Automatic Array Inlining in Java Virtual Machines[*]

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

ABSTRACT

Array inlining expands the concepts of *object inlining* to arrays. Groups of objects and arrays that reference each other are placed consecutively in memory so that their relative offsets are fixed, i.e. they are *colocated*. This allows memory loads to be replaced by address arithmetic, which reduces the costs of field and array accesses. We implemented this optimization for Sun Microsystems' Java HotSpot^{TM} VM. The optimization is performed automatically and requires no actions on the part of the programmer.

Arrays 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 whenever the array is reallocated. We present an efficient code pattern that detects these changes and allows the optimized access of such array fields. It is integrated into the array bounds check. We also claim that inlining array element objects into an array is not possible without a global data flow analysis.

The evaluation shows that our dynamic approach can optimize frequently accessed fields with a reasonable low compilation and analysis overhead. The peak performance of SPECjvm98 is improved by 10% on average, with a maximum of 25%.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Memory management, Optimization*

General Terms

Algorithms, Languages, Performance

Keywords

Java, object inlining, array inlining, just-in-time compilation, garbage collection, optimization, performance

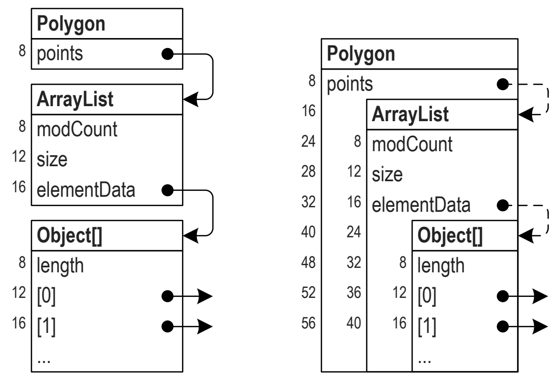
[*]This work was supported by Sun Microsystems, Inc.

1. INTRODUCTION

Arrays play an important role in many object-oriented applications. While objects are used to decompose the functionality into well-understandable small parts, arrays are ideal for the implementation of dynamic data structures. Objects of business logic classes use arrays to reference a variable list of related objects. This dynamic list of children is encapsulated in collection classes that are part of almost every class library. Because of the additional layer between the business object and the array, many memory accesses are necessary to load an array element. Object and array inlining reduce this overhead by placing such objects and their child arrays consecutively in the heap and by replacing the memory accesses with address arithmetic. Our solution addresses the time overhead of field accesses, but not the space overhead. We keep all object and array headers and require additional fields for dynamic arrays, which increases the memory footprint.

Figure 1 illustrates the idea of object and array inlining. A `Polygon` object uses the collection class `ArrayList` of the package `java.util` to maintain a dynamic list of points. The `ArrayList` stores its data in an `Object[]` array. When new elements 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.

To access a point, at least two field loads of the fields `points` and `elementData` and one array access are necessary. Object and array inlining combine the objects to a larger group so that a point can be accessed with a single array access, without loading the fields `points` and `elementData`.



(a) unoptimized objects (b) object and array inlining

Figure 1: Motivating example for array inlining.

The field `points` is initialized only once, e.g. in the constructor of the `Polygon` class, and never changed. This simplifies inlining the `ArrayList` object into the `Polygon` object. In contrast, the field `elementData` of the `ArrayList` is changed whenever the list is resized. The array inlining algorithm described in this paper is capable of handling such changing array fields. This optimization goes beyond the possibilities of *value objects* in languages like C++ and C# where the size of value objects must be fixed at compile time.

We distinguish between parent classes whose objects reference arrays with the same fixed length, parent classes whose objects reference arrays with different but fixed lengths, and parents that reference arrays with varying lengths, i.e. parents whose array fields are reassigned multiple times such as the `elementData` field in the example. Furthermore, we look at the inlining of array elements, e.g. at inlining the points referenced by the example's `Object[]` array into the array. We show that such an optimization is not possible in general without a global data flow analysis because the Java bytecodes for accessing array elements have no static type information.

The paper presents results for an implementation that is integrated into Sun Microsystems' Java HotSpot^{TM} VM. The analysis, however, can be generally applied to all stack-based bytecode languages, such as the *common intermediate language (CIL)* [12] that is part of the standardized *common language infrastructure* and used for example by the Microsoft .NET framework. The bytecodes for accessing object fields and array elements have roughly the same structure in all such languages.

To the best of our knowledge, we describe the first approach that allows inlining of arrays at run time without requiring actions on the part of the programmer. Additionally, we are not aware of any system that allows array inlining for fields that are changed several times. This paper contributes the following:

- We propose to integrate automatic array inlining into a Java virtual machine. We show that the inlining of array fields is possible and reason about the difficulties when trying to inline array elements at run time.
- We present a code pattern for the optimized access of array fields that are changed at run time.
- We evaluate our implementation using standard benchmarks and report results for different configurations.

2. SYSTEM OVERVIEW

Our implementation is based on the Java HotSpot^{TM} VM of Sun Microsystems [21]. The default configuration for interactive desktop applications uses a fast just-in-time compiler, called the *client compiler* [8, 14], and a generational garbage collector with two generations. This structure is similar for most modern Java VM implementations, even though the just-in-time compilers and the garbage collection algorithms vary greatly. Our implementation and all machine code examples in this paper are based on Intel's IA-32 architecture.

The just-in-time compiler operates in the background and compiles methods whose invocation counters reach a certain threshold. The compiler performs aggressive optimizations such as inlining of dynamically bound methods. The dynamic features of Java like dynamic class loading can invalidate such optimizations. Therefore, the VM can *deoptimize* [10] the machine code of certain methods and revert their execution back to the interpreter. The heap is divided into two generations. New objects are allocated in the young generation, which is collected by a stop-and-copy algorithm. Long-living objects are promoted to the old generation, which is usually larger than the young generation. A mark-and-compact algorithm collects the entire heap when the old generation fills up [13].

