Automatic Feedback-Directed Object Inlining in the Java HotSpot™ Virtual Machine *

Christian Wimmer Hanspeter Mössenböck

Institute for System Software
Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler University Linz
Linz, Austria
{wimmer, moessenboeck}@ssw.jku.at

Abstract

Object inlining is an optimization that embeds certain referenced objects into their referencing object. It reduces the costs of field accesses by eliminating unnecessary field loads. The order of objects in the heap is changed in such a way that objects that are accessed together are placed next to each other in memory so that their offset is fixed, i.e. the objects are *colocated*. This allows field loads to be replaced by address arithmetic. We implemented this optimization for Sun Microsystems' Java HotSpot™ VM. The analysis is performed automatically at run time, requires no actions on the part of the programmer and supports dynamic class loading.

We use read barriers to detect the most frequently accessed fields that are worth being optimized. To safely eliminate a field load, the colocation of the object that holds the field and the object that is referenced by the field must be guaranteed. Two preconditions must be satisfied for a field before it is optimized: the objects must be allocated together, and the field must not be overwritten later. These preconditions are checked by the just-in-time compiler to avoid a global data flow analysis. The garbage collector ensures that groups of colocated objects are not split: it copies groups as a whole to their new locations.

The evaluation shows that our dynamic approach successfully identifies and optimizes frequently accessed fields for several benchmarks. The improved peak performance of 9% in average for SPECjvm98 (with a maximum speedup of 51%) justifies the startup overhead of 3% in average (with a maximum slowdown of 11%) that is caused mainly by the read barriers and the additional compilation of methods.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Incremental compilers, Memory management, Optimization

General Terms Algorithms, Languages, Performance

Keywords Java, object inlining, object colocation, just-in-time compilation, garbage collection, optimization, cache, performance

* This work was supported by Sun Microsystems, Inc.

© ACM, 2007. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in the Proceedings of the 3rd International Conference on Virtual Execution Environments, pp. 12–21.

VEE'07, June 13–15, 2007, San Diego, California, USA.
http://doi.acm.org/10.1145/1254810.1254813

1. Introduction

Object-oriented programming encourages programmers to decompose applications into a large number of small classes with a well-understandable and well-testable functionality. While this paradigm improves the code quality, it can have a negative impact on the application's performance because it increases the number of objects that reference each other as well as the number of field loads that access the referenced objects. This can be avoided by placing such objects consecutively in the heap and replacing the field loads by address arithmetic, which is called *object inlining*.

Java™ source code [6] is compiled to platform-independent bytecodes that are executed by a virtual machine [15]. To achieve a high execution speed, modern Java virtual machines translate the bytecodes of frequently executed methods to optimized machine code using a just-in-time compiler. This approach offers the opportunity to integrate object inlining into the just-in-time compiler and apply it dynamically for fields that are frequently accessed. The garbage collector of the virtual machine that moves live objects to new locations can be used to ensure that optimized objects are located consecutively in the heap.

Figure 1 illustrates the idea of object inlining. A `Rectangle` object references two `Point` objects to specify its vertices. Normally, the three objects are independent. Two field loads are necessary to access a coordinate, e.g. of the field `p1` and the field `x`, because the address of the point must be loaded first. The objects are spread over the heap, which leads to a bad cache behavior. Object inlining combines the three objects to a larger group so that a coordinate can be accessed without loading the point first.

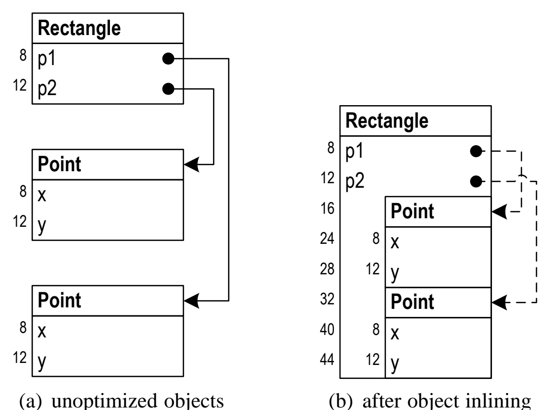

Figure 1. Motivating example for object inlining.

In programming languages like C++, this merging can be performed by the programmer using value objects. The same class can be used both to define reference objects and value objects. This is error prone because of the different semantics, e.g. when variables are assigned. C# avoids this ambiguity: the class designer must explicitly specify if a class is used for reference objects or value objects, which prohibits a flexible re-use of classes. Java does not offer value objects at the language level in favor of a simple object model.

In general, the access pattern of objects depends on how an application is used and which classes are dynamically loaded. Therefore, fields that are worth being optimized must be detected at run time. We use read barriers that increment a per-field counter to identify frequently accessed "hot fields". The analysis which fields can be inlined is done by the just-in-time compiler: methods that affect inlining are scheduled for compilation, and the compiler reports feedback data to the object inlining system. Additionally, we use the garbage collector to ensure that related objects are placed next to each other in the heap. When all preconditions are satisfied, the just-in-time compiler removes field loads for inlined fields.

To the best of our knowledge, our approach is the first that applies object inlining at run time in a virtual machine without requiring actions on the part of the programmer. Most existing solutions perform object inlining in a static compiler and use a time-consuming global data flow analysis to identify fields that should be inlined. This paper contributes the following:

- We propose to integrate automatic object inlining as a feedback-directed optimization into a Java virtual machine and to apply it at run time for frequently accessed fields.
- We support Java's dynamic class loading. Instead of a global data flow analysis, we use the just-in-time compiler for both the analysis and optimization of methods.
- We evaluate our implementation using several standard benchmarks and report results for different configurations.

The rest of the paper is organized as follows: Section 2 gives a short overview of the relevant subsystems in the Java HotSpot™ VM and introduces the components of our object inlining system. Section 3 explains the optimization process for a field and presents the analysis steps. Section 4 comments on the details of some important aspects. Section 5 presents the benchmark results. Section 6 and 7 deal with future and related work, and Section 8 concludes the paper.

2. System Overview

We build on Sun Microsystems' Java HotSpot™ VM, which is part of the JDK 6 [21]. Figure 2 shows the structure of the default configuration for interactive desktop applications. It uses a fast just-in-time compiler, called the *client compiler* [7], and a generational garbage collector with two generations. It is available for Intel's IA-32 and Sun's SPARC architecture, but object inlining is currently only implemented for the IA-32 architecture because of platform-dependent code patterns in the just-in-time compiler.

After the bytecodes of a method have been loaded by the class loader, they start being executed by the interpreter. If the invocation counter of a method reaches a certain threshold, the bytecodes are compiled to optimized machine code. The compiler uses a graph-based high-level intermediate representation (HIR) with an explicit control flow graph for global optimizations, and a low-level intermediate representation (LIR) for linear scan register allocation [23].

