Fuse Method Handles for Dynamic JVM Languages

Shijie Xu
IBM Centre of Advanced Studies
University of New Brunswick
Fredericton, New Brunswick, Canada
shijie.xu@unb.ca

David Bremner
IBM Centre of Advanced Studies
University of New Brunswick
Fredericton, New Brunswick, Canada
bremner@unb.ca

Daniel Heidinga
IBM Ottawa Lab
Ottawa, ON, Canada
Daniel_Heidinga@ca.ibm.com

Abstract

A Method Handle (MH) in JSR 292 (Supporting Dynamically Typed Languages on the JVM) is a typed, directly executable reference to an underlying method, constructor, or field, with optional method type transformations. Multiple connected MHs make up a Method Handle Graph (MHG), which transfers an invocation at a dynamic call site to real method implementations at runtime. Despite benefits that MHGs have for dynamic JVM language implementations, MHGs challenge existing JVM optimization because a) larger MHGs at call sites incur higher graph traversal costs at runtime; and b) JIT expenses, including profiling and compilation of individual MHs, increase along with the number of MHs.

This paper proposes dynamic *graph fusion* to compile an MHG into another equivalent but simpler MHG (e.g., fewer MHs and edges), as well as related optimization opportunities (e.g., selection policy and inline caching). Graph fusion dynamically fuses bytecodes of internal MHs on hot paths, and then substitutes these internal MHs with the instance of the newly generated bytecodes at program runtime. The implementation consists of a template system and *GraphJIT*. The former emits source bytecodes for individual MHs, while the latter is a JIT compiler that fuses source bytecodes from templates on the bytecode level (i.e., both source code and target code are bytecodes). With the JRuby Micro-Indy benchmark from Computer Language Benchmark Game and JavaScript Octane benchmark on Nashorn, our results show that (a) the technique can reduce execution time of Micro-Indy and Octane benchmarks by 6.28% and 7.73% on average; b) it can speed up a typical MHG's execution by 31.53% using Ahead-Of-Time (AOT) compilation; and (c) the technique reduces the number of MH JIT compilations by 52.1%.

VMIL'17, October 24, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5519-3/17/10…$15.00
https://doi.org/10.1145/3141871.3141874

CCS Concepts • **Software and its engineering** → *Interpreters*; *Interpreters*; *Just-in-time compilers*;

Keywords invokedynamic, method handle, bytecode generation, just-in-time, object fusion

ACM Reference Format:
Shijie Xu, David Bremner, and Daniel Heidinga. 2017. Fuse Method Handles for Dynamic JVM Languages. In *Proceedings of ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3141871.3141874

1 Introduction

A method handle (MH) is a typed, directly executable reference to an underlying method, constructor, field, or similar low-level operation, with optional transformations of arguments or return values [12]. These transformations include such patterns as method argument insertion, removal, and substitution. Method handles, together with the Java Virtual Machine (JVM) instruction, *invokedynamic*, were first proposed in the Java Specification Request (JSR) 292 to resolve "pain points" when implementing dynamic JVM languages (e.g., JavaScript, Ruby and Python) [3, 14, 18].

As a method referred by an MH can have references to other MHs, multiple connected MHs make up a Method Handle Graph (MHG) [24]. An MHG has only one root, and it can be executed by invoking the root's *invokeExact* method, which in turn triggers the execution traversal of the whole or partial graph. With the *invokedynamic* instruction, a dynamic method call can be resolved to real method implementations by executing the linked MHG at the call site. In Figure 1, the root holds two child MH 2 and MH 3, and its execution can resolve a dynamic invocation to real method implementations that are referenced by leaf MHs.

1.1 Motivation

The disadvantage of an MHG is its complex structure for traversal, especially when the graph (or equivalent graphs) is large and repeatedly seen at multiple call sites. To resolve a dynamic method, a number of indirections from the root have to be done. In Figure 1, the JVM has to execute at least four MHs (i.e., 1, 3, 4, and 6) from the MH 1 to the leaf MH 7. Considering the prevalence of dynamic method invocations, these four indirections from root to leaves would be repeated many times at runtime, and it is worthwhile to seek a more

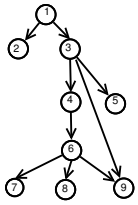

Figure 1. Method Handle Sample

efficient method to reach leaf MHs, instead of traversing on the original path strictly.

Large MHGs are also inefficient for Just-In-Time (JIT) compilations on the JVM. JIT compilation is a procedure that translates a frequently executed bytecode method (or trace) into native machine code. In the J9 JVM (i.e., IBM's JVM implementation), the MH JIT compilation, conducted by Testarossa JIT (TRJIT) compiler, uses multiple phases, two of which are a translation of individual MHs with little optimization (i.e., **warm JIT**) and a translation of an MHG with aggressive optimization (i.e., **hot JIT**), when these MHs are hot enough [5, 11]. Thus, both profiling tasks and warm JITs are conducted on individual MHs, and the cost is approximately proportional to the number of MHs in a graph. For example, there are 9 profiling targets and 9 warm JIT compilation startups for the MHG in Figure 1, if they are all hot enough. Consequently, the accumulated expenses of profiling and compilation startups on individual MHs are not trivial at program runtime, when MHGs become large. Further more, the large number of MH compilation tasks that are submitted to JIT system would increase competition among compilation tasks (e.g., MH tasks and other non-MH tasks) for JIT threads.

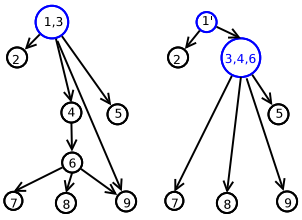

Figure 2. a) Merge 1 and 3; b) Merge 3, 4, and 6

Addressing MHG traversal and its JIT compilation overhead, this paper proposes a dynamic *graph fusion* to compile an MHG into another equivalent but simpler MHG, and optimization opportunities (e.g., graph node selection policy and inline caching). The graph fusion dynamically generates new MH transformations by fusing hot internal MHG nodes' bytecodes; it creates MH instances of the new transformations; it sets up a new MHG with MHs from the source MHG and these newly created MHs; and it substitutes the new MHG for the source MHG at the dynamic call site for execution. For example, two MHGs in Figure 2 are the compilation output for the MHG in Figure 1. In both MHGs, MHs, i.e., MH(1,3), MH(3,4,6) and MH(1'), are newly generated, and the distances from root to MH 7 are reduced from 4 to 3, and from 4 to 2, respectively.