2.1 Definitions

Object and array inlining operate on groups of objects and arrays in the heap that are in a parent-child relationship. An *inlining parent* contains a reference to the *inlining child*. A child has exactly one parent, but a parent can have multiple children. Additionally, there can be inlining hierarchies, i.e. an inlining child can again be an inlining parent itself. In the example shown in Figure 1, the `ArrayList` object is both the child of the `Polygon` object and the parent of the `Object[]` array.

If the inlining parent is an object, an inlining child is referenced by a field of the object. Such a field is called an *inlined field*. The declared type of the field can be either an object type or an array type. We therefore differentiate between *object fields* and *array fields*. From the point of view of the bytecodes, there is no technical difference between object and array fields because the superclass of all arrays is `Object`. However, objects and arrays have different characteristics that are relevant for inlining. For example, the size of an object is constant, while the size of an array is unknown until the allocation.

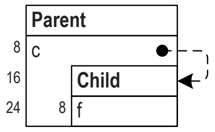
If the inlining parent is an array, the inlining child is referenced by an *array element*. Section 3.2 shows that this case cannot be handled efficiently.

2.2 Dynamic Object Inlining

Array inlining is based on a modified version of our object inlining algorithm [23, 24]. To allow automatic inlining at run time without any programmer interaction, it is necessary to detect hot fields that are worth to be optimized. For this, the just-in-time compiler inserts read barriers that increment field access counters per field and class. When the garbage collector moves objects that are linked by hot fields, it groups these objects so that the parent object and its children are consecutive (*object colocation*).

Two preconditions must be fulfilled for an object field to be inlined. First, the parent and the child object must be allocated together and the field store that installs a reference from the parent to the child must happen immediately after the allocation (*co-allocation of objects*). Secondly, subsequent field stores must not overwrite the field with a new value. All analysis steps are embedded into the compiler and operate on a per-method basis. No global data flow analysis is necessary. When the two preconditions are satisfied for a field, it is possible to optimize field loads, i.e. to replace field loads by address arithmetic.

The optimization phases are partly overlapping because method execution, compilation, and garbage collection are asynchronous. The object layout is not changed during the optimization to limit the number of VM subsystems affected by object inlining and to allow a sliding transition to optimized machine code. Both the object headers and the fields that connect parents and children are preserved.

Allocation:
```
p = new Parent();
p.c = new Child();
```
Field access:
```
… = p.c.f;
```


Unoptimized machine code:
```
// eax: p
mov ebx, [eax+8]
mov ecx, [ebx+8]
// ecx: p.c.f
```
Optimized machine code:
```
// eax: p
mov ecx, [eax+24]
// ecx: p.c.f
```

Figure 2: Object inlining.

Figure 2 shows an example for object inlining. The field `c` of the class `Parent` contains a reference to an object of the class `Child`. The right-hand side of the figure shows the layout of the resulting group of objects when the field `c` is inlined. The field offsets are printed left to the field names. All objects have 8-byte headers and are aligned at 8-byte boundaries. The two field loads that are necessary for the normal access of `p.c.f` are optimized to a single load with a larger offset.

3. POSSIBILITIES FOR ARRAY INLINING

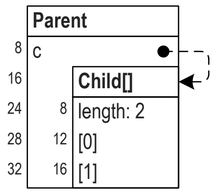
Java integrates array types smoothly into the object class hierarchy. However, there are certain differences between objects and arrays that affect inlining. When arrays are used as inlining children, it must be considered that the size of arrays is not necessarily a compile-time constant. Using arrays as inlining parents is complicated because the Java bytecodes for accessing array elements have no static type information.

The analysis to detect candidates for array inlining and to guarantee the preconditions is similar to the algorithm for object inlining. The just-in-time compiler combines three instructions of the intermediate representation to a single instruction for co-allocation: the object allocation of the parent, the array allocation of the child, and the field store of the array field. The length of the allocated array is a parameter of the array allocation instruction. This parameter can refer either to an integer constant, in which case the length is known at compile time, or any other instruction that computes the actual length, in which case the length is variable. More details on the compiler analysis are presented in [24].

3.1 Arrays as Inlining Children

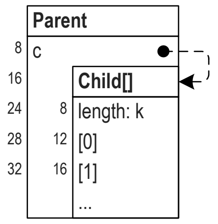
The preconditions for object inlining ensure that an object field references the same inlining child from allocation to deallocation of the object. The class of the inlining child and therefore also its size is known at compile time. The field is not allowed to change because detecting a changed object field would be equally expensive to accessing the field.

The basic algorithm for object inlining can also be applied for the inlining of array fields. The detection of hot fields and the colocation in the garbage collector need no changes. However, the actual inlining algorithm must take the differences between objects and arrays into account. On the one hand, the size of an array is not a compile-time constant in many cases, which hinders array inlining. On the other hand, it is possible to detect whether an array field has been changed at run time with no additional costs by embedding this check into the array bounds check. This increases the number of array fields that can be inlined. We distinguish the three cases of array inlining: *fixed array inlining*, *variable array inlining*, and *dynamic array inlining*.

Allocation:
```
p = new Parent();
p.c = new Child[2];
```
Field and array access:
```
int n = …
… = p.c[n];
```


Unoptimized machine code:
```
// eax: p  esi: n
mov ebx, [eax+8]
cmp esi, [ebx+8]
jae Exception
mov ecx, [ebx+esi*4+12]
// ecx: p.c[n]
```
Optimized machine code:
```
// eax: p  esi: n
cmp esi, 2
jae Exception
mov ecx, [eax+esi*4+28]
// ecx: p.c[n]
```

Figure 3: Fixed array inlining.

