined. Third, all detected transformation sub-chains are merged, and the rule to re-calculate frequency of one transformation sub-chain is

$$sum(k) = \sum_{i,j} o_k \times weight_j \times weight_i.$$

for i^{th} and j^{th} chains. To reduce the computational cost, the transformation chains with a length less than 2 are removed before mining. The reason is that a length-two chain inherently consists of two method handles, i.e., a call-site method handle and a direct method handle, both of which actually do not imply any transformation. Finally, transformation chains are sorted by $sum(k)$, and the most frequent N transformation chains are produced as the mining results.

3.4 Mining Instance Pattern

Instance pattern refers to equivalent method handle subgraphs in given MHGs, and the purpose of their mining is to find equivalencies for future runtime method handle optimizations, e.g., method handle deduplication for efficient memory usage. Two MHGs are equivalent, indicating that all corresponding nodes in the method handle graphs are also equivalent.

Algorithm 1 Two Method Handles's Equivalency Detection

```
1: MH is short for method handle
2: TN is Transformation Name.
3: a: method handle candidate to be added
4: b: The 1st method handle in a equivSet of list equivMap.get(a.TN).
5: procedure Detection(a, b)
6:     if a.countChild()!=b.countChild() then
7:         return false
8:     end if
9:     if a.isDirect() and b.isDirect() then                ▷ Rule 3
10:        return a.targetMethod().isEqual(b.targetMethod())
11:    end if
12:    if a.boxDataCompare(b) is false then                 ▷ Rule 4
13:        return false
14:    end if
15:    for all HM Field's name f in a do                    ▷ Rule 2
16:        MH a' = a.getField(f); MH b'=b.getField(f);
17:        if precheck(a',b') then
18:            continue    ▷ a' and b' previously compared
19:        end if
20:        Ta=a'.getTerminalFuns();Tb=b'.getTerminalFuns()
21:        if !Ta.equal(Tb) then
22:            return false;
23:        end if
24:        if !Detection(a', b') then
25:            return false
26:        else
27:            b'.getEquivSet().add(a')
28:        end if
29:    end for
30:    b.getEquivSet().add(a)
31:    return true
32: end procedure
33: procedure precheck(a', b')
34:    return b'.getEquivSet().contains(a')
35: end procedure
```

Definition 1. *Let G_{mh} denote the method handle graph that starts from root mh. Two MHGs G_{mh} = (V, E) and $G_{mh'}$ = (V', E') are* equivalent *if there exists mappings $f : V \rightarrow V'$ and $g : E \rightarrow E'$, such that $\forall\{u, v\} \in E$ $g(\{f(u), f(v)\}) \in E'$.*

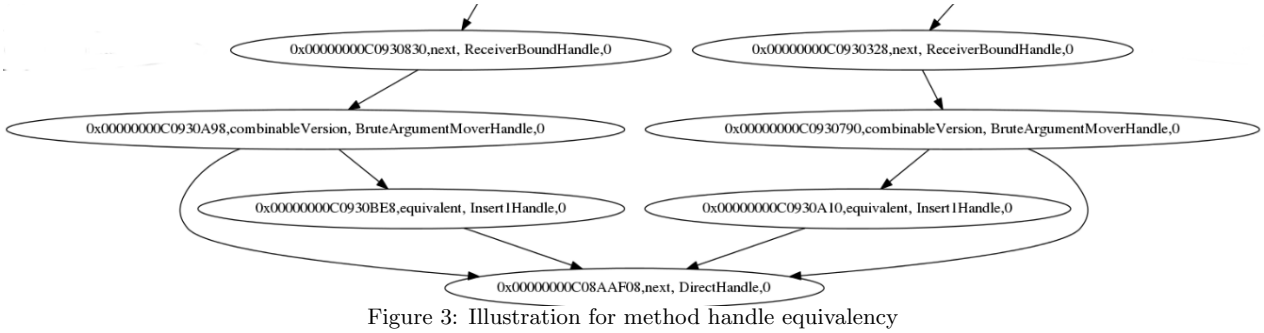


Figure 3: Illustration for method handle equivalency

Therefore, the comparison starts at root method handle, and the combined equivalence predicate is the *and* operation of below four rules.

