Automatic Object Colocation Based on Read Barriers*

Christian Wimmer and Hanspeter Mössenböck

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

Abstract. *Object colocation* is an optimization that reduces memory access costs by grouping together heap objects so that their order in memory matches their access order in the program. We implemented this optimization for Sun Microsystems' Java HotSpot™ VM. The garbage collector, which moves objects during collection, assigns consecutive addresses to connected objects and handles them as atomic units.

We use read barriers inserted by the just-in-time compiler to detect the most frequently accessed fields per class. These "hot fields" are added to so-called *hot-field tables*, which are then used by the garbage collector for colocation decisions. Read barriers that are no longer needed are removed in order to reduce the overhead. Our analysis is performed automatically at run time and requires no actions on the side of the programmer.

We measured the impact of object colocation on the young and the old generation of the garbage collector, as well as the difference between dynamic colocation using read barriers and a static colocation strategy where colocation decisions are done at compile time. Our measurements show that object colocation works best for the young generation using a read-barrier-based approach.

1 Introduction

Object-oriented applications tend to allocate large numbers of objects that reference each other. If these objects are spread out randomly across the heap, their access is likely to produce a large number of cache misses. This can be avoided if objects that reference each other are located consecutively. Changing the object order so that related objects are next to each other is called *object colocation*. It is conveniently implemented as part of garbage collection where live objects are moved to new locations. In general, the access pattern of objects cannot be determined statically because it depends on how the program is used and which classes are dynamically loaded. Therefore the analysis of access patterns must be done at run time.

* This work was supported by Sun Microsystems, Inc.

D. Lightfoot and C. Szyperski (Eds.): JMLC 2006, LNCS 4228, pp. 326–345, 2006.

Figure 1 shows an example of an object graph taken from the benchmark _227_mtrt of the SPECjvm98 benchmark suite [12]. Objects of the instance classes `OctNode`, `Face` and `Point` as well as of the array classes `Face[]` and `Point[]` form an access path that accounts for 70% of all reference field loads. The objects have a size of 16 to 40 bytes, so up to four objects fit in a typical cache line of 64 bytes. Colocating the objects reachable from an `OctNode` object therefore reduces the memory access costs when a `Point` of an `OctNode` is accessed.

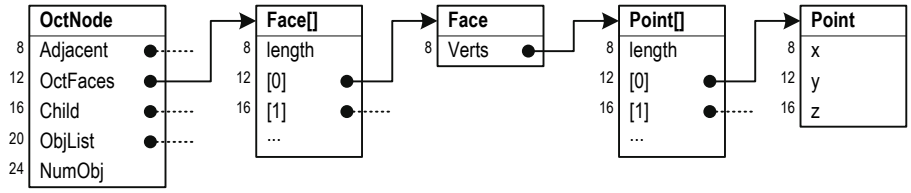

Fig. 1. Motivating example for object colocation

A static colocation strategy, e.g. one that always colocates the object referenced by the first field of another object, is only suitable for simple data structures. For objects with more than one reference field, the first field is typically not the most frequently accessed one. For example, the field `OctFaces` of `OctNode` objects has a 10 times higher access frequency than the first field `Adjacent`.

We implemented a dynamic analysis for Sun Microsystems' Java HotSpot™ VM that identifies frequently accessed "hot fields" on a per-class basis using read barriers. If a field counter reaches a certain threshold, the field is added to the *hot-field table* of the according class. The read barriers are inserted into the machine code by the just-in-time compiler. To minimize the run-time overhead, read barriers that are no longer needed are removed.

The garbage collector uses the hot-field tables to decide which objects should be colocated and assigns consecutive addresses to these objects when they are moved during collection. This goes beyond previous approaches that modify only the order in which a copying collector processes reference fields: We treat a set of colocated objects as an atomic unit and guarantee that it is not separated in a later garbage collection run. This paper contributes the following novel aspects:

- We implemented object colocation in a system with dynamic class loading and different garbage collection algorithms.
- We use read barriers inserted by the just-in-time compiler to get a dynamic field access profile with a negligible run-time impact.
- We evaluate our implementation and compare different configurations of the garbage collector. We also compare the dynamic read-barrier-based approach with a static colocation strategy.

2 System Overview

Figure 2 shows the structure of the Java HotSpot^{TM} VM with the relevant subsystems. We modified the default configuration for interactive desktop applications, called the *Client VM*, which uses a fast just-in-time compiler and a generational garbage collector with two generations. The Client VM is available for Intel's IA-32 and Sun's SPARC architecture, but object colocation is currently only implemented for the IA-32 architecture because the code patterns for the read barriers are platform dependent.

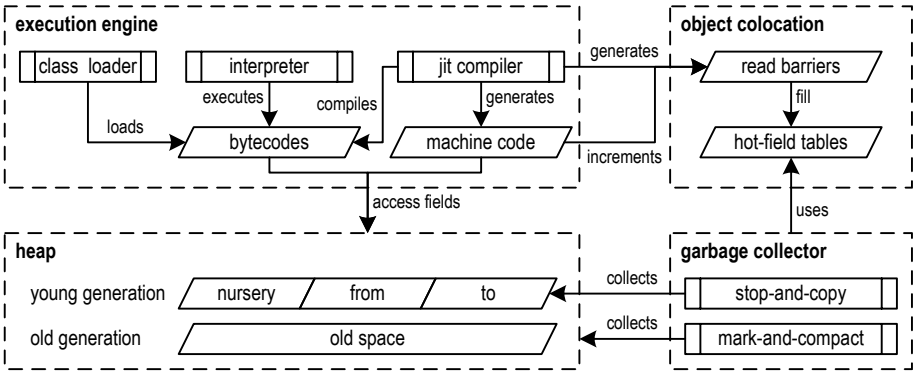

Fig. 2. System structure of the Java HotSpot^{TM} VM

Methods are loaded by the class loader and start being executed by the interpreter. Only frequently executed methods are compiled to minimize the compilation overhead. Both interpreted and compiled methods access objects in the heap, which is divided in a young and an old generation. The young generation is collected using a stop-and-copy algorithm that copies live objects between alternating spaces. A full collection of both generations is done using a mark-and-compact algorithm [7]. Section 4 presents details of these algorithms.

