

Partial Escape Analysis and Scalar Replacement for Java

Lukas Stadler
Johannes Kepler University
Linz, Austria
lukas.stadler@jku.at

Thomas Würthinger
Oracle Labs
thomas.wuerthinger
@oracle.com

Hanspeter Mössenböck
Johannes Kepler University
Linz, Austria
moessenboeck@ssw.jku.at

ABSTRACT

Escape Analysis allows a compiler to determine whether an object is accessible outside the allocating method or thread. This information is used to perform optimizations such as Scalar Replacement, Stack Allocation and Lock Elision, allowing modern dynamic compilers to remove some of the abstractions introduced by advanced programming models.

The all-or-nothing approach taken by most Escape Analysis algorithms prevents all these optimizations as soon as there is one branch where the object escapes, no matter how unlikely this branch is at runtime.

This paper presents a new, practical algorithm that performs control flow sensitive Partial Escape Analysis in a dynamic Java compiler. It allows Escape Analysis, Scalar Replacement and Lock Elision to be performed on individual branches. We implemented the algorithm on top of Graal, an open-source Java just-in-time compiler, and it performs well on a diverse set of benchmarks.

In this paper, we evaluate the effect of Partial Escape Analysis on the DaCapo, ScalaDaCapo and SpecJBB2005 benchmarks, in terms of run-time, number and size of allocations and number of monitor operations. It performs particularly well in situations with additional levels of abstraction, such as code generated by the Scala compiler. It reduces the amount of allocated memory by up to 58.5%, and improves performance by up to 33%.

Categories and Subject Descriptors

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

General Terms

Algorithms, Languages, Performance

Keywords

escape analysis, java, virtual machine, just-in-time compilation, intermediate representation, speculative optimization

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

CGO '14 February 15 - 19 2014, Orlando, FL, USA

Copyright 2014 ACM 978-1-4503-2670-4/14/02 ...\$15.00.

1. INTRODUCTION

State-of-the-art Virtual Machines employ techniques such as advanced garbage collection, alias analysis and biased locking to make working with dynamically allocated objects as efficient as possible. But even if allocation is cheap, it still incurs some overhead. Even if alias analysis can remove most object accesses, some of them cannot be removed. And although acquiring a biased lock is simple, it is still more complex than not acquiring a lock at all.

Escape Analysis can be used to determine whether an object needs to be allocated at all, and whether its lock can ever be contended. This can help the compiler to get rid of the object's allocation, using *Scalar Replacement* to replace the object's fields with local variables.

Escape Analysis checks whether an object escapes its allocating method, i.e., whether it is accessible outside this method. An object escapes, for example, if it is assigned to a static field, if it is passed as an argument to another method, or if it is returned by a method. In these cases the object needs to exist on the heap, because it will be accessed as an object in some other context.

In many cases, however, an object escapes just in a single unlikely branch. Nevertheless, this prevents optimizations. Therefore, we suggest a flow-sensitive Escape Analysis which we call *Partial Escape Analysis*.

The idea behind Partial Escape Analysis is to perform optimizations such as Scalar Replacement in branches where the object does not escape, and make sure that the object exists in the heap in branches where it does escape. Additionally, our Partial Escape Analysis works not on bytecodes but on the compiler's intermediate representation, so that it can be applied, possibly multiple times, at any point during compilation.

This paper contributes the following novel aspects:

- A control-flow-sensitive Partial Escape Analysis algorithm that checks the escapability of objects for individual branches.
- The integration of our Partial Escape Analysis in a Java compiler based on SSA form, speculative optimization, and deoptimization.
- An evaluation of this algorithm on a set of current benchmarks, in terms of run-time, number and size of allocations and number of monitor operations, showing that our algorithm performs well on a variety of benchmarks.

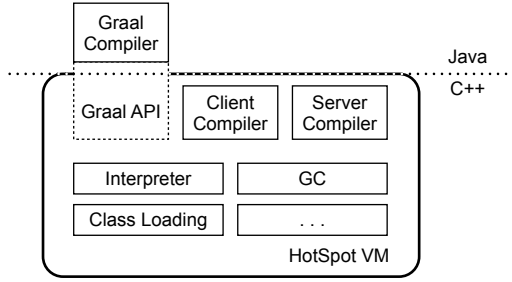


Figure 1: Overview of HotSpot and Graal.

2. SYSTEM OVERVIEW

We implemented our analysis for Graal, which is a Java just-in-time compiler written in Java that runs on top of the HotSpot VM (Figure 1). While it can completely replace the client and server compilers, it reuses all other VM components, such as the interpreter, the garbage collection subsystem and class loading. It is open-source and available via the OpenJDK Graal project [11].

Graal translates Java bytecode into a high-level intermediate representation called Graal IR [5], on which it performs all optimizations. This SSA-based intermediate representation models both control flow and data flow dependencies between nodes. While it explicitly expresses the control flow of the code, many operations are not fixed at specific locations. Rather, the position at which these operations are executed is determined solely by their data flow dependencies. There are usually many positions where an operation could be placed, and it is up to the so-called *Scheduler* to determine a good one.

