Dynamic Class Hierarchy Mutation

Lixin Su and Mikko H. Lipasti

Department of Electrical and Computer Engineering
University of Wisconsin-Madison
lsu@cae.wisc.edu, mikko@ece.wisc.edu

Abstract

Class hierarchies in object-oriented programs are used to capture various attributes of the underlying objects they represent, allowing programmers to encapsulate common attributes in base classes while distributing private attributes in lower-level derived classes. In essence, the semantics of the class hierarchy elegantly capture some of the possible states that a particular object can assume. However, class hierarchies are often poorly designed or evolve in ways that fail to fully capture the stateful behavior of objects. This paper proposes an automated approach for detecting stateful class attributes, and then mutating the class hierarchy dynamically to capture such behavior by creating implicit derived classes that can be specialized for specific object states. Our scheme captures both run-time static behavior, which could have been captured by the programmer by restructuring the class hierarchy at the source level; as well as run-time variant behavior, which cannot be captured using source code transformations. In the latter case, objects transition from one state to another and are dynamically mutated from a derived class to a peer derived class corresponding to the object's new state. These class hierarchy mutations create new opportunities for conventional optimizations such as constant propagation, function specialization, and dead code elimination. For our benchmark set, which includes two versions of SPECjbb, we measure speedups of 1.9% to 31.4% within our Jikes-based prototype implementation, with negligible increases in compilation overhead and object code size.

1. Introduction and Motivation

Modern object-oriented programming languages like C++ and Java support the notion of class hierarchies. Class hierarchies are a powerful programming abstraction that allow object-oriented programs to encapsulate common attributes and operations in base classes, while deriving more specialized classes from those base classes through a process called inheritance. Figure 1 shows an example class hierarchy for animals in a zoo, including two object instances Ling Ling and Quinn, a panda and polar bear whose attributes are stored in the Panda and Polar classes (this example is adapted from [22]). These derived classes inherit attributes from multiple base classes: ZooAnimal, Bear, Herbivore, and Endangered (in the case of Ling Ling). This example demonstrates how a class hierarchy can be organized to encapsulate subtype information in derived classes, while maintaining common attributes in higher-level classes. This type of inheritance-based class design has numerous benefits in terms of structuring code appropriately, avoiding duplication of common functionality, amenability to unit-level testing and validation, code reuse, maintainability, etc., and has been widely adopted in the programming community.

However, even well-written object-oriented programs suffer from incomplete or inadequate class hierarchies, since most programs evolve over time and are modified by multiple programmers. Hence, the benefits of fully encapsulating an object's state by its implicit location in the class hierarchy are often not captured in real programs. Furthermore, a static encoding expressed in the class hierarchy is not able to adapt to run-time behavior. For example, a zookeeper may be interested in tracking whether or not objects of class Polar are in a hungry state, which would likely vary by time of day depending on their feeding schedule. In this case, each Polar instance could have a state field that would record the current state of hunger of that polar bear, and use the value of that state variable to determine whether or not trainers or caretakers were allowed to enter the polar bear cage. Ideally, we would like to encode this state in the class hierarchy by adding a derived class called Hungry Polar Bear (shaded in Figure 1),so that existing methods for resolving state-based variations in behavior could be employed to handle this scenario as well. For example, the routine controlling the cage door lock would just invoke an open method against the resident of the cage, and the method would be overloaded to open the door for non-hungry polar bears, and refuse to open the door for hungry polar bears; the virtual function table lookup would seamlessly invoke one or the other operation, depending on which derived class the object belonged to. However, conventional object-oriented programming languages do not allow run-time reassignment of an object from one derived class to another. Doing so would require the class hierarchy to dynamically mutate or metamorphose to accommodate such changes.

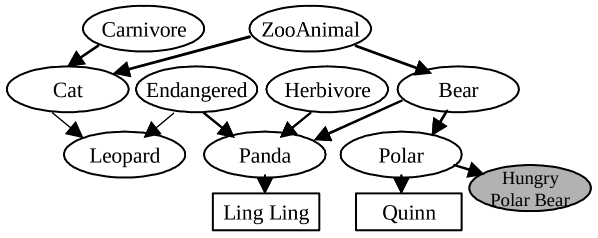


Figure 1. Example Class Hierarchy (adapted from [22]).

In fact, recent work by Maurer [24] showed that precisely this type of metamorphic programming—where the program structure itself adapts to changes in object state—can lead to dramatic speedups for applications where behavior is dependent on object state. However, lack of any language-level support for such metamorphosis makes it extremely difficult to realize this performance benefit in a manner amenable to modern software engineering practices. To the best of our knowledge, this paper is the first attempt to transparently provide some of the benefits of metamorphic programming in the context of a modern high-level language. We do so by leveraging the existing mechanisms that are in place to support class hierarchies, while dynamically mutating the program in response to changes in object state, and to attempt to transparently extract performance benefit from such mutation.

Specifically, we have implemented a prototype of a fully-automatable system that, first, identifies likely candidates for class hierarchy mutation via source code analysis and profiling that identifies hot states for object attributes that control dynamic behavior. Second, our system dynamically mutates the class hierarchy to create new derived classes that are specific to these hot states. Finally, our system employs existing support for aggressive optimizations like constant propagation, function specialization, and dead code elimination, all in the context of a robust java virtual machine—Jikes—that is capable of running complex, multithreaded applications like SPECjbb. Our results indicate that several of the existing benchmarks that we studied contain class hierarchies that are prime candidates for dynamic mutation, resulting in speedups ranging from 31.4% for a microbenchmark (SalaryDB) that is very amenable to class hierarchy mutation, to 4.5% and 1.9% for the full-blown SPECjbb2000 and SPECjbb2005 benchmarks.

The remainder of the paper is structured as follows: Section 2 shows how dynamic class hierarchy mutation works; Section 3 presents the details of the implementation; Section 4 introduces the concepts of object life time constants; Section 5 explains how specialization inlining works; Section 6 describes the experimental methodology; Section 7 presents empirical study results; Section 8 conducts a survey of related work; and Section 9 concludes the paper and discusses future work.

2. Dynamic Class Hierarchy Mutation

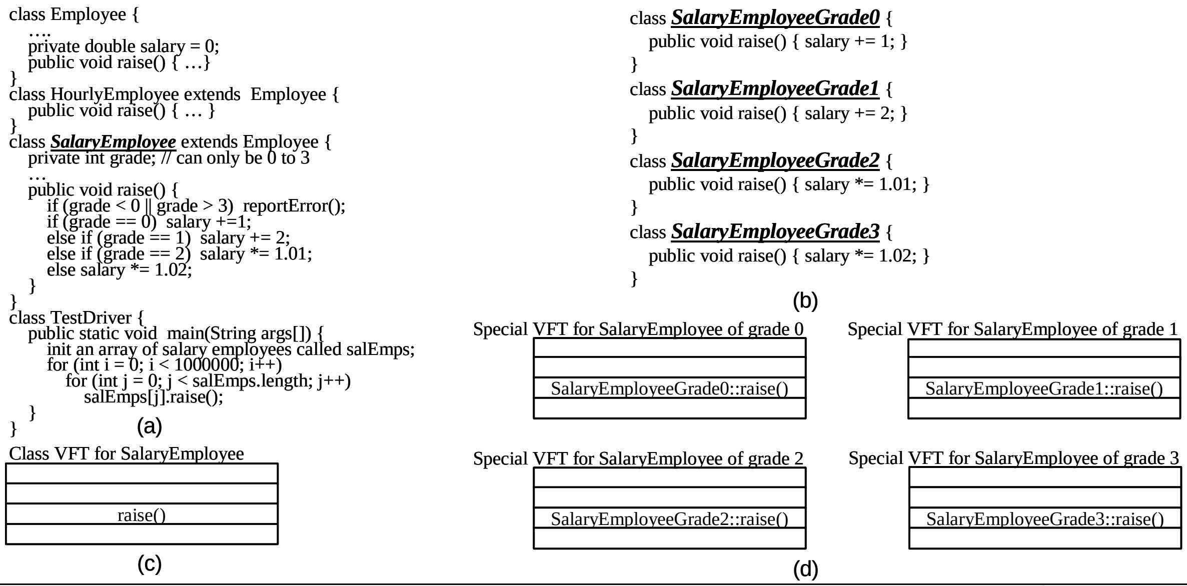
This section describes how class hierarchy is dynamically mutated at runtime and then gives an example to illustrate it.