Allocation:
```
int k = …
p = new Parent();
p.c = new Child[k];
```
Field and array access:
```
int n = …
… = p.c[n];
```


Unoptimized machine code:
```
// eax: p  esi: n
mov ebx, [eax+8]
cmp esi, [ebx+8]
jae Exception
mov ecx, [ebx+esi*4+12]
// ecx: p.c[n]
```
Optimized machine code:
```
// eax: p  esi: n
cmp esi, [eax+24]
jae Exception
mov ecx, [eax+esi*4+28]
// ecx: p.c[n]
```

Figure 4: Variable array inlining.

Fixed Array Inlining. If all objects of a parent class reference arrays with the same fixed length, the inlining of array fields can be handled in the same way as the inlining of object fields. Because the length is constant, all referenced arrays have the same size, which is already known at compile time. In addition to eliminating the field access for the array field, subsequent accesses of the array can be optimized. The constant length is used to simplify the array bounds check, which does not need to load the array length from memory anymore.

The example in Figure 3 shows an array field that always references an array of length 2. Compared to a field access, an array access requires two additional machine instructions for the bounds check. The first instruction compares the index with the length of the array. The second one branches to an out-of-line code block that throws an exception. In addition to eliminating the memory access that loads the field `c`, the memory access of the array bounds check is replaced with the constant 2. Instead of three memory loads in the unoptimized machine code, only one load is necessary in the optimized code.

Variable Array Inlining. If the objects of a parent class reference arrays with different but fixed lengths, the field

Allocation:

```
p = new Data[2];
p[0] = new Data();
p[1] = new Data();
```

Array access:

```
int n = …
… = p[n].f;
```

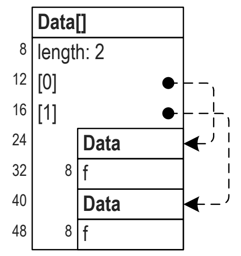


Figure 5: Array as the inlining parent.

accesses can be eliminated in the same way as with object inlining. However, a parent object can only have one such inlining child. Because the size of the array is not known at compile time, an array with variable length must always be the last child of a parent. The inlining offset of a subsequent child would not be fixed otherwise.

Figure 4 is a modification of Figure 3. The length of the allocated array is not known at compile time. The memory access for the array bounds check cannot be eliminated, so two memory loads are necessary in the optimized code. The array must be the last inlining child of the `Parent` class.

Dynamic Array Inlining. When an array field is assigned multiple times, it is no longer safe to eliminate the field access without further checks. In contrast to object inlining, however, it is possible to detect if an array field has been changed at run time without additional overhead for the common case. Section 4 discusses different optimization possibilities.

3.2 Arrays as Inlining Parents

Reference arrays contain pointers to other objects and arrays. Therefore, it would be beneficial to combine an array with the objects that are referenced by the array elements, i.e. to allow also arrays as inlining parents. Figure 5 shows the resulting object structure when the two `Data` objects referenced by a `Data[]` array are inlined. The access `p[n].f` can be performed by the address arithmetic `p+n*16+32`. However, we claim that this optimization is impossible without a global data flow analysis because of the structure of the array access bytecodes.

Java bytecodes [17] are executed using an operand stack. Most bytecodes pop their arguments from the stack, perform an operation, and then push the results back on the stack. Only arguments that are constant for the Java source language compiler, such as numbers of local variables or numeric constants, are part of the bytecodes. Another kind of such constants are field names. The bytecodes for loading and storing fields, `getfield` and `putfield`, include a symbolic reference to the accessed field denoting the name of the field and the class in which the field is to be found. The VM linker converts the symbolic reference to a field offset. With this static information, it is possible to differentiate which fields of which classes are changed at run time.

In contrast, the bytecodes that load and store elements of reference arrays, `aaload` and `aastore`, have no static operands. Both the accessed array and the index of the accessed element are taken from the operand stack. The lack of static type information hinders inlining of array elements. It is not possible to find out which array is modified by an `aastore` bytecode, i.e. it is not possible to find out whether

```
void m1(Parent p, Child c) {      0: aload_0
  p.c = c;                        1: aload_1
}                                 2: putfield Parent.c

void m2(Other p, Child c) {       0: aload_0
  p.c = c;                        1: aload_1
}                                 2: putfield Other.c
```

Figure 6: Example for field stores.

```
void m1(Data[] p, Data c) {         0: aload_0
  p[0] = c;                         1: iconst_0
}                                   2: aload_1
                                    3: aastore

void m2(Object[] p, Data c) {       0: aload_0
  p[0] = c;                         1: iconst_0
}                                   2: aload_1
                                    3: aastore
```

Figure 7: Example for array stores.

the elements of the modified array have been inlined or not. Every `aastore` bytecode can possibly modify any reference array in the heap.

Figure 6 and Figure 7 illustrate the differences between the `putfield` and the `aastore` bytecodes. Figure 6 corresponds to the object structure presented in Figure 2. Assume that the object field `c` of the class `Parent` should be inlined. The bytecode `putfield Parent.c` in Method `m1` tells the object inlining system that the field `c` of class `Parent` is modified here. When the method is called for a `Parent` object that already references a `Child` object, the object inlining system knows that the field is not inlinable because it is modified after the initialization. Similarly, the bytecode `putfield Other.c` in method `m2` affects the inlining of `Other.c`, but not the inlining of `Parent.c`. In other words, the type in the `putfield` bytecode tells us which objects are modified.