Rule 1. *Both mh and mh' have the same transformation name (*TN*) and type (*type*).*

Rule 2. *The child method handles referenced by the same* fieldName *in* mh *and* mh*' are also equivalent.*

Rule 3. *The referenced functions have the same method type and signature if both are direct method handles.*

Rule 4. *If present, corresponding boxed data values are also equivalent.*

According to these rules, two method handles in Figure 3, i.e., *0xC0930830* and *0xC0930328*, are equivalent because they have the same transformation name and equivalent children.

The overhead to detect all equivalency recursively is huge when the size of the MHG becomes large. In order to reduce this cost, we take advantage of method handle types and the unique method handles that have already been identified. In our solution, we use an equivalent method handle set, *equivSet*, to keep method handles that have already been classified as equivalent. All method handles in an *equivSet* have the same transformation name and type, and a method handle will be added to an *equivSet* if it is equivalent to either one already in the set. A global *equivMap* is created to index the *equivSet*s with the same transformation name, which has type

```
1    Map<TransformName, List<EquivSet>>
         equivMap;
```

The main task of equivalency detection is to place a method handle into the right *equivSet*. Therefore, the candidate method handle a is compared to a method handle b from *equivSet*, which is the head of the list *equivMap.get(a.TN)*, and added to the *equivSet* if comparison returns true. Otherwise, a new *equivSet* with a is created and appended to the corresponding list in the *equivMap*. The detailed comparison between a and b is shown in Algorithm 1. One of the procedures is *precheck* that checks whether two child method handles, a' and b', are in the same *equivSet*. The true return value of *precheck* indicates both a' and b' are equivalent, and the comparison has already done previously. Similiarly, the function *getTerminalFuns()* returns all the names of the terminal functions that the method handle, the caller, can reach, and the comparison of *getTerminalFuns*'s result aims to avoid unnecessary recursive comparison. For example, both *0xC0930A98* and *0xC0930790* in Figure 3 have been placed into the same *equivSet*, and this information can be directly used for comparison with *0xC0930830* and *0xC0930328* so that recursive access of child method handles can be avoided.

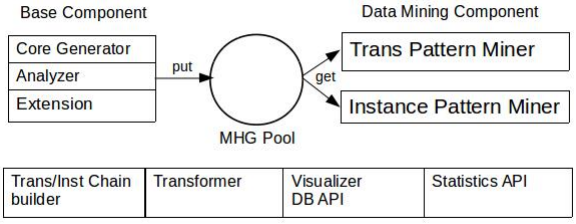


Figure 4: Method Handle Mining System (MHMS) Overview.

4. METHOD HANDLE MINING SYSTEM

A *Method Handle Mining System* (MHMS) is built to implement the data mining task for dynamically typed language implementation on the JVM. MHMS provides a way to collect MHG traces at program runtime and then perform the data mining task offline. Furthermore, MHMS also provides some extensions, e.g., MHG visualizer and trace dumper, for advanced method handle trace analysis if the developers have their own specific purposes.

The whole MHMS mainly comprises two components, *base component* and *data mining component*, which are shown in Figure 4. The base component provides all basic functionality for trace collection and formatting at program runtime, and builds method handle graphs, which will be put in the MHG pool. In contrast to the base component, the data mining component fetches MHG from the pool and conducts data mining tasks, including the aforementioned *transformation pattern mining* and *instance pattern mining*. Besides these two components, an API layer is also provided to facilitate tasks in the base component and data mining component. For example, the *chain builder* module in the API layer provides functions that convert an MHG to transformation chain and instance chains directly, while the *Statistics API* formats the data mining results and generates a report.

4.1 Base Component

The base component consists of three separate modules:

core generator, analyzer, and extension.

Core Generator.

This module consists of a number of configuration files and Python scripts, and its main purpose is to generate JVM core files by scheduling dynamic languages on the JVM. In total, three kinds of configurations are provided here, benchmark script configurations, including scripts location and parameters, dynamic typed language interpreters (e.g., JRuby interpreter), and JVM options that trigger JVM core images before JVM exits. With these configurations, the benchmark scripts can be scheduled on the JVM successfully, and the JVM core files that have snapshots of created method handle graphs are collected.

Analyzer.

