Automatic Feedback-Directed Object Fusing

CHRISTIAN WIMMER and HANSPETER MÖSSENBÖCK
Johannes Kepler University Linz

Object fusing is an optimization that embeds certain referenced objects into their referencing object. The order of objects on the heap is changed in such a way that objects that are accessed together are placed next to each other in memory. Their offset is then fixed, i.e., the objects are *colocated*, allowing field loads to be replaced by address arithmetic. *Array fusing* specifically optimizes arrays, which are frequently used for the implementation of dynamic data structures. Therefore, the length of arrays often varies, and fields referencing such arrays have to be changed. An efficient code pattern detects these changes and allows the optimized access of such fields.

We integrated these optimizations into Sun Microsystems' Java HotSpot^{TM} VM. The analysis is performed automatically at run time, requires no actions on the part of the programmer, and supports dynamic class loading. 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: the objects must be allocated at the same time, and the field must not be overwritten later. These preconditions are checked by the just-in-time compiler to avoid an interprocedural data-flow analysis. The garbage collector ensures that groups of colocated objects are not split by copying groups as a whole. The evaluation shows that the dynamic approach successfully identifies and optimizes frequently accessed fields for several benchmarks with a low compilation and analysis overhead. It leads to a speedup of up to 76% for simple benchmarks and up to 6% for complex workloads.

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

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Java, object fusing, object inlining, object colocation, just-in-time compilation, garbage collection, optimization, cache performance

1. INTRODUCTION

Java source code [Gosling et al. 2005] is compiled to platform-independent bytecodes that are executed by a virtual machine [Lindholm and Yellin 1999]. Modern Java VMs then translate the bytecodes of frequently executed methods to optimized machine code using a just-in-time compiler. A garbage collector may move live objects to new memory locations while reclaiming unreferenced objects. Both the just-in-time compiler and the garbage collector impose a run-time overhead, but they can be used to implement novel feedback-directed optimizations [Arnold et al. 2005] inside the VM that are not possible in a static compiler.

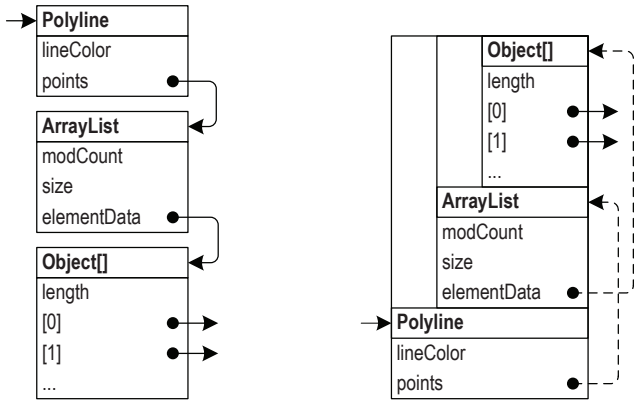
Object-oriented programming encourages developers to decompose applications into multiple classes with well-understood and well-tested functionality. This im-

This work was supported by Sun Microsystems, Inc.
Authors' addresses: C. Wimmer and H. Mössenböck, Institute for System Software, Christian Doppler Laboratory for Automated Software Engineering, Johannes Kepler University Linz, Austria; email: {wimmer, moessenboeck}@ssw.jku.at
C. Wimmer is now with the Department of Computer Science, University of California, Irvine.

Accepted for Publication in ACM Transactions on Architecture and Code Optimization

proves the code quality, but can have a negative impact on performance. It leads to a large number of objects on the heap that are linked together by field references. This increases the number of loads for referencing fields, i.e., the number of memory accesses of the application. In particular, this is evident in Java because Java does not support value objects at the language level. *Object colocation* places objects that are accessed together next to each other on the heap. When the offset between such objects is fixed, *object fusing* replaces field loads by address arithmetic. We describe two different levels of optimization: *instance fusing* deals with fields that reference class instances, and *array fusing* expands the concept to fields that reference arrays. Object colocation requires support from the garbage collector, while object fusing is performed by the just-in-time compiler.

Figure 1 illustrates the idea. A `Polyline` object uses the Java collection class `ArrayList` to maintain a dynamic list of points. Internally, the `ArrayList` stores its data in an `Object[]` array. When new points are added and the size of the array does not suffice, the array is replaced by a larger copy. The array elements reference the points that store the actual coordinates. Several memory accesses are necessary to load a single point. When the objects are scattered on the heap, the cache performance is also negatively affected. Object fusing combines the objects to a larger group so that all fields and array elements can be loaded or stored with a memory access relative to the beginning of the `Polyline` object.



(a) Unoptimized objects (b) After object fusing

Fig. 1. Motivating example for our optimization approach

To allow `Polyline` to have subclasses with additional fields, we place the fused objects in front of the `Polyline` object. This complicates the access of fused arrays, but we present an efficient code pattern to handle this case. Our solution addresses the time overhead of field accesses, but not the space overhead. We keep all headers of objects and require the duplication of the array length, which increases the memory footprint. The memory layout shown in Figure 1 reflects our actual implementation. It is not necessary to use this layout for object fusing, so other approaches can use different layouts as long as objects are combined on the heap.

In programming languages like C++ [ISO/IEC 2003], the merging of objects can be performed by the programmer using value objects. However, this is error-prone because reference objects and value objects have different semantics, e.g., when variables are assigned. Fusing arrays whose size can change at run time, such as the `Object[]` array in our example, is not possible at all in C++ because it is necessary to move objects on the heap. Additionally, value objects are incompatible with polymorphism and dynamic dispatch.

Implementing object fusing as a feedback-directed optimization at run time has several advantages over a static compile-time optimization. First, the optimization can be applied optimistically, i.e., it is possible to revoke the optimization if a precondition is invalidated later on. This also simplifies the data-flow analysis that checks the preconditions. Secondly, modular applications that are compiled, deployed, and installed as several parts are optimized as a whole. There is no distinction between application classes and library classes such as collections, i.e., an object of a library class can be fused with an application object and vice versa. Code that is loaded dynamically or even generated at run time is also optimized. Thirdly, code that exists in libraries but is never actually executed does not affect the optimization because unused classes are not loaded into the VM.

This paper presents results for an implementation that is integrated into Sun Microsystems' Java HotSpot™ VM. The approach, however, can be generally applied to all languages that are executed using a virtual machine with just-in-time compilation and garbage collection, such as the *common intermediate language* [ISO/IEC 2006] that is part of the standardized *common language infrastructure* and used, for example, by the Microsoft .NET framework.

To the best of our knowledge, our approach is the first that applies object fusing at run time in a virtual machine without requiring actions on the part of the programmer. Additionally, we are not aware of any system that allows fusing of arrays whose length can change at run time. This paper summarizes the algorithms that we presented in earlier publications [Wimmer and Mössenböck 2006; 2007; 2008] and illustrates them with an example. In addition to these previous publications, this paper contributes the following novel parts:

—We handle class hierarchies by reversing the fusing order such that field offsets are constant even if a parent class has subclasses.

—We present a code pattern for the optimized access of fused arrays that supports the reverse object order. The address arithmetic for the array access is integrated into the array bounds check.

—We evaluate our implementation, which is integrated into a production-quality Java VM, using several standard benchmarks and report results for different configurations of object colocation, instance fusing, and array fusing. It leads to a speedup of up to 76% for simple benchmarks of the SPECjvm98 benchmark suite (up to 30% when not considering the db benchmark, which is commonly considered as a too easy target for optimizations), as well as up to 6% for complex workloads of the DaCapo benchmark suite.

2. SYSTEM OVERVIEW

Object fusing operates on groups of heap objects that are in a parent-child relationship. A *fusing parent* contains a reference to the *fusing child*. A child has exactly one fusing parent. A parent can have references to multiple children, i.e., it is possible to fuse more than one child with a single parent. Additionally, fusing hierarchies can exist, i.e., a fusing child can in turn be another fusing parent. Consequently, a fusing parent, its direct children, and all its indirect children form a single group of objects. In the example shown in Figure 1, the `ArrayList` object is both the child of the `Polyline` object and the parent of the `Object[]` array.

The reference from the fusing parent to the fusing child is a field declared in the class of the fusing parent. Such a field is called a *fused field*. Based on the declared type of the field, we distinguish between a field referencing a class instance (we write only *instance* for abbreviation) and referencing an *array*. An *optimized field load* of a fused field uses address arithmetic to yield the same result as a normal field load, i.e., it does not require a memory access.

In previous publications, we referred to *object fusing* as *object inlining* [Wimmer and Mössenböck 2007; 2008; Wimmer 2008]. The term *inlining* is however not precise for our approach because the parent and its children remain separate entities. They are only located in a certain defined order so that field accesses can be eliminated.

2.1 Design Principles

Our object fusing system is based on the following design principles. We believe that many of these principles are important not only for object fusing, but are generally applicable for feedback-directed optimizations inside a Java virtual machine.

—*Automatic*: The optimization neither requires any actions on the part of the programmer nor any special tools at compile time or deployment time. All analysis and optimization steps are performed automatically at run time.

—*Dynamic*: We fully support dynamic class loading because it is a key feature of modular Java applications.

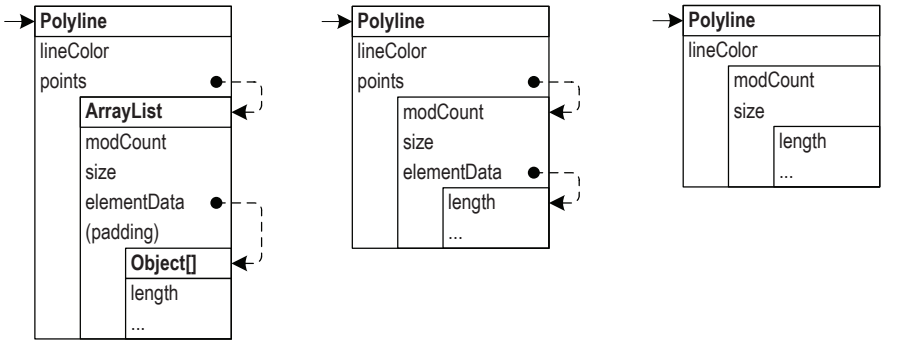
—*Feedback-directed*: To decide which fields should be optimized, we collect profiling data at run time using read barriers that increment field access counters.