Graal is an aggressive and optimistic compiler that often makes assumptions about the state and behavior of the running application. This includes assumptions such as some classes not having subclasses, and some branches never being taken. When one of these assumptions is invalidated, e.g., by loading a new class or by entering an unexpected branch, the execution needs to be transferred from compiled code back to the interpreter (which does not make any assumptions and can execute all code). This switch back to the interpreter is called *Deoptimization* [7] and requires a translation from the machine state (native stack frames) back to the Java VM state, which is used by the interpreter.

The Graal IR contains a mapping to Java VM state for all positions that can cause a deoptimization to occur. This mapping is expressed as **FrameState** nodes and consists of the current position (bytecode index and method), the local variables, the contents of the expression stack and the locked objects.

After inlining, one position can map to multiple Java VM stack frames. A frame state thus contains a reference to an *outer frame state*, which is the caller’s state. This reference is used to create chains of **FrameState** nodes that describe the state of all inlined methods at the current position.

The Graal IR keeps the frame states not at the points where the actual deoptimizations take place, but at the points where side effects may occur. Side effects are operations such as field stores and method calls, which cannot be reexecuted. Operations like an integer addition do not

```

1  class Key {
2      int idx;
3      Object ref;
4      Key(int idx, Object ref) {
5          this.idx = idx;
6          this.ref = ref;
7      }
8      synchronized boolean equals(Key
9          other) {
10         return idx == other.idx &&
11            ref == other.ref;
12     }
13 }
14 static CacheKey cacheKey;
15 static Object cacheValue;
16
17 Object getValue(int idx, Object ref) {
18     Key key = new Key(idx, ref);
19     if (key.equals(cacheKey)) {
20         return cacheValue;
21     } else {
22         return createValue(...);
23     }
24 }

```

Listing 1: Simple example.

have side effects; they can be reexecuted and will lead to the same result.

For example, a field store is associated with a frame state that describes the state *after* the store. A machine code instruction that causes or can cause a deoptimization will be associated with the after state of the last side-effecting instruction. All instructions that were executed in-between do not have side effects and can therefore be reexecuted in the interpreter.

3. ESCAPE ANALYSIS

Escape Analysis checks whether an allocated object escapes (i.e., can be used outside) the allocating method or thread. This happens, for example, if it is assigned to a global variable or heap object, or if it is passed as a parameter to some other method. Compilers use Escape Analysis to determine the dynamic scope and the lifetime of allocated objects. The result of this analysis allows the compiler to perform numerous optimizations on operations such as object allocations, synchronization primitives and field accesses.

Listing 1 shows a small piece of code that will serve as an example to show the benefits of Escape Analysis: The **getValue** method creates a new **Key** object and checks whether it is in the cache. If so, the method returns the cached value. Otherwise, it creates and returns a new value (the method **createValue** is not discussed here).

When **getValue** is compiled, the compiler will most likely perform some inlining, which might cause the actually compiled code to look like Listing 2. The **Key** constructor and the **equals** method have been inlined into the **getValue** method, and a **synchronized** block was created to achieve synchronization on the inlined **equals** method.

When Escape Analysis examines the resulting method, it will come to the conclusion that no reference to the allocated **Key** object escapes from the current compilation scope.

```

1 Object getValue(int idx, Object ref) {
2   Key key = alloc Key;
3   key.idx = idx;
4   key.ref = ref;
5   Key tmp1 = cacheKey;
6   boolean tmp2;
7   synchronized (key) {
8     tmp2 = key.idx == tmp1.idx &&
9           key.ref == tmp1.ref;
10  }
11  if (tmp2) {
12    return cacheValue;
13  } else {
14    return createValue(...);
15  }
16 }

```

Listing 2: Example from Listing 1 after inlining.

```

1 Object getValue(int idx, Object ref) {
2   int idx1 = idx;
3   Object ref1 = ref;
4   Key tmp = cacheKey;
5   if (idx1 == tmp.idx && ref1 ==
6       tmp.ref) {
7     return cacheValue;
8   } else {
9     return createValue(...);
10  }

```

Listing 3: Example from Listing 2 after Scalar Replacement and Lock Elision.

This implies that no references to the object exist after the method has returned, and that no other thread can ever see a reference to this object. The compiler can use these observations to perform a number of optimizations:

- The allocation of the object on the garbage collected heap can be replaced with allocation on the stack or in other non-garbage-collected allocation areas such as zones¹.
- *Scalar Replacement* can be used to eliminate the allocation altogether, by replacing the fields of the object with local variables.
- Since the object’s lock will never be contended, *Lock Elision* can remove the synchronization on **key**.

If the compiler uses Scalar Replacement and Lock Elision, the result might look like in Listing 3. The allocation was replaced with the local variables **idx1** and **ref1**, and the **synchronized** statement was removed entirely.

