RC24827 (W0907-054) July 27, 2009
Computer Science

IBM Research Report

Java Cold Method Refactoring

David Wood
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
USA

Bill Tracey
IBM Software Group
11400 Burnet Road
Austin, TX 78758-3415
USA

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Java Cold Method Refactoring

David Wood
IBM T.J. Watson Research Center
19 Skyline Dr.
Hawthorne, NY 10532
dawood@us.ibm.com

Bill Tracey
IBM Software Group
11400 Burnet Rd.
Austin, TX 78758-3415
wtracey@us.ibm.com

ABSTRACT

We discuss the techniques and benefits of migrating the bodies of loaded, but unused (cold), Java methods to secondary classes which need not be loaded, thus reducing the amount of resources required for Java class loading, storage and verification. The migration is done using binary refactoring of compiled Java classes. We tested two popular applications and showed that under certain conditions we can attain 8% benefit to start up time and almost 7% reduced memory consumption. Benefits were also demonstrated for page faults and working set. We further identify the attributes of class libraries and applications which can affect the benefits of our cold method refactoring.

1. INTRODUCTION

Object oriented programming languages, such as Java, afford the software developer with the ability to group related code together in objects, or classes in the Java language. This is an extremely effective way to design, develop and maintain software; however, it may not always be the most beneficial when considering runtime execution and Java's all-or-nothing class loading semantics. The essence of our work recognizes that although methods may be logically grouped within a given class, they may not be grouped properly with respect to execution ordering or memory layouts. For example, there may be methods that are only called on error conditions or when the program exits. In such cases, it may be desirable to defer or completely avoid the loading of these methods, and more specifically the bytecodes associated with these methods.

The Java Virtual Machine is designed to dynamically load classes, their methods and data, as references are encountered during the course of program execution. Classes make up the primary unit of function within any Java program and are loaded in total, that is, without loading of individual methods. Generally speaking, methods account for the bulk of memory associated with a class and therefore the largest amount of I/O in loading the class. Furthermore, the bytecodes making up the body of each method must be verified regardless of whether or not they are executed. Our experience has shown that as much as 25% of program start up time can be spent in class verification. Therefore to reduce memory and time performance, we would like to identify any opportunities to reduce the size of loaded classes and more specifically, methods. Our effort focuses on achieving this goal through the migration, or refactoring, of methods that are loaded as part of a loaded class, but not used for a given program execution scenario. We refer to such methods as *cold methods*. Our refactoring implementation acted on bytecodes using the JikesBT bytecode toolkit [4].

We used the Eclipse SDK and Tomcat as the benchmarks for evaluation of our techniques. We found that for start up, as much as 50% of the method bytes loaded were not executed. This suggests we might achieve as much as a 12.5% startup time improvement (via reduced verification) and a 50% memory reduction for stored method bytecodes.

2. RELATED WORK

Our work makes use of binary class file refactoring. This technique is now becoming a popular way of weaving new code into existing programs, but also as a technique for implementing program optimizations as covered by Tilevich and Smaragdakis [8].

Much work has been done in the past on software code optimization, and notably a whole domain centered around optimizing object-oriented systems has evolved and [9] provides a valuable survey of techniques in Java. Most techniques focus specifically on execution time [9][10] and in the Java space there are many that address issues of garbage collection performance [11][12]. Our focus has been primarily on Java start up issues and not throughput issues in a steady-state runtime. However, our techniques can be applied to any phase of program execution in which large numbers of new classes are loaded.

Our work is closely related that of Tip et al's [13][15][16] on class hierarchy optimizations for reduced class size. Their work focuses primarily on class hierarchy compression and uses static program analysis to identify unused and closely related classes and unused class members. They are able to demonstrate a considerable reduction in size of an application's class files. While [13] and [15] focused on C++ applications, [16] discusses Java specifically and utilized a number of techniques including method removal to, on average, achieve 51.7% reduction in class file size.

Krintz etl al's [13] work on class file splitting is perhaps the closest to our work. It uses profiling to identify cold methods and fields and then migrates them to secondary classes. Their technique for migration uses inter-class references to link the

warm and cold classes and then changes references to the cold members to reference the secondary class members.

Our work can be differentiated from this past work in a number of ways. In particular, our work:

1. does not remove the unused methods. Instead it leaves the method signatures but reduces their size, thus keeping the original class functionality in tact and allowing non-scenario executions to succeed.
2. adds accessor methods instead of changing the protection levels of class members.
3. focuses specifically on the problem of unused methods,
4. details issues involved in migrating method bodies to secondary (shadow) methods.