2.1. General Approach and Concepts

A class hierarchy can be dynamically altered by mutating certain classes in the hierarchy (in the rest of the paper, class hierarchy mutation is shortened as class mutation.). In order to receive performance benefits, these classes need to have mutable behavior and consume a significant portion of the computation time. Such classes are mutable classes. Mutable classes exhibit their mutable behavior through mutable methods. Mutable methods are methods that behave differently depending on the mutation state of their receiver object at runtime and they also consume heavy computation time. A class's mutation state is decided by the values of static and instance fields that this class uses. These fields are called state fields. Class mutation is implemented at runtime by generating special virtual function tables (VFTs). Each special VFT corresponds to a dynamically mutated class. A special VFT can contain pointers to the compiled code to mutated (specialized) methods. An object in a mutation state can be viewed as being instantiated from a dynamically mutated class. The invocation of its mutable methods is via the special VFT that corresponds to the object's runtime state.

2.2. An Example

In Figure 2, a simple program, SalaryDB, is shown to illustrate how class mutation works. It creates an employee database and then iterates through the employees to raise their salaries. The Employee class and its subclasses have



```
class Employee {
    ….
    private double salary = 0;
    public void raise() { …}
}
class HourlyEmployee extends  Employee {
    public void raise() { … }
}
class SalaryEmployee extends Employee {
    private int grade; // can only be 0 to 3
    …
    public void raise() {
        if (grade < 0 || grade > 3)  reportError();
        if (grade == 0)  salary +=1;
        else if (grade == 1)  salary += 2;
        else if (grade == 2)  salary *= 1.01;
        else salary *= 1.02;
    }
}
class TestDriver {
    public static void  main(String args[]) {
        init an array of salary employees called salEmps;
        for (int i = 0; i < 1000000; i++)
            for (int j = 0; j < salEmps.length; j++)
                salEmps[j].raise();
    }
}
```

(a)

```
class SalaryEmployeeGrade0 {
    public void raise() { salary += 1; }
}
class SalaryEmployeeGrade1 {
    public void raise() { salary += 2; }
}
class SalaryEmployeeGrade2 {
    public void raise() { salary *= 1.01; }
}
class SalaryEmployeeGrade3 {
    public void raise() { salary *= 1.02; }
}
```

(b)

Class VFT for SalaryEmployee
raise()
(c)
Special VFT for SalaryEmployee of grade 0
SalaryEmployeeGrade0::raise()
Special VFT for SalaryEmployee of grade 1
SalaryEmployeeGrade1::raise()
Special VFT for SalaryEmployee of grade 2
SalaryEmployeeGrade2::raise()
Special VFT for SalaryEmployee of grade 3
SalaryEmployeeGrade3::raise()
(d)

Figure 2. An example - SalaryDB. (a) is the original source code. (b) shows the mutated versions of the mutable class - SalaryEmployee. (c) is the virtual function table (VFT) for SalaryEmployee. (d) shows the special VFTs that can be dynamically generated**.**

multiple methods but only the raise() method is shown. SalaryEmployee is at the end of the class hierarchy based on Employee. SalaryEmployee is a mutable class that depends on a state field - grade. A salary employee can have a grade varying from 0 to 3. These are the hot values of grade. The raise() method in SalaryEmployee is a mutable method. Based on the value of grade, a SalaryEmployee object can have four distinct hot (mutation) states. In each hot state, the raise() method behaves differently. Each hot state corresponds to a dynamically mutated class based on SalaryEmployee with grade being set to a known value. We use virtual function tables (VFTs) to achieve this dynamic class mutation. Special VFTs are copied from the class (general) VFT. Each special VFT corresponds to a dynamically mutated class. The function pointers of mutable functions in a special VFT can be set to code specialized according to the specific state the dynamically mutated class is in. In this case, the mutable functions can be compiled with grade specialized to 0, 1, 2, or 3. An object can change its VFT pointer to a special VFT according to the special state it is in. When raise() is executed using dynamic binding, the object VFT pointer is located and then its method offset is used to find the pointer to the compiled code for this method. With the object VFT pointer pointing to a special VFT, the specialized raise() code can be invoked without any extra overhead. Later on, the object VFT pointer can be reset to another special VFT if the state of the object changes. In this case, the salary employee's grade changes, e.g. the salary employee gets promoted. No value guarding is needed for the specialized code generated for each version of a mutable method. There can be thousands of SalaryEmployee objects and all the objects share only four hot states and can be viewed as being instantiated from one of the four dynamically mutated classes based on SalaryEmployee. The dynamic class hierarchy can be viewed as three original classes (Employee, HourlyEmployee, and SalaryEmployee) plus any combination of the four mutated classes (SalaryEmployeeGrade[0...3]).

More specifically, when a SalaryEmployee object is initiated, a value, e.g. 0, is assigned to grade and the state of this object is known. The object can be viewed as being instantiated from a dynamically mutated class - SalaryEmployeeGrade0 (The current dynamic class hierarchy includes three original classes plus SalaryEmployeeGrade0.). All executions of the raise() method are automatically directed to SalaryEmployee0::raise(), which is specialized according to the state the dynamically mutated class is in. When the state of the object changes (grade is reset to 1), the object VFT pointer is reset to the special VFT matching the dynamically mutated class - SalaryEmployeeGrade1 (The current dynamic class hierarchy changes to three original classes plus SalaryEmployeeGrade1).

3. Implementation

This section presents an implementation that can identify the hot states of runtime objects and apply such hot states to dynamically mutate classes. The implementation includes two major steps. First, the hot states of objects are identified via profiling and static analysis. An object state is represented by the values of a combination of static and instance (non-static) state fields. The fields can be declared by a class itself or a class's parent classes. In the second step, the information acquired in step 1 is fed into a Java

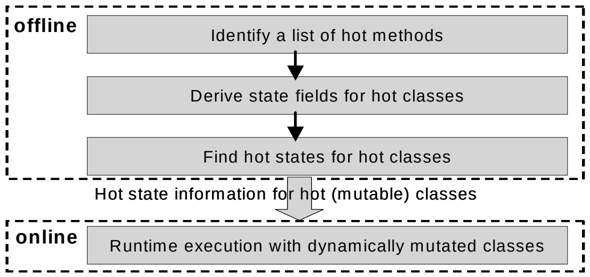

offline
Identify a list of hot methods
Derive state fields for hot classes
Find hot states for hot classes
Hot state information for hot (mutable) classes
online
Runtime execution with dynamically mutated classes

Figure 3. An overview of our implementation.