The garbage collector accesses the hot-field tables that store parent-child relationships of classes whose objects should be colocated. The access profile of fields is collected by read barriers, which are emitted by the just-in-time compiler into the generated machine code and increment a counter for each field load. Fields with high counter values are added to the hot-field tables. Section 3 presents the code patterns used for the read barriers.

A parent object and the child object referenced by the parent's most frequently accessed field are placed next to each other in the heap. If a second field also has high access counts, the corresponding child is placed consecutively to the first one, and so on. Objects referenced by fields with low access counts are not colocated because the optimization of rarely accessed data structures does not pay off.

Accesses to array elements are counted in the same way as field accesses. However, the colocation of objects referenced by array elements is more complicated because all elements are usually accessed with similar frequencies. As a pragmatic solution, we colocate only the object referenced by the first element.

2.1 Hot-Field Tables

A hot-field table is a VM-global data structure that is built for every class with hot fields. It is rooted in the data structure maintained by the VM for each loaded class, which already stores information such as super- and subclasses, fields and methods of the class.

Figure 3 shows a fragment of the hot-field tables for our example benchmark _227_mtrt. The table for a *parent class* stores a list of child entries for its hot fields. Each entry holds the offset (*off*) of the field as well as the field's declared *child class*. The order of the children is important: The first entry of the list is processed first by the garbage collector, so the first child is placed consecutively to the parent.

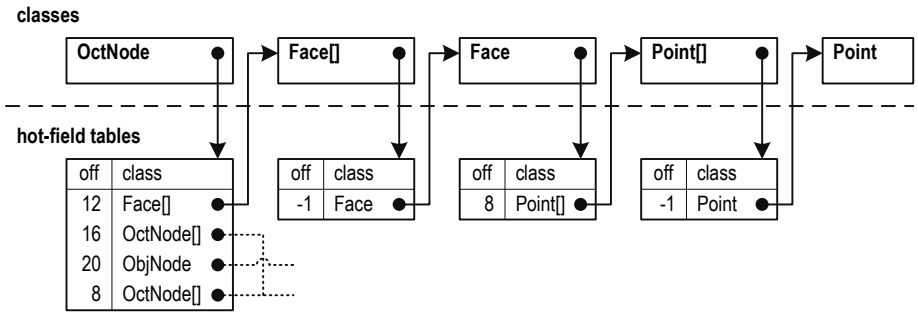

Fig. 3. Example of hot-field tables

The list contains only frequently accessed children because optimizing rarely accessed data structures only introduces overhead to the garbage collector. There is no table for the class `Point` because this class only stores scalar fields. Since array classes do not have a list of fields with according field offsets, the special marker value -1 is used as the offset in the table. It is replaced by the index of the first non-null array element when an actual object graph is constructed.

A hot-field table does not contain child entries for fields declared in a superclass. Instead, the superclass has its own hot-field table. Similarly, a table contains only direct children. Indirect children are only implicitly visible: the class of a child entry also has its own hot-field table. In our example, a `Face[]` object is a direct child of an `OctNode` object, while `Face`, `Point[]` and `Point` objects are indirect children of this `OctNode`. During garbage collection, all direct and indirect children of an object are captured by a separate table, which is discussed in Sect. 4.1.

2.2 Identifying Hot Field Loads

The hot-field tables are filled dynamically at run time. To achieve the best results, they should contain only the most frequently accessed fields in the correct order. We use read barriers emitted by the just-in-time compiler to detect hot field accesses. Section 3.1 presents the code that is inserted by the compiler. We count only field loads and not field stores because a high number of stores can indicate a frequently changing data structure where object colocation is difficult or even impossible. Furthermore, we ignore loads of scalar fields and emit read barriers only for loads of reference fields.

We also experimented with a static approach to object colocation where the just-in-time compiler fills the hot-field tables with all fields that are accessed in compiled methods instead of emitting read barriers. The resulting hot-field tables are bigger, but still useful because only a small fraction of methods is compiled and the tables do not contain fields that are accessed only by interpreted methods. In Sect. 5.2 we compare the two approaches.

3 Read Barriers

Read barriers allow dynamic measurements of an application's memory access behavior. A read barrier is a piece of machine code that is inserted after the code that performs the actual load of a reference field. We use two different kinds of read barriers:

- A *simple read barrier* identifies frequently accessed fields that are worth being optimized by object colocation.
- A *detailed read barrier* collects data for the analysis and verification of the optimizations. It counts the number of field accesses where a parent object and its child objects are colocated as well as in the same cache line.

Simple read barriers are a prerequisite of our object colocation and therefore always enabled. In contrast, detailed read barriers are currently not intended for production use. When analyzing the impact of object colocation, as presented in Sect. 5.2, detailed read barriers are enabled via a VM flag.

The read barriers are inserted by the just-in-time compiler because it has full information about fields: The instruction for a field access in the compiler's intermediate representation contains the class that declares the field (the parent class), the field offset and the type of the field (the child class). With this information, a unique counter is created for each field. When the same field is accessed in different methods, the same counter is used. The few field accesses that are performed by the interpreter are thus not counted, but this does not affect the precision of the measurements.

The address of a counter is statically known and can be directly emitted into the machine code. This allows a read barrier to be efficiently implemented as a

single increment instruction, which nevertheless counts only accesses to a particular field of a class. Section 3.1 shows the details of the emitted instructions.

Our read barriers take compiler optimizations into account: The compiler eliminates a field load if the value of the field is known at compile time or if the load is redundant, and also does not emit a read barrier for these loads. So the resulting counter values can be lower than a naive counting using an instrumented interpreter, but they better reflect the actual behavior of an application.