3. COLD METHOD REFACTORING

The basics of this technique involve first, identifying the methods that we should consider for refactoring and second, migrating the bodies of the identified methods to methods in secondary classes. Identification of non-executed but loaded methods (i.e. cold methods), can be done in various ways – we used program execution traces. The migration of method bodies is done by creating new methods in secondary classes and changing the original method to call the new method in the secondary class. As long as we are careful, the secondary class will not be loaded until the original method is actually executed. Refactoring can be done at either the source or bytecode level. Our work has focused on bytecode refactoring. The specifics of method identification and method migration are discussed in detail below.

3.1 Identifying Target Methods

The target methods that we need to identify are those that are loaded but which are not executed for a given program execution scenario. Execution scenarios could be simple start up scenarios or more involved with longer running executions.

With the execution scenario defined, we used an execution tracing facility to produce a listing of the amount of time spent executing each loaded method. From this we produced a list of unexecuted methods, including full method signatures, which are then used as input to our refactoring tool. We should note at this point that static analysis methods could also be used to determine a list of cold methods. For now, we leave that as future work.

3.2 Migrating Target Methods

The basic concept in refactoring cold methods is to migrate the body of the method to another method in a secondary class which, if done properly, need not be loaded until the original method is executed. For example, in source code:

```
public class A
   public void foo() {
     <body of foo>
   }
}
```

could become

```
public class A
   public void foo() {
     Secondary.foo(this);
   }
}

public class Secondary {
   public static void foo(A a) {
       <modified body of foo>
   }
}
```

In this simple example, there are a number of considerations to note.

1. A shell class was created to hold the new method.
2. The new method was created – we call this the shadow method.
3. The original body is modified in the new shadow method.
4. The original method is modified to call the new method in the shell class.

There are many details to consider in each of these steps. For example, how often should we create a shell class? Should the shadow method be virtual or static? Do constructors need to be handled any differently? What changes, if any, need to be made to the method body when it is migrated to the shadow method? These and other issues are discussed below.

3.2.1 Serializable Classes

Classes marked as serializable through the java.io.Serializable marker interface should be treated with care. Serializable classes use the `serialVersionUID` field as a version stamp for the class, however the definition by the developer is optional and if need be, the JVM will generate a default value. This default value is overly specific and generated based on information including the set of methods in the class. As such, adding accessor methods will change the default `serialVersionUID` and thus unnecessarily disrupt the serializer/deserializer. For this reason, we avoid any refactoring that would require the addition of accessors to classes that do not implement their own `serialVersionUID`.

3.2.2 Creating the Shell Class

The first thing that must be done is to create a secondary class to serve as a container or shell for the shadow method. One aspect to consider is how often to create a shell class. Should we create a new class for each method? Should we put all of a class's migrated methods into a single class? Or is there perhaps something in between? Creating a new class for each method would provide the greatest granularity and assure that only the method bodies for executed methods are loaded. However, there is a non-zero cost to runtime performance (and on-disk sizes) for each additional class. Each new class adds to the JVM's overhead and increases the amount of I/O required. Creating a single class for all of a class's cold methods avoids this overhead but may force the loading of unneeded shadow methods. Without a clear indication, we defined S as the number of shadow methods per class and tested performance for different values. Results are shown below. Lastly, should the class be created in the same or different package, and should it be a peer or, sub-class of, or a class unrelated to, the original class. The answer to this question is addressed below where we consider member access rights and their impact on method body migration.

3.2.3 Creating the Shadow Method

The shadow method can be created as either static or virtual and the method signature, return value, exceptions and parameters must be set according to the original method's signature. The shadow method is declared to return the same type as the original and to throw the same exceptions. The input arguments are also the same except that the parameter list must include a reference to the instance on which the original method was called. If the shadow method is virtual, then an instance of the shell class would need to be created in order to call the method, which would complicate the migration process and negatively impact execution time and memory. Further, from a CPU performance standpoint, static method invocation is more efficient than that for a virtual method. Finally, the choice of static vs. virtual will effect how the body is modified to attain access to the members of the original method's class and its super classes. Our implementation uses static shadow methods, which is the easiest to implement and is also provides the best run-time throughput.

When the target method is synchronized, we could choose to make the corresponding shadow method synchronized as well, however, this would add a fair amount of overhead for the additional semaphore evaluation. One might consider moving the synchronization from the target to the shadow method; however this would only be valid if a) we could arrange to use the same semaphore in the shell class as in the original target class, or b) we can be assured that the original method's semaphore is used only by that method. Our solution is to simply leave the original method synchronized and to not synchronize the shadow method. This does leave open the door for synchronization violations through calls to the shadow method from other than the target method. This could only be achieved by reflection or other examination of the class structures, and so seems unlikely.

3.2.4 Shadow Method Modifications

The largest portion of the refactoring effort is the attention that must be paid to the modifications required when the original method body is migrated (copied plus modifications) to the shadow method. First, the references to the original instance and method parameters must be maintained. Second, all member accesses must be checked to be sure they are still allowed, and if not, member accessor functions must be created and called in place of the original member accesses[1]. Third, each location that used the `super` operator must be modified to call a new accessor method in the original class that makes the `super` reference. Lastly, special care must be taken when class and instance initializers, <clinit> and <init> respectively, are migrated. Each of these issues is detailed below.