—*No interprocedural data-flow analysis*: An interprocedural analysis, e.g., building a call graph, is complicated in Java because most methods are dynamically bound and new classes can be loaded at a later time. Instead, we perform only intraprocedural analyses of methods.

—*Optimization on a per-class basis*: All analysis and optimization steps operate on a per-class basis, i.e., either all or no instances of a certain class are optimized. This reduces the overhead as it is not necessary to distinguish between optimized and unoptimized instances of the same class.

2.2 Memory Layout

When objects are grouped together by object fusing, the header for child objects can be either eliminated or preserved. Similarly, the alignment of child objects is optional. Figure 2(a) and Figure 2(b) illustrate the differences between these two



(a) Unmodified layout (b) Without headers (c) Without headers and fields

Fig. 2. Possible memory layouts for fused objects

cases. We assume that the header size and the object alignment are both 8 bytes and that a pointer requires 4 bytes, which are the usual numbers on 32-bit virtual machines. On 64-bit VMs, the header size and pointer size are even bigger. Eliminating the object headers of the two child objects saves 16 bytes. Eliminating the alignment padding of the `ArrayList` object saves another 4 bytes, so the optimized group of objects is 20 bytes smaller than the original objects.

When the children do not conform to the standard object layout, there is no reason to keep the internal pointers to them. Removing the fields `points` and `elementData` reduces the size of the object group by additional 8 bytes. In total, object fusing can save 28 bytes of memory for each `Polyline` object group. Figure 2(c) shows the resulting memory layout. Reducing the object size improves the cache behavior and reduces the pressure on the garbage collector. Nevertheless, we use the unmodified layout for object fusing because it has several advantages:

—*Object locking*: The object header is used for synchronization. If the header of a child object were removed, we would have to guarantee that the object is never used for synchronization.

—*Side pointers to children*: External references to the child object, e.g., from other objects, expect the header to be there. Removing the object header would change the field offsets and thus disallow such references.

—*No new kind of heap elements*: The elimination of object headers and fields would lead to a new kind of heap elements that are a mixture of instances and arrays. For example, the `Polyline` object would consist of an instance part with three fields and an array part whose array length is stored at the unusual offset 20.

—*No change of unoptimized field loads*: If the pointer between parent and child were removed, it would be necessary to distinguish between normal and fused fields also in code that is rarely executed and thus not worth the optimization.

—*Smooth transition to optimized code*: Our system performs the optimization steps asynchronously. Changing the object layout would require a distinct transition phase where all affected objects are rewritten and the accessing methods are transformed.

—*Smooth deoptimization*: A newly loaded class can invalidate preconditions of already optimized fields. It would be necessary to restore the removed object headers and field pointers when object fusing is revoked.

—*Support for reverse object fusing*: To support subclasses of the parent object's class, we place fusing children in front of the parent (our actual memory layout is shown in Figure 1). Removing the headers of child objects would lead to objects that do not start with a header.

—*Support for dynamic array fusing*: For fields referencing arrays where the field is changed at run time, fusing requires a pointer to the newly allocated (resized) array. The array is accessed via this pointer until the next run of the garbage collector fuses the parent and the array again.

In summary, we believe that changing the memory layout is not compatible with our dynamic fusing approach. In order to reduce the object size at run time, it would be necessary to have a distinct transition phase where objects are rewritten, leading to a more static approach of object fusing that we do not explore here.

2.3 Preconditions for a Field

Before loads of a certain field can be replaced by address arithmetic, it must be guaranteed that all fusing parents containing this field are correctly colocated with their children, i.e., the fused field must always point to a location that can also be computed by address arithmetic. We define the following preconditions that a field must satisfy for a safe application of object fusing:

(1) The parent and all of its fusing children must be allocated at the same time, and the field stores that install references to the children into the parent must occur immediately after the allocation.

(2) The field referencing a child must not be modified after allocation. If the field were overwritten later with a new value, the new object would not be colocated to the parent and an optimized field load would access the old child.

The second precondition is necessary because detecting that a field has been changed is equally or even more expensive than the normal unoptimized field access. A check whether the field has been changed would be necessary before each load, which would require a read barrier consisting of at least two machine instructions to guard an optimized field load. However, object fusing is only beneficial if a fused object can be accessed via address arithmetic without further checks. For references to arrays, this constraint can be relaxed. It is possible to integrate the check for a changed field into the array bounds check (see Section 5), which is required by the Java language specification. The modified bounds check does not need additional machine instructions in the common case.

2.4 Optimization Phases

To detect whether a field is worth being optimized and to guarantee the preconditions for object fusing, each field runs through the optimization phases shown in Figure 3. Assume that the field `f` links a parent object of the class `P` to a child object of class `C`.

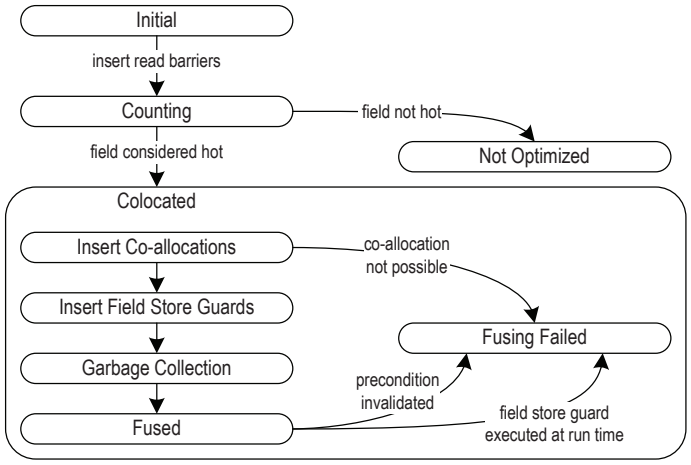


Fig. 3. Optimization phases for a field

—*Counting*: In order to collect a field access profile, *read barriers*, i.e., increments of a per-field and per-class counter, are inserted for all loads of `f`. If the field counter exceeds a certain threshold, the field is considered to be hot and is entered into the *hot-field table* of `P`.

—*Colocated*: When the garbage collector moves objects, it processes a `P` object and its `C` object as a group so that the objects are placed next to each other. This ensures that objects that were placed next to each other during allocation are still next to each other after garbage collection.

—*Insert co-allocations*: To prohibit `P` objects that are not followed by a `C` object, methods that allocate a `P` object must be transformed. We combine the allocation of `P`, the allocation of `C`, and the field store of `f` to a single 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 fusing of `f` fails. This includes situations where `C` is not allocated in all cases, i.e., where `f` can be `null` for some code paths. Co-allocation guarantees the first precondition for object fusing.

—*Insert field store guards*: Field stores that modify `f` after the co-allocation are not allowed, so guards are inserted in front of such field stores. When a guard is executed later, object fusing of `f` fails and the optimization of field loads must be undone. This guarantees the second precondition for object fusing. For references to arrays that are allowed to change, the field store guard marks the old array as invalid so that it is no longer accessed.

—*Garbage collection*: After co-allocations and field store guards have been inserted, the two preconditions are satisfied for all objects that will be allocated in the future. However, the heap can still contain objects that were previously allocated. Such objects are colocated by the garbage collector.

—*Fused*: When the two preconditions are satisfied, loads of `f` are optimized, i.e., the memory access is replaced by address arithmetic. Dynamic type checks can be optimized because the exact type of the object referenced by `f` is known.

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

The list of methods where co-allocation and field store guards are needed is available from the *method table*, a table filled by the class loader. It maps class names to methods that allocate instances of the class, as well as field names to methods that modify the field. The preconditions for object fusing can only be guaranteed for the currently loaded classes. Dynamic class loading can introduce additional methods that allocate parent objects in such a way that co-allocation is not possible. In this case, object fusing has been too optimistic and must be undone by falling back to a version of the methods where field loads are not optimized.

When examining the preconditions, we consider only 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 modified using *reflection* or the *Java Native Interface*. Because these cases are difficult to handle and rather rare, we are conservative and disable fusing for all fields affected by such operations.

2.5 Example

Figure 4 shows the Java source code for the example introduced in Figure 1, which is used as the running example in this paper. The class `Polyline` uses the collection class `ArrayList` of the package `java.util` to manage a list of points. The source code of the class `ArrayList` is reduced to the parts relevant for the example. It uses an `Object[]` array for the data, referenced by the field `elementData`. The field `size` contains the number of array elements currently in use. When a new element is added using the method `add()` and the capacity of the array does not suffice, the array is replaced by a larger copy, i.e., the field `elementData` is changed.

The field `points` of the class `Polyline` is declared using the interface type `List` instead of the implementation class `ArrayList`. This allows the list implementation to be exchanged by modifying only one line of the source code. For our object fusing algorithm, the declared type is irrelevant and only the implementation class is considered. The generic parameter `<Point>` improves the type safety of the Java code, but the `ArrayList` still uses an `Object[]` array.

2.6 The Java HotSpot™ VM

Our implementation is based on the Java HotSpot™ VM of Sun Microsystems [Sun Microsystems, Inc. 2009]. The default configuration for interactive desktop applications uses a fast just-in-time compiler, called the *client compiler* [Griesemer and Mitrovic 2000; Kotzmann et al. 2008], and a generational garbage collector with two generations. It is available for Intel's IA-32 and Sun's SPARC architecture, but object fusing is currently only implemented for the IA-32 architecture.

```
class Polyline {
  int lineColor;
  List<Point> points;

  Polyline() {
    points = new ArrayList<Point>();
  }

  int getNumPoints() {
    return points.size();

  }

  Point getPoint(int index) {
    return points.get(index);
  }

  void addPoint(Point newPoint) {
    points.add(newPoint);
  }
}

class Test {
  void allocate() {
    Polyline poly = new Polyline();
    // Do something with poly
  }
}
```

```
class ArrayList<E> implements List<E> {
  int modCount, size;
  Object[] elementData;

  ArrayList() {
    elementData = new Object[10];
  }

  int size() {
    return size;
  }

  E get(int index) {
    if (index >= size) throw new ...
    return (E) elementData[index];
  }

  void add(E e) {
    modCount++;
    if (size+1 > elementData.length) {
      int newCapacity = ...
      elementData = Arrays.copyOf(
            elementData, newCapacity);
    }
    elementData[size++] = e;
  }
}
```

Fig. 4. Java source code of the example classes

After the bytecodes of a method have been loaded by the class loader, they begin execution in 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 high-level intermediate representation (HIR) in static single assignment (SSA) form [Cytron et al. 1991] with explicit data-flow and control-flow graphs for global optimizations, and a low-level intermediate representation (LIR) for linear scan register allocation [Wimmer and Mössenböck 2005]. Compilation is done in the background while the method continues to run in the interpreter. The compiler performs aggressive optimizations such as inlining of dynamically bound methods. If an optimization is invalidated later, e.g., because of dynamic class loading, the VM can *deoptimize* [Hölzle et al. 1992] the machine code and continue the execution of the current method in the interpreter.