Compilation is done in the background in a separate thread. The compiler performs aggressive optimizations such as inlining of dynamically bound methods. If an optimization is invalidated later,

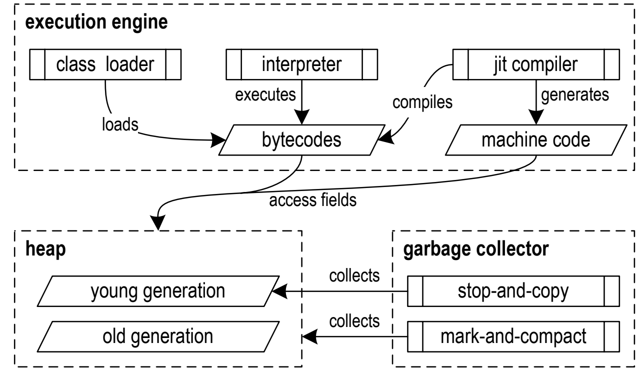

Figure 2. System structure of the Java HotSpot™ VM.

e.g. because of dynamic class loading, the VM can *deoptimize* the machine code and continue the execution of the current method in the interpreter [8].

The heap is divided into a young and an old generation. The young generation is collected using a stop-and-copy algorithm that copies live objects between alternating spaces. Objects that survive a certain number of collections are promoted to the larger old generation. If the old generation fills up, a full collection of both generations is done using a mark-and-compact algorithm [10].

2.1 Definitions

Object inlining operates on groups of objects that are in a parent-child relationship. The *parent object* contains a field that points to a *child object*. The child is also called the *inlined object*. A child has exactly one parent, but a parent can have multiple children. Therefore, a group can consist of more than two objects.

Object inlining always involves a field that is declared in the class of the parent object and that points to the child object. We denote this field as an *inlined field*. In the example, the `Rectangle` object is the parent object and the `Point` objects are the children of the rectangle. The points are referred to by the fields `p1` and `p2`, so these are the inlined fields.

Object inlining should not be confused with method inlining: method inlining is a compiler optimization that replaces a call to a method by a copy of the method body. This eliminates the overhead of method dispatching.

2.2 Design Principles of Object Inlining

Our object inlining system is designed to be *automatic* (i.e. it does not require programmer interaction), *dynamic* (i.e. it is applied at run time and supports the dynamic class loading of Java), and *feedback-directed* (i.e. it uses profiling information collected at run time to decide which fields are to be optimized). Because all analysis and optimization steps are happening during the execution of the application, the time necessary for the analysis adds to the total run time. Therefore, the analysis must be fast, which precludes a global data flow analysis and requires a conservative algorithm in several cases.

Object inlining affects several subsystems of the VM. Most of our modifications concentrate on two parts where changes are inevitable: the just-in-time compiler and the garbage collector. Other subsystems, for example the class loader, the interpreter and the internal representation of classes, contain small changes such as notifications when new classes are loaded. Many other subsystems are completely unchanged, including the locking scheme for synchronizing objects [17] as well as the handling of threads and safepoints.

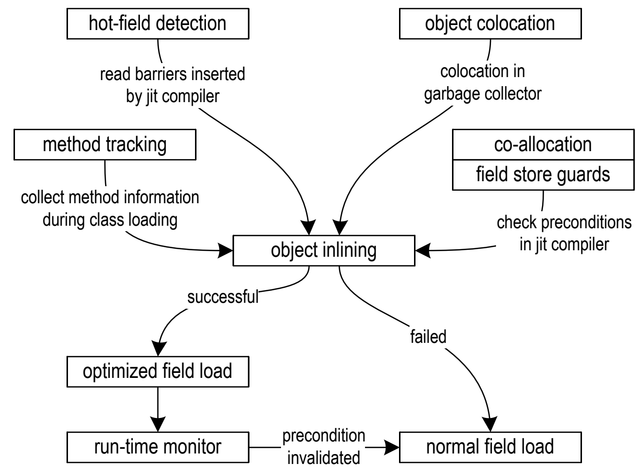

Figure 3. Components for object inlining.

The layout of objects is not changed during object inlining. The parent object and its children remain distinct objects, but we guarantee that they are located consecutively in the heap. The parent's fields that point to the children are still present, and the object headers of the children are preserved. This allows the normal access of optimized fields in subsystems that would not benefit from object inlining, such as the interpreter and the synchronization system. It also allows a smooth transition to optimized machine code. In the example in Figure 1, this is indicated by the dashed lines between the inlined objects.

We define two sufficient preconditions that a field must satisfy to safely apply object inlining:

1. The parent and the child object must be allocated together, and the field store that installs a reference to the child object into the parent object must happen immediately after the allocations.
2. The field referencing the child must not be modified after the allocation. If the field were overwritten later with a new value, the new object would not be colocated to the parent and the optimized field load would still access the old child object.

2.3 Components for Object Inlining

Figure 3 shows the components that are required for object inlining as well as their interactions. This section contains a short description of the components, and Section 3 presents details and examples. The components are not invoked sequentially, but the phases are partly overlapping because method execution, compilation and garbage collection are asynchronous.

- *Method tracking*: The class loader builds the *method table* that contains information about methods that allocate objects as well as methods that modify fields.
- *Hot-field detection*: When a method is compiled, *read barriers*, i.e. increments of per-class counters, are inserted for all field loads. If a counter for a field `f` exceeds a certain threshold, `f` is considered to be hot and is entered into the *hot-field table*.
- *Object colocation*: When the garbage collector moves objects, it processes groups of objects that are linked by hot fields together so that the parent object and its children are consecutive.
- *Co-allocation*: For every hot field `f` linking a parent object of class `P` to a child object of class `C`, the methods that allocate `P` objects are compiled. If possible, the compiler combines the allocations of `P` and `C` objects to a co-allocation. This ensures that the newly allocated objects are placed next to each other in the correct order. If co-allocation is not possible, then object inlining of `f` fails. This includes the cases where `C` is not allocated in all cases, i.e. where `f` can be `null` for some code paths. Together with object colocation, co-allocation guarantees the first precondition for object inlining.
- *Guards for field stores*: For every hot field `f`, the methods that modify `f` are compiled. The compiler inserts guards before the field stores. When a guard is executed later, object inlining of `f` fails and methods with optimized field loads are deoptimized. This guarantees the second precondition for object inlining.
- *Optimized field load*: When the preconditions for a field are satisfied, loads of the field are optimized, i.e. the memory access is replaced by address arithmetic. In some cases, it is also possible to optimize array bounds checks and dynamic type checks.
- *Run-time monitor*: It is possible that object inlining fails after optimized field loads were emitted, e.g. when a class is loaded later that invalidates a precondition. The run-time monitor detects these cases. All methods with optimized field loads are deoptimized, and execution continues with normal field loads.