3.2.4.1 Instance and Parameter References

The Java Specification defines an ordering for method arguments and an instance reference (for virtual methods) on a method's incoming stack. For example, the virtual method `foo(int a, int b, int c)` in class A has an incoming stack that looks like the following:

3	c
2	b
1	a
Stack location 0	Reference to instance of A

And if the method is static then the stack is as follows:

2	c
1	b
Stack location 0	a

If we can maintain an equivalent incoming stack in the shadow method we will minimize necessary modifications to the method body that might otherwise require complex flow analysis and bytecode manipulation. For the migration of a static method to a static method, the signatures do not need to change. For a virtual method migrated to a static shadow method, we can achieve an equivalent stack by prepending an instance reference to the shadow method's formal parameter list. For example,

```
public void foo(Type1 a1,…, TypeN aN) {
        <foo body>
}
```

defined in class A, becomes the following in the shell class,

```
public static void foo(A a, Type1 a1,…, TypeN aN) {
        <foo body + modifications>
}
```

The modifications referred to above are discussed in the following sections.

3.2.4.2 Member Access

Class and instance members referenced in the original method may no longer be permitted when copied to the shadow method. For example, where the original method had access to the original class's private members, the shadow method will not. In fact, the bulk of the access problems come with access to members in the class hierarchy of the original class. To address this problem, we create member accessor functions, ideally in the class containing the member, but if that is not possible, then in the class containing the original method. For fields we add getter and setter methods and for methods we add caller methods.

In order to reduce memory impacts and improve execution performance, we would like to minimize the number of new accessor methods required. As we consider this, we should remember that the original method was assumed to compile and so had rights to access members referenced from within it. This means for example, that we won't need to provide access to private or protected members outside the class hierarchy.

The first step to minimizing the number of accessors is to make the shell class a subclass of the original and put it in the same package as the original. This will allow a virtual method access to all non-privates in all super classes, and allow a static method access to all of the original class's non-privates. A static method's access rights to the original class's super class members will depend on whether or not the super class(es) are in the same package or not. For example, an original method making an access to a protected member of a super class outside the same package will require an accessor when migrated to a new static method.

[1] This step is closely related to what the Java compiler does in creating accessors that allow inner classes to reference otherwise inaccessible outer class members.

An implementation consideration is that we may not always have a complete or correct definition for a given class and/or its methods. JikesBT will try and derive a class's members as it encounters them in other classes, but may not be able to determine the access rights. In this case the member is marked as a 'stub'. If we encounter a stub member while analyzing the method body to determine the need for required accessors, we treat the method as private and inaccessible. This makes it fairly important to provide the implementation with a full set of class definitions in order to minimize the number of accessors created. Table 1 summarizes the accessor requirements when the shell class is defined in the same package and as a subclass of the original class.

Table 1 Accessor requirements for secondary class as a subclass in the same package.

Class Accessed By Migrated Code		Accessors Required for New Method	
		Static Implementation	Virtual Implementation
Unrelated Class		None.	None.
Original Class (A)		A's private members	A's private members
Original Class.super()		All super() accesses	All super() accesses.
Super class in same package	**Have Definition**	None.	None.
	Missing Definition (i.e. stub)	None.	None.
Super class in different package	**Have Definition**	All protected accesses.	None.
	Missing Definition (i.e. stub)	All accesses.	None.

When the original class is declared as `final`, however, we can not define the secondary class as a subclass. In this case, the access rights are similar for static methods and only change for the virtual methods, which now need accessors for protected members in other packages. Table 2 summarizes the accessor requirements when the shell class is defined in the same package, but NOT as a subclass of the original class.

Table 2. Accessor requirements for secondary class as a non-subclass in the same package.

Class Accessed By Migrated Code		Accessors Required for New Method	
		Static Implementation	Virtual Implementation
Unrelated Class		None.	None.
Original Class (A)		A's private members	A's private members
Original Class.super()		All super() accesses	All super() accesses.
Super class in same package	**Have Definition**	None.	None.
	Missing Definition	None.	None.
Super class in different package	**Have Definition**	All protected accesses.	All protected accesses.
	Missing Definition	All accesses.	All accesses.

We can see that implementing the shadow method as a virtual method affords a slight advantage with respect to the need for accessors in the case of non-final classes which have super classes outside of their own package.

Accessor definition is fairly straight forward and we won't go into detail except to define, in general, the basic methods. Any implementation will need to properly handle the input arguments and return values based on the type sizes.