Virtual Machine at the startup of the JVM. At runtime mutated classes can be dynamically created to match the hot states of hot objects. The methodology is shown in Figure 3.

3.1. Offline Profiling and Static Analysis

In our initial prototype, the information about hot states of hot objects are obtained using static profiling and analysis. Three steps, as shown in Figure 3, are performed to derive the hot state information for hot classes. In future work, we will implement on-line value profilers similar to [4] or [9] to further investigate the feasibility of dynamic profiling and analysis.

A list of hot functions for a program is generated using the Intel Vtune performance analyzer[19]. Vtune gives detailed information about the call frequency and execution times of each function. Then, JikesTM, an open source static Java compiler from IBM, is modified to perform simple static analysis to find state fields (fields that can affect the state of an object) for a hot class. In order to find state fields that can affect a program's performance, the following assumptions are made. First, a field affecting an object's state tends to be used in branches since the value of the field affects the state that an object is in and the tasks that an object needs to perform. Second, the field needs to be used in a hot function to have a performance impact. Last, the assignment of the field should occur in a cold function. Otherwise, there will be no opportunity exploring the benefit of knowing an object's state if the field changes its value frequently. The last assumption can be relatively relaxed if it is known that a field is always assigned to the same value in a hot function. Therefore, the importance of a field that can potentially affect an object's state is characterized by (EQ 1), where L_i is the loop nest-

$$V = \sum^{n} L_i H_i - R \sum^{m} l_i h_i \qquad \text{(EQ1)}$$

ing level of a branch that uses this field, H_i is the hotness of the function where the use of this field occurs, l_i is the loop nesting level of an instruction that assigns this field, h_i is the hotness of the function where the assignment occurs, R is a tunable number. The higher V is, the more likely a field can have an impact affecting an object's state and affecting a program's performance. Only fields that affect the states of hot classes, which contain hot methods, are of concern since they are more likely to affect the program's performance.

The above profiling and analysis lead to a set of state fields that affect the states of hot objects. However, the number of states and the hotness of each state for all hot classes are still unknown. In order to obtain the state field values and the hotness of each state, the Jikes RVM is augmented to generate the possible values for each field and the distribution of the values of a field over time. This can be easily done in Jikes RVM and its performance impact is not a concern since it is done offline. Sampling code is inserted to get the values of the state fields for hot classes. Each field has a number of values sampled, the frequency of the occurrence of each value is recorded. At the end of the run, we can find the possible states of each class. Surprisingly, many classes analyzed have a distinct hot state.

After offline profiling and static analysis, information about hot classes and hot states are obtained. Hot classes that have distinct hot states are mutable classes. The hot states of mutable classes are decided by the values of state fields. Such information is then sent to Jikes RVM for use in optimization decisions.

3.2. Online Jikes RVM Implementation

This section describes how Jikes RVM is modified to dynamically mutate classes based on hot state and mutable class information obtained through profiling.

3.2.1. Jikes RVM

Jikes RVM is an open source research virtual machine developed by IBM. The version used for the implementation is Jikes RVM 2.3.4.

Jikes uses a compile-only approach to execute Java bytecode. In the production configuration (the FastAdaptiveGenMS configuration), a baseline compiler and an optimization compiler with three different optimization levels are used. Non-native methods are initially compiled using the baseline compiler and then recompiled by the optimization compiler with different optimization levels if they are detected hot by the adaptive optimization system. The adaptive optimization system performs tasks such as method sampling, hot method recompilation and adaptive inlining.

Jikes has a global structure called the Jikes Table of Content (JTOC) that contains the pointers to the compiled code for static methods. Each class has a structure called the Type Information Block (TIB) that contains the pointers to the code for instance methods. The TIB serves as a virtual function table (VFT). The pointers to the compiled code for interface methods can either be directly embedded into the TIB or be stored in another data structure accessed indirectly via the TIB. Detailed interface method invocation information for Jikes can be found in [2].

A unique object model [7] is created for each class. The object model is used by the memory management system to allocate an object instance whenever an object is created. Each created object has its own TIB pointer. In Jikes RVM 2.3.4, all the object instances have their TIB pointers set as the class TIB pointer.

Each non-native method in Jikes can have at most one valid compiled method at a time. The compiled method can either be generated by the baseline compiler or the optimization compiler. When a new compiled method is generated for a method, the existing compiled method is replaced and invalidated. The replacement occurs in the JTOC if the method is static or in the class TIB and the subclasses' class TIBs (if the method is not private or overridden by the subclasses) if the method is non-static.

Jikes supports both static heuristics-guided inlining and dynamic profile-guided inlining. The static heuristic makes decisions based on the callee bytecode size, the callee's arguments, the root compilation method bytecode size, the inlining depth, and if the inlining needs to be guarded. Profile-guided inlining applies sampled call site information to help achieve better inlining results. Guarded inlining is also supported in Jikes when there is not a single precise target callee at a call site. [18] has more information about Jikes RVM's inlining implementation.

3.2.2. Implementation Details

The core of the implementation is the special TIB management and the special compiled code management. The implementation is mainly based on a distributed dynamic class mutation algorithm shown in Figure 4 and

```
At the end of the constructors for a mutable class:
 If (the object's state is dependent on any instance field)
    If (the values of instance state fields  match a specific hot state)
      set the object's TIB pointer to the correspondent special TIB pointer;
For each assignment of an instance state field in a
non-constructor method for a mutable class:
 If (the values of instance state fields match a specific hot state)
    set the object's TIB pointer to the correspondent special TIB pointer;
 else if (the values of instance state fields don't match any hot state)
    set the object's TIB pointer to the class TIB pointer;
For each assignment of a static state field:
foreach (mutable classes whose states are dependent on this static field) {
   if (the values of static state fields match a hot state) {
      foreach (mutable instance methods that have special compiled code) {
         if (the class's state is also dependent on at least one instance field)
            set the compiled code pointer in the correspondent special TIB
            to the correspondent special compiled code;
         else
            set the compiled code pointer in the class TIB to the
            correspondent special compiled code;
      }
      foreach (mutable static methods that have special compiled code)
         set the compiled code pointer in the JTOC to the correspondent
         special compiled code;
   } else if (the values of static state fields do not match any hot state) {
      foreach (mutable instance methods that have special compiled code) {
         if (the class's state is also dependent on at least one instance field)
            set the compiled code pointers in the special TIBs to the
            general compiled code;
         else
            set the compiled code pointer in the class TIB to the general
            compiled code;
      }
      foreach (mutable  static methods that have special compiled code)
         set the compiled code pointer in the JTOC to the general compiled code;
   }
}
```

Figure 4. Distributed dynamic class mutation algorithm part I: actions at state field assignments.

Figure 5. The algorithm takes actions in different locations such as state field assignments and high-level opt recompilations of mutable methods.