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 [Jones and Lins 1996]. All objects have a header of 8 bytes and are aligned at 8-byte boundaries.

The structure of the Java HotSpot^{TM} VM is similar to most modern VM implementations, even though the just-in-time compilers and the garbage collection algorithms vary greatly. Our implementation does not rely on special features of the compiler, so the algorithms presented in Section 4 and Section 5 can be integrated

```
tid__instruction_____________________________________________
a1   parameter this
a2   a1._12  // Polyline.points
     read barrier: inc counter at 5000h
i3   a2._12  // ArrayList.size
i4   return i3
```

Fig. 5. Read barrier in method `Polyline.getNumPoints()`

into any just-in-time compiler. The ideas presented in Section 3 can be generalized to any garbage collector that moves objects during collection. The only feature of the Java HotSpot™ VM that is not common in other VMs is deoptimization. Deoptimization is a simple and convenient way to undo optimistic optimizations when preconditions of the optimizations change. It can be emulated using code duplication and code patching, however deoptimization simplifies all kinds of aggressive compiler optimizations.

3. OBJECT COLOCATION

Object colocation places related objects next to each other on the heap so that spatial locality is improved. In order to know which objects should be colocated, we detect fields that are frequently loaded and optimize only objects connected by such fields. Read barriers are used to collect the field access statistics.

3.1 Read Barriers

The read barriers are handled by the compiler. When it generates code for loading a reference field, it emits a read barrier that increments a counter. The compiler instruction for a field load contains the class that declares the field, the field offset, and the type of the field. With this information, a unique per-field and per-class counter is created, and the counter address is embedded into the machine code. The low number of field accesses performed by the interpreter are not counted.

Figure 5 shows the high-level intermediate representation (HIR) of the client compiler for the method `getNumPoints()`. The method `size()` is inlined, so the loads of the two fields `points` and `size` end up in the same method. Each instruction has a type t (a for object and i for integer) and a unique id number. For example, the first instruction of Figure 5 loads the `this` parameter into the value `a1` (an object with the id 1).

The field load instruction `a2` loads the field at offset 12 from the object `a1`, i.e., the field `Polyline.points`. When code is generated for this field load, a counter increment is emitted. Assume that the counter for this field is located at the address `5000h`. No read barrier is needed for the field load `i3` because it loads a field of a primitive type, which is not of interest for object fusing.

3.2 Hot-Field Tables

The field counters of all read barrier entries are checked at regular intervals. If a counter exceeds the threshold explained below, the field is considered hot and recorded in the hot-field table of the parent class. If the counter does not cross the threshold in several successive measurement intervals, the field is considered unimportant and ignored in all further optimization steps. In most cases, we use

the time between two garbage collections as the measurement interval. Only if this timeframe is too long, we check the counters using timer interrupts.

As a heuristic, a field is considered hot if it accounts for more than 5% of all field loads within a single interval. This mechanism fills the tables iteratively. When processing the counters for the first time, fields with an exceptionally high access frequency are added to the hot-field tables. Their read barrier counters are then deactivated and ignored when computing the percentages in successive runs of the algorithm, so the next fields with still a high access frequency are added. This is repeated until a stable state is reached in which most fields have similar access frequencies, i.e., no single access frequency is above 5%.

Incrementing a counter for each field load involves a run-time overhead. Therefore, it is necessary to remove read barriers as soon as they are no longer needed, i.e., when it is known that a field is either hot or when the access count has been low for a long time. This is done by recompiling all methods that increment the counter of the read barrier. No counter increments are emitted in the new code. Because of this pattern, read barriers do not have an impact on the peak performance, but only affect the startup time of an application. They produce field access statistics of a sufficient quality to guide our optimizations, so it is not necessary to use more advanced but also more complex profiling techniques like edge profiling and path profiling [Ball et al. 1998].

The hot-field tables are a VM-global data structure with a table for every class that has hot fields. This table is registered in the class descriptor. The table stores a list of entries for the hot fields of the class. Each entry holds the offset of the field as well as the field's declared type, which points to another class descriptor. The entries are sorted in decreasing field-access frequency. This is automatically accomplished by the iterative processing algorithm for the field counters that detects hot fields in the order of their importance.

For array classes such as `Object[]`, the marker value -1 is stored instead of the field offset. A single read barrier counter is used for all elements of an array. For instance classes, a hot-field table does not contain entries for fields declared in a superclass or a subclass. Instead, these classes have their own hot-field tables. Classes without frequently accessed reference fields do not have a hot-field table.

3.3 Modifications of the Garbage Collector

Object colocation is an optimization that groups heap objects together and sorts them so that their order in memory matches their access order in the program. Our implementation uses the information in the hot-field tables to adapt the order of objects in the garbage collector. Object colocation has two goals:

(1) *Improve the cache behavior*: If objects that are accessed together are placed next to each other on the heap, the spatial locality is improved. It is more likely that objects end up in the same cache line. This is a statistical optimization, so a small ratio of unoptimized objects for a certain class does not impact the performance significantly.

(2) *Guarantee preconditions for object fusing*: Memory loads can be replaced by address arithmetic only if the colocation of a parent with its children is guaranteed for all instances of the parent class. If a single parent object cannot

be colocated with its children, the optimized access for all objects of this class must be revoked.

To achieve both goals, we distinguish between fields that *should* be colocated to improve the cache behavior and fields that *must* be colocated for object fusing. Such fused fields are preferred if not all hot fields can be optimized.

We use a modified 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, they are moved together and the grouping is established. The object groups are also promoted as a whole to the old generation if they survive a certain number of copying cycles.

The mark-and-compact algorithm is used to collect the entire heap. Object colocation in the stop-and-copy algorithm also affects the old generation because colocated objects are promoted together, i.e., they end up colocated in the old generation. The basic mark-and-compact algorithm does not change the order of objects and therefore preserves this optimized order. Therefore, modifications of the mark-and-compact algorithm are only necessary for the second goal of our object colocation algorithm: guaranteeing the preconditions for object fusing. Before optimized field loads are possible, one full garbage collection cycle is necessary to colocate objects that were promoted before the field was detected as hot, i.e., before object colocation was activated in the stop-and-copy algorithm (see Section 4.4). This run of the mark-and-compact algorithm can change the object order to establish the colocation.

4. INSTANCE FUSING

Object fusing is an optimization that replaces memory loads by address arithmetic when a field of an object is known to point to another object that is colocated with the first one. This section deals only with class instances (instance fusing), while the Section 5 extends the concepts to arrays. Section 2.1 defined two preconditions for optimizing a field. We use the just-in-time compiler to guarantee these preconditions. Methods that are relevant for a field are compiled with additional compiler phases.

4.1 Object Layout

Intuitively, one would place a child so that it immediately follows its parent in memory. However, this order is not possible if the parent's class has subclasses. In this case, the child would end up at different offsets from the parent, depending on how many additional fields the subclasses have. Conceptually, fusing also adds fields to a parent, namely the fields of the children. All field offsets must be compile-time constants, otherwise field accesses would be inefficient. Using the intuitive object order, fields of subclasses and fields of fused objects would compete for the same offsets.

Figure 6 illustrates the problem. Assume that the class `Polygon` is a subclass of `Polyline` that adds the field `fillColor`, i.e., instances of the subclass are 8 bytes larger. If the intuitive object order were used, this would affect the offsets of the fused children. For example, the increased size of `Polygon` objects would change

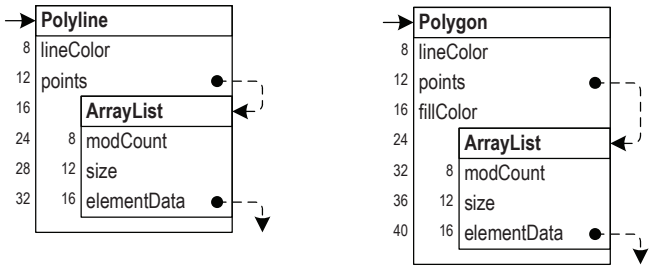


Fig. 6. Inconsistent fusing offsets with class hierarchies

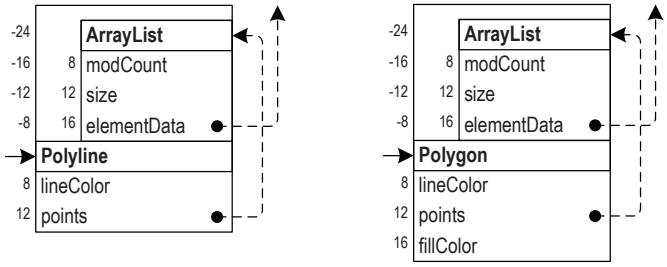


Fig. 7. Reverse object order to support class hierarchies

the offset for the optimized field access of the field `size` from 28 to 36. Since a variable of type `Polyline` can also refer to a `Polygon` object, the offset would no longer be fixed.

We solve this problem by reversing the object order, i.e., we place the child objects in front of the parent object on the heap. With this reverse order, the offsets of the children become fixed negative values. Figure 7 shows the offsets used to access fields of fused objects. For example, the optimized field access of the field `size` uses the offset -12, which is equal for `Polyline` and `Polygon` objects.

Subclasses of a child object's class do not incur problems because the exact class is known from the co-allocation. If different co-allocation sites allocate different classes of child objects, our optimization is not possible.

The reverse order affects object colocation slightly. It changes the order of objects on the heap and therefore the cache performance. However, our measurements indicate that no order is particularly better than the other—depending on the benchmark characteristics, reverse order either degrades or improves the performance. Modern processors that perform automatic memory prefetching can also handle memory access patterns in both directions [Hegde 2008].

