MHDeS: Deduplicating Method Handle Graphs for Efficient Dynamic JVM Language Implementations

Shijie Xu and David Bremner
IBM Centre for Advanced Studies (CAS Atlantic)
University of New Brunswick, Canada
{sxu3, bremner}@unb.ca

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

ABSTRACT

A method handle (MH) is a reference to an underlying Java method with potential method type transformations. Multiple inter-connected method handles form a method handle graph (MHG). Together with the Java Virtual Machine (JVM) instruction, *invokedynamic*, the implementation of MHGs is significant to dynamically typed language implementations on the JVM.

Addressing the abundance of equivalent MHGs during program runtime, this paper presents an MHG equivalence model and an online Method Handle Deduplication System (MHDeS). The equivalence model determines the equivalence of two MHGs in terms of two kinds of keys, i.e., MH key and MHG key, which uniquely identify the transformation of an MH and an MHG, respectively. MHDeS is an implementation of the equivalence model. Instead of creating equivalent MHGs and then detecting their equivalence, MHDeS employs a light-weight structure, the MHG index key, which wraps constructor parameters of an MH. MHDeS uses a transformation index, fast-path comparison, and filters, to speed up the equivalence detection of an MHG index key. Our experimental results with the Computer Language Benchmark Game (CLBG) JRuby micro-indy show that 1) MHDeS with filtering off can speed up the benchmark by 4.67% and reduces memory usage by 7.19% on average; 2) the deduplication result can be affected by the choice of MH transformations for filtering; 3) MHDeS can have the MH JIT compilation performed earlier; and 4) as much as 32% of MHG index keys are detected as non-unique and eliminated by MHDeS, and the expense for a single detection is trivial.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICOOOLPS'16, July 18 2016, Rome, Italy

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4837-9/16/07…$15.00
DOI: http://dx.doi.org/10.1145/3012408.3012412

Keywords

invokedynamic, dynamic JVM language, method handle, deduplication, graph key, graph matching

1. INTRODUCTION

Compiling dynamically typed languages into typed JVM bytecodes involves many "pain points" [1, 21, 4]. To emit typed JVM bytecodes, an operand's type information has to be determined before compilation, which is impossible for dynamically typed languages. Consequently, for a dynamic method call, interpreters (e.g., JRuby) atop JVM have to employ extra indirections (e.g., Reflection) to retrieve the right types for method linkage. These indirections might defeat JVM attempts to predict and inline call targets. Addressing these "pain points", JSR292 proposes method handles and the *invokedynamic* instruction, allowing customized method linkage.

1.1 Method Handle and Method Handle Graph

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 [7]. These transformations include method argument insertion, removal, and substitution, and their implementations are provided by MH's APIs, such as *dropArguments*, *insertArguments* and *guardWithTest*, in [8].

An MH has a type (i.e., method type), specifying the argument and return types of the MH. Its transformation comes along with an MH's execution when its invoker methods (e.g., *invokeExact*) are called. The sample code in Listing 1 shows how to build MHs (dynamic language interpreters atop the JVM build MHs in a similar way and link them to dynamic sites). In the sample, the MH g's type is `()boolean`. Another *FilterReturnhandle d0* filters *gwt*'s return value by *f*, which appends *"yahoo"* to the parameter.

As a Java method referenced by an MH can have multiple references to other MHs, multiple inter-connected MHs form a directed Method Handle Graph (MHG) [24].

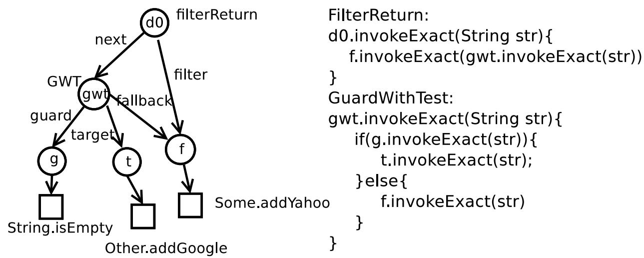

Figure 1: MHG sample

In an MHG, a node represents a method handle instance; a directed edge represents that the method referenced by the source MH has a reference to the target MH; and a label associated with the edge is the reference name of the target MH. According to JSR292, an MHG is responsible for transferring a dynamic invocation to a number of real method invocations via transformations. For example, *d0* and *gwt* in Figure 1, corresponding to Listing 1, represent *FilterReturn* and *GuardWithTest* transformation, respectively. This MHG transfers the invocation at *d0* to *f*, *g*, and either of *t* and *f* finally.

```
MethodHandle g = lookup().findVirtual(
    String.class, "isEmpty", methodType(
    boolean.class));
MethodHandle t = lookup().findStatic(
    Some, "addYahoo", methodType(String.
    class, String.class)); //append
    google to the tail.
MethodHandle f = lookup().findStatic(
    Other, "addGoogle", methodType(
    String.class, String.class)); //
    Append yahoo to the tail.
MethodHandle gwt = MethodHandles.
    guardWithTest(g, t, f);
MethodHandle d0 = MethodHandles.
    filterReturn(gwt, f);
assertEquals((String)d0.invokeExact(``
    str''), ``Strgooglegoogle'');
assertEquals((String)d0.invokeExact(``'
    '), ``yahoogoogle'');
```

Listing 1: Method Handle Graph

1.2 Motivation

A key optimization is MHG deduplication, which detects and eliminates equivalent MHGs—MHGs having similar graph structure and corresponding nodes with equivalent attributes[1]—at program runtime. This idea is directly motivated by the finding that as much as 28.87% of method handles are detected to start similar MHGs [24]. Intuitively, the existence of these equivalent MHGs at runtime burdens a program's memory usage, and slows down its execution, due to repeated constructions of equivalent MHGs.