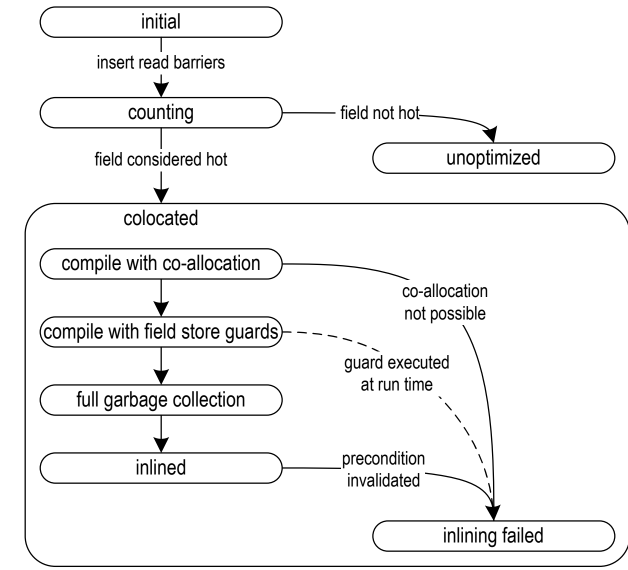

Figure 4. Optimization phases for a field.

If a precondition for a hot field cannot be guaranteed, object inlining is not possible and the field loads must remain. However, it is still possible to colocate the objects during garbage collection to improve the cache behavior. Therefore, object colocation in the garbage collector processes all hot fields regardless of their object inlining state.

3. Optimization Process for a Field

To detect if a field is worth being optimized and to guarantee the preconditions for object inlining, each field runs through the optimization phases shown in Figure 4. This section explains the details for each phase using the example started in Figure 1. Figure 5 shows the corresponding Java source code that defines the data classes as well as a test class that allocates an object. Our object inlining algorithm succeeds to inline the two `Point` objects into the `Rectangle` object and to eliminate the field loads for the fields `p1` and `p2` in the method `area()`.

```java
class Point {
  int x, y;
}

class Rectangle {
  Point p1, p2;

  Rectangle() {
    p1 = new Point();
    p2 = new Point();
  }

  int area() {
    return Math.abs((p1.x - p2.x) * (p1.y - p2.y));
  }
}

class Test {
  void allocate() {
    Rectangle rect = new Rectangle();
    // do something with rect
  }
}
```

Figure 5. Java source code for the example classes.

3.1 Method Tracking

The class loader maintains a so-called method table, which maps class names to methods that allocate objects of this class, as well as field names to methods that modify this field. This activity is called *method tracking*. Information from the method table is used in subsequent steps to decide which methods must be compiled with co-allocation (see Section 3.4) and field store guards (see Section 3.5).

All names are fully qualified, i.e. the class name contains the package name, and the field name contains the class name. Unfortunately, it is not possible to use the internal metadata objects of the VM as keys instead of the names. Because linking is not done during class loading, but at the first execution, the metadata is not available yet. This introduces a small imprecision because two classes with the same package name and the same class name loaded by different class loaders (which is allowed in Java) cannot be distinguished.

The method tracking data is stored in a single hash table that is indexed by class and field names. Figure 6 shows the method table for our example classes. Because every class and field is allocated or stored only in one method of our example, the method list on the right-hand side contains only one element for each entry.

	key (class or field)	**value** (list of methods)
1	new Point	Rectangle.Rectangle()
2	new Rectangle	Test.allocate()
3	putfield Rectangle.p1	Rectangle.Rectangle()
4	putfield Rectangle.p2	Rectangle.Rectangle()

Figure 6. Method table for the example.

To guarantee the preconditions of object inlining for the fields `p1` and `p2`, both `Test.allocate()` and the `Rectangle` constructor must be compiled. `Test.allocate()` allocates an object of the parent class `Rectangle` (entry 2) and must therefore be compiled with co-allocation. If the co-allocation were not possible, object inlining would fail. The constructor of `Rectangle` modifies the fields (entries 3 and 4), therefore it must be compiled with field store guards. If it were ever executed, object inlining would fail. The first entry is never used: object allocations of the child class `Point` are always allowed, even if they allocate objects that are not combined with a parent.

3.2 Hot-Field Detection

At first, all methods are executed in the interpreter that maintains method invocation counters. If a method counter reaches a threshold, the method is scheduled for compilation. When the method `area()` that loads the fields `p1` and `p2` is compiled, the compiler emits read barriers that increment field counters, and the state of these fields goes from *initial* to *counting*. Subsequent executions of this method increment the field counters.

If a field counter reaches a certain threshold, the field is recorded in the *hot-field tables* [24]. The state of the field goes from *counting* to *colocated*. Because the fields `p1` and `p2` are always accessed together, they are also detected to be hot at the same time.

The read barriers impose a significant run-time overhead, so they must be removed when they are no longer needed. This is accomplished by recompiling the method `area()` without inserting read barriers. Because object inlining is possible, the method is recompiled anyway to perform the optimized field load. However, even if object inlining of a field fails, methods are recompiled with normal field access, but without the read barriers.

3.3 Object Colocation

The garbage collector uses the information from the hot-field tables for object colocation. If a field is accessed frequently, the parent and the child objects that are connected by the field are likely to be accessed together. To improve the cache behavior, it is beneficial to colocate these objects even if object inlining is not possible. If the objects are colocated, they are probably in the same cache line so that accessing the parent also brings the child into the cache.

The Java HotSpot™ VM uses a generational garbage collector with two generations. We modified the stop-and-copy algorithm for the young generation to copy groups of objects instead of individual objects. This ensures that a parent object is copied together with all its child objects. If the parent and the children are not consecutive, which can happen before the first garbage collection, they are moved together and the grouping is established. The object groups are also promoted together to the old generation if they survive a certain number of copying cycles.

The mark-and-compact algorithm for the old generation preserves the object order during collection. During compaction, objects are moved towards the beginning of the heap, but they do not change their order. Therefore, the optimized order of the promoted objects is retained, and no special handling for object colocation is necessary. However, objects that were promoted before the field was identified as hot are not colocated. This can be ignored for the overall cache performance, but is a problem for object inlining, as described in Section 3.6.

The code patterns for read barriers, the processing of the read barrier counters, and the detailed algorithms for object colocation in the garbage collector, i.e. the detection and copying of object groups, are described in [24], so the details are omitted here.

3.4 Co-allocation of Objects

Object colocation during garbage collection ensures that a parent object and its child objects are consecutive after the first garbage collection run. For object inlining, however, it is a precondition that the objects are already consecutive immediately after their allocation. Therefore, *co-allocation* allocates groups of objects together, and object colocation ensures that the groups are not separated during garbage collection. We integrated co-allocation into the client compiler.

The high-level intermediate representation (HIR) of the client compiler consists of a control flow graph of basic blocks and a data flow graph of instructions. An instruction represents both an operation and its computation result, so that operands can be represented as pointers to previous instructions.