The provided graph fusion solution works on the bytecode level, and is implemented by two components: an MH template system and GraphJIT. In the template system, templates of individual MH transformations produce source bytecodes. GraphJIT is a JIT compiler for a *Frequently Traversed* but *Stable Directed Acyclic* (FTSDA) graph simplification. Within J9 JVM, GraphJIT fuses source bytecodes of MHs on hot paths, and moves leaf MHs close to the graph root by dynamically generating bytecodes. By running JRuby Micro-Indy benchmark from Computer Language Benchmark Game [2] and JavaScript Octane [16] benchmark on Nashorn [15], our evaluation shows that the technique a) can reduce execution time of Micro-Indy and Octane benchmark by 6.28% and 7.73% on average, respectively; b) can speed up a typical MHG's traversal from Micro-Indy benchmark by 31.53% using AOT compilation; c) as much as 53.84% MHs have been fused by GraphJIT; d) the technique reduces the number of MH JIT (warm JIT) compilations by 52.1%.

1.2 Contributions

Our contributions are

- bytecode level graph fusion for reduction of MHs and potential JIT compilation efforts.
- the integration of GraphJIT with J9 JVM for the method handle graph fusion, and the identification of multiple optimization opportunities (e.g., sharing of the generated bytecodes, and graph node selection policies) to balance the compilation cost and potential benefits.
- an extensive evaluation with a quantitative analysis of how graph fusion impacts performance and MH JIT compilation. We show that graph fusion can speed up method handle graph execution, and reduces the number of MH warm JIT compilations in J9 JVM.

The rest of this paper is organized as follows. Section 2 provides general background for the *invokedynamic* instruction and method handles. Section 3 and Section 4 overview the system structure, and describe the template system. Section 5 and Section 6 provide the GraphJIT integration with method handles in J9 JVM and runtime optimizations (e.g., hot MH selection and inline caching for mutable MHs) during compilation. Section 7 shows our evaluation results. Finally the related work, future work, and conclusions are given in Section 8, Section 9 and Section 10, respectively.

2 Background

2.1 *Invokedynamic* Instruction

Many "pain points" (e.g., failed inlining and polluted profiles) have been reported when implementing dynamic languages on the JVM [18]. Different from statically typed languages,

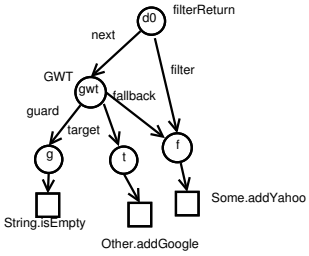

Figure 3. Sample MHG

variables in dynamically typed languages do not reveal much type information at compilation time. As a result, for dynamic method invocations, the JVM has to rely on many indirect ways to calculate types at runtime, which increase runtime overhead.

Therefore, JSR292 introduces a bytecode instruction *invokedynamic* for dynamic method invocations when implementing dynamic JVM languages. This instruction allows language implementers to define the way (i.e., MHG) to resolve a dynamic method invocation. For a dynamic method invocation, dynamic JVM language interpreters generate an *invokedynamic* instruction, and associate it with a bootstrap method. When the JVM first sees an *invokedynamic* instruction, it calls the bootstrap method, which creates and links a method handle (or MHG) to the call site. Later, the dynamic invocation will be resolved to real method implementations via MHG execution traversal.

2.2 *Invokedynamic* and Method Handle Graphs

An MHG consists of MHs, and each MH represents a method type transformation. JSR 292 provides predefined transformations, such as *dropArguments*, *filterReturn*, *guardWithTest* (GWT), and *insertArguments* [18]. Each transformation is represented by a Java class in the J9 JVM. Via MH combination, complex MHGs can be built to resolve a dynamic method invocation. For example, the bootstrap method in Listing 1 creates a call site and connects it with the MHG in Figure 3. The newly created MHG, starting at *d0*, is capable of resolving the invocation at the call site to the real method implementations (i.e., *String.isEmpty()*, *Some.addYahoo()* and *Other.addGoogle()*) via the *GuardWithTest* transformation and *filterReturn* transformation.

An MH can be executed by invoking its *invokeExact* method; it has an attribute: method type, indicating method arguments and return type that the MH accepts; and it is associated with a method type transformation.

3 System Structure

Within the J9 JVM, MH graph fusion consists of a template system and GraphJIT, as shown in Figure 4. The template system emits source bytecodes (i.e., bytecode instructions of a class) for selected MHs in the MHG produced by dynamic

```
1  CallSite bootstrapMethod(...){
2    CallSite cs = new MutableCallSite();
3    MethodHandle g = lookup().findVirtual(String
         .class, "isEmpty", methodType(boolean.
         class));
4    MethodHandle t = lookup().findStatic(Other,
         "addGoogle", methodType(String.class,
         String.class));    //Append google.
5    MethodHandle f = ... //f refers to Some.
         addYahoo
6    MethodHandle gwt = guardWithTest(g, t, f);
7    MethodHandle d0 = filterReturn(gwt, f);
8    //assertEqual((string)d0.invokeExact(args),
         (string)f.invokeExact((String)gwt.
         invokeExact()))
9    cs.setTarget(d0);
10   return cs; }
```

Listing 1. Sample Bootstrap Method for MHG creation

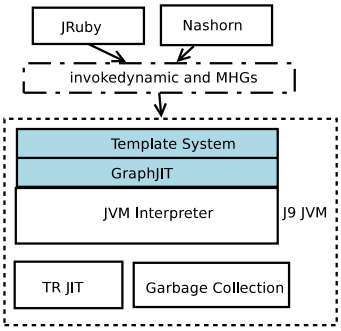

Figure 4. High Level MHG Fusion: Template System, GraphJIT

JVM language interpreters (e.g., JRuby, Nashorn); GraphJIT processes these source bytecodes and fuses them together as new classes; a new MHG is built from these new classes, and replaces the source MHG for execution.