[1]Similarly, equivalent MHs mean that the graphs starting from those MHs are equivalent.

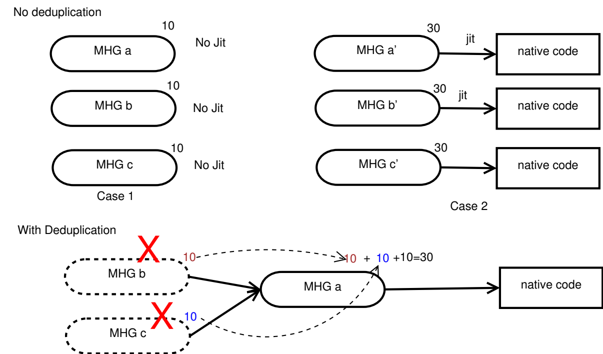

Figure 2: MHG JIT sample

An MHG Just-In-Time (JIT) compilation is an MHG translation from bytecode to machine code, when the number of its invocations exceeds a given threshold. This translation in prevalent compilers (e.g., Testarossa (TR) in IBM J9 [17], *Jalapeno* in the Jikes VM [11, 12] and Client/Server compiler in HotSpot) normally takes two phases. In J9, an MH is translated into a compiled version, called a shared thunk, if it is frequently executed (e.g., its *invocationCount* exceeds a threshold). To reduce the repeated transitions between compiled MH and interpreted MH, these compiled MH versions of a graph are later inlined together as a single native method (i.e., custom thunk) [9], which is set as a member of the root MH or a dynamic site. In other words, an inlined MHG is exclusively used by the root MH in the graph or a site, and is not shared among MHGs. The conclusion from Client/Server compiler in HotSpot [23] is also similar.

Owing to this lack of inlined MHG sharing, the abundance of equivalent MHGs constrains the exploitation of JIT compilation. As shown in Figure 2, three equivalent MHGs, *a*, *b*, and *c*, are not eligible for JIT compilation in Case 1, because all of their *invocationCount*s are less than the JIT threshold, i.e., 30. Similarly, three other equivalent MHGs, *a'*, *b'*, and *c'*, are JITted separately as each MHG is executed 30 times, which results in three complete JIT compilations for the same bytecodes and three equivalent compiled codes. Thus, a motivation for equivalent MHG deduplication is to avoid redundant JITted MHGs by substituting *a* for *b* and *c*, so that all *invocationCount*s can be aggregated, as shown in *with deduplication* part in Figure 2. This would also drive earlier JIT compilation of *a* since *a*'s *invocationCount* will more easily reach the threshold.

This paper provides an MHG equivalence model and an online MHG Deduplication System (MHDeS). In the paper, two kinds of keys, MH key and MHG key, are introduced to uniquely identify an MH's transformation and an MHG's transformation, respectively. MHDeS implements the equivalence model in J9 (it is also portable to other JVMs as it does not use any J9 specific APIs), and it consists of a Method Handle Pool (MH pool), a purger, a detector, and filters. MHDeS organizes all de-

tected unique MHs in the MH pool, and employs MHG index key, a light-weight structure that wraps constructor parameters of an MH, to represent an MH about to be created. By comparing whether an MHG index key is equivalent to an existing MH in the pool, MHDeS reduces the number of recursive traversals and avoids creating an equivalent MH. To minimize its memory burden, MHDeS also purges infrequently matched MHGs from MH Pool at runtime. Tested with the Computer Language Benchmark Game (CLBG) JRuby Micro-indy benchmark [3], our results show that 1) MHDeS can speed up this benchmark by 4.67% and reduce memory usage by 7.19%; 2) the deduplication result can be changed dramatically by choosing MH transformations for filtering carefully; 3) MHDeS can have the MH JIT compilation performed earlier; and 4) as much as 32% of MHG index keys are detected as non-unique and eliminated by MHDeS, and the mean expense for a single detection is less than 1ms for the majority of tests.

1.3 Contribution

- We formulate a graph equivalence model that defines two MHGs' equivalence by their MHG keys. Based on the model, we show that the complexity of finding equivalent sub-MHGs for a given MHG is only quadratic in the MHG's size.
- MHDeS improves the equivalence model by 1) an MHG index key that avoids creating equivalent MHs at program runtime; 2) a transformation index that speeds up transformation chain lookup and reduces the MHG comparison space; and 3) a fast-path comparison and an MH pool that prevents time-consuming graph traversal by detecting non-equivalent MHGs as early as possible.
- We quantify the analysis of MHDeS performance in terms of CPU time, memory usage, MHG reduction effectiveness and time expense for deduplication.

The rest of this paper is organized as follows. Section 2 presents the equivalence model. Section 3 overviews MHDeS and design decisions. Section 4 shows evaluation results from perspectives of measures and MHDeS expense. Section 5 and 6 provide related work and future work, respectively. Finally Section 7 concludes this paper.

2. MODEL

2.1 Method Handle Graph (MHG)

A method handle graph has following features

- An MHG has only one root, via which all MHs are accessible. In this paper, an MHG is represented by its root MH.

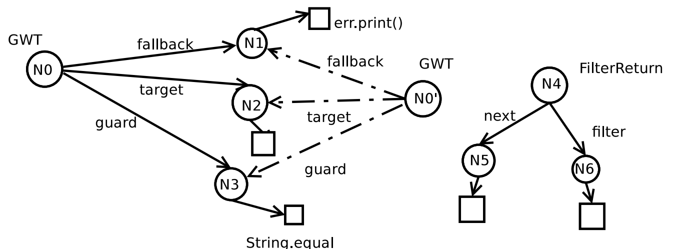

Figure 3: Equivalent MHG sample

- The leaves of sub-MHGs are a subset of the original MHG's leaves.
- Most MHGs (the MHGs that can be linked to a *MutableCallSite* are the exception) are constant once they are created.