For mutable classes that are dependent on instance fields, a number of special TIBs are created. Each special TIB corresponds to a specific hot state. The special TIB can hold the pointers to the special compiled code for instance methods. The compiled code is specialized for the hot state that this special TIB matches. The special TIB can also contain the pointers to the general compiled code for instance methods. In fact, the special TIB is exactly the same as the class TIB when the class is initially instantiated. The existence of special TIBs does not disable lazy compilation, an important technique to avoid the compilation of unused methods. The special TIBs just need to be exactly the same as the class TIB until certain methods are hot, e.g. compiled in optimization level 1 or level 2, and certain conditions are matched. The number of special TIBs is equal to the number of hot states for this class. For mutable classes that are only dependent on static fields, no

At the recompilation of a hot method in the adaptive system:

```
If (the recompilation optimization level is high &&
    the method is mutable) {
    all special compiled code and the general compiled code of this
    method are generated;
    if (the values of static state fields match a specific hot state) {
        if (the method is an instance method) {
            if (the class's state is dependent on at least one instance field) {
                set the compiled code pointers in the correspondent special
                TIBs to the correspondent new special compiled code;
                set the compiled code pointer in the class TIB to the new
                general special compiled code;
            } else
                set the compiled code pointer in the class TIB to the new
                 special compiled code;
            propagate the new general compiled code to the sub classes'
            TIBs if the method is not private;
        } else
            set the compiled code pointer in the JTOC to the correspondent
             new special compiled code;
    }  else {
        if (the method is an instance method) {
            set the compiled code pointers in all TIBs to the new general
            compiled code;
            propagate the new general compiled code to the sub classes'
            TIBs if the method is not private;
        } else
            set the compiled code pointer in the JTOC to the new general
            compiled code;
    }
}
```

Figure 5. Distributed dynamic class mutation algorithm part II: actions at the generation of compiled code for mutable methods.

special TIB is needed since all the instantiated object instances from this class will share the same mutation state if the values of the static fields match a hot state of this class. In this case, pointers to special compiled code are directly updated in the class TIB. For static methods of a mutable class, pointers to their special compiled code are always directly updated in the JTOC since static methods can only use static fields.

Each mutable method of a mutable class can have multiple specialized versions, equal to the number of mutation states of this class. When a method is compiled at a high optimization level, e.g. opt1 or opt2, the specialized versions are generated at the same time.

In several places, object instances' states need to be monitored and corresponding updates to the JTOC, the class TIB, and the special TIBs need to be made. Figure 4 shows the first part of the distributed dynamic class mutation algorithm. It is activated at state field assignments. State fields can be divided into static and instance. They need to be handled differently. In Jikes, the GC can arbitrarily move a pointer to an object instance and thus there is no way to book-keep all the object instance pointers for mutable classes without modifying the GC and introducing GC overhead. Fortunately, at instance field assignments, object pointers are available and they can be used to obtain the values of instance state fields. If the values match a hot state (the values of instance fields in this state), the object TIB pointer can be changed to the special TIB pointer that corresponds to this hot state. If the values do not match any hot state, care needs to be taken to make sure the object TIB pointer is the class TIB pointer. However, the change of the object TIB pointer to a special TIB does not mean that this object is already in the hot state or that special compiled code needs to be used to execute the mutable methods of this mutable class. If the class has any static state field that does not match any hot state (the value of this static field in any hot state), this object instance is not mutated yet and the general compiled code needs to be used for mutable methods in the special TIB. At this time, the compiled code pointers in the special TIB for mutable methods are still the pointers to the general compiled code. The code added at the assignments of static state fields is more complicated. It needs to handle both mutable static methods at the JTOC and mutable instance methods at the special TIBs. The update on the JTOC is easy since static methods are only affected by static fields. For the special TIBs, the change in a static state field means that they now can hold the pointers to the special compiled code for mutable methods if the values of all static state fields match a hot state. Otherwise, the special TIBs have to hold the pointers to the general compiled code for mutable methods.

Figure 5 shows the second part of the distributed dynamic class mutation algorithm. The special compiled code is generated when a mutable method is recompiled using the optimization compiler in a high opt level. After both the general compiled code and the special compiled code are generated, the corresponding entries in the JTOC and the TIBs need to be updated. The special TIBs are updated with the special compiled code when the values of all static state fields match a hot state. This includes two cases: First, there exist static state fields affecting the hot state of a mutable class and their values match the static state field values of the hot state. Second, there are no static state fields affecting the hot state of the mutable class and we assume this is a default match. Otherwise, they are updated with the general compiled code. In the current implementation, the general compiled code instead of the special compiled code is propagated to the sub classes. Only the methods declared by a mutable class are candidates for mutation for this class. Figure 6 gives an example about how to mutate a specific class in a class hierarchy. Offline analysis indicates that only class B is a mutable class. Methods m2 and m3 declared by class B are mutation method candidates. Method m2 declared by class B overrides m2 declared by class A. The special compiled code of method m2 and m3 do not propagate to class C.

```
class A                 class B extends A       class C extends B
{                        {                       {
  ......                   ......                   ......
  public method m1;        public method m2;     }
  public method m2;        public method m3;
  ......                   ......
}                        }
```

Figure 6. An example for specializing a specific class in a class hierarchy.

3.2.3. Further Implementation Issues

For a method invoked using "invokespecial" in Java bytecode, the invocation of the method is statically bound via the class TIB pointer of the method's declaring class. This is for the invocation of an instance initialization method, a private instance method, or a method invoked with "super". Due to correctness issues [21], the invocation can not use dynamic binding via the object TIB pointer. Therefore, the dynamic class mutation technique does not work for such invocations. However, a private instance method can still be mutated if its declaring class is solely dependent on static state fields. In this case, the class TIB itself can be specialized and no special TIB is needed.

The mutation of interface methods in a mutable class is not supported currently. However, Jikes can be easily extended to handle this case. The default in the Jikes production configuration uses an indirectly accessed interface method table (IMT) to help access interface methods implemented by a class even though the pointers to the compiled code for interface methods are also directly stored in the TIB. The TIB holds an entry that has a pointer to the head of the indirectly accessed IMT. The IMT has a fixed number (a static compilation constant) of slots. Each IMT slot can correspond to one or multiple interface methods. If a slot corresponds to one interface method, the pointer to the compiled code for this method is also stored in the IMT slot. If a slot corresponds to multiple interface methods, the slot contains a stub function that can look up all the interface methods corresponding to this slot and find the offset to the TIB pointer for an interface method. Then the stub function uses the object TIB pointer and the offset to find the compiled code for the interface method. In order to support mutation for interface methods, the only modification is to have the IMT slot corresponding to one interface method store the offset to the TIB pointer for this method's compiled code. When invoking an interface method for a mutable class, another load needs to be inserted to get the compiled code pointer after obtaining the content of the IMT slot. This modification is only needed for mutable classes and no extra loads need to be inserted for the invocation of the interface methods for other classes. With this modification, all the special TIBs and the class TIB share the same IMT and no additional IMTs need to be generated.

For type checking/casting bytecode operations such as "instanceof" and "checkcast", Jikes uses the object TIB pointer and the class TIB pointer to check if an object is an instance of a class. In the presence of special TIB pointers, this is not true any more. The TIB has an entry that contains the type information for a class. The special TIB is a replicant of the class TIB and then changes the compiled code pointers for mutable methods to the special compiled code pointers. It has the same type information entry as the class TIB. For mutable classes, the type information entries instead of the TIB pointers need to be used to decide if an object is an instance of a class.

The Jikes adaptive system collects the sampling information for each compiled method and uses it to indicate the hotness of the compiled method's correspondent method since each method can only have one valid compiled method at a time and the new compiled method inherits the sampling information from the old compiled method at recompilation for a method. When the special compiled methods exist, the sampling information must be shared between the general compiled method and the special compiled methods. Otherwise, the dilusion of sampling information can delay the recompilation of a hot method.