The procedure is shown in Figure 5. First, each internal MH in the MHG on the left is translated into source bytecodes via the template system. Then these bytecodes (i.e., *C_root*, *C_4* and *C_5*) are fused together as a transformation method for a newly generated MH class (i.e., *DYNGuardWithTestHandle*) by GraphJIT. An instance of *DYNGuardWithTestHandle* replaces the origin *G_root* at dynamic call sites after it is set up with leaves in the source graph. The method in the new class is JITted into native code by TR JIT compiler in J9 JVM, if this instance is frequently interpreted.

4 Method Handle Template System

An MH's source bytecodes represent the transformation that a single MH does, and these bytecodes form a class node, which is made up of a class declaration instruction, field instructions, and method instructions. A class node can be

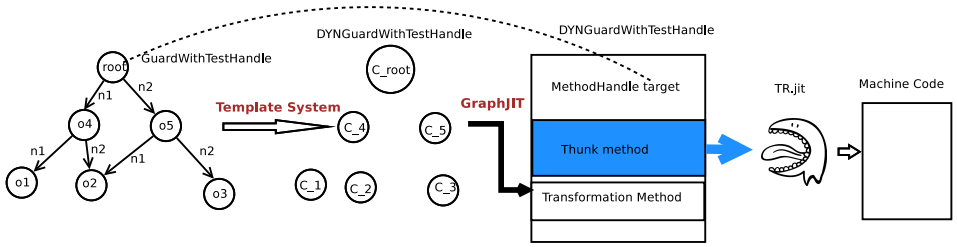

Figure 5. Workflow for Template System, GraphJIT and JIT

uniquely mapped to a Java class after its bytecodes are loaded by a class loader.

An MH's template is a set of Java methods that emit source bytecodes. For an MH, the execution of the corresponding template will produce source bytecodes, the execution of which is equivalent to that of the original MH (i.e., calling that MH's *invokeExact* method). The template system hides the different JVM's implementations for MHs.

4.1 Template Implementation

As different MH transformations vary, a template is only associated with a single method handle transformation (i.e., an MH class). For example, an MH of *guardWithTest* transformation transfers the invocation to its child *trueTarget* by a testing child *guard*. Thus, its template's execution will generate source bytecodes, the decompilation of which is shown in Listing 2.

```
1 T  invokeExact(args){
2   if(guard.invokeExact(args)){
3     return (T)trueTarget.invokeExact(args);
4   }else{
5     return (T)fallback.invokeExact(args);
6   }
7 }
```

Listing 2. Decompiled result for *guardWithTest* template

A template's execution requires a method handle's method type. It accepts a method type and other optional data that the MH has as parameters. The method type determines the number of variables should be loaded to the operand stack, and which instructions should be used for individual variables in the source bytecodes. For a *guardWithTest* MH with a method type *(Object, int)Object*, two instructions: *ALOAD* and *ILOAD*, are emitted to load first two arguments into operand stack.

4.2 Source Bytecodes Sharing

To avoid unncessary template execution, source bytecodes generated by a template can be shared, if corresponding MHs have the same method type. As shown in Figure 6, both MH 1 and MH 2 have the same transformation and method type. Therefore, each MH transformation maintains an internal cache, which remembers the source bytecodes by

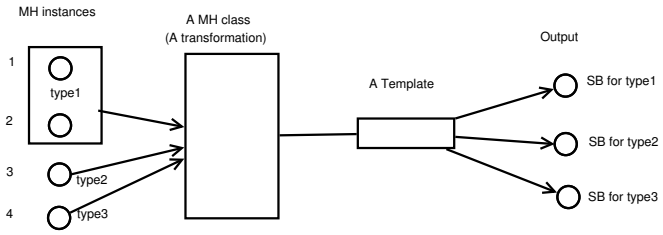

Figure 6. Source Bytecodes (SB) and Method Types

the corresponding method type. The source bytecodes, once they are generated, are inserted into the cache. Later, for an MH transformation, the cached version of source bytecodes is directly used, instead of regenerating everything from scratch, if the method type is hit in the internal cache.

5 GraphJIT Compilation

GraphJIT is a JIT compiler that performs a graph fusion by compiling source bytecodes of internal graph nodes on hot paths. The compilation is on the bytecode level (i.e., both source and target are bytecodes). During compilation, the source bytecodes from MHs on hot paths are selected and fused together as new bytecodes (i.e., one or more Java classes); a new MHG from these new bytecodes is created and set up with leaves or infusible nodes from the source graph; and finally the new graph replaces the source graph for execution.

Figure 7 illustrates how GraphJIT works for the graph in Figure 1. First, source bytecodes of MH 4 and MH 6 are fused for a single MH (4,6); then, source bytecodes of MH 3 and MH (4,6) are fused as the MH (3,4,6); next, the bytecodes of the root MH 1 are also fused with the MH (3,4,6), and a new MH (1, 3, 4, 6) is created. Compared to the source graph *G0* , the new graph has 4 less MHs, and the distance from root MH to the leaves 7 and 8 is shortened by 3.

The main three steps to complete MHG fusion are field context construction, dynamic bytecode generation, and new graph activation. To connect these three steps, a queue is built for each non-leaf fusible MH in the graph, and will be filled up with bytecodes during bytecode generation step. Different from the source bytecodes, which only represent the transformation that a single MH has, bytecodes in an

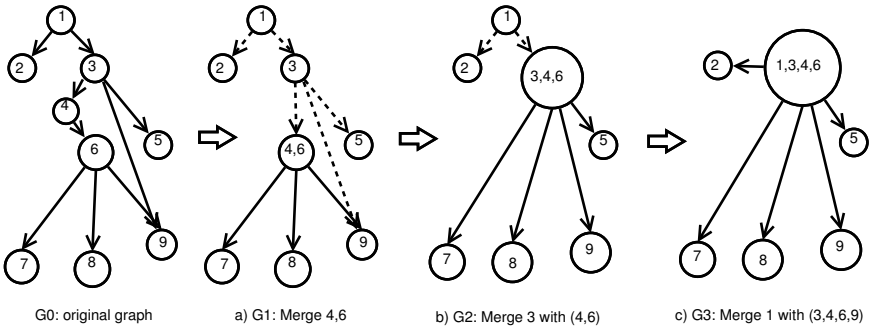

Figure 7. Graph Compilation Illustration for Figure 1 (The merging of two MHs here is equivalent to fusion of their bytecodes, instance creation of the fused bytecodes, and replacement of both MHs by the new instance)