In this paper, $G_{mh}(V, E)$ defines an MHG starting from root MH *mh*, where V is a set of MHs and E is a set of directed edges that connect these method handles together. The creation of an MHG involves the creation of individual MHs in the graph and edges' setup among these MHs. The creation of terminal MHs can be completed via a reflective API *MethodHandles.Lookup* while non-terminal MH creation involves *invokespecial* invocations and a number of method type checks.

Equivalent MHGs are graphs that have similar graph structure and the corresponding MH nodes with the same transformation. For example, there are three MHGs in Figure 3 and only two of them, G_{N0} and $G_{N0'}$ are equivalent.

2.2 MH Key and MHG Key

Method Handle (MH) key.

An MH key, $\mathrm{MH_key_{mh}}$, is a unique identifier for an MH. It is made up of an MH's transformation *name*, a method type that the transformation applies to, and optional data. The transformation characterizes the type of the transformation, and it is the API that creates this MH, e.g., *guardWithTest*, *insertArguments*, *filterReturn*, if it is not a terminal MH. The optional parameters are only necessary for some special transformations. For example, the *insertArguments* transformation requires the existence of two variables: *pos*, the position where the insert occurs, and *values*, an array that indicates what to insert. For a terminal MH (e.g., a *DirectHandle*) that does not have any transformation, its transformation is *null*.

Method Handle Graph Key.

An MHG key, $\mathrm{MHG_key_{mh}}$ is associated with the root MH, *mh*, of an MHG, and uniquely identifies the transformation that the whole MHG performs. Different from an MH key, an MHG key is made up of MH keys of all method handles in the graph, and their connections. In this paper, the MHG key for graph $G_{mh}(V, E)$

is defined as

$$\text{MHG_key}_{\text{mh}} = \begin{cases} \text{MH_key}_{\text{mh}}, \text{if mh is a terminal MH} \\ \{\text{MH_key}_{\text{mh}}, \text{S}_{\text{mh}}\} \end{cases} \quad (1)$$

where S_{mh} is *mh*'s ordered set of child MHs, where child order is customized by *mh*'s transformation. For example, *guardWithTest* has three children and its S_{mh} is a list [*guard, target, fallback*], while S_{mh} of *filterReturn* is [*next, filter*] instead of [*filter, next*].

Thus, an MHG can be represented by its MHG key, which is recursively built from MH keys and the structure of the graph. Two MHGs can be classified as equivalent if both MHG keys are equivalent, while the equivalence of two MHs, both of which have the same MH keys, does not mean that the graphs starting from both are equivalent.

2.3 MHG Equivalence Model

Our method handle equivalence solves two questions.

Are two given MHGs equivalent?.

The model defines an MH equivalence function $F : (m, n) \rightarrow \{1, 0\}$, where m and n are the roots of MHG G_m and G_n, respectively. If $F(m, n) = 1$, then the MHGs G_m and G_n are also equivalent. The equivalence MH function $F(m, n)$ can be formulated as

$$F(m, n) = (m = n) \vee (f'(m, n) \wedge (f''(m) = f''(n))) \quad (2)$$

where $f''(m) = \text{MH_key}_\text{m}$. The function $f'(m, n)$ compares S_m and S_n, of the MHs m and n. Together with MH key, a simplified $f'(m, n)$ is shown in Equation 3.

$$f'(m, n) = \begin{cases} \bigwedge_{i=1}^{|S_m|} F(S_m(i), S_n(i)), \\ \quad \text{if } S_m \neq \emptyset \wedge S_n \neq \emptyset \wedge |S_m| = |S_n| \\ 1, \quad S_m = S_n = \emptyset \\ 0, \quad \text{otherwise} \end{cases} \quad (3)$$

Based on Equation 2, the complexity to determine two MHs' equivalence is only $O(|G|)$ since MHs in a child MH set are ordered in advance.

Do equivalent sub-MHGs exist?.

Based on the previous conclusion, this question can be answered by comparing the given MHG to individual MHGs in the potential sub-MHG set.

This problem is not NP hard. Assume the given MHG with n MHs, the target MHG with m MHs ($m \gg n$). First, according to MHG's features in Section 2.1, there are at most m sub-MHG pairs to compare (one is the given MHG and the other is a sub-MHG from the target MHG). Second, for each pair, the complexity of comparison is $O(n)$, that is the total number of MH nodes in the given MHG. Thus, the total complexity is $O(n*m)$, which can be simplified as $O(m^2)$.

3. SYSTEM DESIGN

This section discusses our prototype design, Method Handle Deduplication System (MHDeS), which is capable of deduplicating equivalent method handles at program runtime, and our implementation decisions. Instead of conducting equivalence detection for method handles directly, MHDeS determines uniqueness of a method handle about to be created by its MHG index key (Section 3.2), and creates this method handle only if it is unique after detection. MHDeS extends the MHG equivalence model by 1) reducing detection overhead by only comparing the MHs with the same transformation in the pool, 2) an MHG index key, and 3) a fast-path comparison to avoid unnecessary comparison by detecting non-equivalent MHs as early as possible.

MHDeS consists of two layers. The lower layer mainly consists of three components: the MH pool, the purger thread (purger), and configurations. MH pool organizes all unique method handle references at runtime; the purger thread periodically removes MH references that are not frequently hit during comparison. The higher layer is mainly made up of a detector and a number of filters associated with the detector.

