Run-Time Support for Optimizations Based on Escape Analysis*

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

Abstract

The Java™ programming language does not allow the programmer to influence memory management. An object is usually allocated on the heap and deallocated by the garbage collector when it is not referenced any longer. Under certain conditions, the virtual machine can allocate objects on the stack or eliminate their allocation via scalar replacement. However, even if the dynamic compiler guarantees that the conditions are fulfilled, the optimizations require support by the run-time environment.

We implemented a new escape analysis algorithm for Sun Microsystems' Java HotSpot™ VM. The results are used to replace objects with scalar variables, to allocate objects on the stack, and to remove synchronization. This paper deals with the representation of optimized objects in the debugging information and with reallocation and garbage collection support for a safe execution of optimized methods. Assignments to fields of parameters that can refer to both stack and heap objects are associated with an extended write barrier which skips card marking for stack objects. The traversal of objects during garbage collection uses a wrapper that abstracts from stack objects and presents their pointer fields as root pointers to the garbage collector.

When a previously compiled and currently executing method must be continued in the interpreter because dynamic class loading invalidates the machine code, execution is suspended and compiler optimizations are undone. Scalar-replaced objects are reallocated on the heap lazily when control returns to the invalidated method, whereas stack-allocated objects must be reallocated immediately before program execution resumes. After reallocation, objects for which synchronization was removed are relocked.

*This work was supported by Sun Microsystems, Inc.

1. Introduction

The growing popularity of Java is in large part based on its portability, which is achieved by translating source code into platform-independent bytecodes. In order for a Java program to run on a platform, a Java virtual machine must exist that executes the bytecodes for this platform. Early implementations relied only on interpretation. They had to fetch, decode and execute bytecode by bytecode and thus suffered from poor performance. In the meantime, a lot of work and research effort has been spent on more efficient implementations. Modern virtual machines dynamically compile bytecodes to machine code and therefore run at high speed.

It does not make much sense to compile all methods when an application is started. First of all, typical applications spend the majority of their time executing only a small part of their code. Secondly, Java allows dynamic loading of classes on demand that may not yet be available at startup. Thirdly, time spent on compiling a method is time that could have been spent on interpreting the bytecodes. Therefore, mixed-mode run-time environments combine interpretation with dynamic compilation [1]. They first interpret all methods, which results in fast startup times, and then compile only the "hot" parts of a program, e.g. the most frequently called methods or methods that contain long-running loops.

Mixed-mode run-time environments complicate the implementation of compiler optimizations. Since compilation time adds to run time, the dynamic compiler must achieve a trade-off between the duration of optimizations and execution speed of the resulting machine code. Besides, compiled methods may call or may be called by interpreted methods, which are not aware of compiler optimizations. More aggressive optimizations usually require run-time support, i.e. modifications of the run-time environment. If the optimizations affect the allocation of objects, the garbage collector must be adapted as well.

The Java HotSpot™ VM uses a generational garbage collector. New objects are allocated in the young generation, which is collected by a stop-and-copy algorithm. When objects have survived a certain number of collection cycles, they are promoted to the old generation. In the default configuration, the old generation is collected by a mark-and-compact algorithm. Starting from the set of root pointers, the garbage collector marks all reachable objects, assigns consecutive memory slots to them, adjusts pointers, and finally moves the objects to their new locations.

We developed a new escape analysis algorithm and various optimizations to reduce the costs of object allocation, deallocation and synchronization [15]. We integrated our implementation into a current snapshot [21] of the Java HotSpot™ VM, which facilitates aggressive compiler optimizations because it is able to revert to interpretation when conditions that hold at compile time are broken later. This mechanism is referred to as *deoptimization* [10].

To the best of our knowledge, our approach is the first that reallocates and relocks objects when dynamic class loading invalidates optimized machine code. Most existing implementations are either based on a closed-world assumption, make pessimistic decisions, or require checks everytime optimized code is executed. This paper deals with our run-time support that allows a safe execution of optimized methods. It contributes the following:

- It examines which parts of the virtual machine are affected by our compiler optimizations.
- It suggests a representation for optimized objects in the debugging information.
- It explains how stack objects are treated by the garbage collector.
- It describes how objects are reallocated and relocked in the context of deoptimization.
- It evaluates the modifications in terms of memory consumption and the number of optimized objects.

The rest of the paper is organized as follows: Section 2 gives a short overview of the escape analysis algorithm in order to put the run-time support into context. Section 3 describes the representation of optimized objects in the debugging information. The modifications of the garbage collector are presented in Section 4, and the adaptation of the deoptimization framework in Section 5. Section 6 evaluates the modifications, and Section 7 deals with related work.

2. The Escape Analysis Algorithm

If an object created in a method is assigned to a global variable or to the field of a heap object, if it is passed as a parameter to a method or returned from a method, its lifetime exceeds the scope in which it was created. Such an object is said to *escape* its scope. Knowing which objects do not escape a method or thread allows the compiler to perform aggressive optimizations:

- If an object does not escape the creating method, its fields can be replaced by scalar variables. The compiler can eliminate both the allocation of memory on the heap and its initialization.
- Objects that are accessible only by the creating method and its callees can be allocated on the method stack. This is cheaper than an allocation on the heap and reduces the burden of the garbage collector.
- Java provides bytecodes that lock and unlock objects for the synchronization of threads. If an object does not escape a thread, synchronization on it can be removed because it will not be locked by another thread.

Our escape analysis algorithm is especially tailored to the needs of a dynamic compiler. In contrast to other approaches, it does not create an object dependency graph but computes the escape states of objects in parallel to the construction of the intermediate representation, which is faster and requires less memory. Objects whose escape states depend on each other are inserted into a so-called equi-escape set. If one of them escapes, only the escape state of the set and not of all objects must be adjusted. The tracking of field values for scalar replacement is integrated into the computation of the static single assignment form.

