Escape Analysis in the Context of Dynamic Compilation and Deoptimization[*]

Thomas Kotzmann
Institute for System Software
Johannes Kepler University Linz
Linz, Austria
kotzmann@ssw.jku.at

Hanspeter Mössenböck
Institute for System Software
Johannes Kepler University Linz
Linz, Austria
moessenboeck@ssw.uni-linz.ac.at

ABSTRACT

In object-oriented programming languages, an object is said to *escape* the method or thread in which it was created if it can also be accessed by other methods or threads. Knowing which objects do not escape allows a compiler to perform aggressive optimizations.

This paper presents a new intraprocedural and interprocedural algorithm for escape analysis in the context of dynamic compilation where the compiler has to cope with dynamic class loading and deoptimization. It was implemented for Sun Microsystems' Java HotSpot^{TM} client compiler and operates on an intermediate representation in SSA form. We introduce equi-escape sets for the efficient propagation of escape information between related objects. The analysis is used for *scalar replacement* of fields and *synchronization removal*, as well as for *stack allocation* of objects and fixed-sized arrays. The results of the interprocedural analysis support the compiler in inlining decisions and allow actual parameters to be allocated on the caller stack.

Under certain circumstances, the Java HotSpot^{TM} VM is forced to stop executing a method's machine code and transfer control to the interpreter. This is called *deoptimization*. Since the interpreter does not know about the scalar replacement and synchronization removal performed by the compiler, the deoptimization framework was extended to reallocate and relock objects on demand.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Incremental compilers, Optimization*; D.4.1 [**Operating Systems**]: Process Management—*Synchronization*; D.4.2 [**Operating Systems**]: Storage Management—*Allocation / deallocation strategies*

General Terms

Algorithms, Languages, Performance

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

Keywords

Java, just-in-time compilation, optimization, escape analysis, scalar replacement, stack allocation, synchronization removal, deoptimization

1. INTRODUCTION

In Java, each object is allocated on the heap and deallocated by the garbage collector, which is invoked once in a while to examine the heap and release the memory of objects as soon as they are not referenced any longer. Allocation, access and synchronization of objects involve considerable costs:

- An object allocation does not only involve memory reservation. The fields of the object must also be initialized, because the garbage collector presumes that unassigned references are null. Additionally, the Java language specification defines default values for fields. Therefore, the memory occupied by the newly created object is typically cleared out in a loop even if values are explicitly assigned to the fields by the constructor afterwards.

- Reading or writing a field of an object requires one or two memory accesses depending on whether the address of the object is already in a register or not. In the context of a generational garbage collector, the modification of a field that holds a pointer is more expensive. It is associated with a *write barrier*, because pointers from older to younger generations must be recorded in a *remembered set* so that younger generations can be collected without inspecting every object in the older ones [12].

- Java programs typically coordinate multiple threads by synchronizing on a shared object. Much research has been done to reduce the cost of synchronization in JVM implementations, but complete elimination of useless synchronization is still a desirable goal.

Scalar replacement. The costs described above can be reduced by various compiler optimizations. An object that is neither assigned to a non-local variable nor passed as a parameter does not escape the method in which it is allocated. The compiler can eliminate the allocation and replace the object's fields by scalars (see Figure 1). This optimization eliminates memory allocation, initialization, field access and write barriers.

```
float foo(float d) {
  Circle c = new Circle(d / 2);
  return c.area();
}
```

original method

```
float foo(float d) {
  Circle c = new Circle();
  c.r = d / 2;
  return c.r * c.r * PI;
}
```

after inlining

```
float foo(float d) {
  float r = d / 2;
  return r * r * PI;
}
```

after scalar replacement

Figure 1: Example for scalar replacement of fields

Stack allocation. An object that is accessed only by the creating method and its callees does not escape the current thread. Although it must not be eliminated because the callees require a reference to the object, it can be allocated on the stack. Its memory is released implicitly when the stack frame is removed at the end of the method. This means that garbage collection runs faster and less frequently because the young generation of the heap does not fill up that fast. Moreover, synchronization on thread-local objects can be removed.

In this paper we present a new algorithm for escape analysis and related optimizations. The analysis was implemented for the client compiler of the Java HotSpot^{TM} Virtual Machine [22] in the context of a research collaboration between Sun Microsystems and the Institute for System Software at the Johannes Kepler University Linz. The paper contributes the following:

- It presents an intraprocedural and interprocedural escape analysis for a dynamic compiler. The results are used for scalar replacement, stack allocation and synchronization removal.
- It introduces equi-escape sets for the representation of dependencies between objects and the efficient propagation of escape states.
- It describes how the HotSpot^{TM} deoptimization framework was extended to deal with eliminated object allocations and removed synchronization.

Sun's client compiler is a simple and fast compiler dedicated to client programs, applets and graphical user interfaces [9]. Its objective is to achieve high compilation speed at a potential cost of peak performance. The front end operates on a SSA-based [5] high-level intermediate representation (HIR) and performs only few high-impact optimizations, such as constant folding, method inlining and common subexpression elimination. The back end converts the HIR into a low-level intermediate representation, which is the basis for linear scan register allocation [24] and code generation.