Table 3. Generalized accessor methods

Inaccessible Members of A	Generated Accessor in A

void foo(T1 a1…TN aN)	`public static void call$foo(A a, T1 a1,… TN aN) {`
	`  aload0`
	`  aload1`
	`  ….`
	`  aload N`
	`  invokevirtual`
	`  A.foo(T1 a1,…TN aN)`
	`}`

static void foo(T1 a1…TN aN)	`public static void call$foo(T1 a1,… TN aN) {`
	`  aload0`
	`  aload1`
	`  ….`
	`  aload N-1`
	`  invokestatic`
	`  A.foo(T1 a1,…TN aN)`
	`}`

Field f of type T	`public static void set$f(A a, T val) {`
	`  aload0`
	`  aload1`
	`  putfield A.f`
	`}`
	
	`public static T get$f(A a) {`
	`  aload0`
	`  getfield A.f`
	`  areturn`
	`}`

Static field f of type T	`public static void set$f(T val)`
	`{`
	`  aload0`
	`  putstatic A.f`
	`}`
	
	`public static T get$f() {`
	`  getstatic A.f`
	`  areturn`
	`}`

Note that these accessors could have been defined as virtual, but again given performance advantages of the *invokestatic* bytecode, we chose static accessor implementations.

It turns out that replacing the invoke and field accessor bytecodes with their accessor methods is a good match, because each accessor requires the same incoming stack as the bytecode it replaces. Specifically,

`putfield A.foo` → `invokestatic A.put$foo(A a, T v)`

```
getfield A.foo    → invokestatic A.get$foo(A a)
putstatic A.bar   → invokestatic A.put$bar(T v)
getstatic A.bar   → invokestatic A.get$bar()
invokevirtual A.foo(…)   →
                    invokestatic A.call$foo(A a, …)
invokestatic A.bar(…)    →
                    invokestatic A.call$bar(…)
```

3.2.4.3 Super() Method References

References made using the super() operator in the original method, must also be modified in the shadow method. First, super() references are identified when the *invokespecial* bytecode is used to invoke a method that is neither private nor an instance initializer. Once encountered, our approach was to create 'super accessor' functions in the original class which simply uses the *invokespecial* bytecode on the same method. The super accessor looks as follows and must always be defined within the original class:

```
public TypeR super$A$foo(Type1 a1…TypeN aN) {
       aload0
       aload1
       …
       aload N
       invokespecial A.foo(Type1 a1…TypeN aN)
       areturn
}
```

It has the exact signature of the original method called and is always virtual, but must be unique within the class hierarchy to avoid infinite indirect recursion. Our approach to uniqueness was to include the class name in the method name. Finally, if any exceptions are thrown by the target method, then those must be declared for the super accessor as well. Again as with the non-super member accessors the input stack requirements are the same for both the direct invocation to the super method and it's accessors as follows:

```
invokespecial A.foo(…)  → invokevirtual A.super$foo(…)
```

3.2.5 Creating the New Method Body

To create the new method body, the primary task is in arranging to call the shadow method correctly. We consider the three cases of a static method, a virtual method and a class or instance initializer.

3.2.5.1 Static Methods

The static method is perhaps the most straightforward. It takes no instance reference and only a set of arguments, as follows:

```
public class A {
  static TypeR foo(Type1 a1…TypeN aN) {
        <migratable body>
  }
  …
}
```

which becomes the following,

```
public class A {
  static TypeR foo(Type1 a1…TypeN aN) {
        aload0
        aload1
        …
        aload N-1
        invokestatic A$Cold.foo(Type1 a1…TypeN aN)
        areturn
  }
  …
}
```

where A$Cold.foo() was created as the shadow method for A.foo().

3.2.5.2 Virtual Methods

The virtual method is only somewhat more complex. It adds the instance reference as a parameter along with the set of declared arguments. as follows:

```
public class A {
  TypeR foo(Type1 a1…TypeN aN) {
        <migratable body>
  }
…
}
```

which becomes the following,

```
public class A {
  TypeR foo(Type1 arg1…TypeN argN) {
        aload0
        aload1
        …
        aload N-1
        invokestatic A$Cold.foo(A a, Type1 a1…TypeN aN)
        areturn
  }
  …
}
```

where again, A$Cold.foo() was created as the shadow method for A.foo().

3.2.5.3 Class and Instance Initializers

Our approach supports the refactoring of both class initializers (<clinit>) and instance initializers (<init>), however there are two additional considerations.

First, if a class initializer is loaded but not executed, it means 1) that the class was loaded for verification purposes only, and 2) that none of its other methods were executed either. Our results show that about 10% of classes (by count) are affected with this issue. Given that this is really a verification issue and that it is a relatively small problem, we chose to defer this problem to a separate solution not covered in this paper. When this situation is encountered our algoirithm will avoid refactoring any methods in the class.