When objects are passed to another method, the compiler uses interprocedural escape information to determine if the objects escape in the callee. This information is computed during the compilation of a method and stored together with the machine code to avoid reanalyzing the method at each call site. The interprocedural analysis not only allows allocating actual parameters in the stack frame of the caller, but also supports the compiler in inlining decisions: even if a callee is too large to be inlined by default, the compiler may decide to inline it because it thus expects to replace more objects by scalar variables.

In a mixed-mode run-time environment, a method is interpreted several times before its compilation. For this period of time, no interprocedural escape information is available for this method. When the compiler reaches a call of a method that has not been compiled yet, it examines the method's bytecodes to determine if a parameter escapes. The bytecode analysis considers each basic block separately and checks if it lets one of the parameters escape. This is more conservative than a full escape analysis, but faster to compute because control flow is ignored. The analysis stops as soon as all parameters are seen to escape. When the method is compiled later, the provisional escape information is replaced with a more precise one.

```
float foo(Point p1) {
  Point p2 = new Point(3, 4);
  Line l = new Line(p1, p2);
  return l.length();
}
```

(a) original method

```
float foo(Point p1) {
  Point p2 = new Point();
  p2.x = 3; p2.y = 4;
  Line l = new Line();
  l.x1 = p1.x; l.y1 = p1.y;
  l.x2 = p2.x; l.y2 = p2.y;
  return l.length();
}
```

(b) after inlining

```
float foo(Point p1) {
  float x2 = 3, y2 = 4;
  Line l = new Line(); // on stack
  l.x1 = p1.x; l.y1 = p1.y;
  l.x2 = x2; l.y2 = y2;
  return l.length();
}
```

(c) after escape analysis

Figure 1. Example for scalar-replaced and stack-allocated objects.

Figure 1 gives an example for the escape analysis and the optimizations described above. A point is specified by two floating point coordinates, and a line stores the four coordinates of the start and the end point. After inlining the constructors, the object `p2` does not escape the method, so its allocation can be eliminated. Its field values are stored in scalar variables and can probably be held in registers later. The `length` method is too large to be inlined but does not let the object `l` escape globally. Therefore, the line object can be allocated on the stack. The interprocedural escape information for the method `foo` states that the parameter `p1` does not escape in the method and can be allocated in the stack frame of a caller.

3. Debugging Information

In order to perform aggressive optimizations, the Java HotSpot™ client compiler makes optimistic assumptions about the structure of the running program. For example, it inlines a virtual method if class hierarchy analysis [7] yields that currently only one implementation for that method is loaded. If dynamic class loading later invalidates such assumptions, previously compiled methods must be *deoptimized*, i.e. compiler optimizations must be undone. This requires the compiler to describe the performed optimizations in the debugging information. Besides, the compiler creates so-called *oop maps* (see below), which are used by the garbage collector to determine the locations of the root pointers.

Debugging information and oop maps are not generated for every single machine instruction, but only for some *safepoints*, at which method execution may be suspended for garbage collection or deoptimization. After compilation, the generated information is serialized and stored with the machine code in a compressed form.

3.1. Oop Map

Run-time methods of the Java HotSpot™ VM refer to objects in the Java heap via handles. A handle is a data structure that encapsulates a pointer to a Java object and resides in a special memory area where the garbage collector visits the object and updates the pointer if the object is moved. Machine code generated for a Java method, however, refers to objects via direct pointers for efficiency reasons. To distinguish direct pointers from handles, they are called *ordinary object pointers* (oop).

In order to trace through the live objects, the garbage collector needs to know where the machine code stores pointers to heap objects. Therefore, the compiled method includes an *oop map* for each safepoint. The oop map lists all registers and stack slots that contain root pointers. In the context of escape analysis, pointers stored in an object whose allocation was eliminated via scalar replacement become root pointers and need to be registered in the oop map accordingly.

Stack objects must be considered in the oop map as well. Assume that both objects created by the code in Figure 2 are allocated on the stack, but at different positions within the frame. Since the variable `p` always refers to object 1, the positions of its fields are known at compile time. The fields can be accessed relative to the frame pointer without loading the object's address. At safepoint 1, no pointer to the object exists, so pointers stored in fields are root pointers.

```
p = new T();    // object 1
// safepoint 1
if (...) {
  q = p;
} else {
  q = new T();  // object 2
}
// safepoint 2
```

Figure 2. Stack objects at safepoints.

While `p` always refers to object 1 at safepoint 2, `q` may refer to different objects and must therefore be kept as a pointer. If we would register both `q` and the fields of object 1 as roots, the garbage collector would visit the pointer fields of object 1 twice when `q` points to object 1. This is not allowed: the mark-and-compact algorithm updates all pointers before the objects are moved to their new locations, so after the first visit, a pointer field does not point to an object any more.

```
void foo(int n) {
  T p = new T();
  p.f = 5;
  p.g = n + 3;

  bar();


  ...
}
```

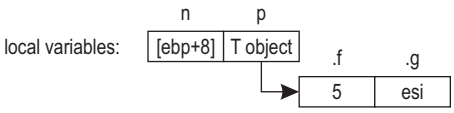
(a) Java source code

```
// parameter n at ebp+8
// allocation eliminated

mov  esi, [ebp+8]
add  esi, 3
// safepoint for method call

...
```

(b) machine code



(c) scope and object entries

Figure 3. Example for an object entry.

Therefore, we create a special entry in the oop map to describe the stack object 1 as a whole instead of its individual fields. When the garbage collector reads the oop map and encounters such an entry, it marks the stack object as scanned before visiting its fields. Even if q provides a root pointer to the same object, the fields will not be processed a second time.

3.2. Object Entries

Dynamic class loading may invalidate machine code generated under optimistic assumptions, even while the affected method is running. In this case, the execution of the method is continued in the interpreter. While the machine code keeps temporary values in registers and spill slots, the interpreter expects all values to be stored in the stack frame. In the case of deoptimization, a new frame for the interpreter must be set up and filled with the appropriate values from registers and spill slots.