2. INTRAPROCEDURAL ANALYSIS

The HIR is built by parsing the bytecodes and performing an abstract interpretation [9]. At first, the boundaries of all basic blocks are determined by examining the destinations of jumps and the successors of conditional jump instructions. Then the basic blocks are filled with HIR instructions. Each HIR instruction that loads or computes a value represents both the operation and the result so that operands appear as pointers to prior instructions. Escape analysis and scalar replacement are performed in parallel with the construction of the HIR.

2.1 Escape States and Fields Array

Before the compiler can optimize the allocation and synchronization of objects, it has to know whether an object allocated in a method can be accessed from outside the method. This knowledge is derived from the HIR via escape analysis. If an object can be accessed by other methods or threads, it is said to *escape* the current method or the current thread, respectively. In the context of our work, possible escape levels are:

- `NoEscape`: The object is accessible only from within the method of creation. In most cases, the compiler can eliminate the object and replace its fields by scalar variables (see Section 2.2). Objects with this escape level are called *method-local*.
- `MethodEscape`: The object escapes the current method but not the current thread, e.g. because it is passed to a callee which does not let the argument escape. It is possible to allocate the object on the stack and eliminate any synchronization on it. Objects with this escape level are called *thread-local*.
- `GlobalEscape`: The object escapes globally, typically because it is assigned to a static variable or to a field of a heap object. The object must be allocated on the heap, because it can be referenced by other threads and methods.

In contrast to the bytecodes of a method, the SSA-based HIR does not contain any instructions for loading or storing local variables. They are eliminated during the abstract interpretation of the bytecodes. For this purpose, the compiler maintains a *state object* which contains a *locals array* to keep track of the values most recently assigned to a local variable [16]. If an instruction creates a value for the local variable with the number n, a pointer to this instruction is stored in the locals array at position n. If an instruction uses a local variable as an operand, it is provided with the corresponding value from the locals array, i.e. a pointer to the instruction where the value was created.

Bytecodes refer to local variables via indices, and the maximum number of variables for a certain method is specified in the class file. As far as fields are concerned, we do not know in advance which or how many fields are accessed within a method. Therefore, the state object is extended by a growable *fields array* which stores the current values of all fields. Figure 2 shows the contents of the locals array and the fields array for a simple sequence of bytecodes.

At the end of the HIR construction, when we know which objects escape and which do not, we substitute the fields of non-escaping objects by the values stored in the fields array. Provided that `a0` does not escape in the example, scalar replacement can eliminate the allocation and substitute any use of `a0.f` by `i1` and any use of `a0.g` by `i3`.

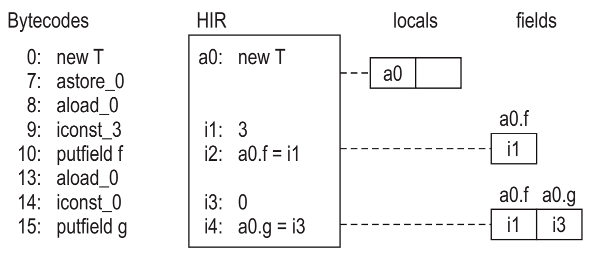

Figure 2: HIR and state object for a sequence of bytecodes

2.2 Analysis of Basic Blocks

In HIR code, an object is represented by its allocation instruction. It contains a field for the escape state, which is initialized with `NoEscape` and updated along with the construction of the HIR.

If an object is assigned to a non-local variable, not only itself becomes accessible by other threads, but also its fields. Therefore, each object keeps track of other objects that are referenced by its fields. If the object escapes, this list is traversed and the escape states of the referenced objects are adjusted.

When the compiler parses an instruction that might cause an object to escape, it adjusts the escape state of the instruction representing the object. The escape state can only increase towards `GlobalEscape`, but never decrease over time. The following paragraphs describe how certain instructions affect the escape state of objects and which actions are necessary for scalar replacement of fields:

- `p = new T():` Initially, `p` starts as `NoEscape`. If, however, the class `T` defines a finalizer, `p` is marked as `GlobalEscape` because the finalizer accesses the object before it is garbage collected. If the class `T` was not loaded yet, `p` is also marked as `GlobalEscape` because no information about the class is available.

- `a = new T[n]:` The escape state of a newly allocated array `a` is `NoEscape` only if the specified length `n` is a constant. Arrays of variable length are not replaced by scalars, because the compiler is not able to guarantee that the array is accessed only with valid indices and that no exception occurs at run time. They cannot be allocated on the stack either, because the maximum size of the stack frame must be known at compile time. These arrays are marked as `GlobalEscape`.

- `T.sf = p:` As soon as an object `p` is stored into a static field `sf`, it can be accessed by other threads and must therefore be marked as `GlobalEscape`. We cannot state anything about the lifetime of `p` or the scope it will be referenced from.