Traditionally, Escape Analysis uses algorithms such as *Equi-Escape Sets* [8] to determine which objects escape from the scope. These algorithms build sets of objects that have the same escape state, with each object initially being in a separate set. By analyzing all operations in the method the system can merge sets (e.g., when an object in one set is

¹Zones are heap areas with a known, limited lifetime. The whole area can be freed in bulk when a certain scope is left.

```

1 Object getValue(int idx, Object ref) {
2   Key key = new Key(idx, ref);
3   if (key.equals(cacheKey)) {
4     return cacheValue;
5   } else {
6     cacheKey = key;
7     cacheValue = createValue(...);
8     return cacheValue;
9   }
10 }

```

Listing 4: Complex example.

assigned to a field of an object in another set), or mark a set as escaping (e.g., when an object in this set is assigned to a global variable).

4. PARTIAL ESCAPE ANALYSIS

In many cases, making a global decision about the escapability of objects does not allow the compiler to perform the above optimizations. For example, the object allocated in Listing 4 escapes into the global variable **cacheKey**, so that Escape Analysis would consider it to be escaping.

However, if we only consider the path through the **true** branch of the **if** statement, the object does not escape. Analyzing the escapability of objects for individual branches is called *Partial Escape Analysis*. Partial Escape Analysis iterates over the code and maintains the current escape state and the current contents of allocated objects during this process. Initially, each allocated object is in the state **virtual**, which means that there was no reason yet to actually allocate it. As the algorithm progresses along the control flow, it updates this state when instructions operate on the allocated object.

The transition from Listing 5 to Listing 6 shows how Partial Escape Analysis lets the compiler optimize the code in this example:

- The allocation in line 2 is removed, and an entry for this object is created that specifies that it is **virtual** and that all fields have their default values.
- The assignments to the fields **idx** and **ref** in lines 3 and 4 are removed, and their effects are remembered by updating the object’s field states.
- When entering the **synchronized** region in line 7, the object is still **virtual**. The monitor enter operation is removed, and the object’s state is augmented with a **locked** flag that specifies that this object would have been locked if it actually existed at this point.
- The accesses to the **idx** and **ref** fields of the **virtual** object in lines 8 and 9 can be replaced using the object’s current field states.
- When exiting the **synchronized** region in line 10, the object is still **virtual**. Thus, the monitor exit operation is removed, and the **locked** flag is removed from the object’s state.
- At the **if** statement in line 11, a copy of the current state is created, because it has to be propagated to both successors of this *control split*.

```

1 Object getValue(int idx, Object ref) {
2   Key key = alloc Key;
3   key.idx = idx;
4   key.ref = ref;
5   Key tmp1 = cacheKey;
6   boolean tmp2;
7   synchronized (key) {
8     tmp2 = key.idx == tmp1.idx &&
9           key.ref == tmp1.ref;
10  }
11  if (tmp2) {
12    return cacheValue;
13  } else {
14    cacheKey = key;
15    cacheValue = createValue(...);
16    return cacheValue;
17  }
18 }

```

Listing 5: Example from Listing 4 after inlining.

```

1 Object getValue(int idx, Object ref) {
2   Key tmp = cacheKey;
3   if (idx == tmp.idx && ref ==
4       tmp.ref) {
5     return cacheValue;
6   } else {
7     Key key = alloc Key;
8     key.idx = idx;
9     key.ref = ref;
10    cacheKey = key;
11    cacheValue = createValue(...);
12    return cacheValue;
13  }
14 }

```

Listing 6: Example from Listing 5 after Partial Escape Analysis.

- When continuing at line 12, the object is still **virtual**, and the **return** statement ends the processing of this branch.
- When continuing at line 14, the object is still **virtual**, but the assignment to the static field **cacheKey** lets the object escape. In order for it to escape, it needs to exist, and therefore the object needs to be created and initialized with the current state of its fields at this point. This process is called *materialization* in our system. The object is transitioned to the state **escaped** at this point, and the state of its fields cannot be used from here on since there could be assignments to the fields from outside the compilation scope.
- Lines 15 and 16 do not affect the state of the object anymore.

In effect, the allocation was moved into one branch of the **if** statement. While this did not lead to fewer allocation sites in the resulting code, it reduces the dynamic number of allocations at runtime. The actual reduction depends on the likelihood of the branch containing the allocation being reached, but there will always be at most as many dynamic allocations as in the original code.