For this purpose, the deoptimization uses *scope entries* that are created by the compiler and specify where the machine code stores which values. The debugging information contains two separate lists of scope entries: one for the operand stack and one for the local variables. Each scope entry describes one value, and its position within the list denotes which stack slot or local variable it refers to.

If a variable refers to an object whose allocation was eliminated, the list of scope entries contains an *object entry* that describes the contents of the object (see Figure 3). The object entry allows the deoptimization framework to later reallocate the object on the heap. It holds a list of scope entries that describe where the machine code stores the field values of the object. The position of a scope entry within this list corresponds to the position of the described field within the object.

A stack-allocated object is also represented by an object entry. In contrast to an eliminated object, however, the object entry only specifies the position of the stack object within the frame. A list of scope entries for fields is not needed because the contents of the object can be copied from the stack into the heap object during reallocation (see Section 5.3).

When the VM deoptimizes a method that inlines other methods, it has to split the frame of the compiled method into several frames of interpreted methods, each with its own operand stack and local variables. Objects, however, are shared among all inlined methods and must be reallocated exactly once during deoptimization. Therefore, the debugging information specifies separate scope entries for each inlined method, but only one set of object entries.

3.3. Monitor Entries

When a compiled method locks an object, it stores information about the object in its stack frame. To convert the information for the interpreter, the deoptimization framework must know which objects are locked. For this reason, the debugging information provides a list of monitor entries. Each monitor entry encapsulates a scope entry that describes the location of a pointer to the locked object.

If synchronization on a thread-local object was eliminated, it must be relocked during deoptimization because the interpreter will later try to unlock it again. Therefore, the compiler has to ensure that there is a pointer to every thread-local object, otherwise the object could not be described in the debugging information.

The corresponding monitor entry specifies the location of the pointer to the thread-local object. In addition, the monitor entry indicates that synchronization was removed and that the object must be relocked during deoptimization. In case of a method-local object whose allocation has been eliminated, the monitor entry encapsulates an object entry instead of a normal scope entry.

3.4. Structure of Debugging Information

After compilation, the machine code and the information for garbage collection and deoptimization must be registered in the virtual machine. For this purpose, the compiler creates a data structure consisting of the machine code, oop

maps, debugging information, method dependencies (see Section 5.1), and exception handler tables. The debugging information provides scope entries and monitor entries for each safepoint. Its structure is shown in Figure 4, and the internal layout in Figure 5.

```
DebuggingInfo  = {SafepointDesc}.
SafepointDesc  = offset ObjectPool {Scope}.

ObjectPool     = {ObjectEntry}.
ObjectEntry    = id type (Fields | stackpos).
Fields         = {ScopeEntry}.

Scope          = methodref bci LocalVariables OperandStack Monitors.
LocalVariables = {ScopeEntry}.
OperandStack   = {ScopeEntry}.
Monitors       = {MonitorEntry}.

ScopeEntry     = ConstantEntry | LocationEntry | ObjectEntryRef.
ConstantEntry  = number | oop.
LocationEntry  = register | spillslot.
ObjectEntryRef = id.

MonitorEntry   = ScopeEntry stackpos eliminated.
```

Figure 4. Structure of debugging information.

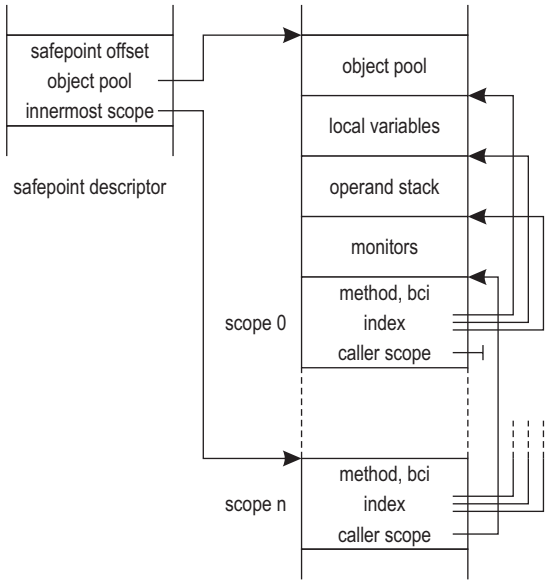


Figure 5. Layout of debugging information.

A safepoint descriptor specifies the offset of the safepoint in the machine code, the object pool, and a set of scope descriptors. Upon deoptimization, the VM looks up the safepoint descriptor whose offset matches the current program counter.

The *object pool* contains object entries for all scalar-replaced and stack-allocated objects. Even if there are multiple scopes due to inlining, the object pool exists only once per safepoint. In addition to a unique identification number and the object's type, an object entry either specifies a list of scope entries for the fields of a scalar-replaced object or the position of a stack-allocated object in the stack frame.

Each inlined method is represented by a separate scope description. It stores a reference to the method and the bytecode index (BCI) of the current instruction in this method. Lists of scope entries are used to describe the values in local variables and on the operand stack. A scope entry either describes which register or spill slot contains the value, or directly stores the value if it is known at compile time. When a value refers to a scalar-replaced or stack-allocated object, the identification number of the appropriate object entry in the object pool is emitted.

For each object that is locked in the current scope, a monitor entry is created. It encapsulates a scope entry for the locked object and specifies the stack slot that is used by the compiled method to store information about the lock. A flag indicates if synchronization on the object was removed.

4. Garbage Collection

Scalar-replaced objects do not affect garbage collection. Their pointer fields are registered in the oop map and therefore treated as root pointers. Stack objects, however, can contain pointers to heap objects that must be processed. We implemented an extended write barrier and a wrapper for pointer traversal, which hide stack objects from card marking and the garbage collector. This facilitated an easy integration of our new optimizations without corrupting the carefully tuned and tested algorithms in place.

4.1. Write Barriers

A card marking mechanism is used to keep track of pointers from the old into the young generation. Each assignment to a field that references an object is associated with a write barrier that marks the modified card as dirty [11].