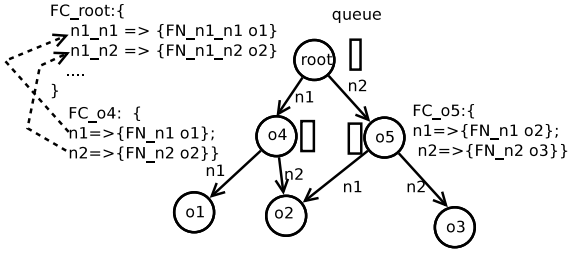

Figure 8. Graph and FieldContexts (FCs)

MH's queue represent transformations of the whole subgraph starting at that MH.

5.1 Field Context Construction

A per-MH *FieldContext* tracks both leaf and infusible internal MHs that the MH can reach, as well as paths to these MHs. During bytecode generation step, a *FieldContext* is mapped to bytecode instructions that define a new MH class. These newly generated instructions are pushed to the MH's queue.

The main component in a *FieldContext* is a map that consists of entries

$$fieldName => \{fieldNode, child\}$$

where the *fieldName* is a string field name, and the *fieldNode* (i.e., a combination of field modifier, field type, field name in a class) is an instruction that creates a field member for the corresponding *child*. Here *child* is an infusible graph node (e.g., a leaf or an infusible node), and is reachable from the MH. The *fieldName* reflects the path from the MH to the corresponding *child*, and the field name component in the *fieldNode* will be consistent with it. For example, the root's *FieldContext*, *FC_root* in Figure 8, has an entry $n1\_n1 => \{FN\_n1\_n1, o1\}$, indicating that the path from the root to *o1* is $n1 \rightarrow n1$ in the source graph.

The order to construct a *FieldContext* is from bottom to top. To build an MH's field context, GraphJIT checks to see whether all its child *FieldContext*s have been set up. Otherwise, it goes to create and set up the MH's child *FieldContext*s.

Field contexts will be fused, if corresponding MHs (e.g., *root* and *o4*) are fusible. To set up an MH's *FieldContext*, GraphJIT copies entries from its child *FieldContext*s to its map, with field name transformations. For an entry in a child *FieldContext*, GraphJIT prefixes the field name component with a path from the current MH to that child. Take the *root* and *o4* in Figure 8 for an example; the path between them is *n1*. Thus, two entries in *FC_o4* are prefixed with the path *n1_*, before both are added to the *FC_root*'s map. The copied entry *n1_n1* for the MH *o1* in *FC_root* indicates that JVM can reach *o1* from *root* via the path *n1_n1*. Meanwhile, the corresponding field name component in the *FN_n1_n1* is also updated, so that it would be consistent with the new key name *fieldName* in the map.

In case a child is not fusible (i.e., a leaf or an infusible internal node), GraphJIT creates a new entry, in which the field name is the same as the edge name, and *fieldNode* refers to that child directly. For example, a new entry

$$n2 => \{FieldNode\_n2, o5\}$$

is created and added as one entry for the *FC_root*, if *o5* is not fusible.

5.2 Dynamic Bytecode Generation

For an MH in the graph, GraphJIT converts its *FieldContext* into bytecodes, and adds these bytecodes into the queue. These new bytecodes consist of instructions to create a class declaration, fields, and methods. Once completed, these bytecodes form a new MH class, representing the whole subgraph's transformation.

For the class declaration, the generated class is a subclass of the *MethodHandle* class in J9, so that it can reuse the existing method handle implementations in J9. For fields, GraphJIT creates a field for each entry in the *FieldContext* using *fieldNode* information. Take the entry $n1\_n1 \rightarrow$ *{FN_n1_n1, o1}* in *root*'s field context for example; GraphJIT creates a *MethodHandle* field, named *n1_n1*, with modifier information in the *FN_n1_n1*.

GraphJIT generates two methods, a thunk method and a transformation method, for the new class. The thunk method is a J9 specific method, which glues the TR JIT compiler to the transformation method. Owing to the class layout changes, GraphJIT generates the transformation method by fusing the current MH's source bytecodes and bytecodes from current MH's child queues. GraphJIT copies current MH's source instructions, after possibly performing one of following operations:

method inlining for an invocation, if the invocation receiver is a fused child, the queue of which has been filled up with bytecodes from its field context.

field operation instruction (e.g., getField) elimination, if the field name to the target is not in the entries of current MH's field context. A field operation instruction loads an object's field into the operand stack. For example, the source bytecodes for

$$this.n1.invokeExact()$$

in *root* is shown in listing 3. GraphJIT will skip *GETFIELD* instruction, because *n1*, which references child *o4*, in this instruction fails to hit entries in *root*'s field context.

```
1 ALOAD 0
2 GETFIELD MethodHandle.n1:LMethodHandle
3 INVOKEVIRTUAL MethodHandle.invokeExact()LObject
```

Listing 3. Field Operation Elimination

5.3 Activation of New Graphs

After dynamic bytecode generation, bytecodes in an MH's queue are class loaded to create a new Java class. The activation is only for MHs that are either the root or MH's infusible children.

For the generated Java class, GraphJIT creates an instance *O*, and sets its fields based on the corresponding *FieldContext*. For the entry $n1\_n1 \rightarrow \{FN\_n1\_n1, o1\}$ in *root*'s *FieldContext*, GraphJIT sets *O*'s *n1_n1* field to be *o1*. The new graph starting *O* is activated, once it replaces the source MHG.

6 Runtime Optimization

6.1 MH Node Selection Policies

MH Entry Counter An MH has an Entry Counter (EC), whose value increases by one, each time the MH is executed. An MH will be classified as hot, if its EC is greater than *Threshold*, as discussed below. Leaf MHs does not have an EC, as they are infusible and represent empty transformation.