3.1 Code Patterns for Read Barriers

Figure 4 shows the code pattern for a simple read barrier that increments a counter for a field load. Assume that the field at offset 8 is to be loaded, that the object's address is already in register `eax`, and that the counter is located at the fixed address `5000h`. The IA-32 instruction set allows instructions to operate on memory operands [6], so it is not necessary to load the counter value into a register. Only a single instruction is emitted for the increment.

```
          ...                     // eax: base address of object
          mov  ebx, ptr [eax+8]   // access field at offset 8
          inc  ptr [5000h]        // increment counter
          ...                     // ebx: result of field load
```

Fig. 4. Code pattern for a simple read barrier

A simple read barrier is sufficient for identifying hot fields, but for the evaluation of object colocation we are also interested in statistical data about colocated objects and the cache behavior. Figure 5 shows the code pattern for a detailed read barrier that checks if a parent object and a child object are located in the same cache line.

```
          ...                     // eax: base address of object
          mov  ebx, ptr [eax+8]   // access field at offset 8
          dec  ptr [4000h]        // decrement slowcase counter
          jle  slowcase           // slowcase if counter reaches 0
continue: ...                     // ebx: result of field load

slowcase: mov  ptr [4000h], 1000  // reset slowcase counter
          inc  ptr [5000h]        // increment total counter
          lea  esi, ptr [eax+8]   // compute address of field
          xor  esi, ebx           // check if address and value of
          and  esi, 0FFFFFFC0h    //   field are in same cache line
          jne  skip_inc
          inc  ptr [5004h]        // increment cache line counter
skip_inc: ...                     // check for object colocation
          jmp  continue
```

Fig. 5. Code pattern for a detailed read barrier

Executing the complete sequence of more than 15 instructions for each field load would be too expensive. Therefore, the code is placed in a *slow case* [4] that collects data only for every 1000$^{\text{th}}$ field load. A global counter is decremented at each field load. Assume that the address of this counter is `4000h`. When the counter reaches 0, the slow case is executed.

The slow case resets the counter to 1000 and increments a *total counter* at the address `5000h`. The address of the referencing field is loaded to `esi`, and the address of the referenced object is already in `ebx`. These addresses are in the same cache line with a size of 64 bytes if all but their lower 6 bits are identical, which is checked using the `xor` and `and` instructions. In this case the *cache line counter* at the address `5004h` is incremented. Dividing the cache line counter by the total counter yields the percentage of objects that are in the same cache line.

The slow case also computes a *colocation counter* that counts the number of cases in which the parent object and its child are colocated. This part of the code has been omitted from Fig. 5 because it is similar to the code for computing the cache line counter. If the base address of the parent object (which is in `eax`) plus the size of the object (which is retrieved via the class pointer stored in the object's header) equals the address of the child object (which is in `ebx`), the objects are colocated and the counter is incremented.

3.2 Processing of Counters

When the counter of a field has exceeded a certain threshold at the time of the next garbage collection, the field is recorded in the hot-field table of the parent class. The time between two garbage collections is used as the measurement interval. We want to record fields that are accessed frequently in this period, and to filter out the large number of fields that are accessed infrequently. As a heuristic, a field is added to the hot-field table if it accounts for more than 6% of all field loads in the last period.

The heuristic fills the tables iteratively: At the first garbage collection, fields with an exceptionally high access frequency (and therefore a high percentage) are added to the hot-field tables. Their read barrier counters are then invalidated and ignored when computing the percentages at the second garbage collection, so the next fields with still a high access frequency are added. This is repeated until a stable state is reached where most fields have similar access frequencies, so no single one is above 6%.

Incrementing a counter for each field load involves some run-time overhead. Therefore, read barriers are removed as soon as they are no longer needed, i.e. after the corresponding field was added to the hot-field table or if the access count was low for a long time. This is done by recompiling all methods that increment the read barrier's counter. The machine code of those methods is marked so that the compiler is invoked when the method is called the next time. Because read barriers whose counters were invalidated are ignored during compilation, the new code does not contain these read barriers anymore.

4 Modifications of Garbage Collection Algorithms

The Java HotSpot™ VM uses a generational garbage collection system with different collection algorithms. The default configuration uses two generations with a *stop-and-copy* algorithm for the young generation and a *mark-and-compact* algorithm for a full collection of both generations.

When the young generation is collected, live objects are copied between two alternating spaces, called the *from-space* and the *to-space*. After several copying cycles, an object is promoted to the old generation. New objects are allocated in a separate *nursery space* of the young generation that is treated as a part of the from-space during collection.

When the old generation is full, the entire heap is collected by a mark-and-compact algorithm. All live objects are marked and then moved towards the beginning of the heap in order to eliminate gaps between live objects. This takes more time than a collection of the young generation, but it is only necessary if no more space is available for the promotion of young objects.

We integrated our object colocation algorithm into both algorithms and allow switching it on and off independently. This allows us to evaluate the benefits of the optimization in both generations. However, enabling object colocation for the young generation also affects the old generation: Groups of colocated objects are promoted together, so the order of objects in the old generation is also partly optimized. Because the unmodified mark-and-compact algorithm does not change the object order, the optimized order is preserved.

4.1 Colocation Tables

The hot-field tables introduced in Sect. 2.1 are easy to maintain because they store only direct children. However, it is expensive to detect all direct and indirect children that should be colocated to a particular parent object. To limit the overhead during garbage collection, an additional *colocation table* is created from the hot-field table for each class. Figure 6 shows the colocation tables for the running example of _227_mtrt.

Each table contains a flat list of all fields that should be colocated for a given class. It is created once before garbage collection, and filled with objects multiple times during garbage collection. The first entry stores the parent object for which the table is filled; all other entries are direct or indirect children of this object. The columns contain the following information:

- Field offset (*off*): The offset of the field whose value is stored in this entry, or -1 as a marker for arrays.
- Parent entry (*par*): The index of this object's immediate parent in the same table. It is 0 for direct children and greater than 0 for indirect children of the parent object for which the table is filled.
- Object (*obj*): The actual object that is referenced by this field. It is filled in during garbage collection.