- `q.f = p:` When an object `p` is assigned to an instance field of `q`, it inherits the escape state of `q` provided that this state is higher than its own. If other methods or threads are able to obtain a reference to `q`, they can also access `p`. Additionally, `p` is inserted into the list of objects referenced by `q` so that its escape state gets updated in case `q` escapes later on. The fields array is extended by an entry for `q.f` and `p` is stored in the corresponding slot.

- `x = p.f:` This generates an HIR instruction which loads the field `p.f`. The compiler obtains the current value `v` of `p.f` from the fields array and remembers it for this HIR instruction. If `p` has the state `NoEscape` at the end of the HIR construction, `p.f` is replaced by `v`.

- `p == q:` If two objects `p` and `q` are compared, they must both exist at least on the stack, so their escape state is raised to `MethodEscape`. This is also true for an object compared with null.

- `(T) p:` A type cast requires the object `p` to be allocated even if scalar replacement is able to provide a value for any subsequent field access, because the cast might fail and then an exception has to be thrown. Therefore, `p` is marked as `MethodEscape`. Only if the compiler can prove statically that the object is always of the requested type, the cast is eliminated and the escape state remains unchanged.

- `p.foo(q):` A purely intraprocedural analysis is not able to predict what happens to the arguments of a method invocation. In the worst case, the callee assigns them to static fields or passes them on to other methods. Therefore, we mark all actual arguments as `GlobalEscape`. The object receiving the call is treated in just the same way as any other parameter. This rather conservative approach is refined in an interprocedural analysis (see Section 3).

- `return p, throw p:` Return values and exception objects are generally marked as `GlobalEscape` and never allocated on the stack because they can be accessed by the caller when the frame of the callee has already been released. Interprocedural analysis opens up an optimization if a formal parameter is returned which does not escape otherwise. In this case, the actual parameter may be allocated on the stack unless it escapes in the caller.

To check the correctness of scalar replacement and to detect errors in future modifications, our compiler contains verification code. If verification is enabled via a special VM flag, the compiler keeps track of field values but refrains from scalar replacement. Whenever a field is loaded from memory, a run-time assertion is inserted into the machine code which compares the field value with the scalar that the field could have been substituted with. If one of the assertions fails, the JVM displays an error message and halts the program.

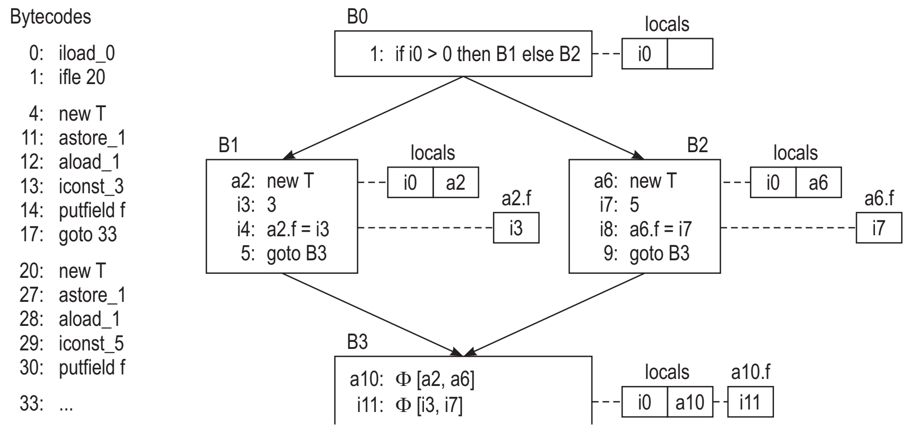

Figure 3: Phi functions for local variables and fields

2.3 Analysis of Control Flow

The SSA form [5] requires that every value has a single point of definition even if different values are assigned to a variable in alternative control flow paths. If multiple values of a variable flow together in a control flow graph, a phi function has to be inserted which merges the incoming values into a new value. In the context of scalar replacement, phi functions must be created both for local variables and field variables (see Figure 3).

The escape states for the operands of a phi function depend on each other. Assume four objects a0 to a3 and the three phi functions

$\Phi_0 = [$a0, a1$]$,
$\Phi_1 = [$a1, a2$]$, and
$\Phi_2 = [\Phi_1$, a3$]$.

Figure 4 shows the corresponding data flow. If some of the objects escaped, it would not make sense to treat the others as non-escaping and thus to replace their fields by scalar variables, because the objects could not be accessed in a uniform way anymore.

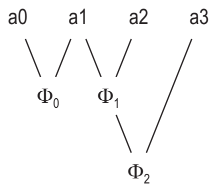

Figure 4: Phi functions and their operands

Instead of determining and propagating the maximum escape state, the operands of a phi function and the phi function itself are inserted into an *equi-escape set* (EES). All elements in the set share the same escape state, namely the maximum of the elements' original states.