This module restores MHGs from the core files. It works as follows. First, it moves the reader cursor to a region where all Java objects reside in the cores and then iterates all objects' call sites which have been linked. Second, for the method handle that is linked to the call site in the core, the analyzer creates a method handle as the root method handler of MHG and initializes it with attributes read from the core. These attributes include field names, transformation name (class type), and method type. Third, similar to the second step, the analyzer visits the method handle structure in the core and adds a newly created method handle to the MHG recursively. This procedure continues until all direct method handles are reached.

Note that the analyzer is built upon two third party libraries: IBM Direct Dump Reader (DDR) tool, which provides a set of Java interfaces for reading J9 structure blob from the core file (running J9 processing), and IBM Diagnostic Tools for Java (DTFJ) [4], which wraps low level DDR interfaces to load and analyze core file structures. Similar to HotSpot, J9 here is IBM's independent JVM implementation [5]. Our analyzer is also exposed as the DDR command: *dumpmethodhandle*.

Extension.

In order to support advanced data mining methods and further work, the base component also provides a default API to write MHG data to an external disk. This extension provides external tool for MHG visualization and future data mining purposes.

The data persistence is accomplished by graph dumping. In our implementation, the data is written to disk file in Graphviz dot format [13]. This dot format data makes any languages with the Graphviz extension capable of visualizing MHG directly.

4.2 Data Mining Component

Following the analyzer tool component, the data mining component fetches the analyzer output, i.e. MHG set, from MHG pool and applies some pre-defined algorithms to these graph data. The purpose of these algorithms is to provide quantitative analysis for both transformation and instance patterns inside the MHG.

In the data mining component, two kinds of patterns, i.e., transformation pattern and instance pattern, are implemented in the *Transformation Pattern Miner* and *Instance Pattern Miner* respectively. In the *Transformation Pattern Miner*, the input, MHG, is first converted to a transformation chain set and then a suffix tree is created for mining. Similarly, an equivalent method handle detector in the *Instance Pattern Miner* is created to compare equivalency of two method handles.

5. EVALUATION

In this section, we first introduce our evaluation methodology including the hardware platform, the evaluated benchmark suite, and execution parameters. Then we show the mining results of transformation patterns and instance patterns, and our findings for these results.

5.1 Evaluation Methodology

Components	Configurations
CPU	4 × Intel Xeon E7520 1.8 GHz
Cores/Threads	16/32
DRAM	64 GB DDR3 800 MHz

Table 2: Hardware Platform Configuration

The mining of transformation and instance patterns is implemented in the MHMS. Our experiments are conducted on an Intel Xeon server; the hardware configuration is shown in Table 2.

In the experiments, we configured MHMS with JRuby interpreter, a Java implementation of the popular Ruby programming language. Correspondingly, the evaluated benchmark is the JRuby Micro-Indy benchmark of Computer Language Benchmarks Game [3] (CLBG), which is used to compare different interpreters' performance of the same languages. The benchmark contains 41 JRuby test cases in total. The Java Virtual Machine used in our experiments is the IBM J9 JVM, and the JRuby interpreter (Version 1.7.6 with 1024M max heap size and 10240K size for thread stack) with the *invokedynamic* feature enabled.

5.2 Transformation Pattern Data Mining

As discussed previously, the transformation pattern data mining can find frequent transformation chains. Meanwhile, the *Statistics* module in the MHMS summarizes the mining results.

In order to quantify the mining results, a metric, called *Relative Frequency (RF)*, is used to indicate the relative frequency of a transformation chain. This metric is defined as

$$\mathrm{RF}_i = \frac{n_i}{\sum_j n_j}, \tag{1}$$

where n_i is the frequency of i-th transformation chain found in all MHGs. According to this definition, the higher the *RF* value is, the more frequently this transformation chain occurs.

Finding 1. *The frequency of the different transformation patterns varies significantly, and a small number of transformation patterns occur much more frequently than the others.*

There are, in total, 412 transformation chains identified by the MHMS and their lengths range from 2 to 10. The 11 most frequent transformation chains are shown in Table 3, in which *BruteArgumentMoverHandle* denotes a special kind