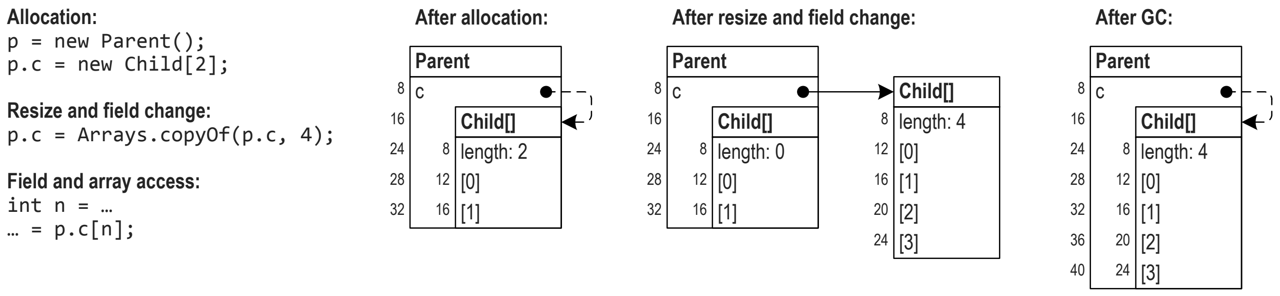
Figure 7 corresponds to the example presented in Figure 5. Because the `aastore` bytecode is not typed, the same bytecodes are emitted for the methods `m1` and `m2`. The method `m2` might modify a `Data[]` array because `Data[]` is assignment compatible with `Object[]`. The method `m2` therefore prohibits array element inlining of `Data[]` and all other array types. Methods like `m2` are common in nearly all applications, therefore our implementation of array inlining cannot handle arrays as inlining parents.

Only a global data flow analysis can solve this problem. It is necessary to know all contexts where the method `m2` is called. It must then be checked whether a `Data[]` array can flow into this method. Although such an analysis would be possible, global reasoning about Java classes is complicated by the dynamic features of Java like lazy class loading and reflection.

Special cases of arrays as inlining parents could be handled without a global analysis, for example the flattening of rectangular arrays to one dimension. However, we did not implement such optimizations.

4. DYNAMIC ARRAY INLINING

Arrays are often used to model dynamic data structures in object-oriented applications. Because the number of fields in an object is fixed, one has to allocate and link multiple objects to model e.g. a list with a variable number of



Allocation:
```
p = new Parent();
p.c = new Child[2];
```
Resize and field change:
```
p.c = Arrays.copyOf(p.c, 4);
```
Field and array access:
```
int n = …
… = p.c[n];
```

Figure 8: Principle of dynamic array inlining.

elements. In contrast, all elements can be stored in a single array with the appropriate length. When elements are added and the length of the array does not suffice, the common solution is to allocate a larger array, copy all existing elements from the old to the new array, and then discard the old one. This strategy is used for example in the Java collection class `ArrayList`.

4.1 Basic Principle

We use the following approach to allow inlining of array fields that can change:

- At allocation, the child array with the initial size is co-allocated with the parent object, so an optimized access is possible.
- After the field has been overwritten with the 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 restores the optimized field order, i.e. the garbage collector colocates the new array to the parent object. Therefore, an optimized access is again possible.

Figure 8 illustrates this strategy. The array field `c` of the class `Parent` that references a `Child[]` array is inlined. In contrast to the examples in Section 3.1, the field can now be changed. Here, the array lengths are the constants 2 and 4 to simplify the example, but they could also be any non-constant value. Because the optimized access is not possible between the resize operation and the next garbage collection, an additional check is necessary before an element of the inlined array is accessed.

Array inlining saves one machine instruction, i.e. the load of the inlined field `c`, therefore it is only beneficial if the additional check does not add new instructions. It is therefore necessary to combine the additional check with the array bounds check that precedes every array access according to the Java specification. We set the length of the old inlined array to 0, which forces the array bounds check to fail. Instead of throwing an exception immediately, we check if the field has been changed and points to a new array. In this case, we access the new array and continue normally. Only if the bounds check for the new array also fails, an exception is thrown.

Figure 9 shows the normal and the optimized machine code for the array access. The optimized machine code is separated into a fast path and a slow path. The fast path

Unoptimized machine code:
```
// eax: p  esi: n
mov ebx, [eax+8]
cmp esi, [ebx+8]
jae Exception
mov ecx, [ebx+esi*4+12]
// ecx: p.c[n]
...
```

Optimized machine code:
```
// eax: p  esi: n
cmp esi, [eax+24]
jae Slowpath
mov ecx, [eax+esi*4+28]
Continue:
// ecx: p.c[n]
...

Slowpath:
mov ebx, [eax+8]
cmp esi, [ebx+8]
jae Exception
mov ecx, [ebx+esi*4+12]
jmp Continue
```

Figure 9: Optimized access of a dynamic array.

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 length of the old inlined array is set to 0, i.e. the memory location `[eax+24]` is set to 0, causing the bounds check to always fail. This case is regarded as uncommon and therefore placed out-of-line at the end of the method. It contains the same code as the unoptimized machine code, i.e. it first loads the field and then accesses the array using the normal offsets. Another bounds check throws the exception if necessary.

Dynamic array inlining is only beneficial if the majority of the array accesses use the fast path. The number of slow path accesses is high if the field is changed frequently or if the timeframe between the field change and the next garbage collection is long. In such cases, the array access should be reverted to the unoptimized machine code. Furthermore, dynamic array inlining does not allow to optimize direct accesses to the array length, e.g. when generating code for the `arraylength` bytecode. The additional check whether the result of the optimized access is 0 would require a compare and a branch instruction, which is more expensive than the normal field access.