Secondly, only initializers are allowed to set the values of fields declared as final. This means that we can not migrate initialization of final fields into the shadow method. Table 4 shows the percentages of initializers setting final fields.

Table 4. Initializer methods setting final fields

	Initializers settting final fields		**Initializer bytes as a percentage of all methods**	
	Eclipse	Tomcat	Eclipse	Tomcat
<clinit>	60%	44%	1.6%	2.3%
<init>	36%	13%	2.0%	1.0%

As might be expected, the percentages for class initializers are larger than for instance initializers. Given the relatively small realizable opportunity available through the migration of instance initializers (less than 1% by size), we decided to forgo migration of instance initializers containing final field initializations. Note that an implementation could attempt migration of all initializations done in instance initializers to the class initializer, but care would have to be taken to be sure values weren't being set differently depending upon which instance initializer was being called.

Lastly, we must consider that instance initializers always include the use of the `super()` operator to invoke the super class's initializer (except for `Object`). A problem arises here in that the super class initializer may not be visible to the shadow method. Normally we would simply use the super accessor strategy discussed in 3.2.4.3. However, we can not use this strategy because initializers are not allowed to call methods (i.e. the shadow method) until super classes have been fully initialized. Given that `super()` is always used within instance initializers, we realized we had to address this complexity. Our solution was to treat only the bytecodes after the super initializer invocation as refactorable, resulting in a partial migration of the initializer body. So for example:

```
public class A extends Object
  public void <init>() {
        invokespecial  Object.<init>
        <migratable body>
  }
}
```

becomes,

```
public class A extends Object
  public void <init>() {
    invokespecial  Object.<init>
    aload0
    invokestatic A$Cold.init(A a)
  }
}
```

with the shadow method as follows:

```
public class A$Cold extends A
  public static void init(A a) {
        <migratable body + accessors>
  }
}
```

```
putfield A.foo  →
       invokestatic A$Cold.put$foo(Type value)

getfield A.foo  →
       invokestatic A$Cold.get$foo()

putstatic A.bar →
       invokestatic A$Cold.put$bar(Type value)

getstatic A.bar →
       invokestatic A$Cold.get$bar()

invokevirtual A.foo(…) →
       invokestatic A$cold.foo(A a, …)

invokestatic A.bar(…) →
       invokestatic A$Cold.bar(…)
```

3.2.6 JVM Dependencies

Finally, there are a number of classes about which the JVM may have specific expectations. This is a particular issue when refactoring the JDK (classes.zip, rt.jar, etc). Although we won't show results for refactored JDKs, our tool was enabled with the ability to define classes and packages which are to be avoid when considering either a) refactoring or b) the addition of accessor methods.

4. CONTROL PARAMETERS

With the framework in placed to produce the cold method refactorings based on a set of identified target cold methods, we find the need for the ability to control which methods are actually selected for refactoring. Recall that one of our primary goals is to reduce the size of the classes that are actually loaded and to increase the percentage of methods (actually bytecodes) that are executed for a chosen execution scenario. At the same time we need to be careful to not create too many shell classes, as these may eventually be loaded and lead to larger overall memory requirements and perhaps reduced CPU performance. To enable some control over these attributes, we provided a number of control parameters as follows:

MinMethodBytes – threshold method size required before a method will be considered for refactoring

MinClassBytes - minimum number of targeted method bytes within a class before the class's methods will be considered for refactoring.

ShellMethods – maximum number of shadow methods per shell class.

ShellMethodBytes – maximum size of the shell class in method bytes

Altering MinMethodBytes allows us to make sure we're not refactoring methods which are already smaller than the resulting method body replacement. Similarly, altering MinClassBytes will enable us to limit the cost of adding a class to cases where the class has considerable opportunity for savings. These two are important for basic refactoring and will have the largest impact on initial class loading. ShellMethods allows us to limit the number of methods per shell class. A lower value may be useful since it will limit the amount of unnecessarily loaded code when one of the cold methods is executed. ShellMethodBytes may be useful in trying to match class sizes with operating system or JVM page sizes. These two parameters control behaviors once a cold method is referenced and so will impact out-of-scenario usage. The next section looks only at the results on varying the values of MinMethodBytes and MinClassBytes.

5. RESULTS

We would like to understand the impact of cold method refactoring on initial application start up. As such, we have limited our examination to the impact of MinMethodSize and MinClassSize on both start up time and memory. To understand the impact we chose two popular java applications as benchmarks:

Eclipse Java SDK 3.1.2 – this is an unmodified version of the Eclipse 3.1.2 SDK from eclipse.org. We opened an empty workspace to the default perspective as our scenario. We used the eclipse.starttime property in debug mode to capture startup times.

Apache Tomcat 5.0.28 – this is a combined HTTP and Servlet engine from Apache.org. Our scenario was to start the server as is configured in the initial download. We made a small modification to capture start up time that is more representative of time since invocation.