RF	Pattern
10.67%	GuardWithTestHandle → GuardWithTestHandle
7.695%	GuardWithTestHandle → BruteArgumentMoverHandle
5.957%	GuardWithTestHandle → GuardWithTestHandle → BruteArgumentMoverHandle
3.754%	BruteArgumentMoverHandle → Insert1Handle
3.274%	GuardWithTestHandle → BruteArgumentMoverHandle → Insert1Handle
2.853%	GuardWithTestHandle → GuardWithTestHandle→ BruteArgumentMoverHandle → Insert1Handle
2.462%	BruteArgumentMoverHandle → DirectHandle
1.988%	BruteArgumentMoverHandle → PermuteHandle
1.949%	GuardWithTestHandle → GuardWithTestHandle → FoldNonvoidHandle
1.498%	GuardWithTestHandle → BruteArgumentMoverHandle → DirectHandle
1.30%	BruteArgumentMoverHandle → Insert1Handle → AsTypehandle

Table 3: Transformation Pattern Statistics

of method handle that can hold additional values to pass to the next method handle. According to this table, the frequency of the different transformation chains varies significantly, and a small number of transformation patterns occur far more frequently than others. In the result, the chain with the largest RF accounts for 10.67% of all identified chains, while the 11^{th} most frequent chain only accounts for 1.3% of all identified chains. The short transformation chains are likely to have larger RF values while the longer chains are opposite. For example, the lengths of the top three chains with largest RF values are either 2 or 3.

Moreover, we only select length-two transformation chains from the mining results and re-calculate their RFs. Our results show that the accumulated RF of the top 5 transformation chains, which are the 1^{st}, 2^{nd}, 4^{th}, 7^{th}, and 8^{th} transformation chains in Table 3, is 68.28% while that of the remaining 214 length-two transformation chains is only 31.72%. This data supports that a small number of chains occurs far more frequently than others.

Finding 2. *The JRuby interpreter has preferences for some specified transformation patterns.*

We can also easily observe that some kinds of method handles, e.g., *GuardWithTestHandle* and *BruteArgumentMoverHandle*, appear very frequently in these transformation chains. This is determined by the implementation of the language interpreter. In other words, the dynamic JVM language implementations have preferences for some specified transformations. In our example, JRuby interpreter is likely to have a number of if-else method invocation transformations (*GuardWithTestHandle*) and argument permutation transformations (BruteArgumentMoverHandle).

Both **Finding** 1 and **Finding** 2 suggest optimization opportunities for both language implementations and JVMs. On the one side, both mining results (especially if generalized to the appropriate dynamic language) make language implementations aware of overall transformation chains during method invocation, and drive them to use it efficiently. On the other side, these results also reduce the number of optimization candidates sharply, and make it feasible for a JVM to intensively optimize these small numbers of transformation chains and method handles within at an affordable cost. That is, a JVM can only concentrate on optimizing the most frequent transformation chains, e.g., *GuardWithTestHandle→GuardWithTestHandle*, and compiling them into native code by Just-In-Time compilers. The unnecessary costs for optimizing infrequent chains are avoided, leading to negligible overhead on the execution time of programs.

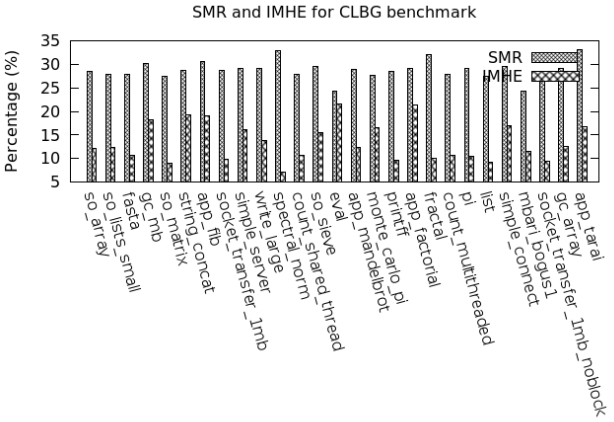


Figure 5: Percentage of saved memory theoretically

5.3 Instance Pattern Data Mining

In order to quantify our instance pattern data mining results, two metrics, *SMR* and *IMHE*, are used. The *SMR* stands for Saved Memory Ratio, which is the ratio of the number of method handles can theoretically be eliminated to the total number of the method handles created during program runtime. With the assumption that all method handles in an *equivSet* can be represented by one of method handles in the *equivSet*, a higher *SMR* value indicates that more method handles can be eliminated, thus increasing efficiency. More specifically, *SMR* can be formally defined as