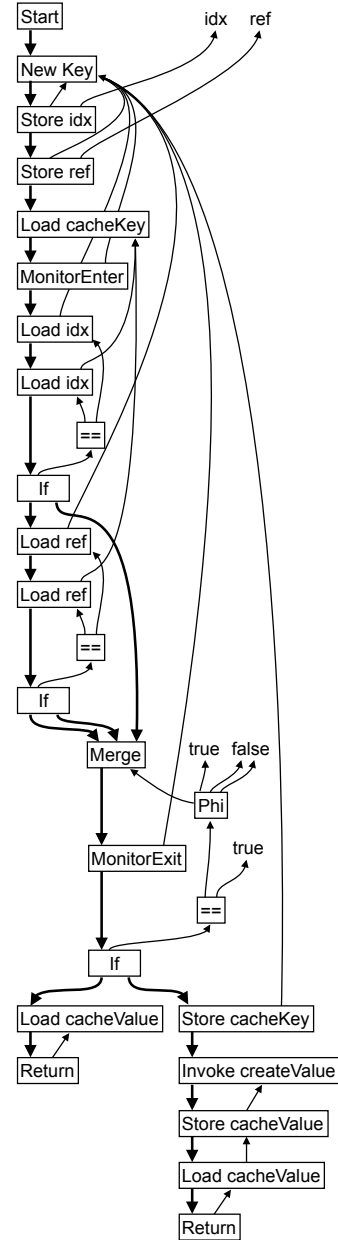


Figure 2: Graal IR of the example in Listing 5 (after inlining).

5. GRAAL PARTIAL ESCAPE ANALYSIS

Partial Escape Analysis is particularly effective if it can interact with other parts of the compiler, such as inlining, global value numbering, and constant folding. In order to do so, it needs to work on the same internal program representation as other optimizations, which, in case of Graal, is the high-level Graal IR [5].

Figure 2 shows the Graal IR² for the example in Listing 5 (after inlining). As the Graal IR is in SSA form, there are no more variables, and the local variable **tmp2** is expressed us-

²The graphical representation is a slight modification of the one used in [5] that omits some aspects of Graal IR that are of no consequence to the algorithms described in this paper.

```

1  class Id extends Node {
2      Class<?> type;
3  }
4  class ObjectState {
5  }
6  class VirtualState extends ObjectState
7  {
8      int lockCount;
9      Node[] fields;
10 }
11 class EscapedState extends ObjectState
12 {
13     Node materializedValue;
14 }
15 class State {
16     Map<Id, ObjectState> states;
17     Map<Node, Id> aliases;
18 }

```

Listing 7: The state that is propagated through the IR.

ing a phi function. Control flow dependencies are expressed as bold arrows pointing downwards, and data flow dependencies as thin arrows pointing upwards.

Graal’s Partial Escape Analysis starts iterating the IR graph at the **Start** node, and processes each node as soon as all its control flow predecessors have been processed. This means that it will follow the control flow, branch at control splits, and process **Merge** nodes as soon as all predecessors have been visited. Iteration stops at *control sinks* such as **Return** and **Throw** nodes.

During this iteration, the system maintains a state that keeps track of previously encountered object allocations. For each node that is visited, the system takes the predecessor state and updates it by any effects of the current node. **Merge** nodes and loop entries are special in that there are multiple predecessor states (from merged branches and loop back edges), which need to be merged into one consistent state before processing the node.

If there was no reason yet to actually create (*materialize*) an allocated object, it is considered to be **virtual**. This implies that the state of all fields and the number of held locks is known and correct. When a previously **virtual** object needs to be created in the heap, an actual allocation needs to be inserted, which is considered to be the *materialized value*.

5.1 Allocation State

Listing 7 shows a simplified version of the allocation state maintained during the control flow iteration. Each object allocation encountered is represented by an **Id** object. For each of these **Ids** there is an **ObjectState** describing the current knowledge about this allocation, stored in the **states** map. If the allocation is still **virtual**, the state is a **VirtualState** representing the field values and the lock count, if an allocation escaped, the state is an **EscapedState** representing the materialized value. Finally, **aliases** contains a mapping from Graal IR nodes to **Ids**. It will initially map from the **New** node of the original program to the allocation’s **Id**, but during further analysis more aliases for the same allocation might be added.

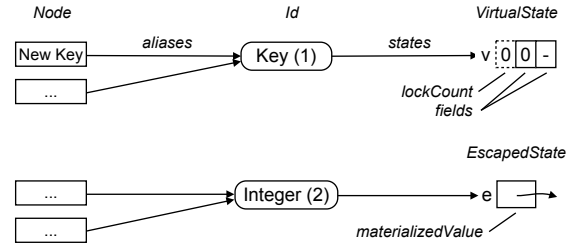


Figure 3: Visualization of the allocation state used in the rest of this paper.

Figure 3 shows a visualization of the state described in the last paragraph. Multiple nodes can be associated with one object **Id**, and one **Id** is always associated with exactly one **ObjectState**. The rounded rectangle representing an object **Id** contains the object’s type and a unique identifier. The **VirtualState** contains the **lockCount** and the values of the object’s fields. The **EscapedState** contains the **materializedValue**, which is a reference to the node that will create the actual object at runtime.