Memory statistics were collected using *PsList* from Microsoft[19]. In neither case was the JDK itself refactored.

5.1 Application Characterization

To understand the nature of the classes making up these applications and the nature of any refactoring impact, we need to understand the relative amounts of warm and cold method code. Eclipse consists of 8.65MB of total class methods, while Tomcat consists of 2.39M. Figure 1 shows the breakdown of the loading

of these methods for our start up scenario. Perhaps surprisingly, cold methods are as common as warm methods.

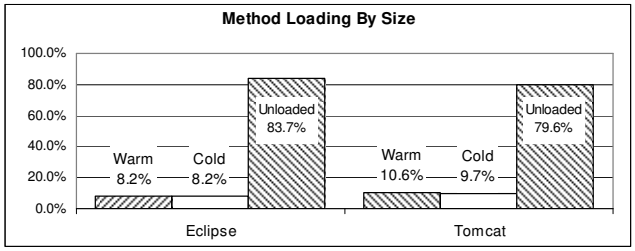

Figure 1. Method usage

Eclipse shows the greatest percentage of cold methods, but both are close to 50% of total method bytes loaded. Figures 2 and 3 show the relative size distributions of the loaded methods.

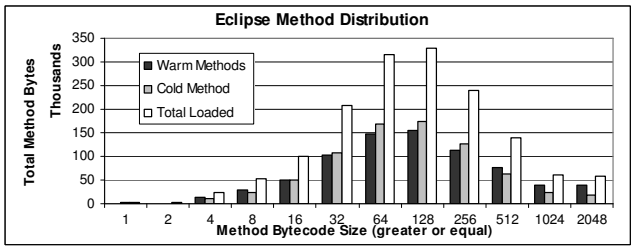

Figure 2. Eclipse loaded method distribution

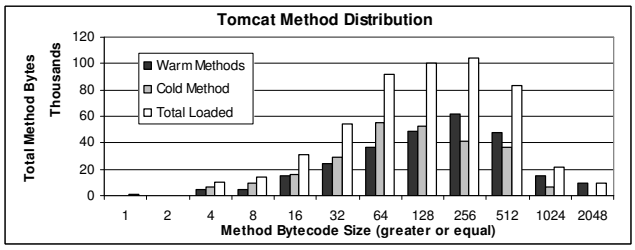

Figure 3. Tomcat loaded method distribution

The distributions are fairly similar; however we can see that Eclipse has a lower average method size. Also, while Eclipse's cold and warm methods have an equivalent distribution, we can see that Tomcat's warm methods are generally larger than its cold methods and may help to account for the lower ratio of cold method bytes to warm method bytes for Tomcat.

5.2 Class Object Impact

We looked at the refactoring results with both MinMethodBytes and MinClassBytes set to 0 to remove any concomitant limitations on refactoring. With these settings, we were able to successfully refactor 83.5% of the Eclipse cold methods and 82% of Tomcat methods (by size). The primary factor accounting for not attaining higher success rates is the occurrence of serializable classes that do not define their own `serialVersionUID`. We avoid refactoring these classes because adding accessor methods, which is generally required, would change the default `serialVersionUID` thus causing serialization issues with the resulting code.

Next, we consider the impact on cold methods sizes as we vary the value of MinMethodBytes while holding MinClassBytes at the value of 0. The values at ∞ represent the non-refactored case.

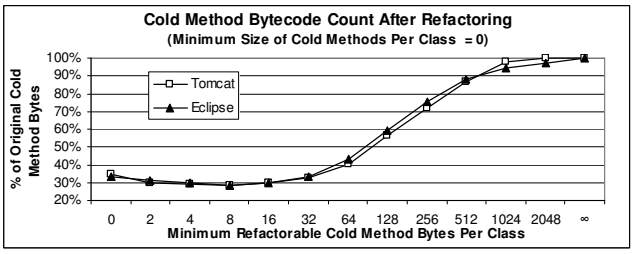

Figure 4. Total cold method size vs. minimum refactorable method size

Here we clearly see that a value of 8 for MinMethodBytes achieves the greatest reduction in cold method bytes, just over 71%, for both Eclipse and Tomcat. Note that for values smaller than 8 we are actually creating more cold method bytes by replacing small methods bodies with larger ones to call the shadow methods.

Next, we consider the impact on cold method sizes as we vary the value of MinClassBytes while holding MinMethodBytes at the optimal value of 8[2].