$$\text{SMR(j)} = \frac{N_j - S_j}{T_j}, \tag{2}$$

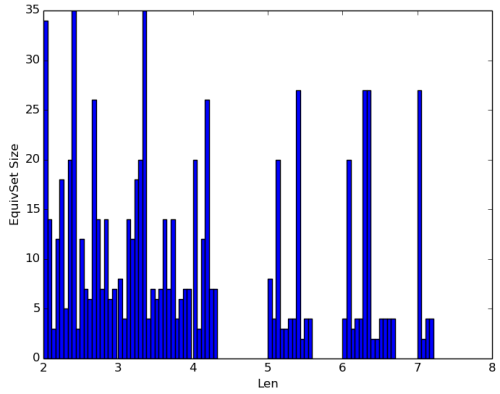
where j is a benchmark test name, N_j is the number of equivalent method handles[4] of j, S_j is the number of equivalent sets *equivSet*s, and T_j is total number of method handles for the benchmark.

IMHE, Inverse of Method Handles per Equivalent Set, represents a distribution of equivalent method handles per equivalent set. The smaller this value is, the more equivalent method handles there are per equivalent set. *IMHE* can be formally defined as

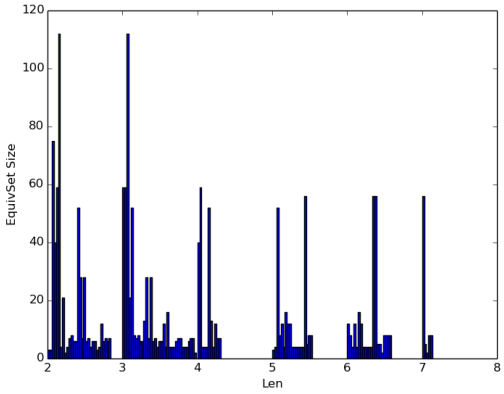
$$\text{IMHE(j)} = \frac{S_j}{N_j}. \tag{3}$$

Finding 3. *A large number of equivalencies exist among*

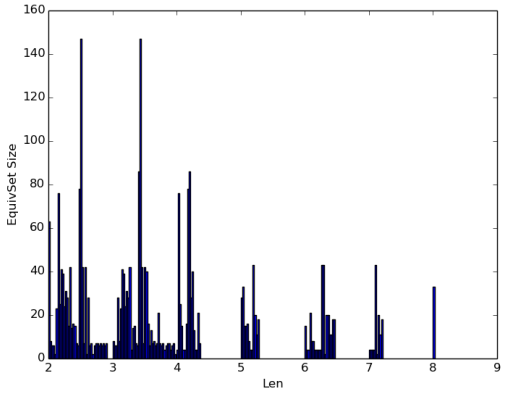
[4]An equivalent method handle is the one that is in a equivalent set *equivSet*, and there must be more than one other method handle, which is equivalent to it.