5.2 Effects of Nodes on the Allocation State

While iterating over the control flow, Partial Escape Analysis looks for operations that have an effect on the allocation state. There are three categories of nodes that require some action:

- Allocations create new **virtual** objects, therefore they always modify the state by adding new elements.
- If any of the inputs of a node is a key in the **aliases** map, then the node needs to be examined.
- **Merge** nodes and **LoopBegin** nodes (which represent loop headers) merge multiple states.

Figure 4 shows the node patterns that introduce **virtual** objects or change the state of an existing **virtual** object:

- For each allocation, new **Id** and **VirtualState** objects are created, and the **VirtualState** object is initialized with default values. Also, new entries in the **aliases** and **states** maps are created that point from the allocation to the **Id** and from the **Id** to the **VirtualState**.
- Storing a value which is not in the **aliases** map into a field of a **virtual** object sets the field value in the corresponding **VirtualState** object. Loading a (non-**Id**) value from a field of a **virtual** object replaces the **Load** with the value from the corresponding field of the **VirtualState** at all its usages.
- Entering a synchronized region (**MonitorEnter** node) with the locked object being a **virtual** object increments the **lockCount**, and (d) exiting the synchronized region decrements the **lockCount**.
- Storing a **virtual** object into a field of another **virtual** object puts a reference to the **Id** of the stored object into the **fields** array of the target **virtual** object.
- Loading a **virtual** object from a field of another **virtual** object inserts a new entry into the **aliases** map

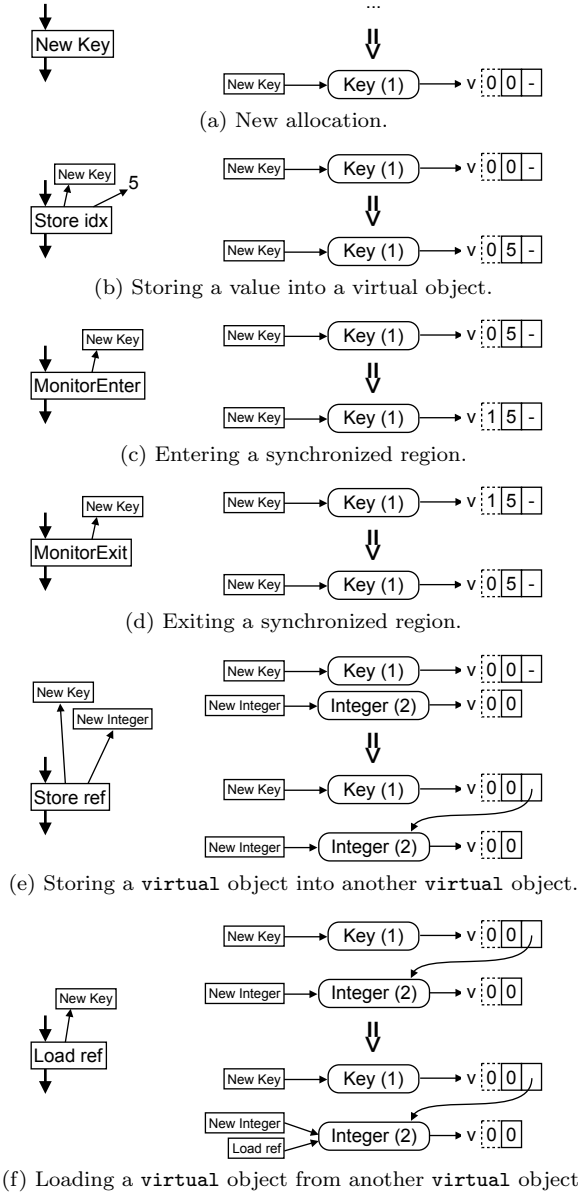


Figure 4: Operations performed on virtual objects.

so that the Load node can be recognized as referring to the virtual Integer during further processing.

All these operations are removed from the IR after they have been processed.

Many other operations can also be replaced with constant expressions based on the precise information the state provides. Equality checks on object references are always **false** when exactly one of the inputs is **virtual**. If both inputs are **virtual**, the check will produce **true** if they refer to the same Id, **false** otherwise. Type checks on **virtual** objects can also be performed at compile time, since the exact type is known.

Figure 5 shows an example of a Store operation where the input is an **escaped** object. In general, inputs that refer to **escaped** objects are handled as if they were normal values,

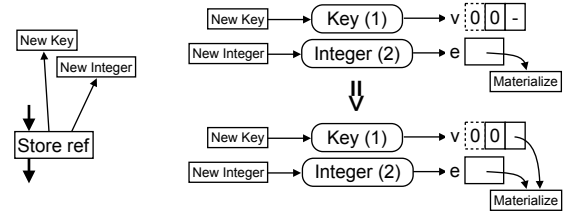


Figure 5: Store operation performed on an escaped object.

but they are replaced with the **materializedValue** during processing.

Any operation that is not explicitly handled is assumed to require an actual object reference. Therefore, any **virtual** object that is referenced from such an operation will be materialized, and the input that maps to the Id of the now **escaped** object is replaced with the materialized value.