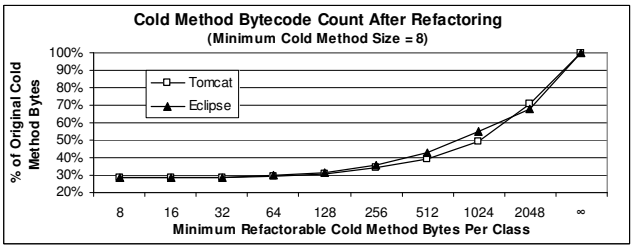

Figure 5. Percent change in cold methods bytecodes

Figure 5 shows the resulting cold method bytes as percentages of the original. We can see that in both cases we are able to reduce the size to about 29% of the initial size – representing a reduction from 706Kb to 202Kb for Eclipse and from 232Kb to 67Kb for Tomcat. The total size of all methods is increased by only about 1% (99Kb for Eclipse and 26Kb for Tomcat). This increase is due primarily to the addition of the accessor methods, which are also cold since they are created only to be called from the original set of cold methods.

5.3 Performance Results

Our runtime performance results were acquired on the following platform:

CPU: Pentium 4, 2.2 GHz

Memory: 1GB

Disk Drive: 60GB SCSI

OS : Windows XP SP2

JVM: Sun 1.5.07 and IBM J9 2.3

[2] There should be no change in cold method bytes for values of MinClassBytes that are less than MinMethodBytes, since selected methods are already restricted to be larger than the total class method bytes.

Benchmarks were run consecutively resetting disk and memory caches between runs and so should be considered cold started. Our measurements were designed to identify the optimal value of MinClassBytes, with MinMethodSize set to 8 per our earlier analysis. Runs of each application and each JVM were done for various values of MinClassBytes. Start up timing results for both Eclipse and Tomcat running on both Sun and IBM JVMs are shown below.

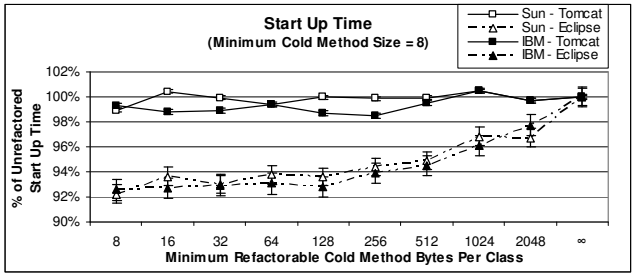

Figure 7. Start up time

We can see that the Eclipse start up time can be reduced by over 7%, with significant loss of impact for values greater than 256. However, the benefit for Tomcat is quite a bit smaller at roughly 1%. Both JVMs are roughly equivalent in both cases.

Next we show the memory impact measured in "peak private bytes" as reported by the *pslist* utility [19].

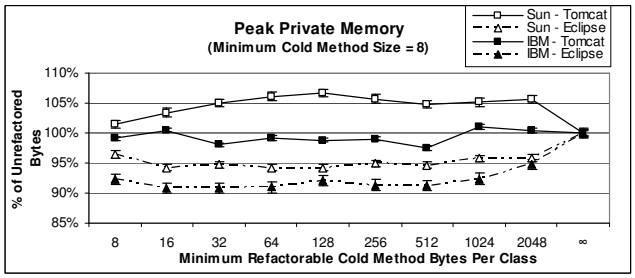

Figure 8. Memory

We see a significant impact for Eclipse with a reduction in memory requirements by about 6-7%, just fewer than 3MB. Again, Tomcat does not see the same benefit.

We believe the difference in results between Tomcat and Eclipse is due primarily to the difference in the rate of class bytes loaded during the measured startup time.

The rate of class byte loading is a good indication of the relative amount of time being spent loading classes, and thus the total opportunity afforded by reducing cold method sizes. We find that Eclipse loads roughly 195Kb/sec during its start up time and Tomcat only 88.4Kb/sec. This factor of over 2 will have an impact on the expected benefit of our refactoring. Note that although Eclipse loads a larger number of classes, it starts up in roughly 7.2 seconds compared to 5.5 seconds for Tomcat, so although it does twice the rate of class loading it does not take twice as long to start up. In general, these numbers suggest that Tomcat does more non-class-loading work before being completely started, and therefore realizes less relative benefit over that time from cold method refactoring.

Instead of measuring the class loading rate number directly, a good proxy for this measurement maybe the speed up achieved by running the target application with and without the Java class verifier enabled (using the –noverify option). Applications that experience a better speed up without verification use a greater percentage of their start up time loading and verifying classes and as such are more greatly impacted by our refactoring technique. In accordance with this idea is our measurement of a 26% speed up on Eclipse and only a 4.7% speed up on Tomcat when running without class verification.

6. SUMMARY AND CONCLUSIONS

We have shown that a sizable portion of the methods that are loaded for start up execution scenarios of two well-known and popular Java applications, Eclipse and Tomcat, are not actually executed. Both applications demonstrated that the cold methods account for about 50% of the total method bytes loaded during start up. Based on this and the fact that applications can spend as much as 25% of their start up time loading and verifying classes, we postulated the ability to reduce start up time by as much as 12.5% if the application could be made to load only the warm (executed) methods.