The default selection policy is based on an MH's EC. GraphJIT only selects MHs on hot paths, to avoid compilation cost on cold MHs. When the JVM executes an MH, it checks the MH's EC, and fuses the MH with its child MHs, if all ECs exceed *Threshold*. This policy is based on the assumption that an MH path will continue to be hot, if it has been hot for a long time.

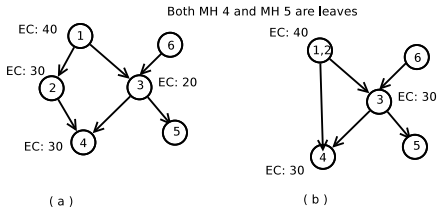

Figure 9. Incremental Generation (Threshold: 30)

The *Threshold* is dynamically determined. GraphJIT requires two parameters *MAX_Threshold* and *t*, which is in a range of [0, 100]. The *MAX_Threshold* is often configured to be the threshold that can trigger the first MH JIT compilation. *Threshold* is calculated as the value, so that t% of existing counters' values are greater than it, when the first MH JIT happens.

Maximal Bytecode Queue Size and Re-Generation The generation is incremental, once *Threshold* is calculated. For example, both MH 1 and MH 2 are first fused in Figure 9.a. Later, the MH 3 might also be fused into the MH (1,2), when JVM executes the MH (1,2), because the EC of MH 3 reaches *Threshold* in Figure 9.b.

To determine whether to fuse MH 3 or not, GraphJIT uses a parameter, named maximal bytecode queue size, to control the size of queue bytecode. For MH (1,2), GraphJIT calculates the number of its source bytecode instructions, which is roughly the sum of queue bytecodes of MH 1, plus the number of the source bytecodes instructions of MH 3. If the sum is still less than the maximal bytecode queue size, MH 3 will be fused into MH(1,2). Otherwise, the fusion of MH 3 has to be delayed until one of its other parents (e.g., MH 6) is executed and that parent's EC also reaches *Threshold*. The TR JIT compiler is sensitive to the complexity (e.g., the combination of loops and if-conditions) of bytecode blocks. By maximal bytecode queue size, GraphJIT controls the complexity of bytecodes, as a larger bytecode block in the queue is likely to indicate a complex graph (e.g., a subgraph's complexity).

6.2 Mutable Nodes and Inline Caching

GraphJIT detects and handles mutable MHG nodes, the children of which are mutable. An MH is classified as potentially mutable, if its source bytecodes have a *PUTFIELD* instruction that resets its child. In other words, a mutable MH's child can be dynamically reset at runtime, and this results unstable graph structure. For example, the *PUTFIELD* in line 4 of Listing 4 sets field *this.n1* to the variable 1. To handle a mutable MH, Inline Caching (IC) is used to remember the receiver during the fusion. GraphJIT creates a cached field member that statically references the current receiver when fusion is ongoing, and generates the byetcodes that check whether the real receiver is equivalent to the cached one at runtime.

```
1 root's source bytecode:{
2 ALOAD 0
3 ALOAD 1
4 PUTFIELD MethodHandle.n1:LMethodHandle  // this.
      n1 = variable 1
5 ....
6 ALOAD 0
7 GETFIELD MethodHandle.n1:LMethodHandle
8 INVOKEVIRTUAL MethodHandle.invokeExact()LObject
9 }
10 Decompiled code:
11 {
12   final static MethodHandle cached=Map.get("key
      ");
13   public void invokeExact(){
14     ...
15     if(n1==cached){
16       // inline cached.invokeExact() in this
          branch
17     }else{
18       n1.invokeExact()
19     }}
```

Listing 4. Mutable MH cases

For example, the decompiled bytecodes for the invocation instruction at line 8 of Listing 4 are shown from line 15 to 19. The variable *cached* at line 13 remembers the receiver by a constant key, and it is initialized during class loading. The remembered receiver normally has been intensively optimized before these bytecodes are generated. At runtime, *cached* is compared with the real invocation receiver, and will be used, iff the comparison succeeds.

7 Evaluation

7.1 Benchmarks

JRuby Micro-Indy Benchmark from CLBG JRuby is a Java implementation of the Ruby programming language, and JRuby is the first language that adopted *invokedynamic* on Java 7. At runtime, the JRuby interpreter emits bytecodes from Ruby source, and then executes these bytecodes on the JVM. Both method handles and *invokedynamic* are only related to dynamic method invocations.

The Micro-Indy benchmark has 32 Ruby tests, and each test has 5 to 150 lines. Most tests have one or two intensive dynamic method invocations via for-loops or recursive calls, and the maximal MHG size is less than 200. During runtime, both verbose GC and JIT logs are collected for performance analysis.

JavaScript Octane Benchmark for Nashorn Nashorn is a JavaScript engine for the JVM. The Octane benchmark measures a JavaScript engine's performance by running a suite of tests, which represent certain use cases in JavaScript applications. In the benchmark, there are 16 JavaScript tests.

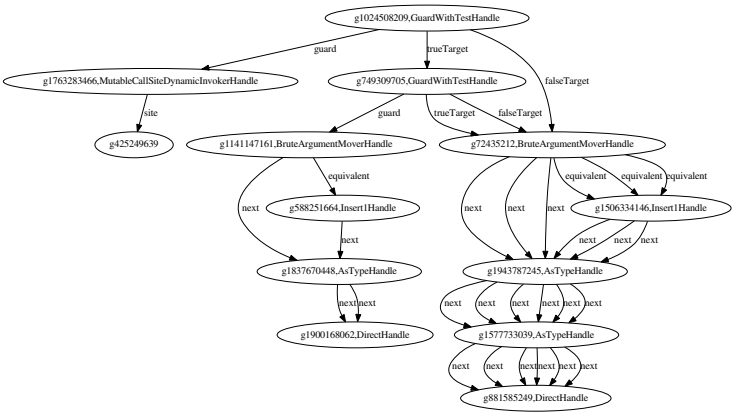

Figure 10. Typical MHG *a* for AOT

As all these tests are evaluated in a single JavaScript application in Octane, we only collect execution time measurement for each test. Similar to the JRuby Micro-Indy benchmark, method handle graphs are only created and traversed for the dynamic method invocations.