4.2 Non-Destructive Approach

The basic principle described above has one important drawback: overwriting the array length destroys the array, i.e. it is no longer accessible because the bounds checks for all following accesses fail. This is no problem if the only reference to the array was the inlined array field, because this field has already been changed. However, the array could also be referenced by other fields or by root pointers. In this case, overwriting the array length is not allowed. An

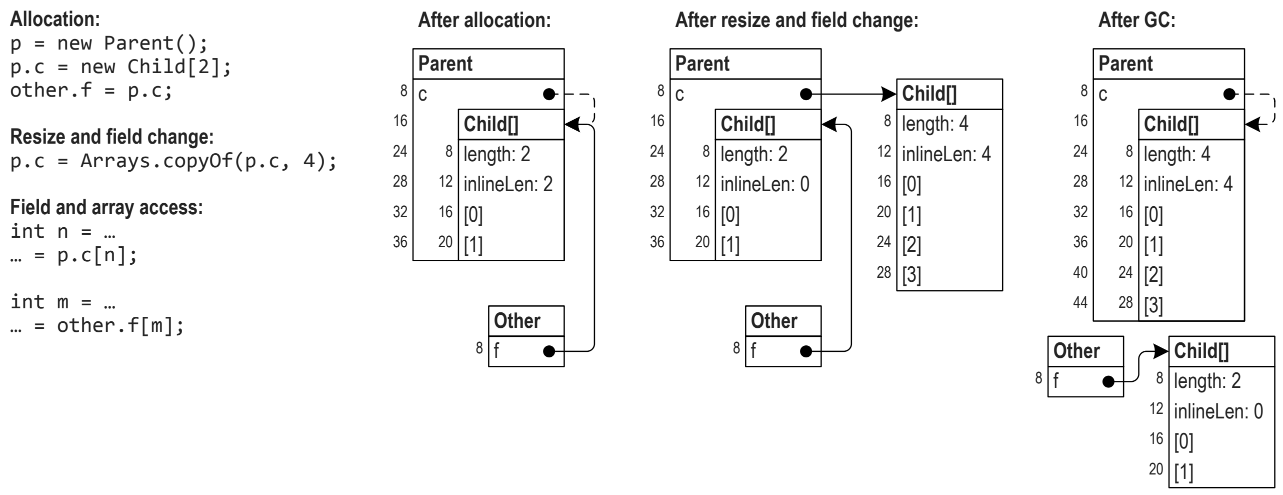


Figure 10: Non-destructive dynamic array inlining.

additional analysis must check if such accesses are possible before dynamic array inlining is initiated.

This can be done using a global data flow analysis. However, an analysis of frequently used classes like `ArrayList` shows that also a method-local bytecode analysis suffices for most cases. This analysis checks that a child array loaded from the inlined field is used only for an array access and not assigned to other fields or returned by the method. Nevertheless, additional analysis steps are necessary and the number of inlinable array fields is reduced.

A non-destructive approach avoids this problem by cloning the array length. Instead of overwriting the regular array length that is also accessed by normal bounds checks, a copy of the array length accessed only by the optimized machine code is overwritten. Figure 10 shows this approach. Each array has two length fields: `length` and `inlineLen`. In contrast to the previous example, the `Child[]` array is now also accessible from the field `f` of an `Other` object. When the field `Parent.c` is changed, the `inlineLen` of the inlined array is set to 0, but the `length` is unchanged. Therefore, the array access `other.f[m]` is still possible. The next run of the garbage collector restores the colocation of the `Parent` object and the new `Child[]` array. The old `Child[]` array is preserved and moved to another location.

To allow a uniform array access, all arrays in the heap must have the second length field, which increases the required heap space. It would also be possible to place the copy of the length field not into the array, but at the end of the parent object. This reduces the required memory, but leads to a more complicated inlining process because the size of parent objects is changed during inlining. Our implementation uses the simple non-destructive approach where all arrays have two length fields.

4.3 Garbage Collection

To be beneficial, dynamic array inlining requires that the timeframe between the modification of an array field and the next garbage collection is short. The Java HotSpot^{TM} VM divides the heap into two generations. A small young generation is collected frequently because most objects die young. The stop-and-copy collector copies young objects between two alternating spaces and increments an age field in each object. When the age exceeds a threshold, the object is promoted to the old generation. The old generation contains only long-living objects. It is larger than the young generation and collected rarely.

Therefore, it makes a difference whether the parent object is in the young or in the old generation. The new array is always in the young generation because it was recently allocated. However, the colocated object order can only be restored by a collection of the young generation when the parent is also still in the young generation. Heuristics in the age calculation can keep the parent object in the young generation as long as the inlined array field is changing. For example, we use the age of the child array instead of the age of the parent object for deciding whether the parent object should be promoted. When the array field is changed frequently, the age of the array is always low and the parent object is not promoted.

4.4 Array Bounds Check Elimination

Array bounds check elimination removes checks of array indices that are proven to be in the valid range (see for example [2] and [18] for Java implementations and [26] for an algorithm integrated into the Java HotSpot^{TM} client compiler). When the index variable is guaranteed to be below the array length, the check can be completely omitted (*fully redundant* checks). When the check is in a loop, the array length is loop invariant, and the maximum value of the index variable is known, the check can be moved out of the loop (*partially redundant* checks).

We use the bounds check also to detect changes of the array field. If the bounds check is eliminated, the optimized array access is no longer possible. Therefore, there is an optimization conflict between bounds check elimination and dynamic array inlining. In practice, such conflict situations are however unlikely. Bounds check elimination requires static information about the length of an array. If the array was just loaded from a field, which is the pattern optimized by array inlining, such information is usually not available and bounds check elimination therefore not possible. When bounds checks are moved out of a loop, the check whether the array field has been changed can still be performed outside the loop.