To try and achieve this we implemented a bytecode refactoring algorithm that migrated a set of known cold method bodies to secondary, or shell, classes. The original methods were left in place but their bodies were modified to call the new (shadow) methods contained within the shell classes, thereby greatly reducing the size of the original cold methods. To fully enable the migration, method and field accessor methods were added as needed to the original class in order to provide the shadow method with access to any inaccessible fields or methods. The shadow method bodies were modified as necessary to call these accessor methods.

We characterized the set of methods from each application. Eclipse consisted of a total of 1.4MB of loaded method bytecodes 50% of which were not executed, while Tomcat loaded 486KB of which 48% were not executed. We found that the algorithm should never refactor cold methods that were smaller then 8 bytes; otherwise we would actually increase the overall size of cold method bytes. We demonstrated that our algorithm was able to reduce the amount of cold method bytes, including added accessor methods, to about 29% of the original for both Eclipse and Tomcat. This means that we shifted the ratio of warm methods to cold methods from 1:1 to 4:1 (about 80% of the bytecodes loaded are executed instead of only half prior to refactoring).

We showed that the refactoring could be tuned based on the minimum cold method bytes within a class and that benefits begin to drop off at a minimum size of 128 bytes.

Our performance analysis using both the Sun 1.5.07 and IBM J9 2.3 JVMs showed that there were both time and memory benefits when applying the refactoring. The greatest benefit was seen for Eclipse with 7% reduction in start up time and almost a 7% reduction in memory. Tomcat showed minimal improvements for both time and memory and even a degradation of memory performance under the Sun JVM.

We believe the difference in performance benefits is due to the relative amount of class loading that occurs during the measured start up time. An indicator for how much application performance

might benefit is the size of the start up time reduction when running without class verification. For values near 26%, which was the case of Eclipse, one can expect good results from refactoring. Tomcat only saw a 4.7% reduction in start up time with verification turned off and so was not able to reap the same benefit from refactoring

In summary, our work has demonstrated that for the right application a fair performance benefit can be attained by refactoring for cold methods.

7. REFERENCES

[1] Lindholm, Tim and Yellin, Frank. The Java Virtual Machine Specification, Second Edition 1999.

[2] Sun Microsystems Inc. (2004), Java Object Serialization Specification

[3] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996

[4] Jikes Bytecode Toolkit - http://w3.alphaworks.ibm.com/tech/alphabrief/jikesbt

[5] X. Leroy. Java bytecode verification: an overview. G. Berry, H. Comon, and A. Finkel, editors, Proceedings of CAV'01, number 2102 in LNCS, pages 265-285. Springer, 2001

[6] Tilevich, E, Smaragdakis, Y. Binary Refactoring: Improving Code Behind the Scenes, ICSE'05

[7] BACON, D. F., FINK, S. J., AND GROVE, D. Space- and time-efficient implementation of the Java object model. European Conference on Object-Oriented Programming (June 2002), B. Magnusson, Ed., vol. 2374 of Lecture Notes in Computer Science, pp. 111--132.

[8] T. Mens and T. Tourwe, "A Survey of Software Refactoring",IEEE Trans. on Software Engineering 30(2): 126-139 (2004).

[9] Kazi, I. H., Chen, H. H., Stanley, B., and Lilja, D. J. 2000. Techniques for obtaining high performance in Java programs. *ACM Comput. Surv.* 32, 3 (Sep. 2000), 213-240.

[10] Nathaniel Nystrom. Bytecode-level analysis and optimization of Java class files. Master's thesis, Purdue University, West Lafayette, IN, May 1998

[11] BRECHT, T., ARJOMANDI, E., LI, C., AND PHAM, H. Controlling garbage collection and heap growth to reduce execution time of Java applications. In ACM Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'01) (Nov. 2001)

[12] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? *High Performance Garbage Collection in Java with MMTk*. In ICSE 2004.

[13] C. Krintz, B. Calder, and U. Holzle. *Reducing Transfer Delay Using Java Class File Splitting and Prefetching*. In ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), November 1999.

[14] Frank Tip and Peter F. Sweeney. Class hierarchy specialization. In ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, 1997.

[15] Peter F. Sweeney and Frank Tip. A study of dead data members in C++ applications. In SIGPLAN '98 Conference on Programming Language Design and Implementation, pages 324-332, June 1998.

[16] Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical Experience with an Application Extractor for Java. IBM Research Report RC 21451, IBM Research, 1999

[17] W. Pugh. Compressing Java class files. In Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation, May 1999.

[18] E. Sirer, A. Gregory, and B. Bershad. A practical approach for improving startup latency in Java applications. In Workshop on Compiler Support for Systems Software, May 1999.

[19] PsList Documentation - http://www.microsoft.com/technet/sysinternals/utilities/PsList.mspx