5.3 Merge nodes

Whenever multiple branches meet at a **Merge** node, there are also multiple states that need to be merged into one consistent state. A so-called **MergeProcessor** is responsible for doing so, as shown in Figure 6. It first (a) creates the intersection of the **aliases** maps of all merged states, which implies that only Ids that exist in all predecessor states and have at least one common alias will survive the merge.

For each Id, the **MergeProcessor** looks at the Id's **ObjectState** in all predecessor states:

- If the Id escaped in all predecessors states (b), a new **EscapedState** for this Id is added to the merged state, with the **materializedValue** pointing to a newly created Phi function that merges the **materializedValues** of the predecessor states.
- If some predecessors Ids are in the **virtual** state and some in the **escaped** state, then all **virtual** states need to be materialized at the corresponding predecessor in the control flow, and processing continues in the previous case.
- If all predecessors Ids are in the **virtual** state, then the new state of the Id will also be **virtual**, and all field values need to be merged. For each field, the **MergeProcessor** looks at the value of this field in all predecessor **VirtualObjects**:

- If all field values are identical, this value will be the value of the field in the new **VirtualState**. Note that this applies to Ids that represent allocations as well: if all predecessor **VirtualStates** reference the same Id, then so does the new one.
- If some field values differ, the **MergeProcessor** creates a new Phi node for this field. This requires that all field values are actual values available at runtime, so none of them can reference a **virtual** object. To ensure this, the **MergeProcessor** checks each value whether it is an Id, and if so, whether the corresponding **ObjectState** is a **VirtualState**. A **virtual** object needs to be

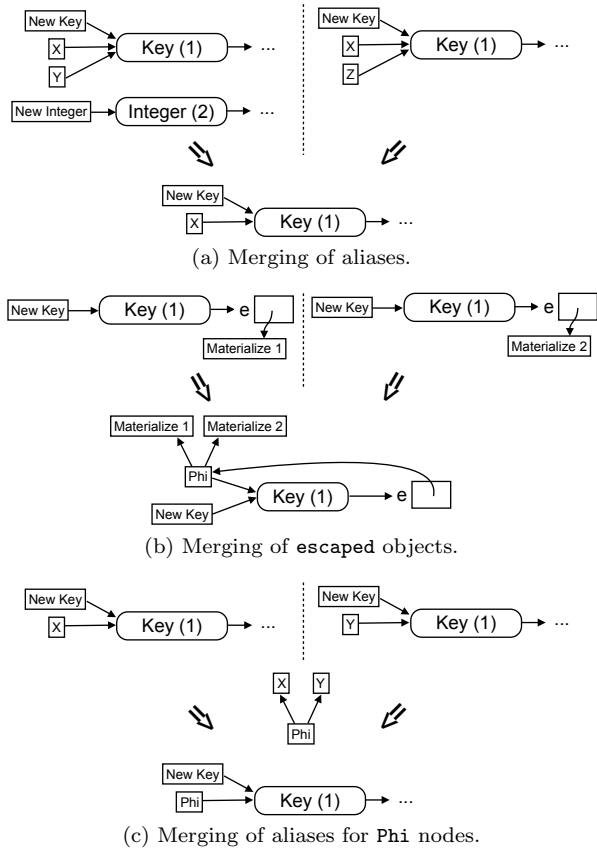


Figure 6: Operations performed by the MergeProcessor.

materialized before it can serve as an input to a Phi node.

The **MergeProcessor** also examines every already existing Phi nodes that is attached to the **Merge** node. It needs to look for inputs of the Phi node that are aliased with Ids:

- If all inputs are aliased to the same Id by their **aliases** maps, the Phi node will be added as an alias of the Id, as shown in Figure 6 (c).
- Otherwise, any input that is aliased with a **virtual** object needs to be materialized, and the input in the Phi is replaced with the materialized value.
- If an input is aliased with an **escaped** object, the input in the Phi is replaced with the **materializedValue**.

During this process of merging states some **virtual** objects might be turned into **escaped** objects, which can invalidate assumptions about which objects are **virtual** taken previously during the merge process. The whole process is therefore iterated until no additional materializations happen during merging, at which point a stable state has been reached.

5.4 Loops

Loops are special in that iteration needs to traverse a loop before its back edges are processed. Graal's Partial Escape Analysis solves this by processing loops iteratively. At the

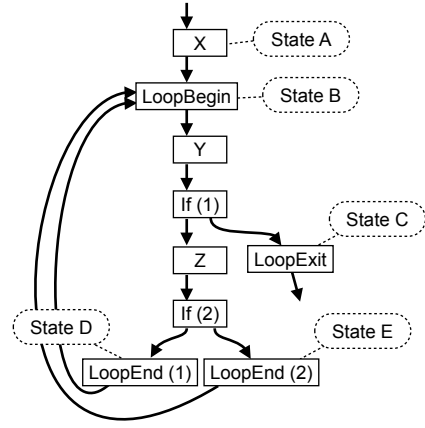


Figure 7: Example loop.

first iteration, the loop body is processed with a *speculative* state, which is taken from the loop's predecessor. Iteration will stop at the loop's back edges and at loop exits. As soon as the loop body has been processed, and the states at all back edges are available, the **MergeProcessor** is used to merge the states of the loop's predecessor and the loop back edges.