4.2 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 following their allocation. For object fusing, however, the first precondition requires that the objects are already consecutive immediately after their allocation. Therefore, co-allocation combines the allocations for a group of objects, and object colocation ensures that

```
tid__instruction_____________________________________________
a1   new Polyline
a7   co-allocation of a1, a2, a4
a2   new ArrayList
i3   10
a4   new Object[i3]
a5   a2._16 := a4  // ArrayList.elementData
a6   a1._8 := a2   // Polyline.points
// Do something with a1
```

Fig. 8. Co-allocation in method `Test.allocate()`

the groups are not separated during garbage collection. Our co-allocation is integrated into the just-in-time compiler. Therefore, all methods that allocate parent objects must be compiled. The list of methods is retrieved from the method table. In our example, the method `Test.allocate()` must be compiled for the fusing of the field `points`, and the method `Polyline.<init>()` must be compiled for the fusing of the field `elementData`.

To detect all allocation instructions that are applicable for co-allocation, we iterate over the field store instructions of a method because field stores connect the allocations. A field store instruction has a reference to two other instructions as its parameters: the object that is modified and the new value of the field. Co-allocation is possible if the object and the value are both allocation instructions, i.e., both refer to objects allocated within the same compiled method. Additionally, the resulting structure must be acyclic, a child object must only have one parent, and the field store must be executed in all possible code paths.

A new HIR instruction for co-allocation is inserted after the first allocation. For this instruction, the back end of the compiler generates LIR operations that allocate a single chunk of memory at once. The original allocation instructions are omitted.

Figure 8 shows the HIR for the method `Test.allocate()`. The instructions refer to the state after all constructors were fused, so there are no more method calls. The allocations of `Polyline`, `ArrayList`, and `Object[]` are in the same method, together with the field stores that install the fusing children into the parent objects. The single co-allocation instruction `a7` is inserted by the additional compiler phase. It allocates only one chunk of memory large enough for all objects and then installs the object headers and field pointers appropriately.

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

4.3 Guards for Field Stores

The second precondition for object fusing specifies that a field referencing an instance must not be modified after co-allocation. Therefore, we compile all methods that modify the fused field and instrument the field store to revoke object fusing 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 be too conservative

```
tid__instruction_____________________________________________
a1   parameter this
a2   new ArrayList
a8   co-allocation of a2, a4
i3   10
a4   new Object[i3]
a5   a2._16 := a4  // ArrayList.elementData
a6   field store guard: revoke fusing of Polyline.points
     a1._8 := a2   // Polyline.points
v7   return
```

Fig. 9. Field store guard in constructor `Polyline.<init>()`

because field stores for fused fields are allowed as long as they are not executed. The static check would inhibit object fusing for all fields that are assigned inside a constructor. Even though constructors are mostly inlined into the allocating method, they also remain as distinct methods and are separately compiled. Because initializing fields in the constructor is a recommended and frequently used code pattern in Java, the static check would lead to nearly no fields that can be fused.

In our example, field store guards are necessary for the field `points` when the constructor `Polyline.<init>()` is compiled. Figure 9 shows the HIR of the constructor. In contrast to Figure 8 of the previous section, the `ArrayList` object cannot be co-allocated with the `Polyline` object because the `Polyline` is not allocated inside the method, but passed in as the `this` pointer in instruction `a1`. Co-allocation is only performed for the `ArrayList` object and its child `Object[]` array, connected by the field store `a5`.

Because it modifies the field `points`, the field store instruction `a6` must be guarded. A call into the VM is emitted in front of the machine code that performs the field store, i.e., just before the `move` instruction. The VM method revokes object fusing for the field `points`. It is, however, unlikely that this revocation ever happens. The constructor `Polyline.<init>()` was inlined in the method `Test.allocate()`, which is the only method that allocates `Polyline` objects in the bytecodes. Therefore, the constructor itself is only executed if the application allocates a `Polyline` object using reflection or the Java Native Interface.

The field `elementData` of the class `ArrayList` is modified by two methods: `ArrayList.<init>()` and `ArrayList.add()`. Both methods are compiled with field store guards. Because the field `elementData` references an array and is thus allowed to change, the guards have different semantics. They do not revoke object fusing, but mark the old array as inaccessible before the field is overwritten with the pointer to the new array. The optimized array load checks this mark so that the old array is no longer accessed (see Section 5.1). The guard inserted into `ArrayList.add()` is likely to be executed several times because this method increases the capacity of the `ArrayList`.

4.4 Transition to Object Fusing

After all methods that allocate parent objects or store to a fused field are successfully compiled, the two preconditions are satisfied for all objects that will be allocated in the future. However, the heap can still contain objects that were pre-

viously allocated and that are not colocated yet. Such objects become colocated by the garbage collector. Because the entire heap must be processed, a run of the mark-and-compact algorithm is necessary. In this run, the order of objects is changed in the old generation when it is necessary to colocate objects. The optimizations described in the next section are delayed until the full collection has completed.

Our entire optimization system is completely thread safe: it supports multi-threaded applications as well as optimization and deoptimization concurrent with running application threads. Object colocation in the garbage collector does not raise any concurrency issues because all application threads are already stopped during garbage collection. The just-in-time compiler runs concurrent with the application threads, so it is only necessary to wait until all relevant methods have been compiled with co-allocation and field store guards before the next optimization phase is started. The existing infrastructure of the Java HotSpot^{TM} VM ensures that other thread-critical operations are correctly synchronized. For example, revoking object fusing implies that some previously compiled methods must no longer be executed. This is achieved by patching the entry points and call sites of these methods, which requires that all application threads are suspended for a short time.

4.5 Optimized Field Loads

When the preconditions for a field are satisfied, loads of the field can be replaced by address arithmetic. This is performed by the just-in-time compiler. Optimizing field loads in the interpreter does not pay off because because the interpretation overhead is much higher than the possible gain of optimized field loads. In contrast to the previous sections, it is also not necessary to compile all methods that load a fused field. Only frequently executed methods that load the field are optimized, i.e., methods that were previously compiled. There are two possible ways to optimize field loads: load folding and address computation.

In many cases, a field load that yields a child object is immediately followed by a field access of the child. Load folding merges the two memory accesses. The resulting access uses a different offset, which is the memory distance between the parent and the child plus the offset of the second field access. Figure 10 shows load folding for the example method `Polyline.getNumPoints()`. The unoptimized HIR contains the two field load instructions. Load folding eliminates the first field load. The address of the `ArrayList` object is never explicitly present. Instead, the field `ArrayList.size` is accessed relative to the `Polyline` object. The combined field offset is -12, as visualized in Figure 7.

Load folding benefits from method inlining. In the example, the two field loads would be in different methods without method inlining. The second field access can be a load or store of any type. Load folding can merge a load of a fused field and a store into the child object to a single store with a larger offset. Similarly, more than two field accesses can be folded to a single one when fusing children are nested, i.e., when an indirect child of a parent object is accessed.

If the address of the child object is needed as an explicit value, the load of the fused field cannot be eliminated, but the memory access can be replaced by address arithmetic. The distance between the parent and the child in memory is added to

```
tid__instruction_____________________
a1   parameter this
a2   a1._12  // Polyline.points
i3   a2._12  // ArrayList.size
i4   return i3
```

(a) Unoptimized HIR

```
tid__instruction_____________________
a1   parameter this
i3   (a1-24)._12  // ArrayList.size
i4   return i3
```

(b) Optimized HIR

Fig. 10. Load folding in method `Polyline.getNumPoints()`

```
tid__instruction_____________________
a1   parameter this
i2   parameter index
a3   a1._12  // Polyline.points
a4   a3.invokeinterface(i2) List.get
a6   return a4
```

(a) Unoptimized HIR

```
tid__instruction_____________________
a1   parameter this
i2   parameter index
a3   (a1-24)  // Polyline.points
a4   a3.invokestatic(i2) ArrayList.get
a6   return a4
```

(b) Optimized HIR

Fig. 11. Address computation in method `Polyline.getPoint()`

the address of the parent. Figure 11 shows the address computation for the example method `Polyline.getPoint()`. The load of the fused field `Polyline.points` yields an `ArrayList` object, which is used as the receiver object in a non-inlined method call. In the optimized HIR, the load is replaced by the address computation `a3 = a1 - 24`. The total number of executed instructions is not reduced, but nevertheless one memory load is eliminated. From a technical point of view, load folding could also be considered as a combination of address computation and constant folding.

4.6 Usage of Static Type Information

The analysis for object fusing increases the amount of static type information for fused fields. The co-allocations guarantee that a fused field is initialized with a child of the same type in all parent objects, and the guarded field stores ensure that this field is never changed later on. Therefore, the dynamic type of the field is known, which is more precise than the declared type defined at compile time in the Java bytecodes.

In our example, the field `points` of the class `Polyline` has the declared type `List`, which is an interface of the Java collections library. The just-in-time compiler does not know the implementation class that is actually used. Therefore, a virtual call of the interface method `List.get()` is necessary in the example of Figure 11(a). Co-allocation discovers that this field is always initialized with an `ArrayList` object. The compiler can use this information and eliminate the overhead of dynamic binding or even inline the method. In the optimized HIR of Figure 11(b), the virtual call is replaced by a static call.

5. ARRAY FUSING

Java integrates array types smoothly into the object class hierarchy. Arrays are considered as objects and inherit from the common base class `Object`. However,

there are certain differences between class instances and arrays that affect fusing, such as the variable size of arrays.

Reference arrays contain pointers to other instances or arrays. Therefore, it would be beneficial to combine an array with the objects that are referenced by the array elements, i.e., to have arrays act as fusing parents. However, this is not possible without an interprocedural data-flow analysis because the bytecodes for array accesses do not contain any static type information. A concept similar to our field store guards is not possible for arrays [Wimmer and Mössenböck 2008]. Because of this, we allow arrays only as fusing children.

5.1 Basic Principle

The preconditions for instance fusing ensure that a field references the same instance throughout the whole lifetime of the parent. The class of the fusing child and therefore its size is a compile-time constant. The field is not allowed to change because the new child would no longer be colocated and the change of fields referencing instances cannot be detected efficiently because it would require a read barrier before each optimized field load.