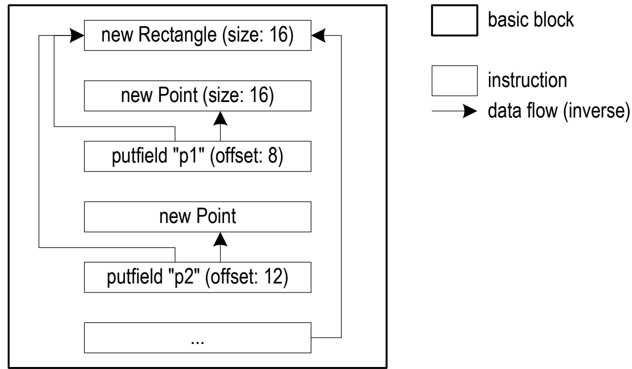

Figure 7. HIR fragment of method `allocate()`.

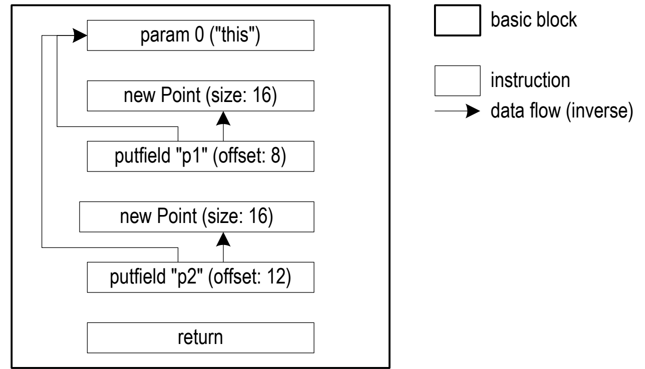

Figure 8. HIR of constructor `Rectangle()`.

Figure 7 shows the HIR for the method `allocate()`. The instructions refer to the state after method inlining. The constructor of the class `Rectangle` is inlined, so the two allocations of `Point` objects and the two field stores of the constructor are in the same block as the allocation of the `Rectangle` object.

To perform co-allocation, we search for `putfield` instructions where both the object containing the field and the assigned value refer to `new` instructions that are in the same basic block. The allocations are combined to a co-allocation instruction. In the example, the two `putfield` instructions satisfy the criteria. Because both fields are contained in the same object, the three `new` instructions are combined to a single co-allocation instruction that is inserted before the first allocation.

The machine code generated for the co-allocation allocates only one chunk of memory with 48 bytes that suffices for the three objects with 16 bytes each. Then the object headers are installed appropriately—the first `Point` starts at offset 16, the second at offset 32. No machine code is generated for the `new` instructions; they only yield the already created objects. The machine code of the `putfield` instructions remains unchanged.

Co-allocation relies heavily on method inlining. The allocations can be combined only when the methods that allocate the child objects are inlined into the method that allocates the parent object.

Object inlining for a field requires that all methods that allocate objects of the parent class are compiled with co-allocation. If the co-allocation of one method fails, then the field is not optimized and the analysis stops. The compiler reports this information as feedback data to the object inlining system. This avoids data flow analysis during object inlining.

3.5 Guards for Field Stores

The second precondition for our object inlining defined in Section 2.2 states that a field must not be modified after the co-allocation. To guarantee this, we compile all methods that modify the field and instrument the field store to revoke object inlining before the field value is changed at run time. Field stores that are already part of a co-allocation are ignored.

A static check at compile time would not be sufficient because field stores for inlined fields are allowed as long as they are not executed. In our example, field store guards are necessary when the constructor `Rectangle()` is compiled. Figure 8 shows the HIR for the constructor. In contrast to the method `allocate()` from the previous section, a co-allocation is not possible here: the `Rectangle` object is not allocated in the constructor, but passed in as the first method parameter.

Therefore, the two `putfield` instructions of Figure 8 are guarded. A call into the VM is emitted in front of the machine code that performs the field stores. The VM method revokes object inlining for the fields `p1` and `p2`. It is, however, very unlikely that this ever happens. The constructor `Rectangle()` was inlined in the method `allocate()`, which is the only method that allocates `Rectangle` objects. Therefore, the constructor itself is executed only when the application allocates an object using e.g. reflection.

3.6 Transition to Object Inlining

After all methods that allocate parent objects or store the field are successfully compiled, the two preconditions are satisfied for all objects that are allocated in the future. However, the heap can still contain objects that were allocated before the methods were compiled and that are therefore not yet colocated. Such objects are colocated by the garbage collector.

Therefore, it is necessary to wait for a full garbage collection that collects both generations. In this run of the mark-and-compact algorithm, parent objects are handled specially: the order of objects is changed also in the old generation if this is necessary to colocate objects. Before this collection, optimized field loads are not possible. Therefore, the optimizations described in the next section are delayed until the full collection has finished.

3.7 Optimized Field Loads

The optimization of loads for inlined fields is again performed by the just-in-time compiler. Methods that load the field and that were already compiled are compiled a second time to apply the optimization. This recompilation also eliminates the read barriers that previously counted the field loads. There are two possibilities to optimize field loads:

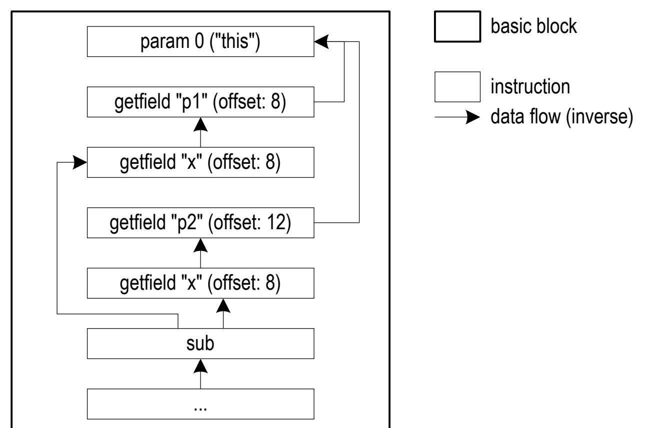

Figure 9. HIR fragment of method `area()`.