The state produced by the **MergeProcessor** is only valid if the speculative start state is correct. Therefore, the new state is compared to the speculative state. If they differ, the new state is used as the speculative start state, and the loop is re-processed. Once the state produced by the **MergeProcessor** equals the speculative state, processing continues at the loop's exits.

Figure 7 shows an example of a loop with one exit and two back edges. When the iteration encounters the loop, only state A is known. In order to be able to start processing the loop, it is assumed that B equals A. As iteration continues processing all other nodes in the loop, it creates the states C, D and E. The **MergeProcessor** merges A, D and E to create a new state B'. If B' equals B, then C is correct and the iteration can continue at **LoopExit**. If B' does not equal B, then B is replaced with B' and the loop is reprocessed.

5.5 Handling Frame States

The HotSpot interpreter cannot work with **virtual** objects. Therefore, all **virtual** objects need to be materialized whenever a deoptimization occurs. The information required to create the objects needs to be added to the **FrameState** nodes that describe the mapping from machine state to Java Virtual Machine state whenever a **virtual** object is referenced by the frame state.

Figure 8 shows two Graal IR fragments with frame states, corresponding to Listing 8. The **FrameState** nodes are expressed as dashed boxes marked with "@"; they contain the method name and the bytecode position. Their inputs describe the local variables and the contents of the expression stack. It is important to note that the expression stacks in this example are empty.

Figure 8 (a) contains a field store which is associated with position 9 in the constructor of **Integer**. At this point there are two local variables: the newly allocated **Integer** object and the value x. The constructor was inlined into the method **foo**, so it has a reference to the outer frame state at

```

1 static Object global;
2 void foo(int x) {
3     Integer i = new Integer(x);
4     global = null;
5     ...
6 }

```

Listing 8: Example shown in Figure 8.

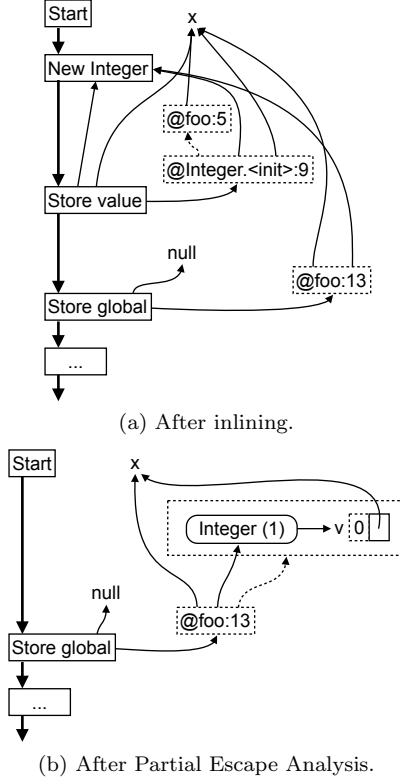


Figure 8: Example from Listing 8 with FrameStates.

position 5 in `foo`. This outer frame state has only one local variable, namely the value `x`.

Figure 8 (b) shows the same Graal IR fragment after applying Partial Escape Analysis. The first field store was removed due to Scalar Replacement, which also removed the associated `FrameState` nodes. The second field store, however, was not removed. Its associated frame state contains a reference to the `virtual` object whose allocation was removed. This reference to the `New Integer` node is replaced with a reference to the `virtual` object’s `Id`. To be able to restore the object during deoptimization, a copy of the current `VirtualState` for the `Id` is added to the frame state.

6. EVALUATION

We evaluated our implementation of Partial Escape Analysis, which is built on top of the Graal compiler, by running and analyzing a number of benchmarks. The original Graal compiler does not perform any kind of Escape Analysis. All benchmarks were executed on server-class Xeon E5-2690 CPUs, with the Java VM configured to use up to 2GB of heap.

Our benchmark process runs each of the 14 benchmarks of the DaCapo suite³ and each of the 12 benchmarks of the ScalaDaCapo suite⁴, warming them up for enough iterations to arrive at a stable peak performance. In addition to that, it runs the SPECjbb2005 benchmark, which also contains a warmup phase.

This process was executed 10 times for a configuration without Partial Escape Analysis and 10 times for a configuration with Partial Escape Analysis. The numbers we report in this paper are the averages of the benchmark results for the 10 runs. In separate runs, we also collected statistics for the size and number of allocations and the number of lock operations for each benchmark.