The basic algorithm for instance fusing can also be applied for array fusing, but there are also certain differences. On the one hand, the size of an array cannot be determined at compile time in many cases, which complicates array fusing. On the other hand, it is possible to integrate the check whether a field referencing an array has been changed into an array access with no additional costs by embedding it into the array bounds check. We use the following approach to allow array fusing:

—At allocation, the child array with the initial length is co-allocated with the parent instance, so an optimized access is possible. It is not necessary that the length be a compile-time constant.

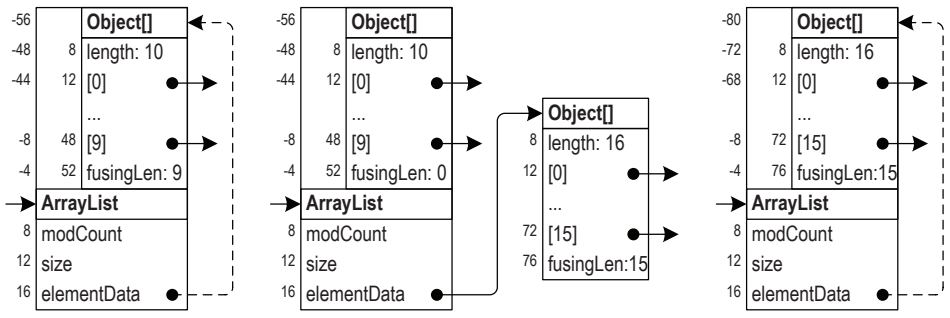
—After the field has been overwritten with a reference to a new array, an optimized access using address arithmetic is no longer possible because it would still access the old array.

—The next garbage collection colocates the new array to the parent. Therefore, an optimized access is again possible.

Only one field referencing an array with variable length can be present in a group of optimized objects. Additionally, the array must be the last element of the group. If an array with variable length were located between two instances in a group, the offsets for the second instance would no longer be compile-time constants.

Figure 12 illustrates the approach using the `ArrayList` object and the `Object[]` array. The field `elementData` is changed by the method `ArrayList.add()`: the old array with the length 10 is replaced by a larger copy with the length 16. Because the optimized access is not possible between the resize operation and the next garbage collection, an additional colocation check is necessary before an element of the array is accessed.

Array fusing saves one field load, therefore it is only beneficial if the check does not require additional instructions. It is necessary to combine the colocation check with the array bounds check that precedes every array access according to the Java specification. When a fused field is modified, we set the length of the old fused



(a) After allocation (b) After resize and field change (c) After GC

Fig. 12. Basic principle of array fusing

array to 0, which forces the array bounds check to fail. Instead of throwing an exception immediately, we check whether the field has been changed, i.e., whether it points to a non-colocated array. In this case, we access the new array and continue normally. Only if the bounds check for the new array also fails is an exception thrown. However, it is not possible to set the normally used `length` field to 0 for several reasons:

—The old array could also be referenced by other objects or by root pointers. Overwriting the `length` field would make the array elements inaccessible because bounds checks of non-optimized array accesses would also fail.

—The garbage collector must iterate the heap. Without the correct array length, it would not be possible to compute the array size and thus to compute the address of the object following the array in memory.

—The optimized array access must load the array length relative to the parent. Because of the reverse object order (see Section 4.1), the array is placed in front of the parent. Therefore, the offset of the `length` field depends on the array length and is not a compile-time constant.

To solve all these issues, we place a copy of the array length, called `fusingLen`, at the end of the child array. It can be loaded using the fixed offset -4 relative to the parent, as shown in Figure 12. This copy is accessed only by the optimized code and can be freely manipulated based on the demands of array fusing. Note that the `fusingLen` is smaller than the `length` in our example, as explained in the next section. To load the element at the index `n` from the parent `p` without loading the address of the array first, the following address arithmetic is necessary (the element size of the array is 4 bytes):

```
p - 8 - fusingLen * 4 + n * 4
```

The computation can be transformed to:

```
p + (n - fusingLen) * 4 - 8
```

With this transformation, no additional machine instruction is necessary. The multiplication by 4 and the subtraction of the constant offset are performed by the indexed addressing mode of the memory access. The subtraction `n - fusingLen` is

```
nr__operation_____________________________
// ecx: this  edx: index
12  move [ecx + 16] -> eax
14  cmp  edx, [eax + 8]
16  branch aboveOrEqual Exception
18  move [eax + edx*4 + 12] -> eax
22  return eax
```

(a) Unoptimized LIR

```
nr__operation_____________________________
// ecx: this  edx: index
12  sub  edx, [ecx - 4] -> edx
14  branch aboveOrEqual S1
16  move [ecx + edx*4 - 8] -> eax
20  return eax

S1  add  edx, [ecx - 4] -> edx
    move [ecx + 16] -> eax
    cmp  edx, [eax + 8]
    branch aboveOrEqual Exception
    move [eax + edx*4 + 12] -> eax
    jump 20
```

(b) Optimized LIR

Fig. 13. Array fusing in method `ArrayList.get()`

folded into the array bounds check in the following way: Normally, the array bounds check compares the array index `n` with the `fusingLen`. On most architectures, the comparison of two numbers is internally implemented as a subtraction whose result is discarded. When the compare operation of the bounds check is replaced by a subtraction operation, both the bounds check and the address arithmetic are performed at once.

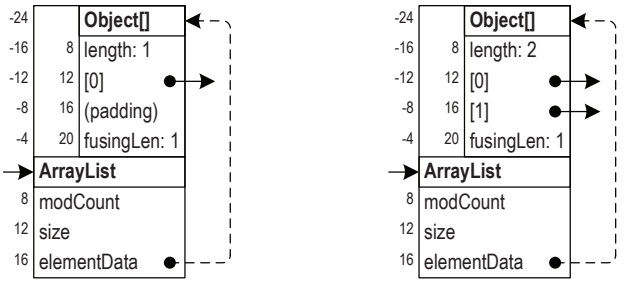
Figure 13 shows a fragment of the low-level intermediate representation (LIR) for the method `ArrayList.get()`. The if-statement of the method is not shown because it is not relevant for the array access. The unoptimized LIR first loads the field `elementData` with the offset 16. The following array load requires three instructions: a compare and a branch for the bounds check, as well as the memory access itself.

The optimized LIR does not need to load the field. Instead, the bounds check uses the `fusingLen`, which is accessed using the offset -4 relative to the `ArrayList` object. The LIR operation 12 in Figure 13(b) now uses a `sub` operation instead of a `cmp` operation for the bounds check. The result `n - fusingLen` is written to the register `edx`. This value is negative, so the memory access of the LIR operation 16 uses a negative offset.

The LIR code is split into a fast path and a slow path. The fast path code performs the optimized array access. One memory load is saved compared to the unoptimized code. When the field is overwritten with the reference to a new array, the `fusingLen` of the old fused array, i.e., the memory location `[ecx - 4]`, is set to 0, causing the bounds check to always fail. This case is regarded as uncommon, so the code is placed out-of-line at the end of the method in the slow path `S1`. It contains the same operations as the unoptimized code, i.e., it loads the field and then accesses the array using the normal offsets. Additionally, it must undo the subtraction by adding the subtracted `fusingLen` to the register `edx`. Another bounds check throws the exception if necessary.

5.2 Object Alignment

The 8-byte alignment of objects complicates the basic scheme for the reverse order of arrays. Because the array size is rounded up to the next multiple of 8, arrays



(a) Array with length 1 (b) Array with length 2

Fig. 14. Memory layout with 8-byte alignment

with different lengths can have the same size. The padding is inserted between the array elements and the field `fusingLen`. The padding must be incorporated into the only part of the address arithmetic that is loaded from the actual array: the field `fusingLen`.

Figure 14 illustrates the exact memory layout of fused arrays with the lengths 1 and 2. The offset of the first array element relative to the `ArrayList` object is -12 in both cases. Therefore, the field `fusingLen` of both arrays must be equal in order to yield the same array element offsets in the address computation shown in the previous section. To make the array bounds check safe, the minimum of the two lengths must be used, i.e., the array with a `length` of 2 has an `fusingLen` of 1.

The reduced `fusingLen` does not impact the correctness because the slow path is used to access the last element of all fused `Object[]` arrays with an even length. In cases such as the `ArrayList` where the last array element is not in use for most lists, this introduces no performance penalty. In summary, array fusing uses the array bounds check for several purposes:

—*Detection of field changes*: The slow path is used if the field was modified and an optimized array access is no longer possible, i.e., if the `fusingLen` is 0.

—*Address computation for the first array element*: The compare instruction of the bounds check is replaced by a subtraction instruction.

—*The 8-byte alignment of objects*: The slow path is used to handle the rounded `fusingLen` appropriately.

—*The actual bounds check*: Indices that are out of the valid array bounds are detected.

In the fast path of the optimized array access, the field load of the fused field is eliminated, so one memory access is saved. However, a slow path that performs the unoptimized array access is needed for correctness. The overall code size is increased in favor of a shorter and faster code for the common case. In our example, the method `Polyline.getPoint()` can access the address of a point relative to the `Polyline` object without loading the address of the `ArrayList` object and the `Object[]` array first. The fields `points` and `elementData` are no longer accessed in the common case.

6. EVALUATION

Our implementation is integrated into Sun Microsystems' Java HotSpot™ VM, using the snapshot release b21 of the upcoming JDK 7 [Sun Microsystems, Inc. 2009]. We use the default configuration for client applications. In addition to the optimizations of the current product version, our client compiler also performs array bounds check elimination based on the algorithm of Würthinger et al. [Würthinger et al. 2009]. The optimizations described in this paper can be selectively enabled using command line flags. We use the following four configurations for the evaluation:

—*Baseline*: All of our analyses and algorithms are disabled.

—*Colocation*: This configuration combines the impact of read barriers for detecting hot fields and object colocation during garbage collection for improving the cache behavior. Instances and arrays are optimized uniformly by the garbage collector.

—*Instance fusing*: In addition to the previous configuration, fields referencing instances are fused and field loads are removed. Fields referencing arrays are not fused, but are still colocated by the garbage collector.

—*Array fusing*: In this configuration, fields referencing both instances and arrays are fused.