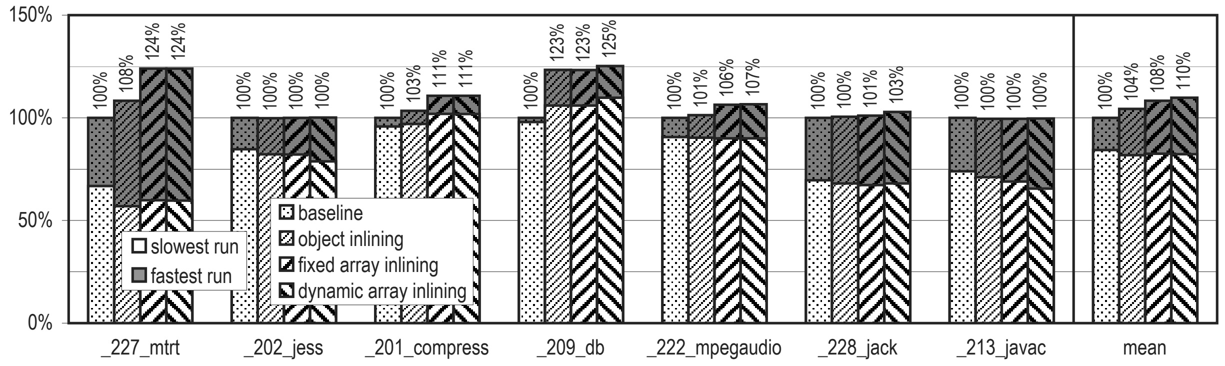


Figure 11: Speedup compared to baseline for SPECjvm98 (taller bars are better).

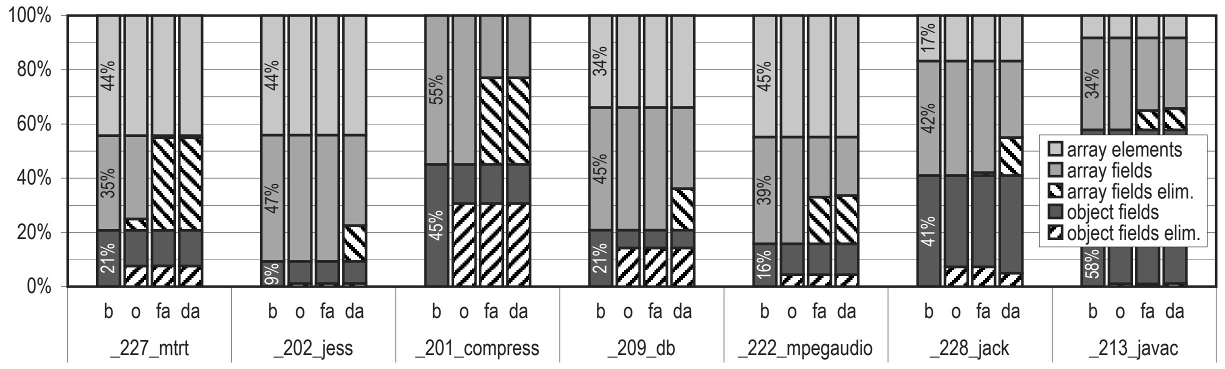


Figure 12: Total and eliminated field and array loads at run time.

5. EVALUATION

We integrated our object and array inlining algorithm into Sun Microsystems' Java HotSpot^{TM} VM, using an early snapshot version of the upcoming JDK 7 [21]. In addition to the optimizations of the snapshot version, our version of the client compiler also performs array bounds check elimination based on the algorithm of Würthinger et al. [26], which eliminates fully redundant checks and moves partially redundant checks out of loops.

All measurements were done on an Intel Core^{TM}2 Quad processor Q6600 with four cores at 2.4 GHz, running Microsoft Windows XP. Each core 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.

We evaluate our work using the SPECjvm98 [20] and DaCapo [1] benchmark suites. Each benchmark is executed five times in the same VM instance, and the slowest and the fastest runs are reported. The slowest runs, which are always the first in our case, represent the startup speed of the VM and include the time necessary for compilation, while the fastest runs show the peak performance after all optimizations have been applied.

5.1 Impact on Run Time

Figure 11 shows how object and array inlining affect the performance of SPECjvm98. We present results for the individual benchmarks, as well as the geometric mean of all benchmarks. The slowest and fastest 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 runs are shown relative to the same baseline. We compare four different configurations that can be selected using command line flags.

For the first column, *baseline*, read barriers are used to detect hot fields and then object colocation for these fields is performed, as described in [23]. The numbers include the overhead for the read barriers as well as the benefit of object colocation. In comparison to a version without these analyses, the slowest runs of our baseline are about 2% slower and the fastest runs about 5% faster in average. Read barriers and object colocation are a prerequisite for object inlining, so they are enabled in all subsequent configurations.

The second column, *object inlining*, shows the results when the analysis described in Section 2.2 is performed. The benchmarks _227_mtrt and _209_db show a speedup for the fastest runs. The slowest runs of these benchmarks are affected differently. For _227_mtrt, the analysis steps for object inlining and the compilation of the necessary methods run in the background during the whole first run, which causes a slowdown of about 15%. In contrast, the analysis succeeds early enough for the _209_db benchmark, so the optimized field loads outweigh the overhead.

For the column *fixed array inlining*, the same analysis is performed for array fields where the array length is a compile-time constant. In addition to the elimination of field loads, array bounds checks are simplified. The benchmarks _227_mtrt, _201_compress, and _222_mpegaudio are significantly improved. A detailed analysis shows that the simplified bounds checks contribute as much to the speedup as the eliminated field loads.

The last column, *dynamic array inlining*, contains the results for the non-destructive array inlining approach described in Section 4. It uses the extended bounds checks to detect changes of the array field. The benchmarks _209_db, _228_jack, and _222_mpegaudio show a small speedup. Because array bounds checks of dynamic arrays cannot be optimized, the impact is lower than the impact of fixed array inlining.

The same optimizations can be applied for variable arrays and for dynamic arrays. Therefore, our implementation does not differentiate between the two cases and treats variable arrays as if they were dynamic. The extended array bounds checks are also emitted for variable arrays, although they would not be necessary. Array fields that are initialized only once with arrays of a non-constant length are therefore included also in the last column of Figure 11.