If a formal parameter does not escape a method or its callees, the method may be called both with a stack-allocated and a heap-allocated actual parameter. Since card marking for a stack object would modify a byte outside the card marking array, non-escaping parameters are associated with a modified write barrier that performs a bounds check before the array is accessed (see Figure 6).

The card index is calculated via a right-shift of the object address in the `EAX` register. Then the index of the first card (*firstIndex*) is subtracted. The unsigned check of the result against the array size (*arraySize*) detects both a negative index, which looks like a large unsigned positive number, and an index greater than the size. If the `EAX` register refers to a heap object, the corresponding card is marked

```
        shr  eax, 9
        sub  eax, firstIndex
        cmp  eax, arraySize
        jae  label
        mov  byte ptr [eax+arrayBase], 0
label:  ...
```

Figure 6. Write barrier with bounds check.

as dirty. If the register refers to a stack object, the bounds check fails and card marking is omitted.

Although the heap may grow on demand, the size of the card marking array can directly be emitted into the machine code. At startup, the Java HotSpot™ VM reserves virtual address space for the maximum size of the heap, although not all pages of it need to be allocated. The same is done with the card marking array, so the write barrier can use the maximum array size in its bounds check.

On Intel platforms [12], the extended write barriers take five instead of the usual two machine instructions, but they are only required for non-escaping formal parameters. As long as fields of a stack object are accessed within the allocating method, no write barriers are emitted at all. The object is registered in the oop map, so its fields are automatically processed at every collection cycle. For escaping objects, traditional write barriers are emitted. The interpreter must execute the bounds check for all objects, because a stack object may also be passed to an interpreted method.

4.2. Wrapper Closure

The garbage collector implements an operation to be performed for all objects as a so-called *oop closure* following the Visitor design pattern. Starting with the root pointers, an oop closure is recursively applied to all reachable oops. Oop closures are used for the stop-and-copy collection, as well as for the marking and pointer adjustment phase of the mark-and-compact algorithm.

When stack-allocated objects are live, the garbage collector must process their fields and visit referenced heap objects. Since existing algorithms do not expect to see pointers into the stack, the iteration of root pointers was modified to use a wrapper closure. It abstracts from stack objects and presents their pointer fields as root pointers to the underlying closure (see Figure 7).

Since the heap is a contiguous area in memory, the `is_in_heap` test can be implemented as a comparison of the object address against the start and the end of the heap. If the test succeeds, the wrapped closure is invoked on the heap pointer. Heap objects never reference stack objects, so the wrapped closure will not see a stack object from this point on. If the test fails, the wrapper closure is applied to the oop fields of the stack object, which may again point to stack objects.

```
void do_oop(oop obj) {
  if (is_in_heap(obj)) {
    wrapped_closure.do_oop(obj);
  } else if (!obj.has_been_scanned()) {
    obj.set_has_been_scanned();
    obj.iterate_oop_fields(this);
  }
}
```

Figure 7. Wrapper for oop closures.

A stack object can directly or indirectly reference itself, so the garbage collector must remember which stack objects have already been scanned. We could set the mark bits to identify scanned objects, but this would require an extra pass to reset them. Instead, we use two bits in the header word of stack objects to encode three values. The bits are initialized with 0, which means that the object has not been scanned at all. At the beginning of every iteration over the root pointers, we toggle a global value between 1 and 2. The `has_been_scanned` test compares the header field with the global value for equality, and `set_has_been_scanned` copies the global value into the header of the stack object.

5. Deoptimization

Both inlining and the use of interprocedural escape information require that the compiler identifies the called method statically despite polymorphism and dynamic method binding. Apart from static and final callees, this is possible if class hierarchy analysis determines that currently only one suitable method exists. When a class is loaded later that provides another suitable method, the previously compiled method must be deoptimized. This means that all optimizations of the compiler are undone and execution of the method is continued in the interpreter. Upon its next invocation, the method will be recompiled without inlining the callee.

5.1. Method Dependencies

Assume that class `B` was not loaded yet when machine code for the method `calc` in Figure 8 is generated. The compiler optimistically assumes that there is only one implementation of the method `foo` and inlines `A.foo` into `calc`. When class `B` is loaded later which overrides `foo`, the VM must deoptimize the method `calc`. In other words, a dependency is introduced between `calc` and `A.foo`. This dependency is recorded during the compilation of `calc` and stored both in the compiled method `calc` and the class `A`.

Even if `A.foo` is not inlined, for example because of its size, it can be statically bound to save the dispatching

```
class A {
  void foo(Point p) { ... }
}

class B extends A {
  void foo(Point p) { ... }
}

static float calc(float x, float y) {
  Point p = new Point(x, y);
  A q = create();
  q.foo(p);
  return p.x * p.y;
}
```

Figure 8. Example for method dependencies.

overhead. Then the compiler uses interprocedural escape information to find out that the parameter `p` does not escape and can be allocated in the stack frame of `calc`. When class `B` is loaded, `calc` must be deoptimized not only because `A.foo` was statically bound, but also because the `Point` object may escape in `B.foo`.

However, it is not sufficient to record a dependency between `calc` and `foo`. Assume that the parameter `p` does not escape `A.foo` just because the compiler inlines the virtual method `bar` in `foo`. When another class is loaded that overrides `bar`, the method `foo` is deoptimized due to its dependency on `bar`. The machine code for `calc` must be invalidated as well, because `p` may escape in the newly loaded `bar` method.

If `calc` depended only on `foo`, it would not be deoptimized because method dependencies are not processed transitively. In other words, the compiler must record a dependency between two methods m and m' if

- m inlines m' or calls it with static binding, or if
- a direct or indirect callee of m inlines m' or calls it with static binding, and m passes at least one stack-allocated object to this callee.

Every compiled method stores a list of its dependencies. When the compiler parses the method call of `foo` in `calc` and uses interprocedural escape information of `foo` to allocate some parameters on the stack, it not only records a dependency between `calc` and `foo`, but also inherits all dependencies between `foo` and its callees.