4. Object Lifetime Constants

Many instance state fields are only assigned once and assigned to constants in their declaring classes' constructors. Figure 7 shows some code in two mutable classes in SPECjbb2000. DisplayScreen has two instance state fields (rows and cols) and they are assigned to constants in a constructor. The two fields are never assigned outside the constructor in DisplayScreen. They are object lifetime constants for objects instantiated from this constructor if their values are never changed after the object creation. Identification of object lifetime constants can help reduce the amount of code patched at state field assignments and help the inlining heuristics make better decisions about inlining mutable methods.

```
package spec.jbb.infra.Util;
...
public class DisplayScreen {
    ...
    int rows, cols;
    public DisplayScreen() {
        rows = 24, cols = 80;
    } ...
} ...
```

```
package spec.jbb;
...
class DeliveryTransaction extends Transaction {
    ...
    private DisplayScreen deliveryScreen;
    public DeliveryTransaction(...) {
        deliveryScreen = new DisplayScreen();
    } ...
} ...
```

Figure 7. An example for object lifetime constants.

In order to capture object lifetime constants, a new algorithm, shown in Figure 8, is designed. The algorithm is based on field analysis and simple escape-like analysis. The algorithm only targets private reference fields that point to object instances of mutable classes. The overhead of the algorithm is very low since only a few reference fields and classes are analysis targets and Jikes has already implemented the field analysis framework for our algorithm.

```
foreach (mutable classes) {
    identify constructor assigned instance state fields and
    record <field, constructor, value> tuples for fields
    that are assigned to constants in constructors;
}
foreach (private reference fields of an exact type
        (in other classes) referring to a mutable class) {
    if ( the private reference field is always assigned
         by "new" using the same constructor) {
        foreach (instance state fields that are assigned to
                 a constant in this constructor) {
            prove that the private reference field's declaring
            class does not modify the field;

            prove that the private reference field does not
            escape its declaring class;

            the field is an object lifetime constant for the
            private reference field if the above two proofs
            succeed;
        }
    }
}
```

Figure 8. An object lifetime constant analysis algorithm.

The algorithm includes the following two major steps. Step one is to perform field assignment analysis for instance state fields in mutable classes. The goal is to identify fields that are only assigned in constructors. A tuple <field, constructor, value> is saved if a field is assigned to a constant in a constructor. These fields are possible object lifetime constants. The next step is the analysis for private reference fields of exact types (in other classes) that refer to mutable classes. The Jikes optimization compiler can already identify private reference fields that are of exact types. Step two starts by finding fields that are always assigned by "new" using the same constructor that initializes some instance state fields to constants. These fields are possible object lifetime constants for the private reference field. To prove that they are object lifetime constants, two additional analyses are needed. First, it needs to be proven that the private reference field's declaring class does not modify a possible object lifetime constant. If an instance state field is not accessible from the private reference field's declaring class, this class can not modify the instance state field. In Figure 7, fields like rows and cols have default accessibility and they are not accessible from a different package and thus the DeliveryTransaction class cannot modify them. If the instance state field is accessible from the private reference field's declaring class, more field analysis is needed to make sure that the private reference field's declaring class does not modify the instance state field. Next, escape-like analysis is needed to guarantee that the private reference field does not escape its declaring class. To satisfy this, the following requirements are needed: it has never been assigned to another field; it has never been used as a method call parameter; it has never been returned by a method. The requirements used here are conservative to guarantee correctness and simplify computation. They can be relaxed in the future work. The algorithm assumes that a mutable class does not leak its own "this" pointer, e.g. return its "this" pointer in one of its functions. This rarely happens and the programs studied do not have this behavior. Otherwise, similar analysis needs to be performed to ensure that the "this" pointer does not escape a mutable class that is of the private reference field's type. If a possible object lifetime field for a private reference field is never modified by the private reference field's declaring class and the private reference field does not escape its declaring class, we know that it is a real object lifetime field and can be treated as a constant.

5. Inlining of Mutable Methods

Inlining is an important optimization technique to improve the performance of object-oriented programs. Inlining of specialized versions of mutable methods typically requires value guarding of the specialized fields. The existence of value guarding itself is an overhead. The existence of both the specialized version of inlining and the regular version of inlining can quickly lead to significant code bloat. One solution is to only inline the specialized version of code but this risks the sacrifice of the benefit of inlining at all if the specialized fields do not have the desired values.

We propose a solution to address the problem of inlining specialized versions of mutable methods. The solution combines the static object lifetime constant analysis and the trade-off inlining decision making. The static analysis identifies fields that will never change their values after their enclosing object is created. Such fields can be specialized in method inlining. In addition, the existence of such fields can also lower the inlining cost of a method when the inlining decision is being made by the inlining heuristics, which enhance the possibility of inlining the callee. In Figure 7, rows and cols are object lifetime constants for the private reference field - deliveryScreen. In the DeliveryTransaction class, all methods called with the receiver "deliveryScreen" can be inlined with rows and cols specialized. The existence of object lifetime constants can also lead to partial specialization inlining. For example, if a method has both object lifetime constants and specializable fields that need guards, it can be inlined only with specialized object lifetime constants.

If no object lifetime constants exist for the inlining of a method, a trade-off needs to be made to decide which benefit is larger, inlining or specialization. Inlining and specialization never coexist in this case. A simple heuristics is presented to decide whether inlining or specialization should be performed. Let N be the number of constants passed at the call site and M be the number of fields that can be specialized in the callee. If N is larger than (M+k), where k is a tunable integer, inlining is performed. Otherwise, specialization is performed. If k is a very small negative number, inlining is almost always performed. If k is a very large positive number, specialization is almost always performed.

6. Experiments

All experiments are performed using Jikes RVM version 2.3.4 on a 2.4GHz Intel Pentium 4 based single-processor machine. The processor has a 512KB L2 cache and its hyperthreading is disabled. The machine has 1GB memory. The operating system is Fedora Core 1 with kernel 2.4.22-1.2115.nptlsmp.

The Jikes RVM build uses the production (FastAdaptive-GenMS) configuration. Jikes is built to execute Java 1.3 applications. For all benchmarks, the initial compiler is set to the optimization compiler. The default optimization level is set to opt0. Methods are initially compiled at opt0 and then recompiled at opt1 and opt2 by the adaptive optimization system. Mutation occurs at opt2. When a mutable method is recompiled at opt2, its mutated versions of compiled code are also generated.

The applications examined include the example program used in Section 2, a logic simulator similar to the one in [24], three real applications (CSVToXML[11], Java2XHTML[29], and Weka[36]) found on the internet, and two standard Java transaction processing benchmarks (SPECjbb2000[31] and SPECjbb2005[32]) from SPEC. CSVToXML's version is 1.1, Java2XHTML's version is 2.0, and Weka's version is 3.2.3. The original SPECjbb2005 is designed to test Java 1.5 virtual machines and cannot be run on Jikes. It has been ported to Java 1.3. For all the benchmarks, multiple runs are performed and the best repeatable results are reported. SPECjbb2000 uses a 128MB heap, SPECjbb2005 uses a 384MB heap, and the rest applications use a 50MB heap (the default heap size in Jikes). The chosen heap sizes are relatively small heap sizes required by the programs without putting too much pressure on GC. For both SPECjbb benchmarks, one warehouse is run multiple times and the throughput of the warehouse is used as the performance metric. The measurement time of each warehouse is 120 seconds for SPECjbb2000 and 240 seconds for SPECjbb2005. The warm-up time of each warehouse is 30s for both benchmarks. The measurement and warm-up times are as specified by SPEC. In a typical compliant run for SPEC publications,

TABLE 1. Benchmarks used in the empirical study.