5.2 Access Counts for Fields and Arrays

The speedup of object and array inlining is directly related to the number of eliminated field loads. Figure 12 shows the distribution of field and array loads and the impact of object and array inlining. The same four configurations as in the previous section are used. They are abbreviated as *b* (baseline), *o* (object inlining), *fa* (fixed array inlining), and *da* (dynamic array inlining).

We differentiate between three kinds of heap references. References to other objects and arrays on the heap can be loaded from *object fields*, *array fields*, or *array elements*. We only count loads of references, so the figure does not contain loads of scalar values like `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 the different object and array inlining configurations. The higher the striped bars are, the more field loads are eliminated. As described in Section 3.2, array elements cannot be optimized without a global data flow analysis. Therefore, the topmost bar shows no eliminated loads in any configuration.

Object inlining optimizes only object fields, i.e. the bottommost of the three kinds. For the benchmarks _227_mtrt, _201_compress, _209_db, _222_mpegaudio, and _228_jack, a significant part of the object field loads is eliminated. For _227_mtrt, the number of accessed array fields is reduced slightly though no array fields are optimized. This is a beneficial side effect of object inlining on other optimizations like global value numbering. When it is known that a field is never changed, some subsequent array loads are identified as redundant and therefore eliminated.

Fixed array inlining eliminates loads of array fields when the lengths of the arrays are compile-time constants. Such arrays are frequently accessed in the benchmarks _227_mtrt, _201_compress, and _222_mpegaudio. Together with object inlining, 75% of all reference field loads of _227_mtrt and 63% of _201_compress are eliminated. This is also reflected by the high speedups for these benchmarks.

Dynamic array inlining is effective for the benchmarks _202_jess, _209_db, and _228_jack where mostly the array-based collection class `Vector` is optimized. For _228_jack, dynamic array inlining reduces the number of eliminated object field loads. The benchmark contains a parent class with three object fields that can be inlined using object inlining. Each of these fields references a collection. With dynamic array inlining, the data arrays of the collections are also inlined. Therefore, the offsets of the second and the third child object are no longer fixed, so inlining of them fails and only one object field is inlined.

	total		object		fixed array		dyn. array	
	obj	arr	obj	arr	obj	arr	obj	arr
_227_mtrt	380	77	3	0	3	4	3	4
_202_jess	410	77	5	0	5	0	5	3
_201_compress	352	76	5	0	5	4	5	4
_209_db	350	74	2	0	2	0	2	2
_222_mpegaudio	412	93	7	0	7	9	7	11
_228_jack	395	80	4	0	4	1	2	3
_213_javac	492	92	2	0	2	1	2	2

Table 1: Number of fields that are optimized.

5.3 Statistics of Inlinable Fields

Object and array inlining is performed field-by-field at run time. To limit the analysis overhead, the algorithm processes only important fields for which the time invested in the optimization is outweighed by the later speedup. Therefore, only fields whose access counters exceed a certain threshold are considered as candidates for inlining. Table 1 shows the number of fields that are inlined in the configurations mentioned above.

The first column, *total*, shows the overall number of fields in all loaded classes. The majority of the several hundred fields are unimportant for the performance of the benchmarks. The number of inlined fields shown in the subsequent columns is therefore below 10 for all benchmarks except _222_mpegaudio. For example, the high percentage of eliminated reference field loads for _227_mtrt is achieved with only 7 inlined fields. These fields are the most frequently accessed ones, so optimizing all of the remaining fields can only lead to a small additional speedup.

For some benchmarks like _213_javac, our analysis is not yet capable of inlining the most important fields. Although some fields are inlined, there is nearly no impact on the eliminated field loads and the speedup. To keep the analysis overhead low, the algorithm is conservative in several places. An improved algorithm can possibly inline more important fields for these benchmarks.

5.4 DaCapo Benchmarks

Figure 13 shows the benchmark results for 7 of the 11 DaCapo benchmarks as well as the geometric mean of all 11 benchmarks. The 4 excluded benchmarks do not show a speedup in any of the object and array inlining configurations. The DaCapo benchmarks are more complex than the SPECjvm98 benchmarks regarding code complexity, class structures, and class hierarchies. Therefore, the speedups are lower compared to SPECjvm98. Nevertheless, some benchmarks like luindex, antlr, or eclipse show a considerable speedup.

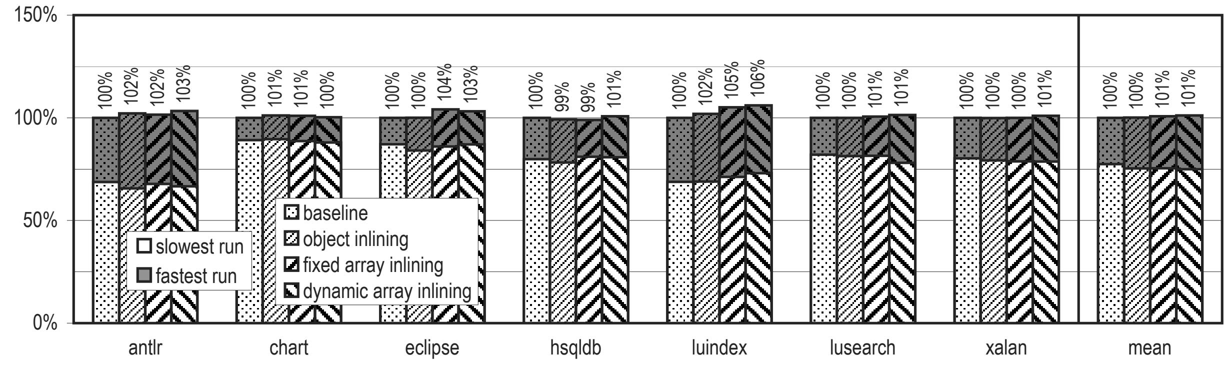


Figure 13: Speedup compared to baseline for DaCapo (taller bars are better).