Typical Method Handle Graph Three MHGs: the typical MHG *a*, shown in Figure 10, the MHG compiled from *a* by GraphJIT with inline caching optimization *with_IC*, and the MHG compiled from *a* by GraphJIT without IC optimization (*NO_IC*), are executed 20*250 times. The MHG *a* is a typical MHG structure, when executing the Micro-Indy benchmark test *cal*, which has worst performance with GraphJIT.

7.2 Environment Setup

Our experiment is done on a Intel Xeon machine that has 4 × Intel Xeon E7520 1.8 GHz processors. The machine has 16 cores and 64GB memory. Two dynamic JVM language interpreters, i.e., JRuby [13] (version 9.0.4.0) and Nashorn [15], are used on the IBM J9 JVM. For the evaluation result, GraphJIT is disabled for data with "Orig" (or "Original"), while it is enabled for the measured data with "GraphJIT". For GraphJIT, the *MAX_Threshold* is configured to be the JIT's threshold, while *t* is set to be 90.

7.3 Execution Time and Execution Score

For the Micro-Indy benchmark, *ExecTime* is measured as the elapsed time to complete a test. The comparison measure is

$$speedup = \frac{ExecTime_{orig} - ExecTime_{GraphJIT}}{ExecTime_{orig}} * 100\% \quad (1)$$

AOT of Typical MHG The purpose of building two MHGs is to verify that the fused MHG outperforms the original MHG on the traversal. According to Figure 11, MHG fusion solution speeds MHG traversals. The traversal of the MHG *no_IC* has best performance, and the mean speedup is 31.53%. The performances between *No_IC* and *Orig* are close to each other after 14*250 executions, because native code generated

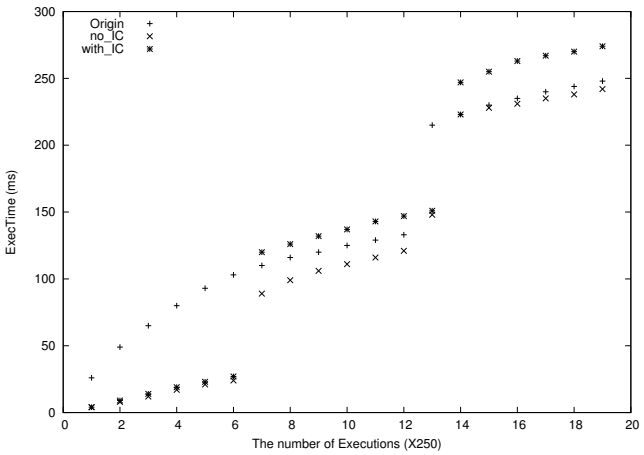

Figure 11. AOT ExecTime (Mean $Speedup_{no\_ic}$: 31.53%)

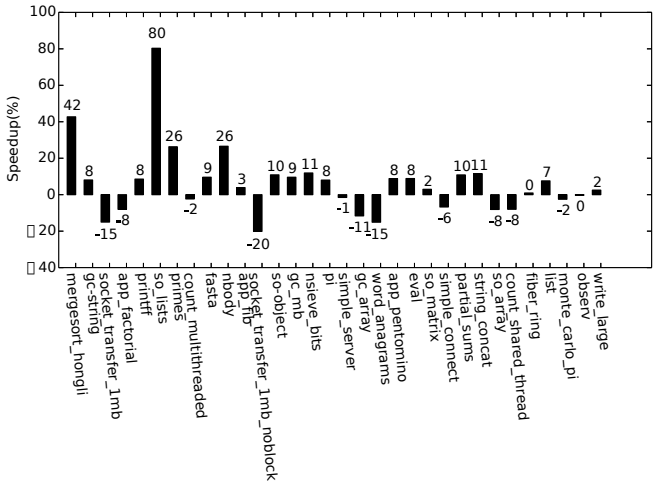

Figure 12. Speedup for Micro-Indy Tests (Mean: 6.28%)

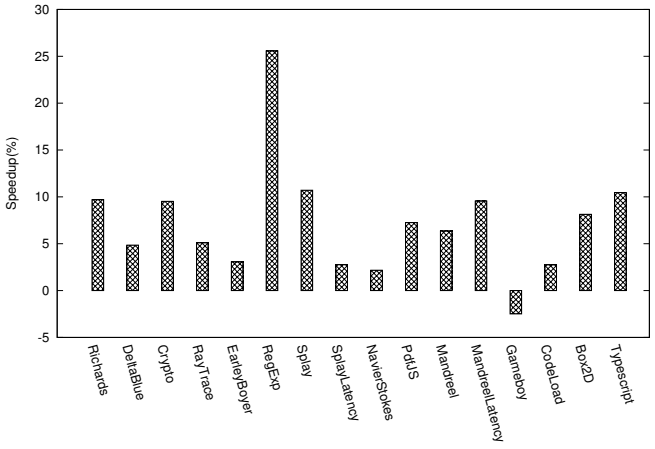

Figure 13. Speedup for Octane Tests (Mean: 7.73%)

from both MHGs are similar to each other. The performance of *with_IC* is worst after 7*250 executions, because inlining cache is not effective for the completely static MHG *a* here. For a static small graph, inline caching adds an extra overhead, and may prevent TR JIT from doing other optimizations, due to the number of checking and comparison operation at runtime.

Runtime GraphJIT Evaluation In this evaluation, both the JRuby Micro-Indy and the JavaScript Octane benchmark are executed on J9 JVM with GraphJIT. Besides MHG and *invokedynamic*, the generated bytecodes from scripts also include other computing tasks. Therefore, the execution time for a test is mixture of costs for MHG traversal, GC, GraphJIT's workload, and other computing tasks.

As shown in Figure 12 and Figure 13, both the JRuby Micro-Indy and the JavaScript Octane benchmark see performance speedup with MHG fusion solution. On the average, the speedups for Micro-Indy and Octane are 6.28% and 7.73%, respectively. However, the speedup for individual tests varies significantly, especially for Micro-Indy benchmark tests. For JRuby benchmark, some tests have a clear performance improvement, but some others get worse. For Octane benchmark, the result is more consistent among all tests. The explanation is that some transformation patterns (e.g., graph structures) are not suitable for fusion, while node selection policy in GraphJIT only considers a node's hotness and corresponding method size. The selection of these sub-MHGs might make the queue bytecodes too complex for the JIT compiler.