Program	Description	Classes	Methods
SalaryDB	Microbenchmark	3	8
SimLogic	Simple Logic Simulator	3	29
CSVToXML	CSV to XML conversion	5	32
Java2XHTML	Java to XHTML conversion	2	8
Weka	Data mining algorithm tool set	22	423
SPECjbb2000	SPEC Transaction processing benchmark	81	978
SPECjbb2005	SPEC Transaction processing benchmark	65	702

SPECjbb2000 takes about 25 minutes while SPECjbb2005 takes nearly an hour.

7. Results

This section presents the results of the empirical study. It shows that the mutation technique can improve the performance of a variety of applications with little overhead. For SPECjbb, histograms about performance over time are given to show when mutation occurs to improve performance.

7.1. Overall Performance

The mutation technique can effectively improve the performance of the applications studied. Figure 9 shows the speedups for the applications examined. SalaryDB shows a significant speedup of 31.4%. The performance improvement is mainly due to branch elimination and dead code elimination. It shows that the potential ideal speedup

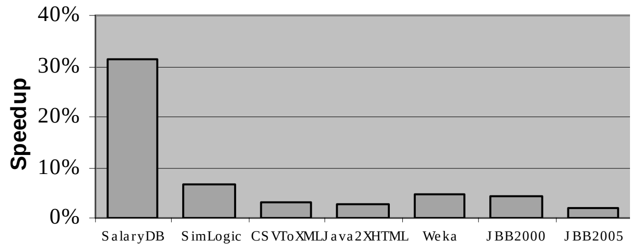


Figure 9. Overall performance improvement.

achieved by the class mutation technique. In other applications, the following factors also contribute to performance improvement: constant propagation, strength reduction, loop unrolling, array bound checking elimination, and specialization inlining. The speedup of the logic simulator is not as high as reported in [24] since the logic simulator in [24] is implemented in C++ and the function pointer is directly manipulated in assembly. For C++, compilation is also done offline and there is no compilation overhead. CSVToXML, Java2XHTML, and Weka show speedups of 3.3%, 2.9%, and 4.7%. These application have one or two distinct mutable classes that account for most of the computation time, so speedups due to class mutation are not surprising. SPECjbb2000 shows a significant speedup of 4.5% while SPECjbb2005 shows a noticeable speedup of 1.9%. For both benchmarks, the throughput of a steady state warehouse is used as the performance metric. In SPECjbb2000, quite a few classes are mutable and mutation creates a lot of opportunities for specialization inlining. With specialization inlining, a lot of further optimizations are available. This explains why a speedup of 4.5% is observed. SPECjbb2005 is a new transaction processing benchmark developed based on SPECjbb2000. The two benchmarks have similar designs. SPECjbb2005 introduces a new heavyweight transaction called CustomerReport and spends less time in mutable methods. In addition, SPECjbb2005 is much more memory aggressive than SPECjbb2000. Its required run heap size is much bigger than SPECjbb2000's. Despite all these, a speedup of about 1.9% is still observed.

7.2. Overhead

To evaluate the overhead introduced by the mutation technique, the compiled code size increase, the compilation time increase and the TIB space increase are measured. Figure 10 shows the increase of the code compiled by the optimization compiler in the presence of the mutation technique. In the experiments, the initial compiler is the optimization compiler and the compiled code generated by the optimization compiler represents the majority of the compiled code for the application. The compiled

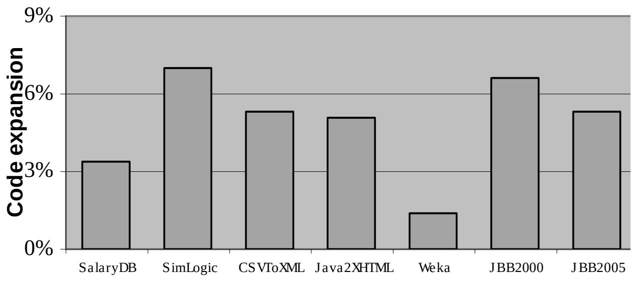


Figure 10. Code size increase. The main contribution to the code size increase is extra versions of compiled code for mutable methods compiled at opt2**.**

code by the optimization compiler includes code compiled at all optimization levels (opt0-opt2). The compiled code size increase is small in all applications, which proves that the space overhead is low for class mutation. Mutation at opt2 and dead code elimination help significantly reduce the size of extra compiled code. Specialization inlining is one main reason for the increase in the size of extra compiled code.

The optimization compiler's overall compilation time increase is shown in Figure 11. Except for SPECjbb2000 and SPECjbb2005, the increase in the compilation time in other applications is less than 8%. Weka has the lowest

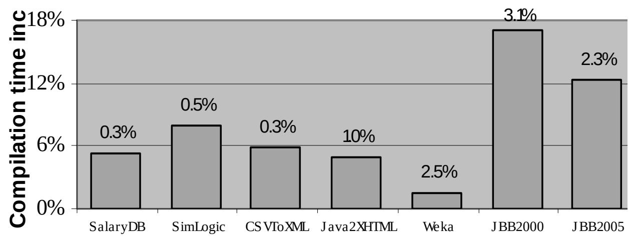


Figure 11. Opt compiler's compilation time increase. The y axis indicates the relative increase of the compilation time due to mutation. The numbers above the bars indicate the compilation time fraction of the total execution time without mutation**.**

compilation time increase, which complies with its small increase in compiled code size. The SPECjbb benchmarks have much more memory pressure. They also have a lot of specialization inlining due to the presence of object lifetime constants and class mutation. Inlining can dramatically increase the compilation time. During specialization inlining, a lot of code is eliminated due to constant folding and strength reduction. This possibly explains why the increase in compilation time is larger than the increase in the compiled code size. The compilation-to-execution time ratios for the studied applications are very reasonable. Especially for SPECjbb2000 and SPECjbb2005, their ratios, 3.1% and 2.3% respectively, are in the range of those that are observed in SPECjbb publication runs using production JVMs.

The additional space occupied by special TIBs is also very small, as shown in Figure 12. The TIB space increase is at worst about 1KB for SPECjbb2000 while it is less than 100 bytes for CSVToXML, Java2XHTML and Weka. TIBs are typically very small and only take about tens of bytes. Duplication of TIBs does not cause any noticeable memory overhead. It is worth noting that memory allocated for TIBs is typically immortal in Jikes, which means that it cannot be garbage collected. Hence, the TIB space should be as small as possible.

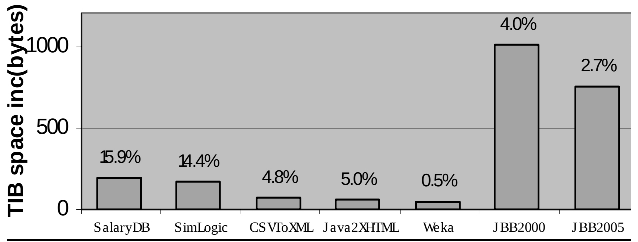


Figure 12. TIB space increase. The y axis indicates the absolute increase of the TIB space. The numbers above the bars represent the relative increase of the TIB space**.**

7.3. SPECjbb Performance

For SPECjbb, we measure how the class mutation technique affects its performance/throughput over time. All the application methods are initially compiled using opt0 and then recompiled at opt1 and opt2 when they are detected hot. When no mutable methods are hot enough, there is no performance benefit from class mutation but performance degradation due to overhead. Gradually, some mutable methods are hot and recompiled at opt2. At this time, mutated compiled code is generated and it can be used to replace the execution of the general compiled code. As shown in Figure 13, for SPECjbb2000, the first warehouse and the second warehouse have lower throughput than the other warehouses. The second warehouse's throughput is lower than the first one's since more compilation occurs in the second warehouse. SPECjbb2005 experiences a similar low throughput period in warehouse one, two and three, as shown in Figure 15. Its low throughput period is longer than SPECjbb2000's since mutable methods are detected hot more slowly than SPECjbb2000's mutable methods. In addition, its steady state performance benefit due to class mutation is not as good as SPECjbb2000's. For data shown in Figure 14, the hotness detection process is accelerated for mutable methods in SPECjbb2000. In such runs, opt1 and opt2 compiled code for mutable methods is generated immediately after their opt0 compiled code is generated. Many mutable methods are recompiled at opt2 in the first warehouse. The early recompilation of such methods causes a sharp drop of the first warehouse's throughput but the steady state throughput arrives earlier in the second warehouse.