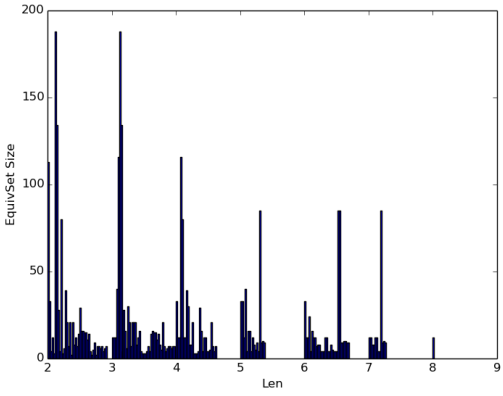
(a) app_tarai



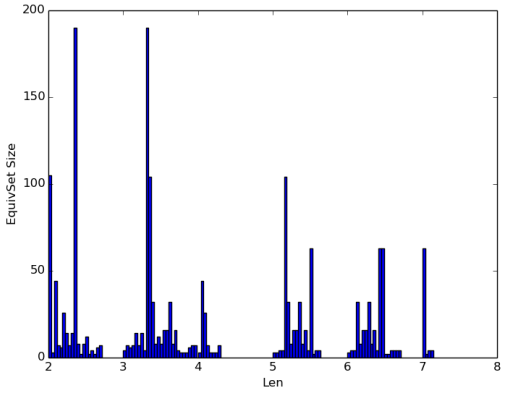
(b) fractal



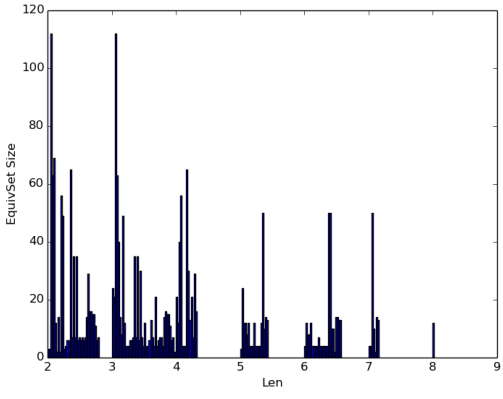
(c) mbari_bogus1



(d) printff



(e) pi



(f) socket_transfer_1mb

Figure 6: Equivalent MethodHandle Set Chain Length Distribution

method handles, and a method handle has 7.4 equivalent method handles on average.

The distributions of both *SMR* and *IMHE* are shown in Figure 5. We can see that *SMR* of different test cases from JRuby CLBG are relatively stable, ranging from 25% to 32%, and the average *SMR* for all cases is 28.83%. This result is very encouraging as it demonstrates that there exists a large percentage of equivalent method handles, and there is a potential to eliminate equivalent method handles for the JRuby interpreter. In contrast to the *SMR*, *IMHE* varies among different test cases (ranging from 5% to 23%), and the average *IMHE* is 13.46% (A method handle has 7.4 equivalent method handles on average). Since an equivalent set *equivSet* is only associated with an unique transformation name (method handles of one equivalent set have the same transformation name and type), it can be explained that method handle preferences for different cases are not the same. For example, the most frequently used transformation for *eval* case is *Insert1Handle*, while it is *BruteArgumentMoverHandle* for case *pi*.

Finding 4. *Similar to* ***Finding*** *1, short instance chains are likely to have more equivalent method handles, and the distribution of equivalent set's size is also uneven even when the chains in these sets have the same length.*

Figure 6 shows the *equivSet*'s size, versus the length of chains in the *equivSet* for six randomly selected CLBG test cases, where *equivSet* is used as equivalent method handle chain set. In these figures, the x axis is the length of the method handle chain, which is calculated as a *floor* function on x axis's value, and the y axis is the size of equivalent chain set (*equivSet*'s size), which is the number of method handle chains in the set. For example, the six bars in Figure 6a when x axis's value is from 4 to 5 represent six unique four-length equivalent chain sets. A short method handle instance chain would not be counted if it is a sub chain of other longer chains.

According to Figure 6, the short chains are likely to have more equivalent chains. As shown in these figures, bars on the left are crowded and tall while they become sparse on the right of the figures. Besides, the equivalent chain sets's size distribution is uneven even when the chains' lengths are fixed. Take length-three equivalent sets in the Figure 6d for example: three sets' sizes exceed 100 while the remaining are less than 45 (The majority of *equivSet*s' size is less than 25).

Furthermore, the distribution of length-two *equivSet*s for 41 benchmark test cases is shown in Figure 7. In the figure, the x axis is the index of the equivalent chain set, and y axis is corresponding equivalent set's size. According to the figure, the distribution is uneven, as 93% of set's size is less than 30. This result, as well as the result in Figure 6, implies that the optimization of the complementary 7% would be the most economical.

All of these conclusions are valuable for dynamically typed language implementers and JVM optimizations, especially if generalized to other dynamic languages. For the former, they can leverage these results by implementing efficient constructions of the method handle chain, while many kinds of method handle optimizations, e.g., method handle deduplication and method handle compilation, can be applied in the future. Another direct implication is that both can only concentrate on the optimization of these small numbers of instance chains to maximize the performance benefits while the cost is minimized.