All measurements were performed on an Intel Core 2 Quad processor Q6600 with 2.4 GHz, running Microsoft Windows XP. Each of the four cores has a separate L1 data cache of 32 KByte. Two cores together share a 4 MByte L2 cache, so there are 8 MByte L2 cache in total. All caches have a cache line size of 64 bytes. The main memory of 2 GByte is uniformly accessed by all cores. The results were obtained using a 32-bit operating system and a 32-bit VM.

6.1 SPECjvm98 Benchmarks

The SPECjvm98 benchmark suite [SPEC 1998] is commonly used to assess the performance of Java runtime environments. It consists of seven programs derived from real-world applications that cover a broad range of scenarios. Each benchmark is executed five times in the same VM instance to allow the VM to apply run-time optimizations, and the first runs as well as the last runs are reported in Section 6.1.3. The first runs represent the startup speed of the VM and include the time necessary for compilation, while the last runs show the peak performance after all optimizations have been applied. We repeated all measurements 12 times (always starting a new VM instance), verified that the results are stable across all executions, and report the mean of all executions. For all benchmarks, the stable state is reached before the last runs start. Therefore, the static field access statistics presented in Section 6.1.2 no longer change in the last runs. The dynamic field access statistics presented in Section 6.1.1 cover only the last runs in order to exclude the startup phase. The heap size was fixed to 64 MByte for all benchmarks.

6.1.1 *Field Access Counts.* Object fusing reduces the number of field loads performed at run time. Figure 15 shows the distribution of field and array loads and the impact of fusing. Because no field loads are eliminated by object colocation, this configuration is not shown in the figure. The remaining three configurations are abbreviated as *b* (baseline), *i* (instance fusing), and *a* (array fusing).

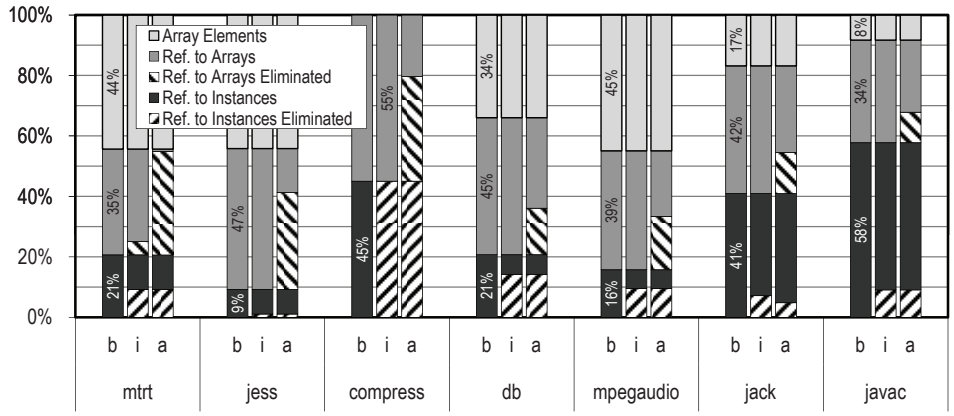


Fig. 15. Dynamic field access statistics for SPECjvm98

	Initial			Counted		Colocated		Fused	
	Inst.	Arr.	Prim.	Inst.	Arr.	Inst.	Arr.	Inst.	Arr.
mtrt	380	77	315	40	19	18	11	4	4
jess	410	77	407	67	24	19	10	2	4
compress	352	76	296	13	16	7	9	7	5
db	350	74	289	23	17	4	6	2	2
mpegaudio	412	93	346	73	35	18	19	9	11
jack	395	80	330	80	23	20	12	2	3
javac	492	92	344	148	36	24	11	2	4

Fig. 16. Static field access statistics for SPECjvm98

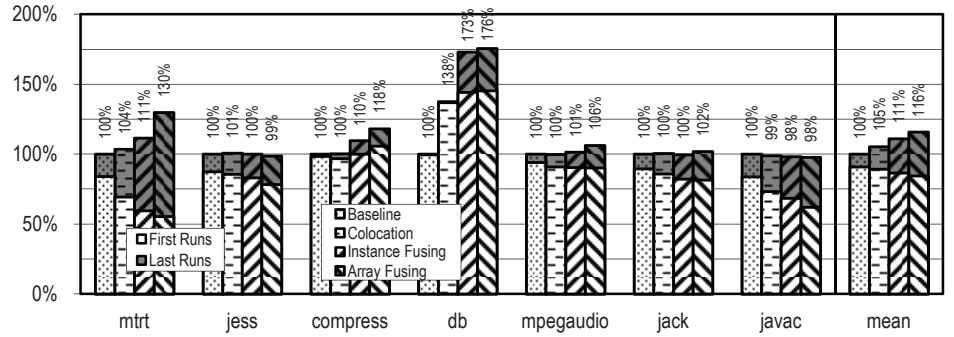


Fig. 17. Speedup compared to baseline for SPECjvm98 (taller bars are better)

We count all loads of references from the heap, i.e., all loads that could be optimized by fusing, and distinguish between three kinds: loads of fields referencing instances, loads of fields referencing arrays, and loads of array elements. The figure does not contain loads of primitive values such as `int` fields or elements of `int[]` arrays. The first column shows the distribution of the three kinds. This distribution is identical in all configurations.

The subsequent columns highlight the percentage of loads that are eliminated by instance fusing and array fusing. The higher the striped bars are, the more field loads are eliminated. Array elements cannot be optimized without an interprocedural data-flow analysis. Therefore, the topmost bar shows no eliminated loads in any configuration.

Instance fusing optimizes only fields referencing instances, i.e., the bottommost of the three kinds. For mtrt, the number of accessed fields referencing arrays is also slightly reduced by instance fusing though no such fields are optimized. This is a beneficial side effect of instance fusing on other optimizations such as global value numbering. If it is known that a field never changes, more subsequent other field loads are identified as redundant and can therefore be eliminated.

All benchmarks except javac perform more loads of references to arrays than to instances. This demonstrates the importance of array fusing. The benchmarks mtrt, compress, and mpegaudio frequently access arrays whose lengths are compile-time constants. The other four benchmarks jess, db, jack, and javac mostly use array-based collection classes that can be optimized.

6.1.2 *Number of Optimized Fields.* Our analysis performs fusing on a field-per-field basis at run time. To limit the overhead, it is important that no time is wasted analyzing fields that are infrequently accessed. Figure 16 shows that detecting hot fields using read barriers acts as an effective filter. The first column *initial* shows the overall number of fields in all loaded classes. Based on the declared type, we distinguish fields referencing instances, fields referencing arrays, and primitive fields. Primitive fields are not relevant for fusing, so they are ignored in the subsequent columns.

The column *counted* shows the number of fields for which read barriers are emitted in the compiled code. The read barrier counters are used to determine whether a field is frequently accessed. The number of frequently accessed fields is shown in the column *colocated*. Only 1% to 5% of the fields referencing instances and 8% to 20% of the fields referencing arrays are colocated. This is essential to keep the overhead during garbage collection low.

The column *fused* shows the number of fused fields when object fusing is performed. Ideally, all colocated fields should also be fused. However, this is not possible because the strong preconditions necessary for object fusing cannot always be met.

6.1.3 *Impact on Run Time.* Figure 17 shows how object colocation, instance fusing, and array fusing affect the performance of SPECjvm98. We present the results for the individual benchmarks as well as the geometric mean of all benchmarks. The first runs and the last runs are shown in the same figure on top of each other: the gray bars refer to the fastest runs, the white bars to the slowest. Both use the

same baseline, i.e., the last runs with all our optimizations disabled. Therefore, it is possible to compare the different configurations as well as the first and last runs. The differences between the first and the last runs in the first column (*baseline*) highlight the overhead of just-in-time compilation in the Java HotSpot^{TM} VM.

The second column of the last runs shows the benefits of object colocation. The second column of the first runs approximates the overhead of the read barriers, i.e., the overhead of the field counters that are incremented at run time. Because field accesses are counted in the first runs, object colocation is not yet performed. Only for the benchmark db, object colocation starts early enough to compensate the read barrier overhead.

The third column shows the impact of instance fusing. The analyses for guaranteeing the preconditions mostly impact the first runs negatively. However, this overhead is justified by the improved peak performance of the last runs. Array fusing, which is shown in the fourth column, increases the impact of instance fusing. For both instance fusing and array fusing, the impact on the first runs depends on the benchmark characteristics: for example, the first run of the benchmark mtrt is negatively affected because of the increased compilation overhead, while the first runs of the benchmarks db and compress are improved because the optimization succeeds early enough in the first run.

For the benchmark compress, seven fields referencing instances and seven fields referencing arrays account for 97% of all field loads. Only two of theses fields are not fused, so nearly the whole object graph is combined to one large group of objects. Similarly, the benchmark mpegaudio has a small working set that fits entirely into the cache. The two most important fields referencing instances and the two most important fields referencing arrays are fused.

The working set of live objects for the benchmark mtrt is much bigger than the cache size. Therefore, object colocation of 29 fields in total leads to a speedup of 4%. Four fields referencing instances and four fields referencing arrays account for over 90% of all field loads. Eliminating the loads of all four fields referencing arrays is the most effective part of the optimization and leads to an overall speedup of 30%. The benchmark db shows the highest speedup of all. It operates on a large number of object groups that consist of two instances and one array. Object colocation is able to avoid cache misses and leads to a speedup of 38%. Eliminating loads of the field that connects the first and the second instance doubles this speedup.

For the benchmark jess, the three most important fields referencing arrays are fused and thus a significant number of field loads are eliminated. However, two of the three fields are used to manage dynamic lists and the fields change frequently. Array fusing for the benchmark jack also optimizes the fields for dynamic lists, but here the change rate of the fields are lower. The slow path is taken infrequently, so array fusing leads to a speedup of 2%. The benchmark javac shows a slowdown in all configurations. The reason for the slight regression of peak performance is mostly the overhead of object colocation during garbage collection. This overhead is similar for all benchmarks, but normally outweighed by the improved cache behavior.

6.1.4 *Impact of Heap Size.* The measurements of the previous section were performed with a heap size fixed to 64 MByte, a size that is reasonable in practice because then garbage collection time is not a major factor of the overall run time.