An EES is implemented as an instruction tree based on the *union-find algorithm* [20]. For this purpose, every instruction has a parent field that points to another instruction in the same set. Each connected component corresponds to a different set. A special role is assigned to the root of the instruction tree. It acts as a representative for the set and specifies the escape state of the set's elements. Accordingly, only the escape state of the root needs to be updated if one of the elements escapes.

Figure 5 visualizes the process of EES creation for the phi functions in the above example. Initially, the objects are not connected to each other. The EES of the first phi function and its operands forms a three-node tree. Assume that the phi function itself is the root of the tree. This choice is arbitrary; any operand could have been put at the root. The second phi function has an operand a1 that is already contained in another set, so a1's representative Φ_0 is linked to Φ_1. After the third phi function has been processed, all objects and phi functions turn out to be elements of one large set.

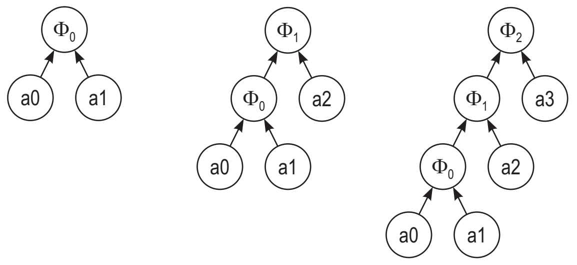

Figure 5: Stepwise creation of an EES

To detect whether two objects belong to the same set, their representatives are compared for identity. Starting from one of the objects at a time, the instruction tree is walked up until its root is reached. In order to speed up the repeated lookup, we make another pass through the tree immediately after the root has been found and set the parent field of each node encountered along the way to directly point to the root. Thus the trees get flattened over time. This is called *path compression* [20].

The escape state of an object loaded from a field usually does not affect the objects stored in other fields. For arrays, however, it is often not possible to prove the equality of two index calculations at compile time and thus to determine exactly which element is loaded. In this case, we conservatively consider all array elements to escape as soon as the object that represents the result of the array load escapes.

3. INTERPROCEDURAL ANALYSIS

The gain of an intraprocedural analysis is limited because objects are rarely used as pure data structures. Instead, they usually act as parameters or receivers of at least one method call. Inlining thus plays a major role in the improvement of escape analysis. Unfortunately, inordinate inlining rapidly fills the code buffer and slows down the compiler.

So interprocedural techniques are vital to tap the full potential of escape analysis. They determine which objects escape neither from the method that allocates them nor from any of its callees. The underlying idea is based on the fact that the compilation of a method produces escape information not only for local variables, but also for the method's formal parameters. In an interprocedural analysis, the compiler does not discard this information but stores it in the method descriptor, so that the escape state of actual parameters can be inferred from it when the method is called later on.

Apart from that, interprocedural analysis supports the compiler in making better inlining decisions. Not every method that can be bound statically is also inlined because the increase in compile time would outweigh the gain in run time. Some of the restrictions, such as the maximum code size, are based on heuristics and may be weakened if we knew that inlining would prevent objects from escaping.

In the course of HIR construction, formal parameters are treated in the same way as objects allocated within the method. They start as non-escaping objects and may be inserted into an EES or stored into fields of other objects. After completion of the HIR, their escape states are encoded by two bit vectors indicating which parameters do not escape and which can be allocated on the stack.

In Figure 6, the first argument `a0` escapes globally because it is assigned to the static field `sf`, while `a1` is compared to null and thus marked as stack-allocatable. The last argument `a2` is not used at all and remains non-escaping. The escape states are encoded by the bit vectors 001 and 011 denoting the set of method-local and thread-local parameters, respectively.

```
static boolean foo(Obj a0, Obj a1, Obj a2) {
  sf = a0;
  return (a1 != null);
}
```

Figure 6: Parameters with different escape states

At each call site, the compiler retrieves the interprocedural escape information from the callee's method descriptor. If the code size of the callee exceeds the maximum inline size within a certain limit, the escape information is used to decide if inlining is desirable nevertheless. It turned out to be a good heuristic to inline methods up to twice as large as the normal threshold if there is at least one parameter that does neither escape the caller nor the callee. After inlining, the allocation of such a parameter can be eliminated.

Even if a method cannot be inlined, the information about its parameters is helpful. If one of the formal parameters does not escape globally, the corresponding actual parameter can be treated as thread-local unless it escapes in the caller. Although it must not be replaced by scalars, it may be allocated on the caller's stack and synchronization on it can be removed.

The Java HotSpot^{TM} VM interprets a method several times before it is compiled. For this period of time, no interprocedural escape information is available. If we reach a call site of a method that was not compiled yet, we perform a fast and conservative analysis on the bytecodes to get escape information for the arguments. Otherwise the arguments of recursive method calls would be treated as escaping globally by default.