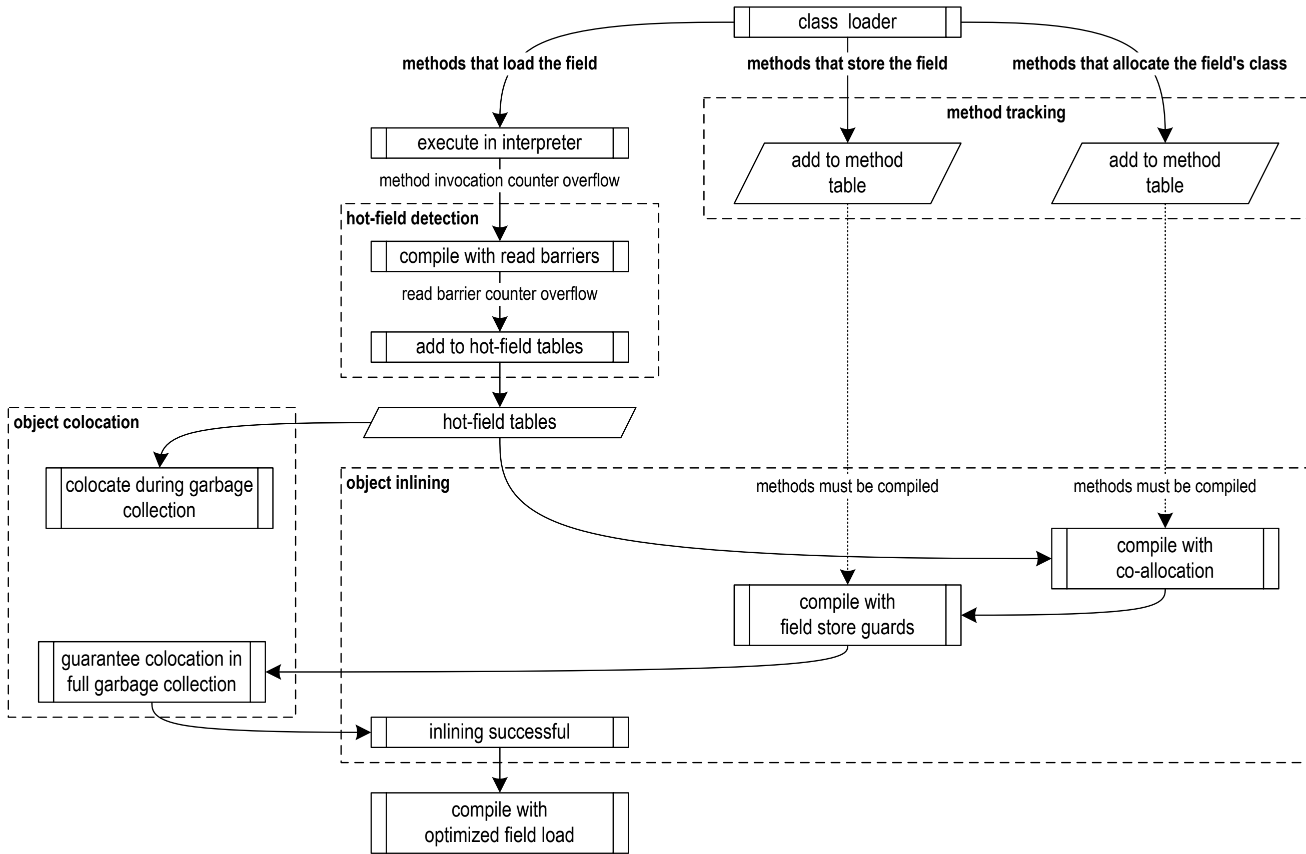

Figure 10. Summarized optimization process for object inlining of a field.

1. *Address computation*: When the address of the child object is necessary, the inline offset is added to the address of the parent object. This replaces a field load with an arithmetic instruction.
2. *Load folding*: When only a field of the child object is accessed, i.e. when the result of the inlined field load is only used as the object for another load, the two field loads can be merged into a single load with a larger offset. This eliminates one field load.

Figure 9 shows the HIR of the method `area()` that loads the inlined fields `p1` and `p2`. In this example, load folding is possible. The first two field loads can be replaced by one load with the offset $16 + 8 = 24$, which is the sum of the inline offset of `p1` and the offset of the field `x`. Analogously, the other two field loads are replaced by one load with the offset $32 + 8 = 40$. The introductory Figure 1 visualizes these offsets.

The correct handling of `null` pointers complicates the optimization. If a field load can throw a `NullPointerException`, it cannot be replaced by address arithmetic. Therefore, object inlining relies on null check elimination done beforehand by the compiler. In the example, the compiler can easily identify that the loads of `p1` and `p2` cannot cause an exception because the `this`-parameter is guaranteed to be non-`null`.

3.8 Run-Time Monitoring

The preconditions for object inlining can only be guaranteed for the currently loaded classes. Dynamic class loading can introduce new methods that allocate parent objects in such a way that co-allocation is not possible. In this case, object inlining was too optimistic and must be undone. This is achieved by deoptimizing all methods that contain an optimized load for the affected field. Fortunately, this happens rarely. Because the hot-field detection using the read barriers and the necessary compilations that guarantee the preconditions take some time, most applications have already reached a stable execution state when a field is inlined.

When examining the preconditions, we only look at methods that allocate objects or store fields using normal bytecodes. However, Java offers several other and more dynamic ways for this. Objects can be allocated and fields can be stored using *reflection* or the *Java Native Interface* (JNI). New objects are also allocated when an object is cloned using `Object.clone()`.

Because these cases are difficult to handle and rather rare, we are conservative and disable object inlining for classes that are allocated and fields that are stored by anything else than bytecodes. We instrument these subsystems so that our code is called before a class or field is accessed. If such an access affects an already inlined field, object inlining is undone using deoptimization.

3.9 Summary

Figure 10 summarizes the steps that are necessary for the successful inlining of a field. The flow chart shows the dependencies of the subsystems and the order in which methods are analyzed and compiled to guarantee the preconditions.

4. Analysis Details

This section addresses details that were omitted from the previous section for clarity. It shows how multiple inlined children, inlining hierarchies, arrays, class hierarchies and interfaces are handled by our algorithm.

4.1 Multiple Inlined Children and Inlining Hierarchies

Typical traces of frequently accessed fields involve more than one parent and child object. The example already showed a parent object that had two inlined children, but it is also possible that an object is inlined into another already inlined object.

The read barriers can detect that more than one field of a class is frequently accessed. The first field whose counter reaches the threshold is the most important one, but the other ones should also be optimized. In our example, the fields `p1` and `p2` are equally important because they are accessed with the same frequency. Inlining just one of them would be an artificial restriction.

Nevertheless, we perform their inlining in two separate steps. We allow only one pending inlining request per class at a time in order to limit the interdependencies during compilation. It would be difficult to check the preconditions of two or more fields concurrently, especially when one inlining is successful and the other one fails.

One object can be parent and child of two different inlined fields at the same time. Assume that a `Rectangle` is used to specify the bounds of a `Figure`, i.e. that `Figure` has a field `bounds` of the type `Rectangle`. In this case, it is beneficial to combine the figure, the rectangle and the two points. The Java code fragment `figure.bounds.p1.x`, which requires three field loads in the unoptimized case, can be optimized to a single field load when the fields `bounds` and `p1` are inlined.

Such inlining hierarchies do not require a special handling in our analysis. It is only required that co-allocation is possible for the whole object group, i.e. the `Figure` object, the `Rectangle` object and the `Point` objects must be allocated in the same compiled method.

4.2 Arrays

In most Java applications, array objects are as important as class instances, but they are more difficult to optimize. They can be both parent objects, i.e. an array element can point to another object, and child objects, i.e. they can be referenced by a field.

Object inlining for arrays that are parent objects is impossible in our approach. It cannot be guaranteed that objects referenced by array elements are located after the array object. The bytecode for storing an array element of a reference array, called `aastore`, does not contain any static type information. A method that contains such a bytecode, and that e.g. stores an element of an `Object` array passed in as a parameter, can modify any reference array of the heap. Array store guards similar to our field store guards are therefore not possible. In contrast, the bytecode for storing an instance field of an object, called `putfield`, contains a static symbolic reference to the field name and to the name of the class that holds the field [15].