3.1 MH Pool

As shown in Figure 4, the MH pool organizes all unique MHs as *MHObject*s. An MHObject consists of a weak MH reference and an integer *count*. The former holds a reference to a unique MH without blocking its Garbage Collection (GC), and the latter indicates the number of times the corresponding MH has been hit during equivalence detection.

The MH pool mainly consists of a transformation index and MHObject chains. In the pool, all detected unique MHs (MHObjects) that have the same transformation type are connected as a chain in descending order by their *count*s. Similar to the MHObject, each chain is labeled with *maxCount*, which indicates the maximum *count* value that MHObjects on the chain have. The MHObject chain is always constructed with a single MH when the corresponding transformation name is missed in the transformation index, and is updated when a new unique MHObject is inserted into the correct position in the chain or existing MHObjects are purged.

Transformation Index.

The transformation index is used to index MHObject chains by the transformation name (for a terminal MH, it is MH's class name (e.g., *DirectHandle* and *VirtualHandle*). As shown in Figure 4, the chain with 10 MHObjects of *InsertArgument*s is indexed by one transformation name *InsertArgument*. With the transformation index, an MHObject chain can be retrieved in one hash table lookup, and a method handle or MHG index key is only compared to MHs in the MHObject chain that have the same transformation type as the given

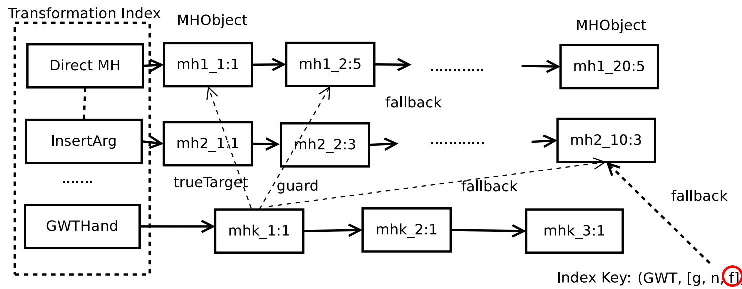

Figure 4: MH Pool Overview

MH.

3.2 Detector

Based on the MH pool, the Detector determines whether an MH about to be created is equivalent to any existing method handle in the pool and deduplicates it if it is equivalent to an existing one. Instead of creating the MH and then comparing it to existing MHs in the pool, the Detector uses an MHG index key, an object that wraps all necessary arguments for creation of that MH, to existing MHs in the pool. If none of the existing MHs in the pool is matched, then the new method handle is created according to the MHG index key and added to the MH pool.

MHG index key.

The motivation for the MHG index key is to avoid wasting CPU and memory resources on non-unique MHs. First, creation of an MHG involves a number of MH creations from the leaves to the root of an MHG. That means, the MHs at the bottom of the graph are first constructed, and then these newly created MHs are passed to *MethodHandles* APIs (e.g., *guardWithTest*) as arguments to create higher MHs in the MHG. Second, using Equation 2 for equivalence comparison requires the existence of MHs first, and to have it not referenced by any existing MHs if it is non-unique, so that these isolated MHs can be garbage collected later. Considering the effort of constructing higher MHs in an MHG (i.e., a number of method type checks and *invokespecial* invocations) and the GC, much of this effort for non-unique MHs can be avoided by using MHG index keys.

An MHG index key is a data object that holds all arguments for a new concrete MH construction, and it can completely represent the MH that can be created from this key. This index key is made up of a transformation (*cls*), an MH key (*mhKey*), and its ordered child MHs (*children*). For example, the *children* in an MHG index key of *GuardWithTest* are [*guard*, *trueTarget*, *falseTarget*], while some other necessary arguments for transformation, e.g., [*pos*, *objects[]*] for *insertArgument* transformation, have been embedded in the *mhKey*. Terminal MHs (leaves) don't have any MHG index key since they do not have any transformation.

Detection Procedure.

Based on the MHG index key, the new MH equivalence function becomes

$$F(m,n) = (m = n) \vee (f'(iKey,n) \wedge f''(iKey) = f''(n)) \tag{4}$$

and

$$f'(iKey,n) = \begin{cases} \bigwedge_{i=1}^{|S_n|} F(S_{iKey}(i), S_n(i)) & |S_{iKey}| = |S_n| \\ 0 \quad f''(iKey) \neq f''(n) \end{cases} \tag{5}$$

where the $iKey$ is the MHG index key of the MH, m, to be created, and n is an existing MH with the same transformation in the pool. The complete detection procedure is shown in Algorithm 1.

In Algorithm 1, there are two kinds of comparisons: fast-path comparison and slow-path comparison.

Fast-Path comparison.

A fast-path comparison only compares the equivalence of MH keys. In Algorithm 1, for an MHObject in the chain, its MH key is retrieved and compared to that of the newly created MHG index key. Slow-path comparison will only be possible when the fast-path comparison succeeds. Thus, the fast-path comparison is an early detection of un-matched MHs to avoid the unnecessary cost of slow-path comparison.

Slow-path comparison.

Slow-path comparison compares an index key (excluding its MH key) to existing MHs in the MH pool. For each MH in the *children* of the index key, it is compared to an existing MH in the pool by the method *equals*, which returns true only if both *tested* and *mh* in Algorithm 1 are the same MH instance, or their MHG keys are equivalent according to Equation 2.

The overhead of the recursive equivalence calculation using Equation 2 is largely reduced by the MH pool. This is because most *children* of an index key are unique MHs and have been placed in the pool. Thus, a comparison result can be quickly made by testing whether both are the same instance, and the *true* is returned if they are. For example, the MH f in the MHG index key

in Figure 4 refers to the MH instance which is unique and has already in the pool.