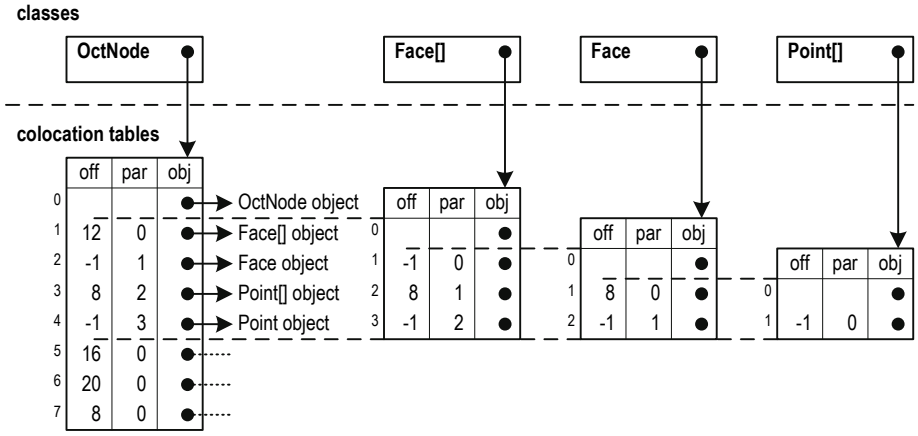

Fig. 6. Example of colocation tables used during garbage collection

In our example, all information required for the colocation of direct and indirect children of an `OctNode` object is contained in the colocation table for `OctNode`: The entries with the indices 1, 5, 6 and 7 denote fields that reference direct children of the `OctNode` object with the index 0. The entries 2, 3 and 4 denote indirect children of the `OctNode` object as shown in Fig. 1. They are direct children of the entries 1, 2 and 3, respectively.

If a `Face[]` object is not colocated to an `OctNode` object, we need the colocation table of `Face[]`. This table is smaller because it contains only the objects that are colocatable to a `Face[]` object. Similarly, there are colocation tables for the classes `Face` and `Point[]`. These tables contain a part of the information of the bigger tables, indicated by the dashed lines in Fig. 6.

The algorithm GetChildren of Fig. 7 is used during garbage collection to fill a colocation table with the children of a specific parent: After the parent object has been stored in the first entry, all its children are iterated. Because the immediate parent of a child is always located before this child, *c.par* has already been added to the table before *c* and the field with the offset *c.off* of the object *c.par.obj* can be accessed.

```
GetChildren(obj)
  tab = colocation table for class of obj
  if tab not found then
    return empty table

  tab[0].obj = obj                  // initialize first entry, which holds the parent
  for i = 1 to tab.length - 1 do    // iterate all entries except the first
    c = tab[i]                      // get the entry of the current child
    c.obj = c.par.obj.fieldAt(c.off)  // access the field at the specified offset

  return tab
```

Fig. 7. Algorithm for filling a colocation table during garbage collection

4.2 Stop-and-Copy Collection of the Young Generation

Figure 8 shows the basic StopAndCopy algorithm. First, all objects referenced by root pointers are copied from the from-space to the to-space using CopyObject. Allocating memory in the to-space requires only an increment of the *end* pointer. Each object of the from-space that has been copied stores a forwarding pointer to its new location. All objects referenced by copied objects are also alive and must therefore be copied as well. The algorithm uses the to-space as a queue and scans all copied objects in sequential order. The forwarding pointer is used to prevent copying an object twice.

```
StopAndCopy
  toSpace.end = toSpace.begin
  for each root pointer r do
    r = CopyObject(r)

  obj = toSpace.begin
  while obj < toSpace.end do
    for each reference r in obj do
      r = CopyObject(r)
    obj += obj.size
```

```
CopyObject(obj)
  if obj is forwarded then
    return obj.forwardee

  newObj = toSpace.end
  toSpace.end += obj.size
  memmove(obj, newObj, obj.size)
  obj.forwardee = newObj
  return newObj
```

Fig. 8. Stop-and-copy algorithm used for collection of the young generation

This breath-first copying scheme is simple and efficient, but it leads to a random order of copied objects in the to-space. An object is copied when the first reference to it is scanned. A depth-first copying scheme, where all referenced objects are copied immediately after the object itself, would result in a better object order, but it would require an explicit stack of objects to be scanned.

We extended the breath-first copying so that it processes groups of objects instead of individual objects. When a parent object has to be copied, the colocation table is filled using the algorithm GetChildren. All child objects in the table are copied together with their parent object. The necessary memory for the object group is allocated at once. Figure 9 shows the modified algorithm for CopyObject. The handling of child objects that are already in the old generation has been omitted from the algorithm; such children are simply ignored.

The root pointers are processed in an arbitrary order. If both the parent object and a child object are referenced by a root pointer, it can happen that the child object is copied before the parent. Because an object must not be copied twice, the two objects cannot be colocated in this garbage collection run. This is checked in the algorithm before colocating a child.

To avoid children that are copied before their parent, objects that were once detected to be colocation children are tagged with a dedicated bit in the object header, referred to as *isColocationChild* in the algorithm. The copying of tagged objects is delayed until the parent object is processed, so the colocation succeeds. Because all children keep the tag for their entire lifetime, objects are guaranteed to stay colocated even if new root pointers to children are introduced.

```
CopyObject(obj)
  if obj is forwarded then
    return obj.forwardee                  // prevent copying an object twice
  if obj.isColocationChild then
    return fixupMarker       // delay copying; a fixup is done when obj is copied later

  tab = GetChildren(obj)                  // get children of obj (may return empty table)
  allocSize = obj.size                    // computation of total allocation size
  for i = 1 to tab.length - 1 do
    tab[i].obj.isColocationChild = true   // tagging of object as colocation child
    if tab[i].obj is not forwarded then
      allocSize += tab[i].obj.size        // only non-forwarded objects can be colocated

  newObj = toSpace.end                    // allocate memory for parent and all children
  toSpace.end += allocSize

  memmove(obj, newObj, obj.size)          // copy and forward parent object
  obj.forwardee = newObj
  offset = obj.size
  for i = 1 to tab.length - 1 do          // copy and forward all children
    if tab[i].obj is not forwarded then
      memmove(tab[i].obj, newObj + offset, tab[i].obj.size)
      tab[i].obj.forwardee = newObj + offset
      offset += tab[i].obj.size

  return newObj
```