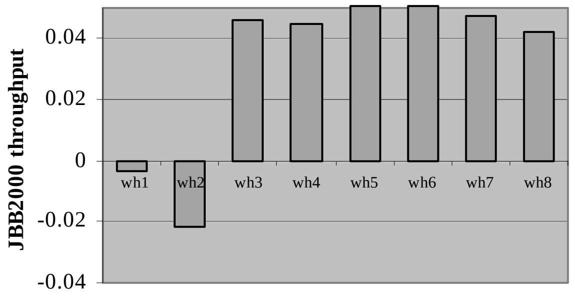


Figure 13. SPECjbb2000's throughput change due to mutation. One warehouse is run eight times with or without mutation. The throughputs of all the warehouses with mutation are compared with those of all the warehouses without mutation**.**

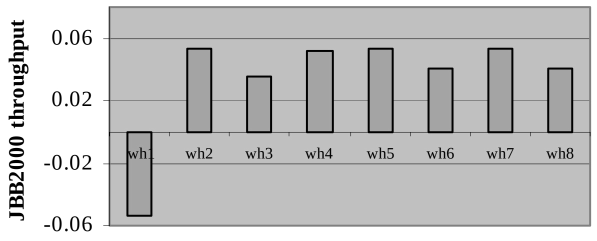


Figure 14. SPECjbb2000's throughput change due to mutation with accelerated mutable method hotness detection. One warehouse is run eight times with or without mutation. The throughputs of all the warehouses with class mutation are compared with those of all the warehouses without mutation**.**

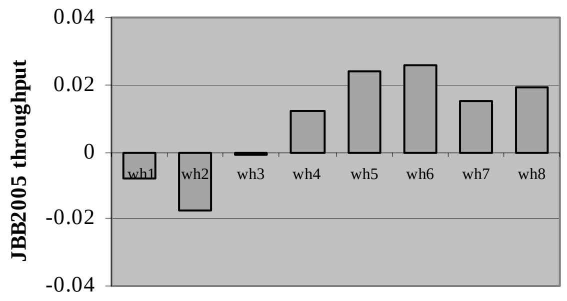


Figure 15. SPECjbb2005's throughput change due to mutation. One warehouse is run eight times with or without mutation. The throughputs of all the warehouses with mutation are compared with those of all the warehouses without mutation**.**

8. Related Work

Research related to our work can be generally divided into three directions: metamorphic programming, code specialization, and static class hierarchy slicing.

8.1. Metamorphic Programming

Research in this direction realizes that a program's behavior keeps changing at runtime (often based on object state changes in object-oriented programming languages). To take advantage of this behavior change, code should be able to dynamically adapt itself to improve its performance. To our best knowledge, [24] first proposed metamorphic programming and it inspired our research. Its main idea is that programs dynamically change their behaviors and they should be able to dynamically adapt themselves to the runtime state to improve their performance. [24] uses the C++ programming language to implement a few programs, including a logic simulator, and inserts assembly code to these programs to change the programs themselves, e.g. change a function pointer to a different implementation. In their examples, they are able to show significant speedups due to the efficiency of C++ and assembly.

[35] also noticed the dynamic code behavior change. It proposed class specialization and the change of specializable methods according to runtime states for the Java programming language. They use a source-to-source transformer to achieve this. Initially, programmers need to

identify specialization classes (N classes) for a particular class in a program. Then these classes are used as the inputs to the source-to-source transformer. The transformer generates an enclosing class and a set of N+1 implementations of the original class. The execution of the original class needs to be directed to the enclosing class first. The enclosing class is responsible for checking the class state and executing the right implementation. In their approach, class specialization is mainly performed by programmers and then specialized classes are linked into the program using a source-to-source transformer before the program is statically compiled to bytecode. This approach also requires the extension to the Java programming language. [35] only conceptually demonstrated their technique's effectiveness for two microbenchmarks.

8.2. Code Specialization

Recent research on code specialization spans across three generations of programming languages: functional, procedural, and object-oriented. Its focus started from pure research exploration to practical product experimentation. Its target optimization scope deals with single instructions, code regions, and methods.

Specialization for functional programming languages such as ML mainly relies on the language's partial evaluation and stage computation features. The FABIUS compiler [20] relies on programmers to identify instructions that can be statically and dynamically generated. Dynamic instructions (mainly for loop invariant removal) are generated at runtime with known invariant information.

Specialization for procedural programming languages focuses on C. Research on C specialization has to add dynamic features to C since C itself does not have any runtime support. The DCG system [15] exposes the intermediate representation to programmers and let them choose code regions to specialize. Tempo [23], the University of Washington dynamic compiler [5], and the tcc system [25] use templates to specify instruction templates that can be replaced at runtime. The DyC system [17] employs dynamic compilation for dynamic code regions identified by static analysis (static profiling and binding time analysis). The runtime optimizations for dynamic code regions are able to show performance in some fairly large applications. DyC also proposes the concept of polyvariant division, which means certain variables may be instantiated to several values at runtime. However, no investigation of this concept is performed in the paper. Besides specialization research on C, [6] proposes a staging analysis algorithm to find specializable variables in Fortran programs. The analysis tries to find variables that are assigned occasionally but used frequently. It is based on loop nesting levels.

In the investigation of specialization for procedural programming languages, specialization for object-oriented programming languages was also being conducted. This research started from Self, Cecil and then Java. [10] specializes the receiver type to minimize the dynamic function dispatch cost in Self. [12] proposes to use static profiling to address overspecialization in Cecil and it also tries to address underspecialization of the receiver type by considering the arguments of a function. [13] uses type group analysis to identify method arguments that are of constant types at call sites. This constant type information can be used in inlining.[30] is the first practical product level exploration of code specialization in a Java Virtual Machine. It performs code specialization in a method basis in the highest optimization level. The implementation is in a proprietary JVM owned by IBM. In this system, the impact analysis is performed to identify specializable fields in a hot method. Then runtime value profiling is performed to collect value information. The method is finally specialized with the values collected. Value guards need to be placed at the beginning of the method. In case of a specialization miss, the JVM needs to recollect the values for specializable fields for the methods and then regenerate the specialization code. This system can truly handle runtime constants efficiently but not semi-invariant fields. Little is disclosed in the paper about how specialization inlining is addressed in the system. Very recently, [27] proposes to use heap store analysis to specialize Java programs. It shows significant speedups for some fairly small programs.