Algorithm 1 Equivalence detection and elimination

```
1: procedure getUnique(cls, mhKey, children, args)
2:     if !filter_on() or (filter_on() and cls is not filtered) then
3:         indexkey = MHGIndexKey.create(cls, mhKey, children, args);
4:         mhList= MH Pool.get(cls)
5:         for all MHObect mho: mhList do
6:             key=mho.getMHKey()
7:             if !key.equal(indexkey.getMHKey()) then      ▷ fast-path f''
8:                 Continue
9:             end if                                      ▷ slow-path f' now
10:            i=0
11:            while i < mho.getChilds().size() do
12:                tested = mho.getChild(i)
13:                mh = get i^th child of indexKey
14:                if !tested.equals(mh) then
15:                    break
16:                end if
17:                i++
18:            end while
19:            if i < mho.getChilds().size() then
20:                mho.incr(); return mho.getMH()
21:            end if
22:        end for
23:    end if
24:    mh = cls.newInstance(args, MHs)
25:    mh.cacheMHIndexKey(indexKey)    ▷ cache the indexKey
26:    Add mh to the tail of mhList
       return mh
27: end procedure
```

3.3 Filtering

MHDeS provides filter APIs to exclude some transformations for deduplication, so that the expense and effectiveness of deduplication can be balanced. According to the finding by Xu et al. [24], equivalent MH ratios of different kinds of method handles—the number of equivalent method handle pairs to the total number of pairs of that kind method handle—vary sharply. Thus, for those kinds of method handles, which have small equivalent MH ratios, their transformation names can be filtered out directly to avoid the detection cost of these transformations as the potential improvement is trivial.

3.4 Purge

Both MHObjects and transformation chains are checked periodically and removed if they are rarely matched during detection. The purges of both kinds of object are conducted in the purge thread which is scheduled at an interval of 5 seconds.

MHObject Purge.

This operation only removes cold MHObjects, the *count* values of which are less than the given threshold. In the pool, different method handles, represented by the MHObjects, have varied hotness, and a cold MHObject is unlikely to match any MHG index keys during detection. Along with the purge, the chain is re-sorted again by MHObject's *count.* Instead of sorting it during detection, the re-sort operation following purge aims to reduce potential waiting time caused by MHDeS.

Transformation Chain Purge.

Similar to the MHObject purge, the transformation chain purge aims to remove the whole transformation chain when all MHObjects in the chain are cold. During chain purge, if *maxCount* is below a threshold, the whole chain will be removed.

4. EVALUATION

In this section, we first describe experimental background. Then we analyze MHDeS' impact in terms of elapsed CPU time, memory, JIT compilation for our benchmark. We also evaluate deduplication effectiveness and the overhead of MHDeS at the program runtime.

We run the JRuby micro-indy benchmark in CLBG [3] atop IBM J9 Java 8. CLBG contains 41 Ruby tests, and most of these tests' lengths are between 10 and 50 lines. When seeing an *invokedynamic* instruction, JVM calls the corresponding *bootstrap* method, which in turn uses MHDeS to create new MHs. Regarding filtering, two arbitrary transformations are chosen blindly and configured in the MHDeS.

4.1 Performance Impact

The elapsed CPU time, CPU_Time, is one of the key measurements for program performance comparison, and it is measured when a test ends as the seconds passed after the start of this test. In theory, for a test with only a single thread, CPU_Time is

$$\mathrm{CPU_time} = \sum_{i=1}^{n_1} \mathrm{T_int_i} + \sum_{i=1}^{n_2} \mathrm{T_exe_i} + \sum_{i=1}^{n_3} \mathrm{T_gc_i} \quad (6)$$

and the definitions of the variables are shown in Table 1. In this Equation, MHDeS impacts the CPU_time by changing bytecode $\sum_{i=1}^{n_1} \mathrm{T_int_i}$, which in turn results in changes of the other three components. For the bytecode interpretation, the added overhead of MHDeS is the execution of deduplication, while the reduced overhead is the creation of equivalent MHGs and the interpretation of the MHGs, the JIT of which occur earlier.

Figure 5 shows our comparison results and performance improvements made by MHDeS, respectively. In these figures, the bars labeled with *MHDeS* represent the data with MHDeS, while the ones with *Orig* represent the runtime without MHDeS. In Figure 5a 5b , the

(a) Filtering Off + CPU *ip* (Median: 0.9%, Mean: 4.67%, std:14.48)

(b) Filtering On + CPU *ip* (Median: 1.55%, Mean: 2.77%, std: 4.84)

Figure 5: CPU time performance comparison

T_int_i	Time expense on interpreting a bytecode;
n_1	the number of interpreted bytecodes.
T_exe_i	Time expense on executing a JITted code;
n_2	the number of executed native code instructions.
T_gc_i	Paused time when doing a global GC
n_3	the number of global GCs.

Table 1: Variables for Elapsed CPU Time

y axis is improved performance *ip*, which is evaluated as

$$\frac{\text{CPU_time}_{\text{Orig}} - \text{CPU_time}_{\text{MHDeS}}}{\text{CPU_time}_{\text{Orig}}} * 100\% \qquad (7)$$

According to Figure 5, MHDeS speeds up benchmark's execution on average. In the figure, the average speedups are 4.67% and 2.77% when filtering is off and on, respectively. Though these average improvements are only moderate, the improvements of the deduplication on tests varies. For example, the maximal execution speedup can reach as high as 77%, and the minimal speedup is only -6% when filtering is off. This dramatic change in improvement is caused by the varieties of different MHG constructors among individual tests. Different tests have their preferences for MHG transformations, and the overhead to interpret and deduplicate these MHG transformations also varies. For example, there is rarely any MHG transformations in tests *eval* and *so_sieve*, while there are more than 8 kinds of transformations in the test *so_matrix*.