5.2. Reallocation of Eliminated Objects

Deoptimization may be necessary when a new class is added to the class hierarchy. The VM examines the superclasses of the new class and marks dependent methods for deoptimization. It also iterates over the interfaces implemented by the new class and looks for methods that relied on the fact that an interface had only one implementor.

Then the VM traverses the stacks of all threads. A frame that belongs to a marked method is not immediately deoptimized. Instead, the machine instruction at the program counter for this frame is patched to invoke a run-time stub. The actual deoptimization takes place when the frame is reactivated after all callees have returned. This is called *lazy deoptimization* [10]. The machine code is marked as non-entrant, so that the VM interprets new invocations instead of executing the patched machine code.

Before execution of the method can continue in the interpreter, which does not know about scalar replacement, objects whose allocations were eliminated by the compiler must be reallocated on the heap. The reallocation code iterates over all object entries in the debugging information and creates appropriate heap objects. Pointers to the heap objects are encapsulated in handles and stored in the object entries. If garbage collection is needed in the middle of reallocation, the newly allocated objects are visited because pointers in handles are treated as root pointers.

The fields of a reallocated object are initialized using the list of scope entries in the corresponding object entry. If a field refers to another eliminated object, the list contains an object entry that encapsulates the pointer to be stored in the field. When the interpreter frame is set up, the pointers in the object entries are copied into the appropriate stack slots for local variables and the operand stack. Deserialization of debugging information takes care that variables referring to the same eliminated object are mapped to identical object entries.

5.3. Reallocation of Stack Objects

If deoptimization occurs, it is guaranteed that no other code in the system tries to access a scalar-replaced object before control returns to the corresponding frame, because escape analysis proved its locality. Therefore, it is acceptable to reallocate method-local objects lazily. Stack-allocated objects, however, may be passed to the newly loaded code which might allow them to escape the thread. They must be reallocated immediately before the execution of the program resumes, because otherwise a reference to a stack object may be stored in a global variable or a heap object.

The stack objects are either referenced by root pointers and other stack objects or represented in the debugging information if no pointer to them exists. Figure 9 shows how stack objects are reallocated on the heap. Note that arrows to a stack object point to the bottom of the rectangle because addresses increase towards the top of the stack. The reallocation is split into three phases:

1. At first, empty objects are allocated on the heap. Starting with root pointers and object entries in the debugging information, deoptimization recursively visits all

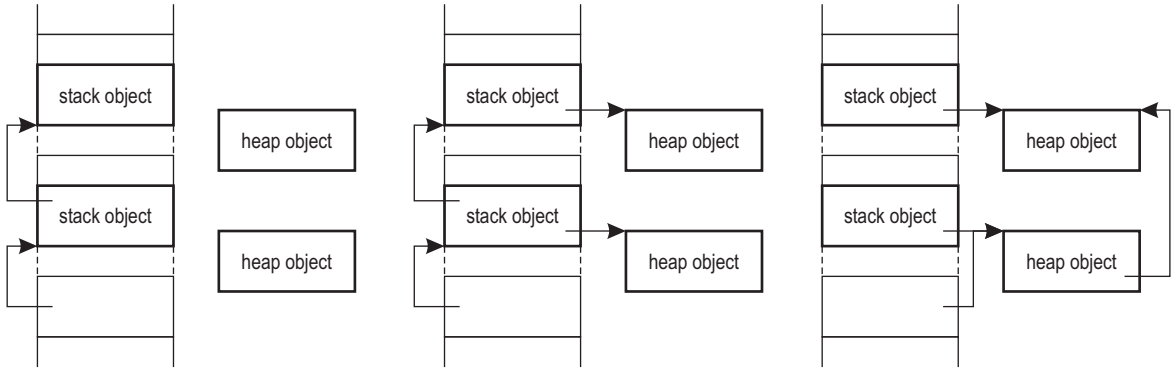


Figure 9. Reallocation of stack objects.

stack objects. This ensures that the complete object graph is reallocated. A data structure maps each stack object to a handle that refers to the counterpart on the heap and keeps the heap object alive if garbage collection is required.

2. Next, all handles are traversed to install forwarding pointers from stack objects to heap objects. The forwarding pointers are used later for the setup of the interpreter frames. The garbage collector must not run during the second and the third phase, because a forwarding pointer would be visited repeatedly if an object is referenced by more than one root pointer or other stack object. This is not allowed for the same reasons as for fields.

3. Finally, the handles are traversed again to copy the contents of stack objects into the heap objects and to replace pointers to stack objects with the forwarding pointers. This phase also adjusts the pointers in stack frames of compiled and interpreted methods.

The first phase is a recursive traversal of all stack objects, but the second and third are linear phases. The forwarding pointers remain in the stack objects until control returns to the frame that hosts them. During lazy deoptimization, the heap pointers are copied into the appropriate slots of the interpreter frame.

5.4. Relocking of Thread-Local Objects

After reallocation, the newly allocated heap objects must be relocked if synchronization on them was removed via escape analysis. Figure 10 shows how objects are internally represented by the Java HotSpot™ VM. The header word, which stores the identity hash code as well as age and mark bits for generational garbage collection, is also used to implement a *thin lock scheme* [2, 3].

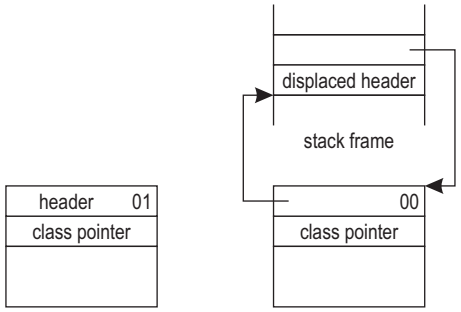


Figure 10. Relocking of objects.

In an unlocked object, the last two bits of the header word have the value 01. When a method synchronizes on an object, the header word and a pointer to the object are stored in a lock record within the current stack frame. Then a pointer to the lock record is installed in the object header. Since stack slots are always aligned at word boundaries, the last two bits of the header word are now 00 and identify the object as being locked. As long as an object is locked by a single thread, the VM gets away with thin locks. Only when another thread synchronizes on an already locked object, the thin lock must be inflated to a heavyweight monitor for the management of waiting threads.