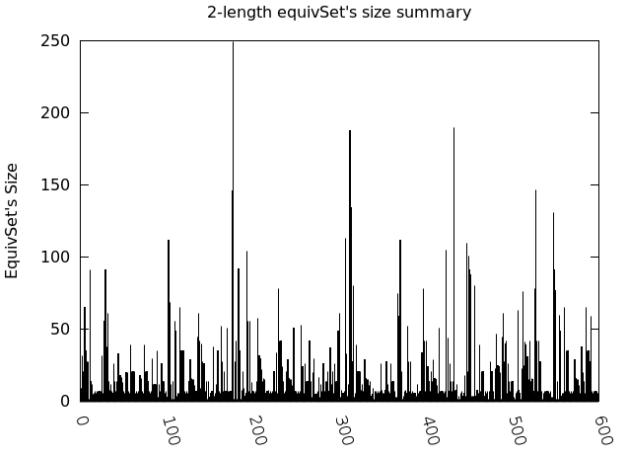


Figure 7: Global length-two EquivSet's Size Summary

6. RELATED WORK

6.1 Method Handle

Some industrial and academical work has been done to optimize dynamic invocation instruction and method handle efficiency. Rose [19] from Sun Microsystems (now Oracle) outlines all aspects of JSR 292 and presents a case study of how inline cache works with the *invokedynamic*, as well as other potential optimization directions. Based on Rose's work, Thalinger *et al.* [23] describe the instruction implementation tactics (and related optimizations) on the Hotspot JVM, which use internally-generated bytecodes as an intermediate language to freeze dynamic call sites and design an "adapter" calling sequence to match caller and callee of different method type signatures. Roussel *et al.* [20] describe their implementation of the JSR 292 in Dalvik, a virtual machine for Android OS. In their work, they explained detailed design idea of the instruction, as well as the method handle combination, on the Dalvik for resource constrained devices.

Some other work has also been done to illustrate the usage of *invokedynamic*. One of the most famous examples is JRuby, a Ruby language implementation on the JVM. Bodden [15] extends their existing Soot framework to support *invokedynamic* instruction for static analysis and transformation of Java programs. Similarly, Ponge *et al.* [18] introduce a dynamic programming language for the Java Virtual Machine (JVM) using the *invokedynamic* instruction and new APIs.

6.2 Program Behavior

Researchers have been interested in the characteristics of dynamically typed languages in the last ten years. In contrast to statically typed languages, their endeavor mainly focuses on either type behavior of dynamically typed languages or pattern distinction between dynamically typed languages and statically typed languages. For example, Li

et al. [17] examine four non-Java JVM languages, including Clojure and Scala, and use exploratory data analysis techniques to investigate differences in their dynamic behavior when they are compared to Java. Their results are based on a number of measurements, which are collected by executing the CLBG project and real-world applications on the JVM. Similarly, Sarimbekov *et al.* [21] apply their own toolchain [22] for workload characterization of dynamically typed languages on the JVM. However, both Li *et al.*'s and Sarimbekov *et al.*'s work is too general to be applied to the method handle patterns.

Little work has been done for research method handle patterns for *invokedynamic* instruction. So far as we know, our work is the first effort to mine patterns of MHG for dynamically typed languages on the JVM.

7. DISCUSSION AND EXTENSIONS

Our current method handle data mining is a mixture of online and offline analysis, which aims to reduce performance degradation caused by the mining process. The work can be extended in at least two areas.

Online method handle data mining.

As the benchmark (Computer Language Benchmark Game) is written intentionally to collect performance measurements, it is different from real applications. Because we have already reached the conclusion that there is much room for method handle optimization, our next step is to apply these data mining methods to a real application at program runtime. Meanwhile, the cost to do this task should be carefully checked.

JVM optimization.

In collaboration with the IBM J9 development team, we will implement some optimizations to take advantage of existing data mining results. Two of our next plans are online method handle elimination, which aims to remove equivalent method handles during program runtime, and dynamic code (bytecode or native code) generation, which compiles the hot transformation chains into a single method (or method handle) at runtime to remove the amount of late binding.

8. CONCLUSION

This paper introduces an MHG mining methodology for dynamically typed language implementation on the JVM. The resultant Method Handle Mining System (MHMS) is capable of collecting and mining useful method handle patterns, transformation patterns and instance patterns. Both patterns have the potential to be utilized to improve the overall performance of dynamically typed languages on the JVM. Based on the proposed mining algorithms of such patterns, we further implemented the MHMS. Our experimental results with the JRuby CLBG benchmark shows that a) the frequency of different transformation patterns varies significantly, b) a large proportion of method handle chains, 28.8%, are equivalent, and the most of these duplicates can be eliminated to reduce consumed memory, and c) distribution of equivalent length-two chains are not even, and only 7% of these chain sets have more than 30 equivalent chains. These conclusions imply that it is possible to optimize a small fraction of these chains to maximize performance benefits.