The analysis on the bytecodes traces escape information only for arguments and not for objects allocated within the method. It considers each basic block separately and checks if it lets one of the arguments escape. This is more conservative but faster to compute, because no phi functions are required and control flow can be ignored. The analysis stops as soon as all arguments are seen to escape. When the method is compiled later, the provisional escape information is replaced with a more precise one. Since the compiler is less conservative than the bytecode analyzer, the escape state of an argument can only change from global escaping to stack-allocatable.

4. THREAD-LOCAL OBJECTS

Objects that do not escape the allocating method nor its callees can be allocated on the stack. The total size of a stack frame must be known at compile time. Therefore, each allocation site of a stack object allocates the object at a unique location in the frame. If an object is created within a loop, it is allocated on the stack only if the same stack slot can be reused in every iteration. Since stack objects do not need explicit memory reservation, machine code is generated only for the initialization of the object fields as required by the Java language specification [8].

Apart from a cheap allocation, stack objects also facilitate an efficient field access. The location of stack objects within the current stack frame is already available at compile time. Since the fields can be accessed relative to the frame's base pointer, the address of the object needs not be loaded.

The assignment to a field that references an object normally requires a write barrier [12]. However, no write barriers are emitted for assignments to fields of stack-allocated objects. They are not necessary because pointers in stack objects are root pointers and must be inspected at every collection cycle anyway.

Even if a formal argument does not escape a method, the method may be called either with a stack-allocated or a heap-allocated actual argument. For assignments to fields of such formal arguments a slightly modified write barrier is emitted, which performs a bounds check before the card marking array is accessed. Figure 7 shows the generated code for the Intel IA-32 architecture [13].

The card index is calculated via a right-shift of the object address in the `EAX` register. Then the index of the first

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

Figure 7: Write barrier with bounds check

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

If a block synchronizes on a thread-local object, synchronization can be removed because the object will never be locked by any other thread. Before Java 5.0, the Java memory model restricted synchronization removal. Each thread has a working memory, in which it may keep copies of the variables that are shared between all threads. According to the old Java memory model [8], locking and unlocking actions caused a thread to flush its working memory, which guaranteed that the shared values were reloaded from main memory afterwards. Therefore, the old Java memory model did not allow useless synchronization to be completely removed, but the new one for Java 5.0 does [15].

When the back end parses an HIR instruction that synchronizes on a thread-local object, it does not emit any machine code for the synchronization. If a method is declared synchronized, locking code must be preserved for the case that the method is called on a shared receiver. Therefore, the compiler tries to inline the method by embedding the body in a synchronized block. If the receiver for this call site turns out to be thread-local, synchronization is removed.

5. RUN-TIME SUPPORT

A method can only be inlined if the compiler is able to statically identify this method despite polymorphism and dynamic method binding. Apart from static and final callees, this is possible if class hierarchy analysis [6] finds out that only one suitable method exists. If, however, a class is loaded later that provides another suitable method, it is possible that the wrong implementation was inlined.

Assume that the class `B` was not loaded yet when machine code for the method `foo` from Figure 8 is generated. The just-in-time compiler optimistically assumes that there is only one implementation for the method `bar` and inlines `A.bar` into `foo`.

If `create` returns a `B` object, the class `B` is loaded and the inlining decision turns out to be wrong. In this case, the machine code for `foo` is invalidated and execution of the method is continued in the interpreter.

The interpreter expects all local variables such as `x` and `p` to be stored in the stack frame, whereas the machine code might keep some of them in registers. Therefore, a new stack frame must be allocated for the interpreter and filled with the variables' current values. This process is called *deoptimization* [11]. It makes use of *debugging information* which is generated by the compiler and specifies the locations of all variables' values for every point in the program where deoptimization can occur.

Debugging information is created by the register allocator, because the locations of local variables are not known before register allocation, and afterwards they are no longer available. For each local variable a *scope value* is created that describes the location of the variable. The list of such scope values makes up the debugging information. When the machine code of a compiled method is installed into the VM, the debugging information is stored together with the native method in a compressed form.

```
class A {
  void bar() { ... }
}

class B extends A {
  void bar() { ... }
}

int foo() {
  int x = 3;
  Point p = new Point();
  p.x = x;
  p.y = 3 * x + 1;
  A q = create();
  q.bar();
  return p.x * p.y;
}
```

Figure 8: Example for optimistic inlining

For scalar replacement and synchronization removal, the deoptimization framework must be able to reallocate and relock objects because the interpreter does not know about these optimizations. Therefore, the debugging information of the original Java HotSpot^{TM} client compiler was extended by information about the fields of eliminated objects and about which objects must be locked.

An eliminated object is represented by a special kind of scope value. It specifies the class of the object and contains a list of scope values itself to describe the locations of the fields. In the example above, the debugging information for the point immediately before the method `bar` is called states that

- the local variable `x` has the value 3,
- the local variable `p` refers to a `Point` object eliminated by scalar replacement, whose field `x` is the constant 3 and whose field `y` is stored in the `ECX` register,
- the local variable `q` refers to an object whose address is stored in the `EBX` register.