6. RELATED WORK

Dolby et al. extended a static compiler for a dialect of C++ with an algorithm for automatic object inlining [6]. Their algorithm clones the code of methods that access optimized objects. Therefore, there can be both optimized and unoptimized objects of the same class, while we require that all objects of a class are optimized. Additionally, they remove object headers and pointers to inlined objects, which reduces the object size. As they use an advanced global data flow analysis, they are able to convert arrays of references to arrays of object values [4]. As described in Section 3.2, our analysis is not capable of such transformations. However, we can handle dynamic arrays where an array field can be changed to reference arrays of different sizes. This is not possible in their approach, as mentioned in [5]. The optimization of such dynamic data structures is the main advantage of a run-time analysis over a static compilation approach. The high compilation time reported for their global data flow analysis would prohibit the integration of their analysis into a virtual machine. The C++ benchmarks show an average speedup of 10%, with a maximum of 50% for some benchmarks.

Laud implemented object inlining in the CoSy compiler construction framework, a static compiler for Java [15]. This algorithm can detect when child objects are replaced with new ones, which could be used for the inlining of dynamic arrays. However, no details regarding arrays are published. In contrast to our algorithm, it is not allowed that a child object is referenced by anything else than the parent object. To the best of our knowledge, no benchmark results are available.

Lhoták et al. provide a good introduction to object inlining [16]. Depending on the access pattern, they differentiate between 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 for several Java benchmarks. Array fields are not evaluated thoroughly in their study. Wile they present the access counts of array fields and compare them with the number of object fields, their subsequent analysis and listing of inlinable fields does not cover array fields. Additionally, they do not distinguish between arrays with constant and arrays with variable length. Inlining of array elements is not evaluated. The study does not describe an implementation for object inlining, so no benchmark results are published, except from three hand-optimized benchmarks.

The algorithm for *object combining* by Veldema et al. groups objects that have the same lifetime [22]. It is more aggressive than object inlining because it also optimizes unrelated objects if they have the same lifetime. This allows the garbage collector to free multiple objects simultaneously. Elimination of pointer accesses is performed separately. Inlining of one array with a variable length per object group is possible. The focus of their optimization is 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 the costs of allocation and deallocation are higher.

Ghemawat et al. use a cheap interprocedural analysis for object inlining and for other global optimizations [7]. Arrays are only inlined when the length is a compile-time constant. Arrays with variable size are not optimized. The analysis was implemented in Swift, an optimizing static Java compiler for the Alpha architecture. There are no timing results with only object inlining enabled.

Several approaches modify the object copying order of the garbage collector to improve the cache performance, but do not optimize field loads. The *online object reordering* of Huang et al. optimizes all fields accessed by frequently executed methods [11]. The *cache-conscious data placement* of Chilimbi et al. uses run-time counters, but does not distinguish between different fields of the same object [3]. Shuf et al. co-allocate objects of frequently instantiated types, called *prolific types*, and then preserve this order during garbage collection [19].

Algorithms for *pointer analysis* make assumptions about possible run-time values of pointers [9]. Most of these static analyses are however unsuitable for object and array inlining because they detect only to which locations a variable *may* point. Wu et al. collect instance-wise points-to information which is precise also in the presence of loops [25]. This analysis could be used to flatten multi-dimensional arrays and to inline array elements into arrays because their *element-wise points-to mapping* connects arrays with the objects referenced by the array elements.

7. CONCLUSIONS

We presented an algorithm for automatic array inlining in a Java virtual machine. Arrays play an important role in most Java applications and are frequently used to model dynamic data structures such as collections. They can be referenced by array fields, and their elements can contain references to objects. Array inlining without a global data flow analysis can only optimize array fields. The optimization of array elements is complicated because the array access bytecodes do not have any static type information.

We distinguish between array fields that are assigned only once, referencing arrays of fixed or variable size, and array fields that can be reassigned over time. The access of such dynamic arrays is optimized by integrating the check whether the field has been changed into the array bounds check. The evaluation shows that array inlining can achieve a significant speedup by only optimizing a handful of fields.

8. ACKNOWLEDGEMENTS

We would like to thank the Java HotSpot^{TM} 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^{TM} VM. We also thank Thomas Kotzmann and Thomas Würthinger for their valuable comments on the work and this paper.

9. REFERENCES

[1] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190. ACM Press, 2006.

[2] R. Bodík, R. Gupta, and V. Sarkar. ABCD: Eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–333. ACM Press, 2000.

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

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

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

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

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

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

[9] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61. ACM Press, 2001.

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

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

[12] ISO/IEC. *Common Language Infrastructure (CLI).* International Standard ISO/IEC 23271, 2nd edition, 2006.

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

[14] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot^{TM}client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 2008.

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

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

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

[18] F. Qian, L. J. Hendren, and C. Verbrugge. A comprehensive approach to array bounds check elimination for Java. In *Proceedings of the International Conference on Compiler Construction*, pages 325–342. LNCS 2304, Springer-Verlag, 2002.

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

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

[21] Sun Microsystems, Inc. *JDK 7 Project*, 2007. https://jdk7.dev.java.net/.

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

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

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

[25] P. Wu, P. Feautrier, D. Padua, and Z. Sura. Instance-wise points-to analysis for loop-based dependence testing. In *Proceedings of the International Conference on Supercomputing*, pages 262–273. ACM Press, 2002.

[26] T. Würthinger, C. Wimmer, and H. Mössenböck. Array bounds check elimination for the Java HotSpot^{TM} client compiler. In *Proceedings of the International Conference on Principles and Practice of Programming in Java*, pages 125–133. ACM Press, 2007.