The transformation for filtering has significant impact on the elapsed CPU time, and should be carefully chosen to maximize potential performance speedup. As shown in Figure 5b, the *ip*'s deviation is reduced from 14.48 to 4.85 when filtering is on, which makes *ip*'s distribution among different tests more smooth. For individual tests in Figure 5b, the *ip* for *so_matrix* is reduced from 77% to 8%, while it increases from 0.5% to 25% for *spectral_norm* when filtering is on. This change can be explained by the distinction of the transformation combination and its deduplication expense. For example, compared to *so_matrix*, the MHG for *spectral_norm* has more *GuardWithTest* MHs, which is more expensive for deduplication than the *AsType* transformation.

Figure 6: Memory usage, (Median: 9.84%, Mean: 7.19%)

4.2 Memory Impact

The evaluated memory measures are memory usage

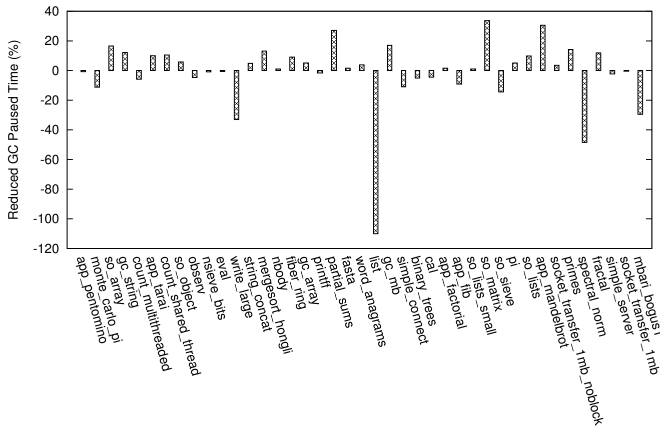

Figure 7: GC Paused Time, Filtering Off (Mean: 1.65%)

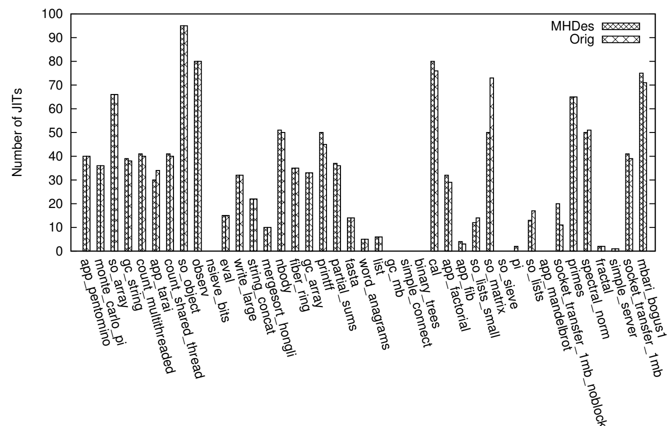

Figure 8: Number of MH JITs, Filtering Off

and GC pause time. On one side, MHDeS alleviates memory consumption as it reduces the occurrences of equivalent MHG creation. On the other side, it creates extra temporary objects, e.g., MHObjects, MHG index keys, etc, during equivalence detection and holds them in the pool, which in turn increases the amount of work for a GC.

The GC policy used in our experiment is the default *gencon* [5], which aims to maximize throughput. With this policy, a heap is divided into a nursery and a tenured area, and objects are first created in the nursery and promoted to the tenured area if these objects survive a certain number of GCs. In the experiment, we configure the JVM to log all GCs (e.g., the occupied memory, time, duration) for analysis.

Memory usage.

Memory usage is occupied memory after the last GC. As shown in Figure 6, MHDeS can reduce memory usage by 7.19% on average. For those benchmarks that have negative reduction percentages, the memory occupied by MHDeS (mainly MHObjects and chains) outweighs the reduced equivalent method handles.

GC Pause Time.

According to Figure 7, the GC pause time is reduced by 1.65% on average, which is only a minor improvement. With MHDeS, there is an increase in short-lived objects, i.e., temporary objects, and a decrease in the number of MHGs. In this case, the incremental GC overhead on temporary objects is largely counteracted by the reduced GC overhead of the equivalent MHGs. Besides, MHs are only a small portion of objects created by the bytecode compiled from those tests. In the figure, the *list* has the worst measurement own to an stack overflow during runtime.

4.3 JIT compilation Impact

Figure 8 shows MHDeS's impact on the number of Method Handle (MH) JIT compilations. According to figure, the number of MH JIT compilations is stable regardless of the presence MHDeS. For test *so_matrix*, the number of JIT compilations is reduced from 75 to 55, which is the maximal change of JIT compilation in our experiment. For other tests, there is trivial change in the number of JITs.

MHDeS also helps move MH JIT compilation earlier. First, the JIT compilation is conducted asynchronously. Second, Equation 6 can be simplified to

$$\text{CPU_time} \approx \sum_{i=1}^{n_1} \text{T_int}_\text{i} + \sum_{i=1}^{n_2} \text{T_exe}_\text{i} \qquad (8)$$

as the percentage of accumulated paused GC time to the total number of *CPU_time* is trivial (the mean is less than 0.8%) when the filtering is off. Since the bytecode method for JIT is nearly fixed, the only explanation for the reduction of CPU_time (4.67%) is earlier JIT compilation. Otherwise, there would be reduced $\sum_{i=1}^{n_2} \text{T_exe}_\text{i}$, but an increase in $\sum_{i=1}^{n_1} \text{T_int}_\text{i} + \sum_{i=1}^{n_2} \text{T_exe}_\text{i}$, which results in a contradiction.