Optimizing arrays that are child objects is similar to normal instance objects. The only difference is that the number of array elements and therefore the size of the array can vary. Therefore, we distinguish two cases:

- When the length of the array is constant at compile time, there are no differences to instance objects. Our co-allocation combines the array with the parent object, and object inlining is possible. A field load that loads the array pointer followed by an array access is folded to an array access with an additional fixed offset. Furthermore, the array bounds check is simplified because the array length is constant, which saves another memory access to load the actual array length.
- When the length of the array is not constant, we do not optimize the field. This is a restriction of the current implementation for our co-allocation, but not a general limitation of the object inlining algorithm.

4.3 Class Hierarchies and Interfaces

Class hierarchies complicate object inlining because subclass objects can be used as if they were superclass objects. Object inlining for fields of a superclass is not possible if a subclass adds additional fields. It is still possible to colocate the children of superclass objects to subclass objects, but the inlining offsets are not known statically because subclass objects are larger than superclass objects. This is the most severe limitation of our dynamic approach where no global data flow analysis is available to identify sites where only superclass objects, but no subclass objects can show up.

Assume that in our example a new class `ColoredRectangle` is added that extends `Rectangle` and adds two fields to store a foreground and a background color. Objects of the new class are 8 bytes larger than objects of the superclass. In contrast to the offsets shown in Figure 1, the first inlined point would start at offset 24, and the second at offset 40. However, a `ColoredRectangle` can be passed to all methods that expect a `Rectangle` and thus access the children with the old offsets 16 and 32, respectively. Therefore, we do not allow object inlining if the class of the parent object has subclasses with fields.

In contrast to this, subclasses of `Point`, for example a class `Point3D` that adds a z-coordinate, do not disturb object inlining. The constructor `Rectangle()` explicitly allocates `Point` objects and not `Point3D` objects. Therefore, we can allow object inlining if the class of a child object has subclasses with fields.

Interfaces do not interfere with object inlining. A field can be declared using an interface type, but during co-allocation, the field is initialized with an object of a class that implements the interface. This class is recorded and can even be used by the compiler to eliminate dynamic type checks because both the field's declared type and its dynamic type are known now.

5. Evaluation

Our object inlining algorithm is integrated into the Java HotSpot™ VM of Sun Microsystems, using the latest product version of the JDK 6 [21]. All measurements of this section were performed on an Intel Pentium D processor 830 with two cores at 3.0 GHz, running Microsoft Windows XP. Each core has a separate L1 data cache of 16 KByte and an L2-cache of 1 MByte. The cache line size is 64 bytes for both caches. The main memory of 2 GByte DDR2 RAM is shared by the two cores. We evaluate our work with the SPECjvm98 benchmark suite [19] and the SPECjbb2005 benchmark [20].

The SPECjvm98 benchmark suite[1] consists of seven benchmarks derived from real world client applications. They are executed several times until the execution time converges. According to the run rules, the slowest and the fastest runs are reported. The slowest run, which is always the first one in our case, indicates the startup speed of a JVM, while the fastest run measures the peak performance after all JVM optimizations have been applied. The speedup compared to a reference platform is reported as the metric for each benchmark, and the geometric mean of all metrics is computed. SPECjbb2005[2] emulates a client/server application. The resulting metric is the average number of transactions per second executed on a memory-resident database.

Scientific applications usually operate on large arrays that cannot be optimized using object inlining. Therefore, no performance gain can be expected for such applications. However, there should also be no slowdown due to the analysis overhead. To verify this,

[1] All SPECjvm98 results are not approved metrics, but adhere to the run rules for research use. The input size 100 was used for all measurements.

[2] All SPECjbb2005 results were valid runs according to the run rules. The measurements were performed with one JVM instance.

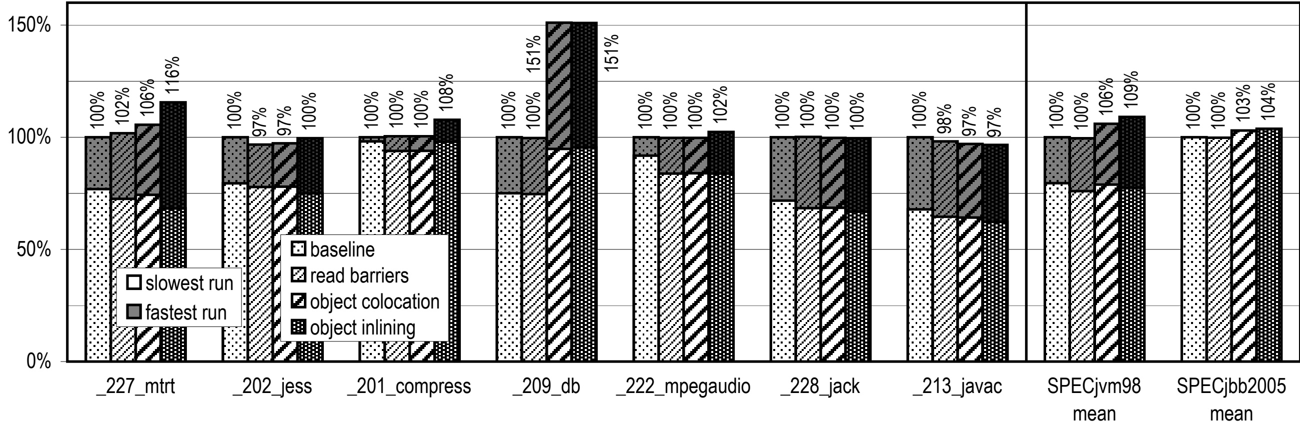

Figure 11. Benchmark results for different object inlining configurations relative to the unmodified JDK 6 (taller bars are better).

we performed all measurements also for SciMark 2.0 [16], a benchmark for scientific applications that executes several numerical kernels. All configurations mentioned in the next sections showed the same results.

5.1 Run-Time Impact of Object Inlining

Figure 11 shows how object inlining and its preceding analysis steps affect the performance of the benchmarks. We report the results for the individual benchmarks of SPECjvm98, as well as the mean of SPECjvm98 and the result of SPECjbb2005. For SPECjvm98, the slowest and the fastest runs are shown on top of each other: the gray bars refer to the fastest runs, the white bars to the slowest. Both runs are shown relative to the same baseline, i.e. the fastest run of the unmodified JDK 6.

The second column, *read barriers*, shows the overhead necessary to detect the hot fields. Read barriers have a negative effect especially on the slowest runs. First, the field counters are incremented at run time. To reduce this overhead, the read barriers are removed when they are no longer necessary by recompiling methods. This nearly eliminates the overhead for the fastest runs. The collected information is not used for any optimizations in this configuration.