In the relocking phase of deoptimization, the VM iterates over all monitor entries in the debugging information and determines if synchronization was removed. In this case, we lock the reallocated object as though the compiled code performed the locking and rely on the existing deoptimization code to convert the lock into the interpreter's representation. Since escape analysis proved the thread-locality of the object, we can always use a thin lock. The compiler reserves space for a lock record in the stack frame even if synchronization was eliminated.

5.5. Setup of Interpreter Frames

When control returns to a method marked for deoptimization, the patched machine code invokes a run-time stub that creates an array of *virtual stack frames*: one for the method to be deoptimized and one for each inlined callee. A virtual frame does not exist on the stack, but stores the local variables, operand stack and monitors of a particular method. Debugging information is used to fill the virtual frame with the correct values from the register map and the memory.

The method stack is adjusted as shown in Figure 11. The frames of the run-time stub and the method to be deoptimized are removed and the virtual frames are unpacked onto the stack. Pointers to reallocated objects are copied from the object entries into the interpreter frames. Then a frame for the continuation of the run-time stub is pushed back onto the stack.

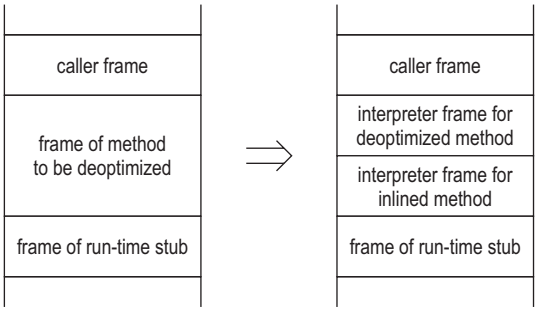


Figure 11. Adjustment of the method stack.

The bytecode index from the corresponding scope descriptor in the debugging information identifies the bytecode in the method that has to be executed next. The address of the interpreter code that handles the bytecode is pushed onto the stack as the return address. When the run-time stub returns, execution automatically continues in the interpreter.

6. Evaluation

The performance gains achieved by scalar replacement, stack allocation and synchronization removal largely depend on the characteristics of the executed application. Programs that allocate a lot of short-lived objects benefit more than programs that primarily perform mathematical computations. The fastest run of the _227_mtrt benchmark, for example, is accelerated by more than 30%, and the speedup for the complete SPECjvm98 [19] is about 5% when it is compiled with escape analysis. The measurements and the impact of escape analysis on compilation time are described in [14] and [15] and will not be elaborated here.

Since deoptimization represents an exceptional situation and thus is not performance-critical, measurements of its time consumption are not really meaningful. Therefore, instead of giving time measurements we chose a variety of counters to evaluate how our modifications in the run-time environment affect the machine code, debugging information, the stack frames and garbage collection.

This section presents the results for four allocation-intensive SPECjvm98 benchmarks and SPECjbb2005 (see Tables 1 and 2). The SPECjvm98 benchmarks were started only once and not repeated until stability in run time was reached. All benchmarks were executed on an Intel Pentium 4 processor 540 with 3.2 GHz and 1 GB of main memory, running Microsoft Windows XP Professional.

_227_mtrt allocates many short-lived objects and thus provides a lot of opportunities for scalar replacement of fields. 74.1% of all object allocations are eliminated. The size of heap-allocated memory decreases by 73%, so that only 52 instead of 165 garbage collection cycles are necessary. 39 methods whose size exceeds the usual inlining threshold are inlined in order to replace more objects by scalar variables. Therefore, the machine code grows by 2.4% at nearly the same number of compiled methods. Debugging information grows by 7.9%, because slightly more method dependencies are recorded and 1,015 object entries must be created, even though a usual run of the SPECjvm98 suite does not require deoptimization.

_209_db yields similar results. 91.5% of object allocations are eliminated, which reduces the size of allocated heap space by 61.9% and the number of garbage collections from 76 to 35. The compiler emits 3 allocation sites for stack objects. Although they are never executed, the compiler needs to reserve frame size and create oop map entries for them. The size of the debugging information slightly increases due to the 97 object entries.

_228_jack allocates a large number of string buffers that do not escape from the creating method and its callees. 16.8% of all objects are allocated on the stack, which reduces the size of allocated heap space by 8.6%. 49.2% of the executed write barriers perform a bounds check. During the 212 garbage collection cycles, only 120 stack objects are live and must be processed by the wrapper closure. 789 object entries are emitted for the reallocation of scalar-replaced and stack-allocated objects.

_213_javac allocates 9.7% of objects on the stack. 48.2% of all executed write barriers perform a bounds check. An independent micro benchmark yielded that extended write barriers are approximately 20% slower than standard write barriers, but the additional optimizations still achieve a speedup of about 3.2% for this benchmark. Although a write barrier with bounds check requires 30 instead of 12 bytes on Intel, the machine code is in total smaller with escape analysis than without, due to eliminated allo-