Fig. 9. Object colocation for the stop-and-copy algorithm

When the copying of a child object is delayed, the references to the child require a later fixup. `CopyObject` returns a fixup marker and the reference is added to a list. When the scan of the to-space is completed, these references are updated to the forwarding pointer of the child object that was set during the colocated copying. In rare cases it can happen that the parent object died, but the child object is still alive because another object holds a reference to the child. Similarly, a field update of the parent object can install a new child object and leave the old one without a parent. Such objects are still uncopied in the fixup phase, so they are copied before the fixup and the colocation bit is cleared.

4.3 Mark-and-Compact Collection of the Old Generation

The stop-and-copy collection of the young generation also affects the old generation because colocated objects are promoted together and are therefore already partly colocated in the old generation. However, a collection of the young generation can only colocate a group of objects if all members of this group are still in the young generation. If a child has already been promoted, it cannot be colocated. In contrast, a collection of the entire heap can colocate all objects.

The mark-and-compact algorithm processes all live objects of the old and the young generation. Because it places all objects contiguously into the old generation, this is called a collection of the old generation. It requires four phases:

1. *Mark live objects*: The heap is traversed recursively starting with the root pointers to mark all live objects.
2. *Compute new addresses*: In a linear walk through the heap each object is assigned a new address, which is stored in the object's forwarding pointer. Because gaps between live objects are removed, objects move towards the beginning of the heap.
3. *Adjust pointers*: All root pointers and inner pointers of objects are updated to point to the new addresses stored in the forwarding pointers of the referenced objects.
4. *Move objects*: In another linear walk through the heap the objects are copied to their new locations. Because objects move only towards the beginning of the heap, the memory of the new location can be overwritten without precautions.

The basic mark-and-compact algorithm preserves the order of objects. This simplifies object colocation because the correct order needs to be established only once. We extended the basic algorithm by modifying the phases 1, 2 and 4 in the following way:

In phase 1, all parents and children are detected. For each object whose class has a colocation table, we use GetChildren (see Fig. 7) to fill the table with the actual children of this object. The children are then tagged as in the stop-and-copy algorithm.

When a parent object is processed in phase 2, GetChildren must be called again because there is only one colocation table per class. All children of this object get consecutive addresses assigned. This may change the order of objects in the heap. With the help of the tags that are set in phase 1, the processing of a child is delayed when it would be processed before its parent. So a child object never gets a new address assigned before its parent. As a result, child objects can now move also towards the end of the heap.

Since objects can now also move towards the end of the heap, phase 4 must take precautions to rescue such objects. They are first copied into a scratch area and then copied back to their final location after all other objects were processed. However, this is only necessary for a small number of objects because at the next collection the object order is already correct, so no reordering and no rescuing is necessary.

5 Evaluation

We integrated our object colocation algorithm in the Java HotSpot™ VM of Sun Microsystems, using a development snapshot of the upcoming Java SE 6 called Mustang [14]. Currently, we work with the Mustang build 66 from January 2006. Compared with the current JDK 5.0, the VM of this build includes optimizations such as a new object locking scheme called biased locking, and an improved

just-in-time compiler using an intermediate representation in static single assignment form and a linear scan register allocator [16].

All measurements were performed on an Intel Pentium D processor 830 with two cores running at 3.0 GHz. Each core has a separate L1 data cache of 16 KByte and an L2-cache of 1 MByte. The cache line size is 64 bytes for both caches. The main memory of 2 GByte DDR2 RAM is shared by the two cores. Microsoft Windows XP Professional was used as the operating system. Both garbage collection algorithms are neither parallel nor concurrent. Therefore, the second core of the processor is idle during garbage collection. We evaluated our work with the SPECjbb2005 benchmark [13] and the SPECjvm98 benchmark suite [12].

The SPECjbb2005 benchmark[1] emulates a client/server application. The resulting metric is the average number of transactions per second executed on a memory-resident database. Since the default maximum heap size of the HotSpot^{TM} VM is too small for this benchmark, the heap was enlarged to 512 MByte via a VM flag.

The SPECjvm98 benchmark suite[2] consists of seven benchmarks derived from real-world applications, which cover a broad range of scenarios where Java applications are deployed. They are executed repeatedly until there is no significant change in the execution time any more. The speedup of the fastest run compared to a reference platform is reported as the metric for each benchmark, and the geometric mean of all metrics is computed.

Scientific applications usually operate on large arrays, so no performance gain can be expected from object colocation. However, there should also be no slowdown due to read barriers or additional garbage collection overhead. In order to verify this, we performed all measurements of the next sections also for SciMark 2.0 [11], a benchmark for scientific applications that executes several numerical kernels. All configurations of read barriers and object colocation showed the same results.

5.1 Read Barriers

Read barriers impose a run-time overhead because additional code must be executed for each field load. Table 1 compares the baseline version where all our changes are disabled, simple read barriers as described in Sect. 3.1 that are always enabled, and simple read barriers that are removed when the counters reach the threshold as described in Sect. 3.2. Object colocation was disabled for all measurements, so no optimizations were performed.

Counting all field loads leads to an average overhead of 30%, with a maximum slowdown of nearly 80% for the field-access-intensive benchmark _227_mtrt. This shows that such a naive read barrier is unfeasible, so we have to remove unnecessary read barriers. When read barriers are removed, the slowdown is reasonably small, with an average of 1%.