The column *object colocation* shows the results when the information about hot fields is used to perform object colocation during garbage collection in order to improve the cache behavior. This has a major impact on _209_db, which accesses a large number of objects that are randomly ordered without object colocation. There is also a speedup for _227_mtrt and SPECjbb2005.

	initial			counting		colocated		inlined	
	obj	arr	scalar	obj	arr	obj	arr	obj	arr
_227_mtrt	378	77	311	55	23	11	10	1	4
_202_jess	411	77	404	71	26	16	11	2	2
_201_compress	353	76	293	21	19	7	9	1	4
_209_db	351	74	286	33	18	3	5	1	0
_222_mpegaudio	413	93	343	76	37	24	22	7	9
_228_jack	396	80	327	79	24	23	13	5	3
_213_javac	493	92	341	151	37	36	11	1	0
SPECjbb2005	1822	446	1553	199	86	49	21	2	0
empty	315	54	205	0	1	0	0	0	0

Table 1. Number of fields that are optimized.

The last column, *object inlining*, reports the final results with all of our code enabled. The benchmarks that showed a speedup with object colocation are further improved by object inlining. Additionally, _201_compress and _222_mpegaudio show a speedup. The impact on the slowest runs depends on the benchmark characteristics. If few methods need to be compiled to guarantee the preconditions, the inlining of a field succeeds early and leads to a speedup. If the additional compilation overhead is higher than the benefit of object inlining, the slowest run is negatively affected. The benchmark _213_javac is the only one that gets slower in each configuration. It is very sensitive to garbage collection time. Because object colocation and object inlining modify the garbage collector to copy groups of objects together, which adds a low but measurable overhead, the result is negatively affected.

5.2 Statistics of Inlineable Fields

Our analysis performs object inlining field-per-field at run time. Each field goes through the phases shown in Figure 4. To limit the analysis overhead, it is important that no time is wasted analyzing fields that are infrequently accessed.

Table 1 shows that detecting hot fields using read barriers acts as an effective filter. Even small Java applications load many classes. The last row of the table shows the field numbers when executing an application that consists only of an empty method `main()`. In this case, all loaded fields are part of the class library and are defined in classes that set up the infrastructure for running an application. Also before `main()` is called, Java code is executed. Two methods of the class `String` are frequently executed and therefore compiled. In these methods, read barriers for the field `String.value` are emitted. The application terminates immediately, so no field is detected as hot.

The first column *initial* shows the overall number of fields in all loaded classes. Based on the declared type, we distinguish object fields, array fields, and all other scalar types like `int` and `float`. Scalar fields are not relevant for object inlining, so they are ignored. Most of the several hundred fields are only accessed by methods that run in the interpreter. The column *counting* shows the number of fields for which read barriers are emitted in compiled code.

The read barrier counters are used to determine if a field is frequently accessed. The numbers of frequently accessed fields are shown in the column *colocated*. Only 1% to 7% of all loaded objects fields and 5% to 24% of the array fields are colocated. This is essential to keep the overhead during garbage collection low.

To guarantee the preconditions for object inlining at run time, our analysis is conservative in several cases. This is reflected by

	baseline	read barriers		obj. colocation		object inlining	
_227_mtrt	171	219	+28%	218	+27%	249	+46%
_202_jess	204	296	+45%	296	+45%	328	+61%
_201_compress	56	75	+34%	75	+34%	93	+66%
_209_db	104	150	+44%	149	+43%	151	+45%
_222_mpegaudio	171	317	+85%	316	+85%	398	+133%
_228_jack	251	362	+44%	362	+44%	410	+63%
_213_javac	641	934	+46%	928	+45%	896	+40%
SPECjbb2005	522	782	+50%	785	+50%	840	+61%
empty	2	2	+0%	2	+0%	2	+0%

Table 2. Number of methods that are compiled.

the low number of fields where object inlining is possible. For some benchmarks, we optimize the most important fields. For example, the 5 inlined fields of _227_mtrt lead to a significant speedup. The single inlined field of _209_db is also the most important one. For other benchmarks such as _202_jess and _213_javac, our analysis is too conservative, so the most frequently accessed fields cannot be inlined yet. An improved algorithm for co-allocation that can e.g. combine allocations of different basic blocks or arrays with non-constant length would improve these benchmarks.

5.3 Compilation Overhead

The just-in-time compiler plays an important role in our analysis. We use it to emit read barriers, to perform co-allocation, to guard field stores, and finally to optimize field loads. This makes it necessary to compile more methods, and also to compile some methods twice. Table 2 shows the number of methods that are compiled in the different configurations.

The removal of read barriers for already hot fields requires that the methods containing those barriers are recompiled. With the exception of _222_mpegaudio, this increases the number of compiled methods by 28% to 50%, i.e. at most every second method is compiled twice. Object colocation affects only the garbage collector, so the number of compiled methods changes insignificantly.

Object inlining is only initiated for the few fields that are frequently accessed. Therefore, the compilation overhead is modest even though multiple methods must be compiled to guarantee the colocation of a field and to optimize the field loads. This overhead is not a bottleneck of our object inlining because the just-in-time compiler is fast enough and compilation is done in the background while the application continues to run.

6. Future Work

The speedup of object inlining is directly influenced by the number of fields that are inlined. Currently, our analysis is conservative in several places, especially regarding the inlining of arrays. We plan to extend the analysis so that arrays with a non-constant length at the allocation site can also be inlined, and to weaken the precondition that an array field must not change after the allocation.

This is justified by the observation that many array fields rarely change. For example, the class `ArrayList` uses an internal array to store the list elements. This array changes only when an element is added and the capacity is not large enough. It would be beneficial to inline the data array so that a list element could be loaded without loading the array address first.

This requires a special handling of methods that enlarge the data array so that the store to the field does not invalidate inlining. The optimized load must detect the change and access the new array instead of the inlined old one.

7. Related Work

Dolby et al. implemented object inlining in a static compiler for a dialect of C++ [2, 3, 4]. The compiler offers a highly sophisticated interprocedural analysis framework. When objects are inlined, all methods that might access these objects are cloned so that methods for optimized and unoptimized objects are available. The time necessary for the analysis varies from one quarter to half of the total compilation time. The C++ benchmarks showed an average speedup of 10%, with a maximum of 50% for some benchmarks. The global data flow analysis and the high compilation time prohibit the integration of such an analysis into a virtual machine.

Laud implemented object inlining for Java in the CoSy compiler construction framework, a static compiler for Java [13]. In contrast to our implementation, this algorithm can detect when child objects are replaced with new ones. It is, however, not allowed that a child object is referenced by anything else than the parent object, e.g. by a field of another object. This is possible in our algorithm. To our knowledge, no benchmark results are published.

Lhoták et al. provide a good introduction to object inlining and illustrate four classes of inlineable fields, depending on the access pattern [14]. Using this classification, they compare the number of fields that are inlineable by the algorithms of Dolby and Laud for several Java benchmarks, including SPECjvm98. According to their terms, our algorithm can optimize fields that satisfy the rules *[contains-unique]* and *[unique-container-same-field]*, which are the same rules that are required by the algorithm of Dolby. Their results show that a significant number of the most frequently accessed fields can be inlined. However, their research is limited to object fields; array fields are ignored. We infer from our evaluation that array fields are equally important as object fields for object inlining. The study does not describe an implementation for object inlining, so no time measurements except from three hand-optimized benchmarks are published.