Because all configurations use the same fixed heap size, the space overhead of array fusing, which requires an additional length field, is included in the benchmark results: The larger arrays fill up the heap more quickly and lead to a more frequent garbage collection, which adds to the total run time we report.

Experiments with different heap sizes showed similar speedups and slowdowns: A large heap of 1 GByte does not change the execution times at all because the larger heap cannot be utilized by the benchmarks. Smaller heaps of 40 MByte and 20 MByte already show a slowdown of some benchmarks, especially javac because it has a high object allocation rate. However, all configurations are affected equally, e.g., javac has consistently 12% slowdown when comparing 20 MByte heap size with 64 MByte heap size. When the heap size is set below a certain limit, garbage collection time begins to dominate the overall run time. For example, with a heap size of 12 MByte mtrt is a factor of 5 slower. No particular pattern is visible for our different configurations of object colocation and fusing: the speedup gained from our optimizations is alleviated by the overhead of the garbage collector. Because the variances of multiple runs of the same configuration are high, the differences are also not statistically significant.

6.2 DaCapo Benchmarks

The DaCapo benchmark suite [Blackburn et al. 2006] consists of eleven object-oriented applications. They are more elaborate than the SPECjvm98 benchmarks regarding code complexity, class structures, and class hierarchies. We evaluate the DaCapo benchmarks (version 2006-10-MR2) in a manner similar to the SPECjvm98 benchmarks and report the same metrics. Again, we executed each benchmark five times and show the result for the first and the last runs. The heap size was fixed to 128 MByte for all benchmarks. A large heap of 1 GByte does not change the benchmark results. Smaller heaps of 64 MByte, 32 MByte, and 24 MByte show gradual slowdowns of some benchmarks, especially luindex and xalan. However, our configurations still show the same behavior. Note that hsqldb requires a heap size of 128 MByte and thus fails in the configurations with a smaller heap.

Figure 18 shows the dynamic field access counts of the DaCapo benchmarks. The higher complexity of the DaCapo benchmarks compared to SPECjvm98 complicates our optimizations. Only a smaller percentage of the dynamic loads of references to instances and arrays are eliminated.

The higher complexity is also visible in the static field access statistics shown in Figure 19. The effective detection of hot fields using read barriers is essential to keep the run-time overhead of object colocation and fusing in an acceptable range. The several thousand fields referencing instances and several hundred fields referencing referencing are reduced to at most 77 and 65 colocated fields, respectively. However, our fusing algorithm is too conservative, so few of the colocated fields are fused.

Figure 20 shows the benchmark results for DaCapo. Because the benchmarks are larger and have a higher number of methods, more methods must be compiled in the startup phase and the differences between the first runs and the last runs are usually higher than for SPECjvm98. This is not related to our optimizations because also the baseline configuration, where all of our modifications are disabled, is affected. The additional overhead of our code for the first runs is still modest.

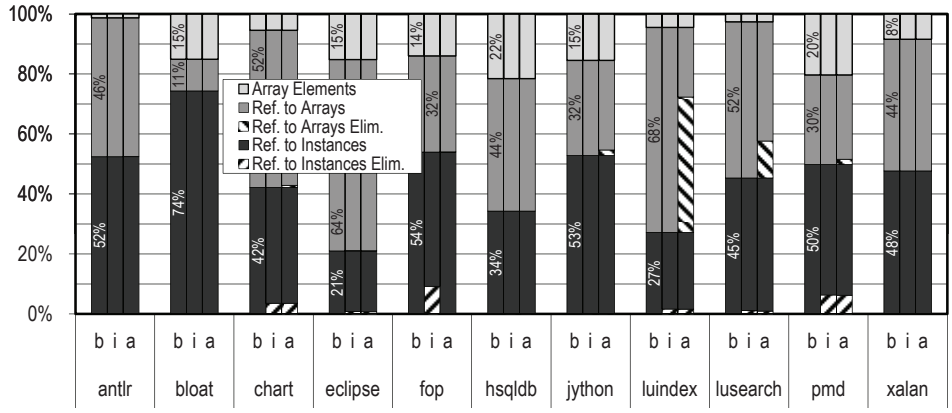


Fig. 18. Dynamic field access statistics for DaCapo

	Initial			Counted		Colocated		Fused	
	Inst.	Arr.	Prim.	Inst.	Arr.	Inst.	Arr.	Inst.	Arr.
antlr	795	126	606	238	43	24	13	1	2
bloat	1008	167	578	251	62	47	12	8	2
chart	1592	274	1296	207	68	32	11	4	2
eclipse	3079	767	2162	771	374	75	65	11	5
fop	1452	158	949	324	48	31	12	1	3
hsqldb	844	209	689	209	92	26	9	2	1
jython	1264	234	841	278	94	77	31	3	7
luindex	709	155	643	191	63	35	19	6	4
lusearch	684	140	623	148	55	22	15	3	4
pmd	1092	173	779	238	58	65	26	9	8
xalan	1335	183	852	449	75	13	11	0	1

Fig. 19. Static field access statistics for DaCapo

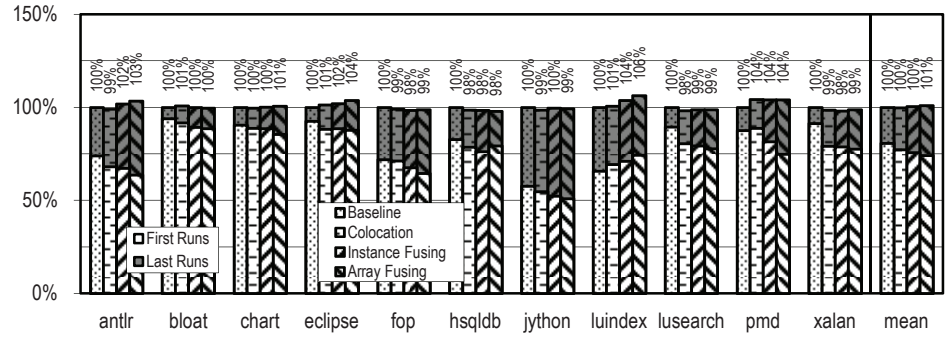


Fig. 20. Speedup compared to baseline for DaCapo (taller bars are better)

The speedup of object fusing is lower compared to SPECjvm98, and there are more benchmarks without a speedup at all. The main constraint of our approach is object co-allocation. When the time frame between the allocation of a parent and its children is too long, the allocations do not appear in the same compiled method so co-allocation is not possible. A complex control flow between the allocations also hinders co-allocation. Additionally, the DaCapo benchmarks use more dynamic data structures where references to instances are changed several times. This cannot be handled by instance fusing.

The benchmark pmd benefits from object colocation. Changing the object order reduces the number of cache misses and leads to a speedup of 4%. Object fusing has no impact because too few field accesses are eliminated.

In contrast, about two thirds of the loads of array references are eliminated for the benchmark luindex. Most of the eliminated loads are actually loads of the same field for the same object, i.e., mostly cache hits. Nevertheless, the large number of eliminated loads still leads to a speedup.

The dynamic field access statistics reflect only one of the optimizations that are performed for fused fields. Other benefits such as the increased amount of static type information affect the benchmark speed. For example, the benchmarks antlr and eclipse show a speedup although no significant number of field loads are eliminated. All other compiler optimizations benefit from object fusing, however the exact impact of the different optimizations is difficult to quantify. Examples of optimizations are the elimination of type checks and improved method inlining because of statically known receiver types.

Our object colocation algorithm is embedded into the garbage collector and introduces a small but measurable overhead. This overhead outweighs the benefit for some of the benchmarks. For example, the benchmark antlr is slightly slower with object colocation, but then benefits from object fusing. For some other benchmarks such as fop and hsqldb, the overhead is higher than the benefit for all configurations. However, no benchmark has a slowdown of more than 2%.

Our baseline, the Java HotSpot^{TM} VM, is a production-quality Java VM. In contrast to research virtual machines, all subsystems are heavily tuned for performance. Even small extensions to critical paths like the garbage collector lead to a measurable slowdown. Additionally, our implementation of object fusing supports all parts of the Java specification, especially dynamic class loading. This requires a conservative handling in some cases. Therefore, the speedup for the benchmarks is lower compared to other implementations mentioned in the related work. However, we believe that our results are meaningful and reflect the actual impact of the optimizations in practice.

7. RELATED WORK

This section discusses other research projects that work on similar optimizations. So far, object fusing was only performed in static compilers (under the term *object inlining*), while object colocation was usually integrated into virtual machines.

7.1 Object Inlining

Dolby et al. extended a static compiler with an algorithm for automatic object inlining [Dolby 1997; Dolby and Chien 1998; 2000]. The input language is ICC++,

a dialect of C++. Their algorithm clones the code of methods that access optimized fields. Therefore, there can be both optimized and unoptimized objects of the same class as long as the same method always works on objects of the same kind. Because they use an advanced interprocedural data-flow analysis, they are able to convert arrays of references to arrays of object values. Our analysis is not capable of performing such transformations. However, we can handle dynamic arrays where a field is changed to point to arrays of different sizes. This is not possible in any of the existing static approaches. A collection of medium-sized applications shows a maximum speedup of 50%.

Laud implemented object inlining in a static Java compiler [Laud 2001]. This algorithm can detect and handle the case when a child object is replaced with a new object. Instead of replacing the inlined field with a reference to the new object, the fields of the new object are assigned to the fields of the old object. It is, however, not allowed that a child object is referenced by anything other than its parent object, e.g., by a field of another object. No details regarding arrays are published. To the best of our knowledge, only the detection of inlinable fields was implemented, but not the necessary program transformations that remove loads of inlined fields. Therefore, no benchmark results are available.

Lhoták et al. provide a good introduction to object inlining and analyze the possible impact on several Java benchmarks [Lhoták and Hendren 2005]. Depending on the access pattern, they distinguish four classes of inlinable fields, and use this classification to compare the number of fields that can be optimized by the algorithms of Dolby and Laud. According to their terms, our algorithm can optimize references to instances that satisfy the predicates *[contains-unique]* and *[unique-container-same-field]*, and references to arrays that satisfy the predicate *[unique-container-same-field]*. Requiring only a single predicate for arrays is an improvement compared to the algorithms of Dolby and Laud, which require at least two predicates to be fulfilled. The study does not describe an analysis or implementation for object inlining, so no benchmark results are published. They only list the most important inlinable fields for SPECjvm98 and some other benchmarks.