	_227_mtrt			_209_db			_228_jack			_213_javac		
	w/o EA	with EA		w/o EA	with EA		w/o EA	with EA		w/o EA	with EA	
methods compiled	123	125	–	65	64	–	232	232	–	535	536	–
machine code size (bytes)	111,024	113,664	+2.4%	53,616	54,048	+0.8%	389,152	357,920	−8.0%	545,984	525,472	−3.8%
average frame size (words)	14.21	14.91	+4.9%	15.20	16.06	+5.7%	14.66	17.03	+16.2%	16.38	17.87	+9.1%
object allocations eliminated	–	4,760,042	74.1%	–	2,892,496	91.5%	–	3,043	0.0%	–	49,283	1.4%
objects allocated on stack	–	147,211	2.3%	–	0	0.0%	–	1,034,112	16.8%	–	356,248	9.7%
objects allocated on heap	6,423,128	1,519,648	23.6%	3,175,480	267,129	8.5%	6,165,896	5,126,541	83.2%	3,690,136	3,259,700	88.9%
stores without barrier	–	64,200	2.2%	–	0	0.0%	–	404,363	3.8%	–	231,913	1.2%
stores with standard barrier	2,839,211	2,254,893	78.4%	29,972,806	26,927,554	99.5%	10,555,098	4,949,064	47.0%	19,181,052	9,694,282	50.6%
stores with extended barrier	–	558,439	19.4%	–	136,593	0.5%	–	5,186,503	49.2%	–	9,241,550	48.2%
safepoints in machine code	272	280	+2.9%	170	167	−1.8%	775	777	+0.3%	1,510	1,512	+0.1%
standard oop map entries	10,530	11,208	+6.4%	2,431	2,517	+3.5%	15,567	14,951	−4.0%	32,220	31,705	−1.6%
stack entries in oop map	–	215	–	–	23		–	504	–	–	464	–
heap memory allocated (kB)	158,078	42,653	−73.0%	73,601	28,069	−61.9%	194,828	178,031	−8.6%	186,519	178,771	−4.2%
garbage collection cycles	165	52	−68.5%	76	35	−53.9%	231	212	−8.2%	222	210	−5.4%
stack objects wrapped	–	192	–	–	0	–	–	120	–	–	43	–
debugging info (bytes)	62,254	67,197	+7.9%	23,621	24,693	+4.5%	133,450	144,281	+8.1%	285,920	301,032	+5.3%
method dependencies	289	310	+7.3%	41	45	+9.8%	235	261	+11.1%	524	539	+2.9%
object entries	–	1,015	–	–	97	–	–	789	–	–	937	–

Table 1. Statistical data for SPECjvm98 benchmarks with and without escape analysis.

cation sites, pointer loads and field accesses. Debugging information contains 937 object entries and 539 instead of 524 method dependencies.

Table 2 shows the same data for the SPECjbb2005 benchmark [18]. For a better comparison, we used a slightly modified variant which executes 50,000 transactions instead of running for a fixed time. 5% of the object allocations are eliminated and 1.4% of objects are allocated on the stack, which marginally reduces the allocated heap space and the number of garbage collection cycles. With escape analysis, an additional 203 methods are inlined, which affects the size of the machine code and the number of dependencies. Debugging information grows by 11.9% and contains 1,067 entries for the reallocation of optimized objects.

In all of these benchmarks, escape analysis helps to reduce the amount of data allocated on the heap. The resulting machine code executes faster, mainly because the overhead for allocation and initialization is eliminated, but also because the garbage collector runs less frequently. Although deoptimization usually does not occur, support for it is indispensable for the general use of escape analysis in the Java HotSpot™ VM. The impact on the performance and memory consumption is rather small and more than outweighed by the gains of the additional optimizations.

	SPECjbb2005		
	w/o EA	with EA	
methods compiled	485	485	–
machine code size (bytes)	451,088	477,632	+5.9%
average frame size (words)	14.87	17.03	+14.5%
object allocations eliminated	–	701,731	5.0%
objects allocated on stack	–	193,788	1.4%
objects allocated on heap	14,055,982	13,141,789	93.6%
stores without barrier	–	113,505	0.3%
stores with standard barrier	41,418,672	16,581,547	42.4%
stores with extended barrier	–	22,400,091	57.3%
safepoints in machine code	962	980	+1.9%
standard oop map entries	23,773	24,020	+1.0%
stack entries in oop map	–	386	–
heap memory allocated (kB)	599,069	577,251	−3.6%
garbage collection cycles	220	212	−3.6%
stack objects wrapped	–	6	–
debugging info (bytes)	208,855	233,808	+11.9%
method dependencies	768	795	+3.5%
object entries	–	1,067	–

Table 2. Statistical data for SPECjbb2005.

7. Related Work

Our escape analysis algorithm was influenced by the work of J.-D. Choi et al. [6], B. Blanchet [4], E. Ruf [16], J. Bogda and U. Hölzle [5], J. Whaley and M. Rinard [23], and D. Gay and B. Steensgaard [9]. The implementation is efficient enough for a dynamic compiler, conservative if a small additional gain would imply a time-consuming analysis, and able to deal with incomplete and partly compiled programs. The adaptation of the run-time environment was influenced by the algorithms and data structures used in the Java HotSpot™ VM.

U. Hölzle et al. present dynamic deoptimization as a conversion of optimized code into unoptimized code [10]. They use dynamic deoptimization to enable the system to debug individual methods at the source code level while executing

others at full speed, as well as to change a running program and immediately observe the effects of the change. The authors also introduce the term lazy deoptimization for deferring deoptimization until control returns to the method to be deoptimized.

K. Ishizaki et al. propose code patching instead of completely deoptimizing or recompiling a method [13]. The compiler analyzes the current class hierarchy to inline or directly jump to a virtual method, but also generates backup code which performs the original dynamic call. When the assumption about the class hierarchy becomes invalid, the machine code is patched to henceforth execute the backup code. This approach involves less run-time overhead than deoptimization, but it is also less flexible and e.g. not suitable for undoing optimizations like scalar replacement or stack allocation.

D. Detlefs and O. Agesen provide an approach for inlining virtual methods without the need to deoptimize [8]. If a method `m1` calls `p.m2`, they suggest to inline `m2` only if `p` already exists before `m1` is called and for example is passed to `m1` as a parameter. When a class is loaded later that overrides `m2`, then either `m1` has not been invoked yet or `p` has already been allocated. New invocations of `m1` may be called with an instance of the new class and thus must not execute the optimized machine code, but currently active invocations are not affected by the class loading and can safely run to the end. This concept could also be used for the interprocedural escape analysis and would eliminate the need to reallocate objects. However, general interprocedural escape analysis and inlining as it is used in the current Java HotSpot™ VM requires a deoptimization mechanism.