4.4 MHG Reduction Efficiency and MHDeS Expense

MHG Reduction Productivity (MRP) is a ratio of the number of times (n_0) that an MHG index key is detected to be equivalent to the total number of MHG index keys (n) that MHDeS receives for equivalence comparison. This measure indicates MHDeS's productivity, and the higher MRP is, the more productive MHDeS is.

$$\text{MRP} = \frac{n_0}{n} * 100\% \qquad (9)$$

The MHDeS expense, which is evaluated as one equivalence detection for an MHG index key, is shown in Table 2. Compared to the whole program runtime, which normally takes more than 10s, the data in the table shows that the deduplication cost is trivial, as the mean

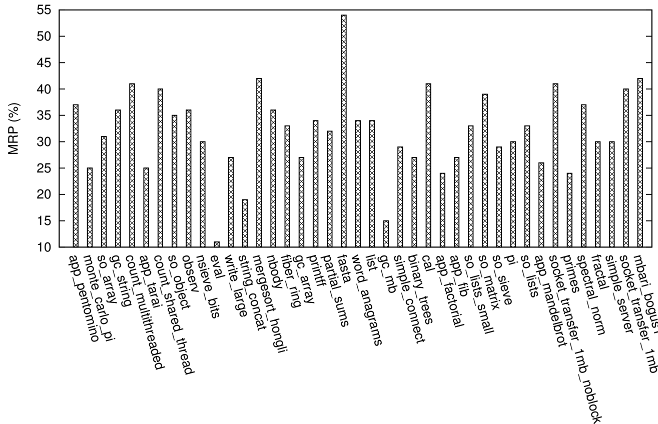

Figure 9: MHG Reduction Productivity, Filtering Off, Mean: 32%

maximal time expense for individual tests is about 10ms (after excluding the dirty data for *list*, the maximal expense is 167ms for *mbari_bougs1*, the *cpu_time* of which is about 26s). Besides, the mean deduplication expense is less than 1ms because the expenses for the majority of deduplication are nearly 0 as the *tested* and *mh* in Procedure 1 refer to the same MH in the MH pool.

	Mean(ms)	**Max**(ms)		**Mean(ms)**	**Max**(ms)
app_pentomino	0.163418	10	monte_carlo_pi	0.415842	11
so_array	0.206452	10	gc_string	0.375	15
count_multithreaded	0.171504	11	app_tarai	0.241667	10
count_shared_thread	0.180247	15	so_object	0.179426	9
observ	0.191558	11	nsieve_bits	0.180233	8
eval	0.468085	9	write_large	0.404762	7
string_concat	0.220339	8	mergesort_hongli	0.222973	9
nbody	0.179562	9	fiber_ring	0.268698	9
gc_array	0.2	9	printff	0.337539	8
partial_sums	0.281437	10	fasta	0.22314	9
word_anagrams	0.197044	9	list	0.945606	296
gc_mb	0.285714	8	simple_connect	0.223214	12
binary_trees	0.297872	10	cal	0.181435	8
app_factorial	0.209302	6	app_fib	0.211009	8
so_lists_small	0.190311	11	so_matrix	0.378261	9
so_sieve	0.272727	8	pi	0.178571	9
so_lists	0.273356	7	app_mandelbrot	0.376	10
primes	0.252137	10	socket_transfer_1mb_noblock	0.170673	1
spectral_norm	0.194774	9	fractal	0.231884	10
simple_server	0.201681	9	socket_transfer_1mb	0.204334	1
mbari_bogus1	0.0388703	167			

Table 2: MHDeS Expense

5. RELATED WORK

5.1 Method Handle and *invokedynamic* instruction

Our work is inspired by the work of Xu et al. on method handle data mining [24], which provided some initial ideas about equivalent method handles in method handle instance patterns. Compared to that work, our work is the first attempt to optimize method handle graphs on the JVM for dynamic JVM language implementations.

The most relevant work with method handles is about the *invokedynamic* instruction and the projects that adopt it. At the beginning of JSR 292, Rose from Oracle outlined all aspects of *invokedynamic* instruction and possible optimization technologies [21]. Though the work described in the paper is now obsolete, the concept is still valid. After Rose, many JSR292 implementations have been proposed. Thalinger and Rose detailed the new instruction implementation that was released with OpenJDK 7 [23]; Heidinga from the J9 team demonstrated IBM's implementation of *invokedynamic* and method handle pipeline design [6, 9]; Gilles Roussel *et al.,* presented a JSR292 implementation in Dalvik, a register-based virtual machine for the Android OS [22], which is much different from the implementation in HotSpot and J9 as Dalvik is a resource constrained VM.

The new instruction has been adopted in many projects. Bodden extends Soot, a framework for static analysis and transformation of Java programs, to support the processing and generating of the *invokedynamic* instruction [13]. Similarly, Ponge *et al.* present the design of Golo, a dynamic programming language for rapid prototyping and polyglot application embedding, and explain how the *invokedynamic* instruction is used in this project [19]. In addition to these research projects, most well-known dynamically typed language interpreters e.g., JRuby (A Java implementation of the Ruby programming language) [1, 2], JPython, and Nashorn (A JavaScript engine based on Da Vinci Machine and released with Java8) [10], are all pioneers of this instruction.