Veldema et al. present an algorithm for *object combining* that groups objects with the same lifetime [Veldema et al. 2005]. Their optimizations are integrated into Manta, a static compiler for Java. Object combining 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 at once. Elimination of pointer accesses is performed separately by the compiler. Similar to our approach, they retain the object headers of child objects, which contain the virtual method table and a flags field. Fusing of a single variable-length array per object group is possible. Their optimizations 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 high. They report a speedup of up to 34% for a set of object-oriented applications.

Ghemawat et al. use a cheap interprocedural analysis for object inlining and for other optimizations [Ghemawat et al. 2000]. Their analysis is integrated into Swift, an optimizing static Java compiler for the Alpha architecture. They collect a variety of properties for each field, e.g., whether the field is never `null` or whether

it is always assigned an object of the same type. Objects can be inlined either with or without their header. The header is necessary when the child object can be referenced from outside the parent. Arrays with variable length are not optimized. There are no timing results with only object inlining enabled, so it is not possible to quantify the impact of object inlining.

Budimlic et al. present a static Java bytecode optimizer that performs object inlining [Budimlić and Kennedy 1997]. When they inline an instance, they eliminate the allocation and replace the fields by local variables, i.e., they perform scalar replacement of instances. For arrays, they replace an array of references by separate arrays of scalar values, one for each field of the inlining child. This is consistent with our definition of inlining because it performs address arithmetic to combine an array access and a field load into a single array access. The evaluation is limited to four small mathematical computations where their optimization is highly effective, leading to a speedup of up to 460%.

7.2 Object Colocation

Huang et al. describe a system called *online object reordering* [Huang et al. 2004], implemented for the Jikes RVM. They use the adaptive compilation system of Jikes that periodically records the currently executed methods. Fields accessed in the most frequently recorded methods are traversed first by the garbage collector. This information is not as precise as our dynamic numbers obtained from the read barriers. By using the existing interrupts of Jikes, their analysis has a low run-time overhead of 2% to 3%.

Chilimbi et al. use generational garbage collection for *cache-conscious data placement* [Chilimbi and Larus 1998] and present results for the dynamically typed and purely object-oriented programming language Cecil. They collect run-time information about accessed objects using a sequential buffer. When an object is accessed, its address is written into the next free position of the buffer. This information is converted to an object affinity graph before garbage collection. Objects that are accessed together with at most one other object access in between are added to the graph. The garbage collector places such objects next to each other in memory. They report a reduced execution time of 14% to 26% for their benchmarks.

Chen et al. use garbage collection as a proactive technique to improve the locality of objects [Chen et al. 2006], i.e., they trigger garbage collection when the locality should be improved. The run-time analysis is similar to the one of Chilimbi, however it is enabled only during short sampling intervals. The implementation is integrated into the Common Language Runtime of Microsoft. The evaluation with several C# applications shows an average speedup of 17%, with an analysis overhead of less than 3%.

Shuf et al. improve the locality of objects in Java applications [Shuf et al. 2002] using frequently instantiated types, called *prolific types*. Their implementation is integrated into the Jikes RVM. The just-in-time compiler uses an allocation profile to co-allocate at most two objects if they have both a prolific type and if they are connected by a field. The garbage collector preserves this optimized order using a modified object traversal algorithm. When only co-allocation is performed, 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 is consistent with our experience that co-allocation itself is not beneficial if object allocations are cheap.

Chilimbi et al. perform structure splitting for structures of a size comparable to the cache line size [Chilimbi et al. 1999]. With splitting, the hot fields of multiple objects end up in the same cache line. They use profiling data collected by an instrumented version of the application to guide a static compiler for Java. Together with cache-conscious object colocation performed by the garbage collector, they report a speedup of 18% to 28%.

8. FUTURE WORK

The benchmark results show that removing field loads is profitable for many applications, and that the overhead is reasonably small if no optimization is possible. In some cases, we are conservative and do not optimize where the expected benefit was too small compared to the implementation complexity. The following small-scale improvements could be implemented without changing the overall architecture of our system:

—Some algorithms handle complicated cases conservatively, mostly the algorithm for object co-allocation. Improvements of co-allocation would directly lead to more fields that can be fused.

—We do not optimize fields that are accessed using the dynamic features of Java, i.e., reflection, the Java Native Interface, and object cloning. However, we did not see a field where optimization is not possible solely because of this constraint.

—In general, fusing of array elements is not possible with our approach. However, some special cases could be handled, e.g., rectangular multi-dimensional arrays.

Some constraints result from the basic design of our approach. We do not support certain optimizations because they would introduce too much complexity that cannot be handled in our dynamic approach. A future implementation of object fusing could rely more on static analysis in favor of more optimizations.

For example, we do not allow optimized and unoptimized instances of the same class to coexist. This is the most severe restriction because a single allocation site where co-allocation is not possible prevents the fusing of a field in all instances and subclass instances. If instances were separated into disjoint groups, the optimization of a single group would be possible. This would require some sort of interprocedural analysis. While an interprocedural data-flow analysis is complicated and expensive because of dynamic class loading, a limited form could be sufficient for this purpose.

Our approach eliminates neither the object headers of fused objects nor the pointers to fused fields. This is necessary because we use a dynamic optimization model that smoothly transitions between unoptimized and optimized machine code. This approach does not allow structural changes of the heap. To support such changes, explicit phases would have to be introduced. For example, removing a field from all objects of a certain class would require a single transition point at which the heap is rewritten.

After this point, accessing the field would no longer be allowed, i.e., all methods that access the field would have to be rewritten at this point. This would require information about all places where the optimized field is loaded, in addition to our

information where the field is stored. If deoptimization is necessary, the heap would have to be rewritten again to reintroduce the field. Future work could investigate whether the complexity of such phase changes is justified by the reduced memory consumption of the optimized application.

9. CONCLUSIONS

We presented a feedback-directed approach for object colocation, instance fusing, and array fusing. Colocation improves the spatial locality of the heap, and fusing replaces field loads with address arithmetic. In contrast to previous static approaches, our optimization is performed automatically at run time when applications are executed in the Java HotSpot™ VM. Following the approach of feedback-directed optimization, we use read barriers to detect frequently accessed fields that are worth being optimized. While the actual optimization that eliminates a field load is quite simple and straightforward, the preceding analysis steps that guarantee the necessary preconditions are challenging. The evaluation with several standard benchmarks showed that a speedup can be achieved for a highly optimized and production-quality Java VM with only optimizing a handful of fields.

Rather than viewing just-in-time compilation and garbage collection as a runtime overhead stealing time that could have been spent executing the application, we consider them as a powerful vehicle for dynamic optimizations. Our array fusing where the fields are modified at run time is one example of an optimization that is not possible in statically compiled languages such as C++.

Acknowledgements

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 and Thomas Würthinger for their valuable comments on the work and this paper.

REFERENCES

Arnold, M., Fink, S. J., Grove, D., Hind, M., and Sweeney, P. F. 2005. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE 93,* 2, 449–466.

Ball, T., Mataga, P., and Sagiv, M. 1998. Edge profiling versus path profiling: The showdown. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, 134–148.

Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, New York, 169–190.

Budimlić, Z. and Kennedy, K. 1997. Optimizing Java: Theory and practice. *Concurrency: Practice and Experience 9,* 6, 445–463.

Chen, W., Bhansali, S., Chilimbi, T. M., Gao, X., and Chuang, W. 2006. Profile-guided proactive garbage collection for locality optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, 332–340.

Chilimbi, T. M., Davidson, B., and Larus, J. R. 1999. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, 13–24.

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

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13,* 4, 451–490.

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

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

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

Ghemawat, S., Randall, K. H., and Scales, D. J. 2000. Field analysis: Getting useful and low-cost interprocedural information. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, 334–344.

Gosling, J., Joy, B., Steele, G., and Bracha, G. 2005. *The Java^{TM} Language Specification*, 3rd ed. Addison-Wesley, Reading.

Griesemer, R. and Mitrovic, S. 2000. A compiler for the Java HotSpot$^{\text{TM}}$ virtual machine. In *The School of Niklaus Wirth: The Art of Simplicity*, L. Böszörményi, J. Gutknecht, and G. Pomberger, Eds. dpunkt.verlag, Heidelberg, 133–152.

Hegde, R. 2008. *Optimizing Application Performance on Intel Core Microarchitecture Using Hardware-Implemented Prefetchers*. Intel Software Network.

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

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

ISO/IEC. 2003. *C++*, 2nd ed. International Standard ISO/IEC 14882.

ISO/IEC. 2006. *Common Language Infrastructure (CLI)*, 2nd ed. International Standard ISO/IEC 23271.

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

Kotzmann, T., Wimmer, C., Mössenböck, H., Rodriguez, T., Russell, K., and Cox, D. 2008. Design of the Java HotSpot$^{\text{TM}}$ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization 5,* 1, Article 7.

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

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

Lindholm, T. and Yellin, F. 1999. *The Java^{TM} Virtual Machine Specification*, 2nd ed. Addison-Wesley, Reading.

Shuf, Y., Gupta, M., Franke, H., Appel, A., and Singh, J. P. 2002. 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*. ACM Press, New York, 13–25.

SPEC 1998. *The SPECjvm98 Benchmarks*. Standard Performance Evaluation Corporation. http://www.spec.org/jvm98/.

Sun Microsystems, Inc. 2009. *JDK 7 Project*. Sun Microsystems, Inc. https://jdk7.dev.java.net/.

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

Wimmer, C. 2008. Automatic object inlining in a Java virtual machine. Ph.D. thesis, Johannes Kepler University Linz.

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

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

Wimmer, C. and Mössenböck, H. 2007. Automatic feedback-directed object inlining in the Java HotSpot^{TM} virtual machine. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*. ACM Press, New York, 12–21.

Wimmer, C. and Mössenböck, H. 2008. Automatic array inlining in Java virtual machines. In *Proceedings of the International Symposium on Code Generation and Optimization*. ACM Press, New York, 14–23.

Würthinger, T., Wimmer, C., and Mössenböck, H. 2009. Array bounds check elimination in the context of deoptimization. *Science of Computer Programming 74,* 5–6, 279–295.