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

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

Table 1. Benchmark results for read barriers (higher is better)

	baseline	read barriers always enabled		read barriers with removal	
SPECjvm98 mean	216.6	150.2	-30.6%	214.4	-1.0%
_227_mtrt	591.9	119.6	-79.8%	583.1	-1.5%
_202_jess	267.2	222.1	-16.9%	260.9	-2.4%
_201_compress	217.7	165.1	-24.1%	219.3	0,7%
_209_db	56.7	53.1	-6.4%	56.7	0.1%
_222_mpegaudio	389.5	323.3	-17.0%	386.8	-0.7%
_228_jack	234.9	208.7	-11.1%	232.1	-1.2%
_213_javac	125.2	110.0	-12.1%	122.8	-1.9%
SPECjbb2005	14,292	9,179	-35.8%	14,152	-1.0%

The maximum slowdown is 2.4% for _202_jess. This benchmark loads a large number of fields with a low frequency, so the corresponding read barriers are not removed because the recompilation overhead would be too high. The slight speedup of some benchmarks is the result of improved optimizations during the recompilation, e.g. a better inlining of methods.

The recompilation of methods increases the total number of method compilations by 23.4% (from 1005 to 1240) for SPECjvm98 and by 48.5% (from 540 to 802) for SPECjbb2005. The additional compilation time has no significant impact on the overall performance, especially for long-running applications.

5.2 Access Counts of Colocated Fields

Object colocation can be performed independently for the young and for the old generation. This allows us to experiment with different scenarios: We measured the impact of object colocation when it is performed only for the young generation, only for the old generation, or for both generations. We also experimented with different strategies for filling the hot-field tables and compared our read-barrier-based approach with a static colocation strategy: Instead of emitting read barriers, the just-in-time compiler adds all fields accessed in compiled code directly to the hot-field tables.

To assess the quality of our object colocation, we counted the number of field accesses where the parent and the child object were colocated and where they were in the same cache line. We use this as an approximation of the memory access costs: When a reference field is loaded, the result of the load is typically used for another field load in the near future. So it is beneficial if the address of a loaded field and the value of the field are in the same cache line. In that case, the subsequent load accesses a memory location that has already been put into the cache during the first load.

Table 2 shows the number of fields and array elements that were loaded for the benchmarks as well as the percentages of the loads that were colocated and that were in the same cache line. The numbers were collected using the detailed read barriers presented in Sect. 3.1. With object colocation the percentages are

Table 2. Field loads of colocated objects and objects in same cache line

	num. loads	baseline		read barriers young gen.		read barriers old gen.		read barriers both gen.		static young gen.	
	(x 1,000)	coloc.	cache	coloc.	cache	coloc.	cache	coloc.	cache	coloc.	cache
SPECjvm98 mean	2,078,900	7%	6%	21%	17%	18%	13%	21%	16%	20%	17%
_227_mtrt	167,000	7%	5%	59%	42%	58%	41%	59%	42%	52%	37%
_202_jess	171,100	19%	13%	39%	29%	19%	13%	39%	29%	41%	31%
_201_compress	774,900	4%	6%	4%	6%	4%	6%	4%	6%	4%	6%
_209_db	356,800	0%	0%	40%	32%	37%	29%	43%	34%	41%	32%
_222_mpegaudio	436,500	9%	5%	9%	5%	9%	5%	9%	5%	9%	5%
_228_jack	61,400	22%	19%	29%	25%	26%	22%	32%	27%	36%	28%
_213_javac	111,200	9%	7%	24%	20%	15%	13%	26%	19%	24%	17%
SPECjbb2005	—	2%	3%	33%	21%	23%	16%	33%	21%	32%	21%

significantly higher for most benchmarks. This shows that object colocation improves both the locality of objects and the cache behavior. The detailed read barriers collect data only for every 1000^{th} field load, so all results are approximate numbers. Because of the large number of loads they are nevertheless significant.

SPECjbb2005 uses a large memory-resident database implemented as trees of objects. Object colocation succeeds to optimize these trees and increases the percentage of colocated objects from 2% to 33%. Performing object colocation in the young generation outperforms object colocation in the old generation because the benchmark accesses a high number of objects located in the young generation. Enabling object colocation in both generations does not improve the numbers further. The number of field loads is not reported because the benchmark does not execute a fixed workload, but runs for a fixed time.

For the benchmark _227_mtrt the percentage of colocated objects increases from 7% to nearly 60%. Table 3 lists the most frequently accessed fields of the benchmark. Four fields form the hot access path and account for 70% of all field loads. These are the fields that were used in the running example of this paper. Object colocation succeeds to colocate a high percentage of them. For the array class `Face[]`, only 18% (about 1/6) of the array accesses load a colocated element because all six elements are accessed with the same frequency and only the first element is colocated.

All three garbage collection configurations basically show the same results. The static colocation strategy leads to the same results as the dynamic strategy for classes with only one reference field. However, it fails to colocate the field `OctNode.OctFaces` because it is not the first one.

The seven fields listed in Table 3 for the benchmark _209_db account for 99.9% of all field loads. In the hot path a `Vector` of `String`s that is stored in an `Entry` of a `Database` is accessed. Colocation is possible for three of the seven fields. The other four fields are typical examples where object colocation is not possible: large arrays (`Entry[]`), frequently changing fields (`Database.index`), fields or arrays of the type `Object` or `Object[]`, and fields of short-living temporary objects such as iterators (`Vector$1.this$0`).