7.4 Effectiveness for Micro-Indy

Percentage of Reduced MHs by GraphJIT For the Micro-Indy benchmark, GraphJIT can effectively reduce the number of graph internal MHs. GraphJIT logs MHs that it has seen and processed. As shown in Figure 14, the number of internal MHs is reduced by 53.84% on the average. This measurement depends on the way that an MHG is built and how frequently an MHG is visited. For GraphJIT, MHGs that have more internal nodes and frequent paths are more likely to be chosen by GraphJIT for compilation.

TR JIT Compilation Impact MHGs compiled from GraphJIT serve as input for TR in J9, and these simplified MHGs will only be JITted if their entry counters exceed a predefined threshold in J9.

Figure 15 shows percentages of reduced MH warm JIT compilations and increased MH warm JIT compilation times. In the figure, the number of warm JIT compilations decreases by approximately 52.1%, which is close to the reduction of MHs, indicating little overhead on the MH JIT profiling and compilation tasks.

However, there is not much improvement on the MH warm JIT compilation time. In Figure 15, only half of tests have negative "percentage of increased MH JIT time". For

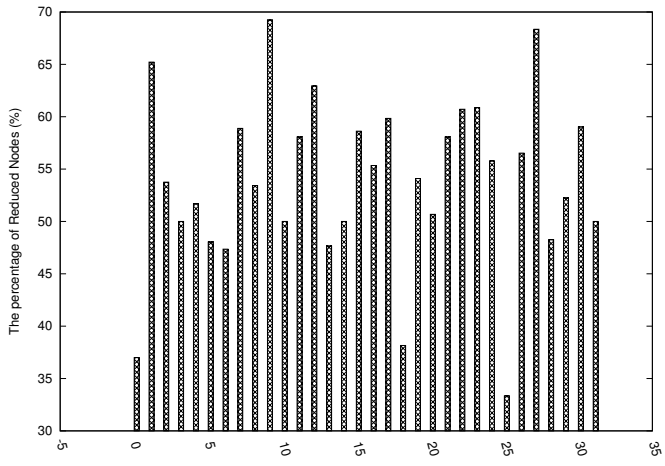

Figure 14. Percentage of Reduced MHs (Mean: 53.84%). x-axis is test-id, the order of which is consistent with x-axis in Figure 12.

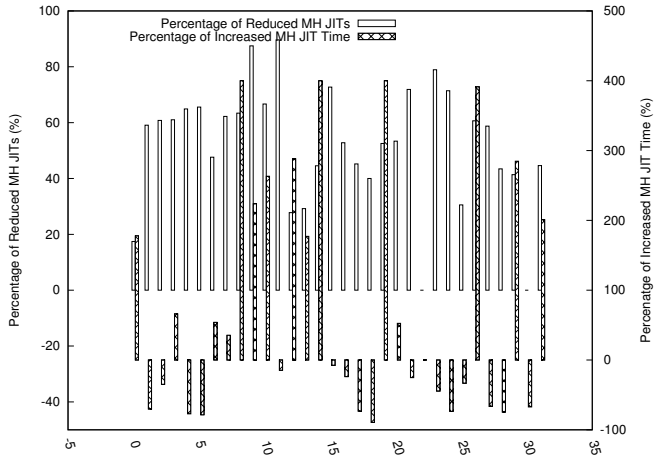

Figure 15. Percentage of Reduced MH Warm JITs vs. Percentage of Increased MH Warm JIT Compilation Time.

some tests (e.g., *14-nsieve-bits* and *8-primes*), MH JIT compilation time increases more than four times. The explanation is that MH JIT compilation time is largely determined by the transformation method's complexity (e.g., the number of if-else branches and exception handlers), while the selection policy with maximal bytecode size could not completely avoid complex MH generation. Although the size of queue bytecodes for fused MHs is still less than maximal bytecode size, these bytecodes might have complex structures and worsen TR MH compilation.

8 Related Work

Both method handles and *invokedynamic* instruction were proposed in JSR 292 for dynamically typed languages on the JVM. An early work is Rose's work [18], which summarizes the "pain points" of implementing dynamic JVM languages, and outlines all aspects of JSR 292 (e.g., implementation and optimization). Thalinger and Rose detail the instruction implementation in OpenJDK 7 [20]. Heidinga demonstrates IBM's implementation of *invokedynamic* and method handle pipeline design [4, 5]. Roussel et al. [19] present a JSR292 implementation in Dalvik, a register-based virtual machine for the Android OS.

JSR 292 has been adopted by many projects, such as the Soot Framework [1], Golo language [17], and Nashorn JavaScript Engine [15]. JRuby [13] is the most well-known project using JSR 292, where JRuby interpreter dynamically compiles Ruby scripts into JVM bytecodes. For dynamic method calls, JRuby interpreter can emit *invokedynamic* instructions and build method handle graphs for method resolution.

Inline caching (IC) is a significant optimization in the programming language area. The idea of inline caching is to remember previous method resolution results at call sites to avoid dynamic method resolution expense in the future. Inline caching was initially implemented in the Self interpreter [9, 10], and GraphJIT applies inline caching optimization to graph nodes, whose children are mutable.

The idea of object inlining (i.e., object fusion) is to transform heap data structures by inlining parent and child data together. Dolby et al. [6–8] implemented object inlining for a dialect of C++, and presented multiple compiler analysis methods (i.e., local data flow, nCFA, and adaptive analysis) to identify inlinable fields. In their solution, a child is selected only if the child would the current node's execution every time. GraphJIT distinguishes to Dolby's work in that GraphJIT targets FTSDA graphs, and its node selection is based on their entry counters and bytecode size.

Wimmer et al. provide another kind of object fusion for HotSpot JVM to replace field loading by address arithmetic. In their solution, garbage collection is modified to colocate frequently accessed objects that have parent-child relationships as groups in a consecutive region; *co-allocation* in the client compiler allocates objects as a group; and the JIT compiler performs field loading optimizations (i.e., address calculation and load folding) [21–23]. However, their solution has two preconditions: 1) both parent and child must be in a consecutive area, once they are created and 2) the relationship should be immutable. Our solution does not have such preconditions, and it only works on bytecode level that is relatively independent from JVM internals.