To reduce the costs of synchronization that cannot be eliminated, the Java HotSpot™ VM performs biased locking [17]. Normally, an object must be locked atomically because two threads may synchronize on it at the same time. In the context of biased locking, a pointer to the current thread is stored in the header of an object when it is locked for the first time. The object is said to be *biased* towards the thread. As long as the object is locked and unlocked by the same thread, synchronization need not be atomic. During deoptimization, the biasing must be revoked for all objects that are currently locked by deoptimized methods. As with reallocation of stack objects, this is not done lazily but immediately before program execution resumes.

R. Veldema et al. use reallocation of objects to support aggressive object combining [22]. The optimization combines two objects `o1` and `o2`, where `o1.f` points to `o2`, by appending the fields of `o2` to the object `o1`. This reduces the overhead of memory allocation and deallocation, as well as the number of pointer indirections because no pointer for the object `o2` is required to access its fields. A new object that is assigned to `o1.f` later is again inlined into `o1`. However, if `o2` is still reachable by another variable, its fields must not be overwritten. This situation is detected at run time. The object `o2` is then reallocated outside of `o1` and all pointers to it are adjusted.

T. Suganuma et al. implemented a region-based compilation technique, which does not compile complete methods but only frequently executed code regions [20]. The selection of regions to be compiled is based on static heuristics and dynamic profiling information. To allow a safe execution of partly compiled methods, the exit from a compiled region must be triggered. Since region-based compilation is performed only at high optimization levels of the compiler, the method in which a region exit occurs is known to be performance-critical. Therefore, it is recompiled instead of being continued in the interpreter. To avoid recursive recompilation, the method is compiled as a whole and prepares entry points for all future possible transitions. Similar to the deoptimization mechanism of the Java HotSpot™ VM, the current stack frame must be replaced with a frame for the recompiled method. Only if the old and the new frame are of the same shape, execution continues without a frame replacement.

8. Conclusions

This paper summarizes the results of our research project on run-time support for optimizations based on an escape analysis in the Java HotSpot™ VM. We introduced a representation for optimized objects used by the compiler to communicate information about scalar replacement of fields, stack allocation and synchronization removal to the garbage collector and the deoptimization framework. We presented an extended write barrier that is emitted when the compiler does not know if an object is located on the stack or on the heap. A wrapper for oop closures abstracts from stack objects and presents their pointer fields as root pointers to an unmodified garbage collector.

The interprocedural analysis neither relies on a closed-world assumption nor on preexistence of method receivers. Class hierarchy analysis is used to identify monomorphic call sites and to perform aggressive optimizations. A lightweight bytecode analysis produces escape information for methods that have not been compiled yet. The deoptimization framework enables dynamic class loading to invalidate machine code even while the invalidated method is running, because the compiler provides sufficient information to undo the compiler optimizations on demand and to continue execution of the method in the interpreter. Method-local objects whose allocations were eliminated are reallocated and initialized lazily, whereas stack-allocated objects are immediately moved to the heap. Objects for which synchronization was removed are relocked before the newly loaded code has the chance to make them accessible for other threads.

The deoptimization of objects is implemented in such a way that the garbage collector can safely run in the middle of reallocation. Handles and forwarding pointers ensure that reallocated objects are not deallocated until pointers to them are stored in the interpreter frame. If the VM runs out of memory during reallocation, it currently terminates because the program does not expect an exception at this point. The situation could be avoided by guaranteeing at method entries that enough heap memory is free to cover potential reallocation during deoptimization. We will address this issue as future work.

Acknowledgments

We want to thank Kenneth Russell, Thomas Rodriguez and David Cox from the Java HotSpot™ compiler group at Sun Microsystems for the close collaboration and the continuous support of our project.

References

[1] O. Agesen and D. Detlefs. Mixed-mode bytecode execution. Technical Report TR-2000-87, Sun Microsystems Laboratories, June 2000.

[2] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An efficient meta-lock for implementing ubiquitous synchronization. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 207–222, Denver, Nov. 1999.

[3] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268, Montreal, June 1998.

[4] B. Blanchet. Escape analysis for Java™: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, Nov. 2003.

[5] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 35–46, Denver, Nov. 1999.

[6] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems*, 25(6):876–910, Nov. 2003.

[7] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 77–101, Århus, Aug. 1995.

[8] D. Detlefs and O. Agesen. Inlining of virtual methods. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 258–278, Lisbon, June 1999.

[9] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *Proceedings of the International Conference on Compiler Construction*, pages 82–93, Berlin, Mar. 2000.

[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, San Francisco, June 1992.

[11] A. L. Hosking and R. L. Hudson. Remembered sets can also play cards. In *Proceedings of the ACM OOPSLA Workshop on Memory Management and Garbage Collection*, Washington, D.C., Oct. 1993.

[12] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual, Volume 2A & 2B: Instruction Set Reference*, 2006. Order numbers 253666-018 and 253667-018.

[13] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for a Java™ just-in-time compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 294–310, Minneapolis, Oct. 2000.

[14] T. Kotzmann. *Escape Analysis in the Context of Dynamic Compilation and Deoptimization*. PhD thesis, Institute for System Software, Johannes Kepler University Linz, Oct. 2005.

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

[16] E. Ruf. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 208–218, Vancouver, June 2000.

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

[18] Standard Performance Evaluation Corporation. *The SPEC JBB2005 Benchmark*. http://www.spec.org/jbb2005/.

[19] Standard Performance Evaluation Corporation. *The SPEC JVM98 Benchmarks*. http://www.spec.org/jvm98/.

[20] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for a Java just-in-time compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 312–323, San Diego, June 2003.

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

[22] R. Veldema, C. J. H. Jacobs, R. F. H. Hofman, and H. E. Bal. Object combining: A new aggressive optimization for object intensive programs. In *Proceedings of the Joint ACM-ISCOPE Conference on Java Grande*, pages 165–174, Seattle, Nov. 2002.

[23] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 187–206, Denver, Nov. 1999.