Table 3. Frequently accessed fields of _227_mtrt and _209_db

	num. loads	baseline		read barriers young gen.		read barriers old gen.		read barriers both gen.		static young gen.	
	(x 1,000)	coloc.	cache	coloc.	cache	coloc.	cache	coloc.	cache	coloc.	cache
_227_mtrt	167,000	7%	5%	59%	42%	58%	41%	59%	42%	52%	37%
Face[]	33,200	0%	0%	18%	9%	17%	9%	18%	6%	18%	9%
Face.Verts	32,900	2%	2%	100%	88%	100%	87%	100%	88%	100%	89%
Point[]	32,600	0%	0%	99%	61%	98%	61%	99%	61%	99%	62%
OctNode.OctFaces	15,700	0%	0%	93%	58%	91%	56%	92%	60%	0%	0%
_209_db	356,800	0%	0%	40%	32%	37%	29%	43%	34%	41%	32%
Entry[]	66,400	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Database.index	61,100	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Vector.elementData	54,900	0%	0%	87%	74%	89%	68%	100%	80%	88%	74%
Object[]	54,500	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Entry.items	51,100	0%	0%	100%	79%	88%	74%	100%	84%	100%	79%
String.value	45,500	0%	0%	100%	75%	87%	65%	100%	75%	100%	75%
Vector$1.this$0	23,200	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%

For _202_jess and _213_javac, performing object colocation in the young generation outperforms object colocation of the old generation. The collection of the old generation comes too late because the benchmarks primarily access objects in the young generation. Enabling object colocation in both generations leads to the same results as performing colocation only in the young generation.

Object colocation cannot optimize applications that require no garbage collection. Both _201_compress and _222_mpegaudio operate on a small, fixed set of objects that are allocated at the beginning of the execution, so the percentages are low for all configurations.

Each benchmark is executed once to collect the counters, so this run also includes the construction of the hot-field tables. For the benchmark _228_jack the static colocation strategy has an advantage because the tables are filled when methods are compiled and not when counters overflow. This is early enough to optimize a larger data structure that is created at startup. However, both strategies show the same results starting with the second execution of the benchmark.

5.3 Run-Time Impact of Object Colocation

Table 4 shows the run-time results of the various object colocation scenarios. Some benchmarks are very sensitive to garbage collection time. Because object colocation requires additional operations for each object copied during garbage collection, the improved cache behavior is countervailed by the garbage collection overhead. The old generation contains much more objects than the young generation, so the overhead is higher when performing object colocation for the old generation. However, there is still potential for optimizing the garbage collection algorithms so that the slowdown for these benchmarks can probably be eliminated in the future.

Table 4. Benchmark results for object colocation (higher is better)

	baseline	read barriers young gen.		read barriers old gen.		read barriers both gen.		static young gen.	
SPECjvm98 mean	216.6	230.0	+6.2%	224.3	+3.6%	227.8	+5.2%	229.5	+6.0%
_227_mtrt	591.9	620.0	+4.8%	582.9	-1.5%	613.3	+3.6%	613.3	+3.6%
_202_jess	267.2	264.4	-1.1%	259.5	-2.9%	263.7	-1.3%	267.3	+0.0%
_201_compress	217.7	217.4	-0.1%	214.6	-1.4%	214.5	-1.5%	217.4	-0.1%
_209_db	56.7	86.6	+52.8%	83.5	+47.2%	86.8	+53.1%	86.7	+53.0%
_222_mpegaudio	389.5	386.8	-0.7%	387.9	-0.4%	388.5	-0.3%	388.9	-0.1%
_228_jack	234.9	230.7	-1.8%	230.7	-1.8%	227.6	-3.1%	229.3	-2.4%
_213_javac	125.2	123.5	-1.3%	117.8	-6.0%	119.6	-4.5%	121.9	-2.7%
SPECjbb2005	14,292	14,599	+2.1%	14,260	-0.2%	14,512	+1.5%	14,394	+0.7%

The benchmark _209_db benefits most from object colocation. The speedup of more than 50% shows that the cache behavior has a major influence on the total performance of the application. _227_mtrt also shows a significant speedup in most scenarios. As shown in Table 3, both benchmarks have a hot path of frequently accessed fields that can be optimized.

SPECjbb2005 shows a speedup of 2.1%, which proves that object colocation also succeeds to optimize a large heap of a long-running application. The overhead of object colocation is influenced by the number of children that are colocated to a parent. Because the static colocation strategy identifies much more children than the dynamic read-barrier-based approach, the dynamic approach has a lower overhead and outperforms the static one.

For SPECjbb2005, the percentages of colocated objects (see Table 2) are similar for the dynamic and the static approach, but the speedup (see Table 4) is 2.1% for the dynamic approach and 0.7% for the static approach. Using the static approach, there are 113 hot-field tables, with a maximum of 23 hot children for the table of a class with 25 reference fields. In the dynamic approach, there are only 38 tables with a maximum of 4 hot children per table. This shows that the static colocation strategy does not scale well for larger applications, so the read-barrier-based approach is inevitable.

5.4 Summary

All in all, the results show that the read-barrier-based object colocation of the young generation leads to the best results. Optimizing only the old generation finds less colocatable objects because the old generation is collected infrequently. Optimizing both generations increases the overhead, but does not improve the heap layout significantly because colocated objects are promoted and the old generation preserves this optimized object order.

The static colocation strategy does not yield information about the most frequently accessed fields. Therefore, too many parent-child relationships are added to the tables, which increases the garbage collection overhead. The impact is evident especially for larger applications such as SPECjbb2005.

6 Future Work

The evaluation showed that our algorithm improves the object order of the heap, but the speedup of some benchmarks is lower than one would expect. Therefore, we plan to extend our object colocation algorithm to do *object inlining*, which will eliminate the field loads for colocated objects and lead to an additional speedup. The counters collected by the detailed read barriers show that for many fields the referenced objects can always be colocated. In such cases, the address of a child object needs not be read from a field, but can be computed by adding a fixed offset to the address of the parent object.

Eliminating field loads can be implemented easily in the just-in-time compiler. However, a safe execution of the optimized code requires additional effort: While failing to colocate a small number of objects of a class is acceptable for object colocation because it does not have a negative impact on the cache behavior, object inlining requires that *all* objects of a class are colocated.

To guarantee this, object allocation must be modified so that colocated objects are already allocated together. Currently, objects are only colocated after the first garbage collection run. Object inlining also requires that field stores which change a parent-child relationship do not happen. Because fields can also be changed via reflection or by native code using the Java Native Interface, these subsystems must be instrumented to detect such cases. Recent research on optimizations in the Java HotSpot^{TM} VM showed that the safe execution of aggressively optimized code requires extensive support of the run-time system [9].