8.3. Static Class Hierarchy Slicing

Research in this direction [28][33][34] is somewhat related to dynamic class mutation. Static class hierarchy slicing notices there are some parts (some fields, methods, or classes) in programs, especially in class libraries, which are not used in the program execution. They can be removed using static analyses. This is a static mutation of the class hierarchy.

9. Conclusions and Future Work

This paper presents an automated framework that can detect stateful class attributes in object oriented programming languages such as Java and mutate class hierarchies dynamically based on these attributes. Mutation classes are implicitly derived by generating special virtual function tables (VFTs) from the general (class) VFT. Each special VFT matches a dynamically mutated class in a specific object state. The framework captures object-oriented programs' metamorphic behavior and creates opportunities for conventional optimizations such as constant propagation, function specialization, and dead code elimination. The framework has been implemented in a Jikes-based system and shows performance improvements ranging from 1.9% to 31.4% in a variety of applications, including SPECjbb2000 and SPECjbb2005. The implementation introduces little space overhead (compiled code size increase less than 8% and VFT storage increase fewer than about 1000 bytes) and a small compilation time increase (17% for SPECjbb2000, 12% for SPECjbb2005, and lower than 8% for other applications).

In future work, we plan to consolidate out tool chain and investigate the feasibility of a complete online Java solution. We will try to move our offline profiling and static analysis to a JVM, possibly a Jikes based system. This will require the development of efficient profiling schemes and light weight static analysis algorithms to minimize the potential online overhead and maximize the performance benefit of class mutation.

10.Acknowledgments

This work was made possible through generous equipment donations and financial support from IBM and Intel, and NSF grants EIA-0103670, CCR-0133437, and CCF-0429854. We would like to thank Kingsum Chow, Trey Cain, and Jong-Deok Choi for feedbacks on the initial idea. We are grateful to Gorden Bell, Natalie Enright, Erika Gunadi and Eric Hill for proof reading the paper. We thank all anonymous CGO reviewers for the detailed and

helpful reviews.

References

[1] B. Alpern, C. R. Attanasio, J.J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno Virtual Machine. IBM Systems Journal, 39(1):211-221, 2000.

[2] B. Alpern, A. Cocchi, S. Fink, D. Grove, and D. Lieber. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, Tampa, FL, Oct. 2001.

[3] M. Arnold, D. Grove, S. Fink, M. Hind, and P. Sweeney. Adaptive Optimization in the Jalapeno JVM. In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, Minneapolis, MN, Oct. 2000.

[4] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation, Snowbird, Utah, June, 2001.

[5] J. Auslander, M. Philiipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, Effective Dynamic Compilation. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Philadelphia, PA, June, 1996.

[6] T. Autrey and M. Wolfe. Initial Results for Glacial Variable Analysis. In Proceedings of the oth International Workshop on Languages and Compilers for Parallel Computing, Aug. 1996.

[7] D. Bacon, S. Fink, and D. Grove. Space- and Time-Efficient Implementation of the Java Object Model. In Proceedings of the European Conference on Object-Oriented Programming, Malaga, Spain, June, 2002.

[8] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Shreedhar, H. Srinivasan, and J. Whaley. The Jalapeno Dynamic Optimizaing Compiler for Java. In ACM Java Grande Conference, June 1999.

[9] B. Calder, P. Feller, and A. Eustace. Value Profiling. In Proceedings of the 30th International Symposiumon on Microarchitecture, Research Triangle Park, NC, Dec. 1997.

[10] C. Chambers and D. Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Languages. In Proceedings of the ACM Conference on Programming Language Design and Implementation, Portland, OR, June, 1989.

[11] CSVToXML v1.1. http://www.dpawson.co.uk/java/csv2xml.html.

[12] J. Dean, C. Chambers, and D. Grove. Selective Specialization for Object-Oriented Languages. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, La Jolla, CA, June, 1995.

[13] J. Dean and C. Chambers. Towards Better Inlining Decisions Using Inlining Trials. In Proceedings of the ACM SIGPLAN '94 Conference on LISP and Functional Programming, Jun. 1994.

[14] A. Diwan, J. E. B. Moss, and K. S. McKinley. Simple and Effective Analysis of Statically-Typed Object-Oriented Programs. In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Language, and Applications, San Jose, CA, Oct. 2000.

[15] D. R. Engler and T. A. Proebsting. DCG: An Efficient, Retargetable Dynamic Code Generation System. In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, San Jose, Oct. 1994.

[16] S. Ghemawat, K. H. Randall, and D. J. Scales. Field Analysis: Getting Useful and Low-Cost Interprocedural Information. In Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation, Vancouver, BC, Canada, June, 2000.

[17] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. An Evaluation of Staged Run-Time Optimizations in DyC. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Atlanta, GA. June, 1999.

[18] K. Hazelwood, D. Grove. Adaptive Online Context-Sensitive Inlining. In Proceedings of the International Symposium on Code Generation and Optimization, San Francisco, CA, March, 2003.

[19] Intel VTune performance analyzer. http://www.intel.com/cd/software/products/asmo-na/eng/vtune/index.htm.

[20] P. Lee and M. Leone. Optimizing ML with Run-Time Code Generation. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Philadelphia, PA, May, 1996.

[21] T. Lindholm and F. Yellin. The Java(TM) Virtual Machine Specification (2nd Edition). 1999. Addison-Wesley Professional. London.

[22] S. B. Lippman. C++ Primer, 2nd Edition. Addison-Wesley Publishing Company, 1991.

[23] R. Marlet, C. Consel, and P. Boinot. Efficient Incremental Run-Time Specialization for Free. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Atlanta, GA, June, 1999.

[24] P. M. Maurer. Metamorphic Programming: Unconventional High Performance. IEEE Computer, volume 37, issue 3, page 30-38, Mar. 2004.

[25] M. Poletto, D. Engler, and M. F. Kaashoek. tcc: A System for Fast, Flexible, and High-Level Dynamic Code Generation. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Las Vegas, NV, June, 1999.

[26] U. P. Schultz, J. L. Lawall, and C. Consel. Automatic Program Specialization for Java. In ACM Transactions on Programming Languages and Systems, Vol. 25, No. 4, pages 452-499, July, 2003.

[27] A. Shankar, S. S. Sastry, R. Bodik, and J. E. Smith. Runtime Specialization With Optimistic Heap Analysis. In Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, San Diego, CA, Oct, 2005.

[28] G. Snelting and F. Tip. Reengineering Class Hierarchies Using Concept Analysis. In Proceedings of the 6th International Symposium on the Foundations of Software Engineering, Lake Buena Vista, FL, Nov. 1998.

[29] S. Steele. Java2XHTML v2.0. http://www2.cs.fsu.edu/~steele/J2X/Java2xhtml.html.

[30] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A Dynamic Optimization Framework for a Java Just-In-Time Compiler. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, Tampa Bay, FL, Oct. 2001.

[31] The Standard Performance Evaluation Corporation. SPECjbb 2000. http://www.spec.org/jbb2000, 2000.

[32] The Standard Performance Evaluation Corporation. SPECjbb 2005. http://www.spec.org/jbb2005, 2005.

[33] F. Tip, J-D Choi, J. Field, and G. Ramalingam. Slicing Class Hierarchies in C++. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, San Jose, CA, Oct. 1996.

[34] F. Tip and P. F. Sweeney. Class Hierarchy Specialization. In Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, Atlanta, GA, Oct. 1997.

[35] E. N. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative Specialization of Object-Oriented Programs. In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, Atlanta, GA, Oct. 1997.

[36] Weka 3.2.3. http://www.cs.waikato.ac.nz/ml/weka.