Despite dependence on the IBM J9 implementation and its library tools, e.g., DDR and DTFJ, MHMS can be extended to different JVM platforms, e.g., HotSpot, by replacing the APIs that read the core files. Also, the idea of data mining method handle patterns is not only applicable for JRuby, but also for other dynamic JVM language implementations, e.g., Javascript and Groovy.

9. ACKNOWLEDGMENTS

This work is supported by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program. Furthermore, we would also like to thank the New Brunswick Innovation Fund for contributing to this project. Finally, we would like to thank the Centre for Advanced Studies - Atlantic for access to the resources for conducting our research.

10. REFERENCES

[1] Charles Nutter. A First Taste of InvokeDynamic. http://blog.headius.com/2008/09/first-taste-of-invokedynamic.html.

[2] Charles Nutter. Invokedynamic in 45 minutes. http://www.jfokus.se/jfokus13/preso/jf13_InvokeDynamic.pdf.

[3] Computer Programming Benchmark Game. http://benchmarksgame.alioth.debian.org/.

[4] IBM Diagnostic Tools for Java. http://www.ibm.com/developerworks/java/jdk/tools/dtfj.html.

[5] J9 VM. https://www-01.ibm.com/support/knowledgecenter/SSYKE2_7.0.0/com.ibm.java.win.70.doc/user/java_jvm.html.

[6] JSR 292: Supporting Dynamically Typed Languages on the $Java^{TM}$ Platform. https://jcp.org/en/jsr/detail?id=292.

[7] JVM instruction set HotSpot. http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html.

[8] JVM instruction set wiki. http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings.

[9] Method Handle-An IBM implementation. http://wiki.jvmlangsummit.com/images/a/ad/J9_MethodHandle_Impl.pdf.

[10] MethodHandle Implemention Tips and Tricks. http://wiki.jvmlangsummit.com/images/6/6b/2011_Heidinga.pdf.

[11] MethodHandles-Java Platform SE7. http://docs.oracle.com/javase/7/docs/api/java/lang/invoke/MethodHandles.html#dropArguments\%28java.lang.invoke.MethodHandle,\%20int,\%20java.lang.Class...\%29.

[12] New JDK 7 Feature: Support for Dynamically Typed Languages in the Java Virtual Machine. http://www.oracle.com/technetwork/articles/java/dyntypelang-142348.html.

[13] The DOT Language in Graphviz. http://www.graphviz.org/content/dot-language.

[14] Trail: The Reflection API. http://docs.oracle.com/javase/tutorial/reflect/.

[15] E. Bodden. Invokedynamic support in soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program*

Analysis, SOAP '12, pages 51–55, New York, NY, USA, 2012. ACM.

[16] J. Gosling, B. Joy, G. L. Steele, Jr., G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.

[17] W. H. Li, D. R. White, and J. Singer. Jvm-hosted languages: They talk the talk, but do they walk the walk? In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, pages 101–112, New York, NY, USA, 2013. ACM.

[18] J. Ponge, F. Le Mouël, and N. Stouls. Golo, a dynamic, light and efficient language for post-invokedynamic jvm. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, pages 153–158, New York, NY, USA, 2013. ACM.

[19] J. R. Rose. Bytecodes meet combinators: Invokedynamic on the jvm. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, VMIL '09, pages 2:1–2:11, New York, NY, USA, 2009. ACM.

[20] G. Roussel, R. Forax, and J. Pilliet. Android 292: Implementing invokedynamic in android. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '14, pages 76:76–76:86, New York, NY, USA, 2014. ACM.

[21] A. Sarimbekov, A. Podzimek, L. Bulej, Y. Zheng, N. Ricci, and W. Binder. Characteristics of dynamic jvm languages. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, pages 11–20, New York, NY, USA, 2013. ACM.

[22] A. Sarimbekov, A. Sewe, S. Kell, Y. Zheng, W. Binder, L. Bulej, and D. Ansaloni. A comprehensive toolchain for workload characterization across jvm languages. In *Proceedings of the 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '13, pages 9–16, New York, NY, USA, 2013. ACM.

[23] C. Thalinger and J. Rose. Optimizing invokedynamic. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 1–9, New York, NY, USA, 2010. ACM.