5.2 Graph Deduplication

Based on graph isomorphism, graph deduplication is a technology to identify and eliminate equivalent graphs at runtime, which is similar to the entity resolution problem in the survey [16]. Existing work on graph identification are rule-based [15], learning-based [20] method and so on. The early work using graph keys to identify graphs is in the work by Pernelle et al. [18], which specifies keys for Resource Description Framework (RDF) data by a combination of objects' and data properties defined over an OWL ontology. Based on graph patterns, Fan et al. [14] introduces a recursive graph key generation. We apply and simplify Fan's work by taking advantage of MH domain information.

6. FUTURE WORK

Our next work for *MHDeS* is adaptive filtering. The overhead of the equivalence detection for different MH transformations varies. In order to maximize the potential benefits and reduce the effort of deduplication, the transformation for filtering should be carefully chosen, and be adaptive to all of kinds of JVM languages, instead of a single JRuby interpreter. Our direct plan is to introduce a profiling module, which evaluates the deduplication cost and potential improvement for each transformation. After training, this module is later used to guide MHDeS what kinds of MHs are most suitable for filtering.

7. CONCLUSION

This paper provides a method handle graph equivalence model and a system, called MHDeS, to eliminate equivalent MHGs at program runtime. MHDeS indexes all unique MHs in the MH pool by their transformation indexes, and detects the uniqueness of an MH about to be created at runtime. During detection and elimination, MHDeS a) creates an MHG index key to represent the MH about to be created during comparison; b) only compares MHG index keys to the MHs that have the same transformation for the purpose of detection overhead reduction; and c) uses fast-path comparison, which detects non-equivalent MHs as early as possible. Our experimental results show that 1) MHDeS with filtering off can speed up dynamic JVM languages by 4.67% and reduces memory usage by 7.19% on average; 2) there is a space to maximize performance speedup by tuning the choice of filtered transformations; 3) MHDeS can have the MH JIT compilation performed earlier; and 4) as much as 32% MHG index keys are detected non-unique and eliminated with MHDeS, and the deduplication expense is trivial when compared to the whole life of the test.

Acknowledgment

This work is supported by the Atlhantic 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.

8. 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] Da Vinci Machine Project. http://openjdk.java.net/projects/mlvm/.

[5] Java technology, IBM style: Garbage collection policies, Part 1. http://www.ibm.com/developerworks/library/j-ibmjava2/.

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

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

[8] Method Handles (Java Platform SE 7). https://docs.oracle.com/javase/7/docs/api/java/lang/invoke/MethodHandles.html.

[9] MethodHandle Compilation Pipeline. https://www.jfokus.se/jfokus15/preso/J9%20MethodHandle%20Compilation%20Pipeline.pdf.

[10] OpenJDK Nashorn Project. http://openjdk.java.net/projects/nashorn/.

[11] Alpern, B., Attanasio, C. R., Barton, J. J., Burke, M. G., Cheng, P., Choi, J.-D., Cocchi, A., Fink, S. J., Grove, D., Hind, M., Hummel, S. F., Lieber, D., Litvinov, V., Mergen, M. F., Ngo, T., Russell, J. R., Sarkar, V., Serrano, M. J., Shepherd, J. C., Smith, S. E., Sreedhar, V. C., Srinivasan, H., and Whaley, J. The JalapeñO Virtual Machine. *IBM Syst. J. 39*, 1 (Jan. 2000), 211–238.

[12] Arnold, M., Fink, S., Grove, D., Hind, M., and Sweeney, P. F. Adaptive Optimization in the Jalapeno JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2000), OOPSLA '00, ACM, pp. 47–65.

[13] Bodden, E. InvokeDynamic Support in Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis* (New York, NY, USA, 2012), SOAP '12, ACM, pp. 51–55.

[14] Fan, W., Fan, Z., Tian, C., and Dong, X. L. Keys for graphs. *Proc. VLDB Endow. 8*, 12 (Aug. 2015), 1590–1601.

[15] Fan, W., Gao, H., Jia, X., Li, J., and Ma, S. Dynamic constraints for record matching. *The VLDB Journal 20*, 4 (Aug. 2011), 495–520.

[16] Getoor, L., and Machanavajjhala, A. Entity resolution: Theory, practice & open challenges. In *International Conference on Very Large Data Bases* (2012).

[17] Grcevski, N., Kielstra, A., Stoodley, K., Stoodley, M., and Sundaresan, V. JavaTM Just-in-time Compiler and Virtual Machine Improvements for Server and Middleware Applications. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3* (Berkeley, CA, USA, 2004), VM'04, USENIX Association, pp. 12–12.

[18] Pernelle, N., SaÃŕs, F., and Symeonidou, D. An automatic key discovery approach for data linking. *Web Semantics: Science, Services and Agents on the World Wide Web 23* (2013), 16 – 30. Data Linking.

[19] Ponge, J., Le Mouël, F., and Stouls, N. 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* (New York, NY, USA, 2013), PPPJ '13, ACM, pp. 153–158.

[20] Rastogi, V., Dalvi, N., and Garofalakis, M. Large-scale collective entity matching. *Proc. VLDB Endow. 4*, 4 (Jan. 2011), 208–218.

[21] Rose, J. R. Bytecodes Meet Combinators: Invokedynamic on the JVM. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages* (New York, NY, USA, 2009), VMIL '09, ACM, pp. 2:1–2:11.

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

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

[24] Xu, S., Bremner, D., and Heidinga, D. Mining Method Handle Graphs for Efficient Dynamic JVM Languages. In *Proceedings of the Principles and Practices of Programming on The Java Platform* (New York, NY, USA, 2015), PPPJ '15, ACM, pp. 159–169.