The debugging information is used to create a state from which the interpreter can continue. At first, the deoptimization framework reallocates all eliminated objects on the heap. In this example, it creates an instance of the class `Point` and initializes its field `x` with 3 and its field `y` with the value of the `ECX` register.

References to reallocated objects are stored in an array and treated as root pointers if garbage collection is needed in the middle of reallocation. After reallocation, the garbage collector must not run until deoptimization has finished, because the array is no longer available and newly created objects would immediately be freed again.

Afterwards, all thread-local objects for which synchronization was eliminated are relocked. Because the representations of locked objects differ between interpreted and compiled code, we lock the objects as though the compiled code performed the locking and rely on the existing deoptimization code to convert the locks into the interpreter's representation. Debugging information also considers the level of synchronization, which allows the deoptimization framework to restore recursive locking.

	without EA		with EA		ratio	
	slowest	fastest	slowest	fastest	slowest	fastest
_227_mtrt	1.391	1.094	1.219	0.859	1.141	1.274
_202_jess	1.907	1.625	1.875	1.594	1.017	1.019
_201_compress	5.969	5.938	5.969	5.922	1.000	1.003
_209_db	11.859	11.625	11.719	11.485	1.012	1.012
_222_mpegaudio	3.047	2.765	3.046	2.765	1.000	1.000
_228_jack	3.359	3.093	3.312	2.953	1.014	1.047
_213_javac	5.516	4.312	5.453	4.219	1.012	1.022

Table 1: Elapsed times of SPECjvm98 benchmarks on Intel (in seconds)

	without EA		with EA		ratio	
	slowest	fastest	slowest	fastest	slowest	fastest
_227_mtrt	2.719	2.200	2.186	1.560	1.244	1.410
_202_jess	4.925	4.450	4.961	4.424	0.993	1.006
_201_compress	11.359	11.252	11.472	11.356	0.990	0.991
_209_db	20.323	19.434	19.893	18.945	1.022	1.026
_222_mpegaudio	8.158	7.673	8.162	7.673	1.000	1.000
_228_jack	6.685	6.006	6.727	5.993	0.994	1.002
_213_javac	11.602	9.340	11.788	9.373	0.984	0.996

Table 2: Elapsed times of SPECjvm98 benchmarks on SPARC (in seconds)

Finally, a stack frame for the interpreter is set up as shown in Figure 9. When execution continues, the interpreter examines the dynamic type of `q` to find out that `B.bar` must be called instead of `A.bar`. Later, the deoptimized method may be compiled again, but this time the method `bar` will not be inlined because both classes `A` and `B` are loaded.

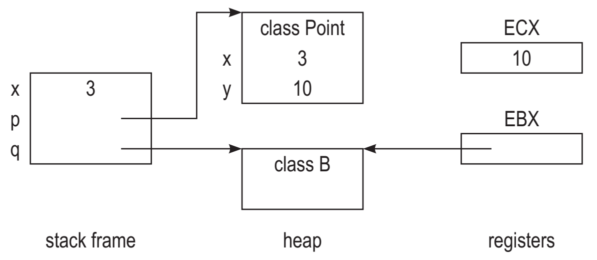

Figure 9: State created by deoptimization

Apart from extending the deoptimization framework to reallocate and relock objects, garbage collection had to be adapted to deal with stack objects. A stack object must not be moved in memory, but the heap objects referenced by its fields need to be visited and kept alive. Therefore, we modified the garbage collector so that it does not visit stack objects but treats pointer fields within stack objects as root pointers.

6. EVALUATION

This section evaluates our escape analysis algorithm on the SPECjvm98 benchmark suite [21]. For this purpose, we extended the Java HotSpot^{TM} client VM of the JDK 5.0 by our optimizations. The benchmarks were executed on an Intel Pentium 4 processor with 3.2 GHz and 1 GB of main memory, running Microsoft Windows XP Professional. The results are not approved SPECjvm98 metrics, but adhere to the run rules for research use.

Table 1 shows the elapsed times of the benchmarks on Intel with and without escape analysis. Three runs were required per benchmark to reach stability. Compilation speed decreases from 236,654 to 208,559 bytes/s. However, even the slowest runs, which invoke the compiler frequently, benefit from escape analysis. The fastest runs are achieved when most methods have already been compiled and thus indicate the quality of the machine code.

The _227_mtrt benchmark implements a multi-threaded ray-tracing algorithm. It mainly benefits from scalar replacement because it uses a lot of short-lived data structures, such as points and vectors, whose allocation can be eliminated. About 3.8 of 5.3 million allocations are eliminated per run. Without escape analysis, the fastest run requires 1.094 seconds. With escape analysis and scalar replacement enabled, the time is reduced to 0.859 seconds which results in a speedup of 27.4%. When the benchmark is compiled with the Java HotSpot^{TM} server compiler, a more highly-optimizing compiler which does not currently implement escape analysis, the fastest run takes 0.985 seconds.