Veldema et al. present an algorithm for *object combining* that puts objects together that have the same lifetime [22]. It is more aggressive than object inlining because it also optimizes unrelated objects if they have the same lifetime. This allows the garbage collector to free multiple objects together. Elimination of pointer accesses is performed separately by the compiler. However, they focus on reducing the overhead of memory allocation and deallocation. This is beneficial for their system because it uses a mark-and-sweep garbage collector where these costs are higher.

Similar to our object colocation, several approaches use profiling data to guide the copying order of the garbage collector. The *online object reordering* of Huang et al. regards all fields accessed by frequently executed methods as hot [9], which is not as precise as our dynamic numbers obtained from the read barriers. The *cache-conscious data placement* of Chilimbi et al. uses a technique similar to our read barriers [1], however it does not distinguish different fields within the same object.

To create and preserve the locality of objects in Java applications, Shuf et al. co-allocate objects and then preserve this order during garbage collection [18]. Instead of detecting frequently accessed fields, they use frequently instantiated types, called *prolific types*, to guide the optimization. Similar to our co-allocation, they build parent-child relationships of objects. A modified traversing of reachable objects during garbage collection preserves this order. For co-allocation, they report speedups of up to 21% with a non-copying mark-and-sweep collector where object allocation costs are high, but they observe no speedup with a copying collector. This corresponds with our experience: co-allocation by itself is not beneficial for the Java HotSpot™ VM.

Escape analysis is another optimization that reduces the overhead of memory accesses. It detects objects that can be eliminated or allocated on the method stack. It is orthogonal to object inlining

because it optimizes short-living temporary objects, whereas object inlining optimizes long-living data structures. Kotzmann et al. implemented a new escape analysis algorithm for the Java HotSpot™ VM [11]. It is fast enough for a just-in-time compiler and handles all aspects of dynamic class loading. Similarly to our approach, they must analyze classes that are loaded after the optimization was performed. If such a class lets a previously optimized object escape its scope, all affected methods are deoptimized [12].

Ghemawat et al. use a cheap interprocedural analysis not only for object inlining, but also for other optimizations such as escape analysis [5]. The analysis was implemented in Swift, an optimizing static Java compiler for the Alpha architecture. There are no timing results where only object inlining is enabled.

In a previous paper, we introduced object colocation for the Java HotSpot™ VM [24]. It presents the code patterns for read barriers and the detailed algorithm for object colocation during garbage collection. However, the analysis did not guarantee the colocation of fields, therefore the optimization of field loads was not possible. This research extends the analysis to guarantee the preconditions of object inlining and applies it by eliminating field loads.

8. Conclusions

We presented an object inlining algorithm that optimizes field loads automatically at run time when applications are executed in the Java HotSpot™ VM. The most frequently accessed fields are identified using read barriers. If certain preconditions are satisfied, i.e. the allocation of the parent and the child object can be combined to a co-allocation and the field is stored only once during this allocation, then the field is inlined and subsequent field loads are optimized.

The optimization does not need a global data flow analysis, but builds on feedback data provided by the just-in-time compiler. All methods that can influence the inlining of a field must therefore be compiled. If the preconditions are satisfied in the currently loaded classes, object inlining is applied optimistically. Deoptimization is necessary if a precondition is invalidated later by dynamic class loading. The evaluation with several standard benchmarks showed that a significant speedup can be achieved with only optimizing a handful of fields.

Acknowledgments

We would like to thank the Java HotSpot™ compiler team at Sun Microsystems, especially Thomas Rodriguez, Kenneth Russell and David Cox, for their persistent support, for contributing many ideas and for helpful comments on all parts of the Java HotSpot™ VM. We also thank Thomas Kotzmann for his valuable comments on the work and this paper.

References

[1] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the International Symposium on Memory Management*, pages 37–48. ACM Press, 1998.

[2] J. Dolby. Automatic inline allocation of objects. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 7–17. ACM Press, 1997.

[3] J. Dolby and A. Chien. An evaluation of automatic object inline allocation techniques. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–20. ACM Press, 1998.

[4] J. Dolby and A. Chien. An automatic object inlining optimization and its evaluation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 345–357. ACM Press, 2000.

[5] 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 Language Design and Implementation*, pages 334–344. ACM Press, 2000.

[6] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™ Language Specification*. Addison-Wesley, 3rd edition, 2005.

[7] R. Griesemer and S. Mitrovic. A compiler for the Java HotSpot™ virtual machine. In L. Böszörményi, J. Gutknecht, and G. Pomberger, editors, *The School of Niklaus Wirth: The Art of Simplicity*, pages 133–152. dpunkt.verlag, 2000.

[8] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–43. ACM Press, 1992.

[9] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: improving program locality. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 69–80. ACM Press, 2004.

[10] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[11] T. Kotzmann and H. Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 111–120. ACM Press, 2005.

[12] T. Kotzmann and H. Mössenböck. Run-time support for optimizations based on escape analysis. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 49–60. IEEE Computer Society, 2007.

[13] P. Laud. Analysis for object inlining in Java. In *Proceedings of the Joses Workshop*, 2001.

[14] O. Lhoták and L. Hendren. Run-time evaluation of opportunities for object inlining in Java. *Concurrency and Computation: Practice and Experience*, 17(5-6):515–537, 2005.

[15] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[16] R. Pozo and B. Miller. *SciMark 2.0*, 1999. http://math.nist.gov/scimark2/.

[17] K. Russell and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 263–272. ACM Press, 2006.

[18] Y. Shuf, M. Gupta, H. Franke, A. Appel, and J. P. Singh. Creating and preserving locality of Java applications at allocation and garbage collection times. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 13–25. ACM Press, 2002.

[19] Standard Performance Evaluation Corporation. *The SPECjvm98 Benchmarks*, 1998. http://www.spec.org/jvm98/.

[20] Standard Performance Evaluation Corporation. *The SPECjbb2005 Benchmark*, 2005. http://www.spec.org/jbb2005/.

[21] Sun Microsystems, Inc. *Java Platform, Standard Edition 6 Source Snapshot Releases*, 2006. http://download.java.net/jdk6/.

[22] R. Veldema, C. J. H. Jacobs, R. F. H. Hofman, and H. E. Bal. Object combining: A new aggressive optimization for object intensive programs. *Concurrency and Computation: Practice and Experience*, 17(5-6):439–464, 2005.

[23] C. Wimmer and H. Mössenböck. Optimized interval splitting in a linear scan register allocator. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 132–141. ACM Press, 2005.

[24] C. Wimmer and H. Mössenböck. Automatic object colocation based on read barriers. In *Proceedings of the Joint Modular Languages Conference*, pages 326–345. LNCS 4228, Springer-Verlag, 2006.