9 Future Work

Our future work mainly focuses on GraphJIT tuning. The main two tasks are to optimize MH selection policy to avoid generating complicated queue bytecodes, and to reduce

GraphJIT runtime expense (e.g., using asynchronous generation).

10 Conclusion

This paper presents a graph fusion solution to compile a method handle graph into another equivalent but simpler MHG on the bytecode level at program runtime, and to use the new MHG to replace the original MHG for execution. In the paper, we described each component (i.e., template system and GraphJIT) for graph fusion, and detailed the selection policy, as well as actions for mutable MHG structures. With JRuby Micro-Indy benchmark and JavaScript Octane benchmark on Nashorn, our experiment show the technique 1) can significantly speed up the typical MHG's traversal by 31.53% via AOT compilation; 2) reduces the execution time of Micro-Indy and Octane benchmarks by 6.28% and 7.73% on the average; 3) fuses approximately 54% of all MHs; and 4) reduces the number of MHs for JIT compilation startup by 52.1%, when GraphJIT is enabled.

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.

References

[1] Eric Bodden. 2012. InvokeDynamic Support in Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis (SOAP '12)*. ACM, New York, NY, USA, 51–55. DOI: https://doi.org/10.1145/2259051.2259059

[2] CLBG. 2017. Computer Language Benchmark Game. http://benchmarksgame.alioth.debian.org/. (2017).

[3] Da Vinci Machine Project. 2017. Da Vinci Machine Project. http://openjdk.java.net/projects/mlvm/. (2017).

[4] Dan Heidinga. 2014. Method Handle-An IBM implementation. http://wiki.jvmlangsummit.com/images/a/ad/J9_MethodHandle_Impl.pdf. (2014).

[5] Dan Heidinga. 2015. MethodHandle Compilation Pipeline. https://www.jfokus.se/jfokus15/preso/J9%20MethodHandle%20Compilation%20Pipeline.pdf. (2015).

[6] Julian Dolby. 1997. Automatic Inline Allocation of Objects. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI '97)*. ACM, New York, NY, USA, 7–17. DOI: https://doi.org/10.1145/258915.258918

[7] Julian Dolby and Andrew Chien. 2000. An Automatic Object Inlining Optimization and Its Evaluation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 345–357. DOI: https://doi.org/10.1145/349299.349344

[8] Julian Dolby and Andrew A. Chien. 1998. An Evaluation of Automatic Object Inline Allocation Techniques. *SIGPLAN Not.* 33, 10 (Oct. 1998), 1–20. DOI: https://doi.org/10.1145/286942.286943

[9] Urs Hölzle. 1995. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. Ph.D. Dissertation. Stanford, CA, USA. UMI Order No. GAX95-12396.

[10] Urs Hölzle and David Ungar. 1994. A Third-generation SELF Implementation: Reconciling Responsiveness with Performance. In *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications (OOPSLA '94)*. ACM, New York, NY, USA, 229–243. DOI: https://doi.org/10.1145/191080.191116

[11] IBM JIT Compiler. 2017. IBM Just-in-time Compiler for Java. https://www-304.ibm.com/partnerworld/wps/servlet/download/DownloadServlet?id=Hvdi$ITAyXHiPCA$cnt&attachmentName=IBM_just_in_time_compiler_for_java.pdf. (2017).

[12] J2SE 7: MethodHandle. 2017. Method Handle (Java Platform SE 7). https://docs.oracle.com/javase/7/docs/api/java/lang/invoke/MethodHandle.html. (2017).

[13] JRuby. 2013. JRuby. http://jruby.org/. (2013).

[14] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2013. *The Java Virtual Machine Specification, Java SE 7 Edition* (1st ed.). Addison-Wesley Professional.

[15] Nashorn. 2017. OpenJDK Nashorn Project. http://openjdk.java.net/projects/nashorn/. (2017).

[16] Octane. 2017. Octane Benchmark. https://developers.google.com/octane/. (2017).

[17] Julien Ponge, Frédéric Le Mouël, and Nicolas Stouls. 2013. 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)*. ACM, New York, NY, USA, 153–158. DOI: https://doi.org/10.1145/2500828.2500844

[18] John R. Rose. 2009. Bytecodes Meet Combinators: Invokedynamic on the JVM. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages (VMIL '09)*. ACM, New York, NY, USA, Article 2, 11 pages. DOI: https://doi.org/10.1145/1711506.1711508

[19] Gilles Roussel, Remi Forax, and Jerome Pilliet. 2014. Android 292: Implementing Invokedynamic in Android. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES '14)*. ACM, New York, NY, USA, Article 76, 11 pages. DOI: https://doi.org/10.1145/2661020.2661032

[20] Christian Thalinger and John Rose. 2010. Optimizing Invokedynamic. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ '10)*. ACM, New York, NY, USA, 1–9. DOI: https://doi.org/10.1145/1852761.1852763

[21] Christian Wimmer and Hanspeter Mössenböck. 2007. Automatic Feedback-directed Object Inlining in the Java Hotspot™Virtual Machine. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE '07)*. ACM, New York, NY, USA, 12–21. DOI: https://doi.org/10.1145/1254810.1254813

[22] Christian Wimmer and Hanspeter Mössenböck. 2008. Automatic Array Inlining in Java Virtual Machines. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '08)*. ACM, New York, NY, USA, 14–23. DOI: https://doi.org/10.1145/1356058.1356061

[23] Christian Wimmer and Hanspeter Mössenbösck. 2010. Automatic Feedback-directed Object Fusing. *ACM Trans. Archit. Code Optim.* 7, 2, Article 7 (Oct. 2010), 35 pages. DOI: https://doi.org/10.1145/1839667.1839669

[24] Shijie Xu, David Bremner, and Daniel Heidinga. 2015. Mining Method Handle Graphs for Efficient Dynamic JVM Languages. In *Proceedings of the Principles and Practices of Programming on The Java Platform (PPPJ '15)*. ACM, New York, NY, USA, 159–169. DOI: https://doi.org/10.1145/2807426.2807440