7 Related Work

Huang et al. describe a system similar to ours called *online object reordering*, implemented for the Jikes RVM [5]. They use the adaptive compilation system of Jikes that periodically records the currently executed methods. Hot fields accessed in these methods are traversed first in their copying garbage collector and thus reordered. The decision which field of a method is hot is based on a static analysis of the method, so it 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* [3] and present results for the object-oriented programming language Cecil. They use a profiling technique similar to read barriers to construct an object affinity graph that guides a copying garbage collector and report an overhead of about 6% for the profiling. They do not distinguish different fields within the same object, which suffices only for small objects and does not allow colocating the most frequently accessed field of bigger objects.

Lhoták et al. compare different algorithms for *object inlining* and report how many field accesses they optimize [10]. All described algorithms are implemented in static compilers and do not handle dynamic class loading. However, the dynamic class loading of the Java HotSpot^{TM} VM asks for algorithms that do not require a global data flow analysis.

The algorithm for *object combining* by Veldema et al. puts objects together that have the same lifetime [15]. It is more aggressive than object inlining because it also optimizes unrelated objects if they have the same lifetime. This allows the garbage collector to free multiple objects together. Elimination of pointer accesses is performed separately by the compiler. However, the focus 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.

Escape analysis is another optimization that reduces the overhead of memory accesses. It detects objects that can be eliminated or allocated on the method stack. It is an orthogonal optimization to object colocation because it optimizes short-living temporary objects, whereas object colocation optimizes long-living data structures. Kotzmann implemented a new escape analysis algorithm for the Java HotSpot^{TM} VM [8]. It is fast enough for a just-in-time compiler and handles all aspects of dynamic class loading. When a class is loaded that lets a previously optimized object escape its scope, all affected methods are deoptimized and recompiled using the same mechanism we use for removing read barriers.

Blackburn et al. measured the dynamic impact of various read and write barriers on different platforms [2]. They focused on barriers that are necessary for current garbage collection algorithms, so a barrier similar to ours that counts field accesses is not measured. A complex conditional read barrier shows an average slowdown of 16% on a Pentium 4 processor, with a maximum slowdown of over 30%.

Arnold et al. presented a general framework for instrumentation sampling to reduce the cost of instrumented code [1]. The framework dynamically switches between the original uninstrumented code and the instrumented code in a fine-grained manner. Instrumentation can be performed continuously with a reported overhead of about 6%. This approach is more sophisticated than our detailed read barriers that always collects data for every 1000th field load, but doubles the code size.

8 Conclusions

We presented an object colocation algorithm implemented for the garbage collector of the Java HotSpot^{TM} VM. The most frequently loaded fields and thus the most promising objects to be colocated are identified using read barriers that are inserted into the machine code by the just-in-time compiler. The read barriers yield precise information about the field access profile with a low run-time overhead of just 1%.

In a generational garbage collection system, object colocation can be performed independently for each generation. Our measurements show that it is sufficient to optimize the young generation. When colocated objects are promoted, they remain colocated in the old generation. A comparison with a static colocation strategy shows that the overhead of optimizing infrequently accessed objects is higher than the benefit.

Acknowledgments

We would like to thank the Java HotSpot™ compiler team at Sun Microsystems, especially Kenneth Russell, Thomas Rodriguez and David Cox, for their persistent support, for contributing many ideas and for helpful comments on all parts of the Java HotSpot™ Virtual Machine.

References

1. Arnold, M., Ryder, B.G.: A framework for reducing the cost of instrumented code. In: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation, ACM Press (2001) 168–179
2. Blackburn, S.M., Hosking, A.L.: Barriers: friend or foe? In: Proceedings of the 4th international symposium on Memory management, ACM Press (2004) 143–151
3. Chilimbi, T.M., Larus, J.R.: Using generational garbage collection to implement cache-conscious data placement. In: Proceedings of the 1st international symposium on Memory management, ACM Press (1998) 37–48
4. Griesemer, R., Mitrovic, S.: A compiler for the Java HotSpot™ virtual machine. In Böszörményi, L., Gutknecht, J., Pomberger, G., eds.: The School of Niklaus Wirth: The Art of Simplicity. dpunkt.verlag (2000) 133–152
5. Huang, X., Blackburn, S.M., McKinley, K.S., Moss, J.E.B., Wang, Z., Cheng, P.: The garbage collection advantage: improving program locality. In: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM Press (2004) 69–80
6. Intel Corporation: IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture. (2006) Order Number 253665-018.
7. Jones, R., Lins, R.: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley & Sons (1996)
8. Kotzmann, T., Mössenböck, H.: Escape analysis in the context of dynamic compilation and deoptimization. In: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, ACM Press (2005) 111–120
9. Kotzmann, T., Mössenböck, H.: Reallocation and garbage collection support for scalar-replaced and stack-allocated objects. Technical report, Institute for System Software, Johannes Kepler University Linz (2006)
10. Lhoták, O., Hendren, L.: Run-time evaluation of opportunities for object inlining in Java. In: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande, ACM Press (2002) 175–184
11. Pozo, R., Miller, B.: SciMark 2.0. (1999) http://math.nist.gov/scimark2/.
12. Standard Performance Evaluation Corporation: The SPEC JVM98 Benchmarks. (1998) http://www.spec.org/jvm98/.
13. Standard Performance Evaluation Corporation: The SPEC JBB2005 Benchmark. (2005) http://www.spec.org/jbb2005/.
14. Sun Microsystems, Inc.: Java SE 6: Mustang Snapshot Releases. (2006) https://mustang.dev.java.net/.
15. Veldema, R., Ceriel, J.H., Rutger, F.H., Henri, E.: Object combining: A new aggressive optimization for object intensive programs. In: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande, ACM Press (2002) 165–174
16. Wimmer, C., Mössenböck, H.: Optimized interval splitting in a linear scan register allocator. In: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, ACM Press (2005) 132–141