The _228_jack benchmark generates a Java parser from a specification file and primarily benefits from lock elision on `StringBuffer` objects. In Java 5.0, strings can be concatenated with the unsynchronized `StringBuilder`. The synchronized `StringBuffer` is still available for compatibility reasons, but forwards almost all method calls to the `StringBuilder`. When the methods of `StringBuffer` are inlined, synchronization can be eliminated. Although 1.2 million object locks are removed per run, the speedup of 4.7% is smaller than the one achieved for _227_mtrt, because the `StringBuffer` object often cannot be eliminated by scalar replacement and synchronization is cheaper than object allocation.

As regards _201_compress and _222_mpegaudio, fast memory allocation is not as crucial as for the two benchmarks described above. They allocate only 406 objects and 361 arrays per run and thus do not provide any optimization opportunities for escape analysis.

	static numbers	dynamic numbers
object allocations eliminated	50	17,679,091
object allocations on stack	214	2,683,179
object allocations on heap	1,849	41,073,677
array allocations eliminated	3	3,202,110
array allocations on stack	4	2,037,655
array allocations on heap	485	23,597,429
locks eliminated	712	6,776,157
locks performed	432	47,320,036
locks at method entry	44	175,081,272

Table 3: Escape analysis statistics for the SPECjvm98 benchmarks

	without EA	with EA	ratio
FFT (1024)	267.0301	267.0301	1.000
SOR (100x100)	446.8820	446.8820	1.000
Monte Carlo	22.9668	90.4432	3.938
Sparse matmult (N=1000, nz = 5000)	283.3377	283.3377	1.000
LU	379.8924	379.8924	1.000
Composite Score	280.0218	293.5171	1.048

Table 4: Scores of SciMark in Mflops

Table 2 presents the results of SPECjvm98 for SPARC. They were measured under Sun OS 5.9 on a Sun Blade 2500 workstation with 2 processors at 1,280 MHz and 2 GB of main memory. _227_mtrt achieves a higher speedup than on Intel, because more fields of eliminated objects can be kept in registers. Other benchmarks benefit less from escape analysis, because the current implementation of bounds checks in write barriers on SPARC is more expensive than on Intel.

Table 3 shows how many allocations and synchronizations are eliminated in SPECjvm98. Static numbers refer to sites in the machine code, whereas dynamic numbers indicate how often these sites are executed. Locks at the entry of synchronized methods cannot be eliminated because the methods are not necessarily invoked on thread-local receivers. Scalar replacement eliminates the allocation of 432 MB on the heap. The total size of generated machine code decreases from 1.19 MB to 1.10 MB (7.6%).

SciMark is another Java benchmark suite for scientific and numerical computations [18]. It mainly measures speed of floating-point operations and therefore does not contain a lot of allocation sites. The Monte Carlo benchmark, however, creates a random number generator object and repeatedly invokes the synchronized method `nextDouble` on it. Escape analysis inspects the bytecodes of this method in order to estimate the escape states for its parameters. Based on the interprocedural escape information, the compiler inlines `nextDouble`, removes synchronization and eliminates the allocation of the random number generator. At run time, about 134 million locks are eliminated. As a result, the score of this benchmark is nearly four times higher with escape analysis than without (see Table 4).

7. RELATED WORK

Jong-Deok Choi et al. from the IBM T. J. Watson Research Center created a framework for escape analysis based on a program abstraction called *connection graph* which captures the relationship between object references and heap-allocated objects [4]. Escape analysis is mapped to a reachability problem over the connection graph. It is used in the context of a static Java compiler for stack allocation of objects and elimination of unnecessary synchronization. In an interprocedural analysis, summary information obtained for a callee is used to update the connection graph of the caller. Therefore, all callees must have been completely analyzed before the caller is processed.

Bruno Blanchet extended the Java-to-C compiler TurboJ by escape analysis [1]. The algorithm transforms Java code into SSA form, builds equations and solves them with an iterative fixpoint solver. Escaping parts of values are represented via integers, which is less costly than graphs and leads to a fast and precise analysis. The results are used for stack allocation and synchronization removal, but not for scalar replacement of fields. Regarding dynamic class loading, the implementation either determines which classes in the class path will actually be loaded or relies on information provided by the user.

Erik Ruf from Microsoft Research implemented an optimization that removes unnecessary synchronization operations in Java [19]. His optimization even eliminates synchronization on objects that are reachable from static fields but accessed only from a single thread. It relies on an equivalence-based representation similar to our equi-escape sets, in which potentially aliased values are forced to share common representative nodes. The optimization is applied to statically compiled programs and was implemented in the Marmot native compilation system for Java which does not support dynamic class loading.

David Gay and Bjarne Steensgaard [7] implemented another escape analysis algorithm for Marmot. The analysis computes two properties for each local variable. The first one specifies if the variable holds a reference that escapes due to an assignment or a throw statement, and the second one if the reference escapes by being returned from the method. Each statement of a program may impose constraints on these properties that must be solved. No attempts are made to track references through assignments to fields, i.e. any reference assigned to a field is assumed to possibly escape from the method in which the assignment occurs. The results of the analysis are used for stack allocation and object elimination.

Jeff Bogda and Ambuj Singh present an incremental shape analysis based on Ruf's approach [3]. It operates on an incomplete call graph and modifies the results as the call graph grows. The authors evaluate three strategies of when to start the interprocedural analysis, and whether to make optimistic or pessimistic assumptions for optimization. Andy C. King also builds on Ruf's analysis and adapts it to deal with dynamic class loading [14]. Any class loaded after a partial analysis of a snapshot of the program is analyzed and incorporated into the system.

Frédéric Vivien and Martin Rinard observed that almost all of the objects are allocated at a small number of allocation sites [23]. For this reason, they present an algorithm which incrementally analyzes only those parts of a program that may deliver useful results. Their algorithm performs an incremental analysis of the neighborhood of the program surrounding selected object allocation sites. It first skips the analysis of all potentially invoked methods, but maintains enough information to reconstruct the results of analyzing the methods when they turn out to be useful. Such an incremental analysis works well for stack allocation, but for synchronization removal, a whole-program analysis promises more optimization opportunities.

V. Krishna Nandivada and David Detlefs from Sun Laboratories developed several techniques based on escape analysis that allow the elimination of unnecessary write barriers supporting the *snapshot-at-the-beginning* (SATB) style of concurrent garbage collection [17]. Two static analyses identify stores to heap locations guaranteed to contain null before the write. The first one does so for fields of objects, and the second one for elements of object reference arrays. Such *initializing* stores do not require SATB barriers. The optimization was implemented in a version of the Java HotSpot^{TM} client compiler without an intermediate representation in SSA form.

8. CONCLUSIONS

We presented a new algorithm for escape analysis that was implemented in a production system. It is especially tailored to the needs of a dynamic compiler which lacks a view of the complete program. The results are used for scalar replacement of fields, to allocate objects and fixed-sized arrays on the stack, and to remove synchronization on thread-local objects. Escape states are efficiently propagated among related objects via equi-escape sets. The deoptimization framework was extended to reallocate and relock objects just before execution continues in the interpreter.

We have implemented a combination of an intraprocedural and an interprocedural analysis. The results of the interprocedural analysis are used to identify objects that can be allocated on the caller stack as well as to estimate where inlining benefits scalar replacement of fields. A light-weight analysis on the bytecodes produces interprocedural escape information for formal arguments of methods that have not been compiled yet.

Future work will focus on a more aggressive escape analysis. A deeper analysis of the call tree will for example increase the number of eliminated and stack-allocated objects. Performance could also be improved by generating unsynchronized versions of some methods which are called whenever the receiver does not escape. Based on the results and experience gained from our implementation, Sun Microsystems plans to add escape analysis also to the Java HotSpot^{TM} server compiler.

9. ACKNOWLEDGMENTS

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

10. REFERENCES

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

[2] 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.

[3] J. Bogda and A. Singh. Can a shape analysis work at run-time? In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, Monterey, 2001.

[4] J.-D. Choi et al. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems*, 25(6):876–910, Nov. 2003.

[5] R. Cytron et al. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[6] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *Lecture Notes in Computer Science*, 952:77–101, 1995.

[7] 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, 2000.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification*. Addison-Wesley, second edition, June 2000.

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

[10] M. Hirzel, A. Diwan, and M. Hind. Pointer analysis in the presence of dynamic class loading. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 96–122, Oslo, June 2004.

[11] U. Hölzle et al. 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.

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

[13] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual, Volume 2A & 2B: Instruction Set Reference*, 2004. Order Numbers 253666 and 253667.

[14] A. C. King. Removing GC synchronisation (extended version). Technical Report 11-03, Computing Laboratory, University of Kent, Apr. 2003.

[15] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 378–391, 2005.

[16] H. Mössenböck. Adding static single assignment form and a graph coloring register allocator to the Java Hotspot^{TM} client compiler. Technical Report 15, Johannes Kepler University Linz, Nov. 2000.

[17] V. K. Nandivada and D. Detlefs. Compile-time concurrent marking write barrier removal. In *Proceedings of the International Symposium on Code Generation and Optimization*, San Jose, 2005.

[18] R. Pozo and B. Miller. *Java SciMark 2.0*. http://math.nist.gov/scimark2/.

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

[20] R. Sedgewick. *Algorithms*, pages 441–449. Addison-Wesley, second edition, 1988.

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

[22] Sun Microsystems, Inc. *The Java HotSpot Virtual Machine, v1.4.1*, Sept. 2002. http://java.sun.com/products/hotspot/.

[23] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, Snowbird, June 2001.

[24] C. Wimmer and H. Mössenböck. Optimized interval splitting in a linear scan register allocator. In *Proceedings of the Conference on Virtual Execution Environments*, June 2005.