6.1 Results

Table 1 shows the results of our evaluation. It omits the number of lock operations because few of these numbers are significant. It also omits the DaCapo benchmarks that do not show a significant change in performance. For each benchmark, the table contains allocation per iteration in MB, millions of allocations per iteration, and iterations per minute metrics, split into without and with Partial Escape Analysis, and change. The “average” row shows the average percentage, including (in case of DaCapo) the benchmarks not shown in the table. Since SPECjbb2005’s iterations are much smaller than the ones in DaCapo and ScalaDaCapo, we scaled these numbers by 10^6 . This makes the table more uniform, but does not influence the relative changes.

Allocated Bytes Most benchmarks show a high allocation rate. DaCapo `lusearch`, `sunflow`, `tradesoap` and `xalan`, ScalaDaCapo `factorie` and `kiama`, and SPECjbb2005 allocated more than 1GB per second. Most of ScalaDaCapo, some of DaCapo, and SPECjbb2005 see a large decrease in allocated bytes per benchmark iteration due to Partial Escape Analysis. ScalaDaCapo `factorie` has the highest decrease at 58.5% or 24.5GB per iteration.

Number of Allocations In general, benchmarks with a high number of allocated bytes also show a high number of allocations. The relative decrease in the number of allocations is usually higher than the decrease in the number of allocated bytes, since the allocations not removed by Partial Escape Analysis often contain large arrays.

Number of Locks We did not observe a significant reduction in the number of lock operations in most benchmarks. DaCapo `tomcat` shows a 4% or 155,000 operations per second reduction, and SPECjbb2005 shows a 3.8% or 2,400,000 operations per second reduction.

Iterations per Minute Most of the benchmarks we executed show some improvement in performance, with many being above 10%. ScalaDaCapo `factorie` benefits the most in terms of performance, with a 33% improvement in iterations per minute. Notably, the DaCapo `jython` benchmark shows a 2.1% decrease in performance. Partial Escape Analysis can in rare cases increase the size of compiled methods, which has a negative influence on this benchmark.

³DaCapo version 9.12-bach [1]

⁴ScalaDaCapo version 0.1.0 (2012-02-16) [14]

		MB / Iteration			MAllocs. / Iteration			Iterations / Minute		
		without	with	Δ	without	with	Δ	without	with	Speedup
DaCapo*	fop	172	166	-3.5%	3	3	-5.6%	150.75	172.41	+14.4%
	h2	1,336	1,267	-5.2%	31	30	-5.9%	11.64	11.98	+2.9%
	kython	2,242	2,057	-8.3%	28	23	-15.2%	25.35	24.80	-2.1%
	sunflow	2,707	2,010	-25.7%	62	43	-30.6%	54.55	55.40	+1.6%
	tomcat	691	685	-0.8%	7	7	-2.4%	46.73	48.78	+4.4%
	tradebeans	3,640	3,354	-7.8%	64	57	-11.1%	9.97	10.61	+6.4%
	xalan	1,289	1,270	-1.4%	10	10	-2.2%	156.25	159.15	+1.9%
	average [†]			-4.9%			-8.0%			+2.2%
ScalaDaCapo	actors	1,866	1,550	-17.0%	56	45	-18.5%	17.10	18.81	+10.0%
	apparat	3,418	3,306	-3.3%	74	70	-5.5%	6.11	6.94	+13.7%
	factorie	43,393	17,996	-58.5%	1,397	547	-60.9%	1.95	2.59	+33.0%
	kiana	642	600	-6.6%	13	11	-11.2%	116.28	135.44	+16.5%
	scalac	758	648	-14.5%	19	15	-22.6%	23.09	24.12	+4.4%
	scaladoc	1,189	1,046	-12.0%	24	18	-24.0%	20.39	20.99	+3.0%
	scalap	68	62	-8.8%	2	2	-12.5%	472.44	555.56	+17.6%
	scalareform	337	292	-13.3%	10	8	-16.5%	127.66	137.61	+7.8%
	scalatest	263	261	-1.0%	4	3	-2.4%	58.14	62.24	+7.1%
	scalaxb	226	212	-5.9%	4	3	-13.8%	100.50	105.26	+4.7%
	specs	588	362	-38.4%	12	3	-72.0%	35.03	36.43	+4.0%
	tmt	2,798	2,698	-3.6%	38	34	-12.2%	13.06	13.50	+3.3%
	average			-15.2%			-22.7%			+10.4%
SPECjbb2005 [‡]		11,608	9,741	-16.1%	180	111	-38.1%	11.07	12.04	+8.7%

Table 1: Evaluation of size and number of allocations, and performance on (Scala)DaCapo and SPECjbb2005.

* omitting benchmarks without significant changes in performance (avrora, batik, eclipse, luindex, lusearch, pmd and tradesoap).

[†] including the benchmarks omitted in this table.

[‡] Scaling factor for SPECjbb2005: 10^6 (numbers are per one million iterations).

6.2 Comparison

The HotSpot server compiler, which is arguably the most widely used jit compiler performing Escape Analysis, benefits less from enabling Escape Analysis than Graal does from enabling Partial Escape Analysis (0.9% vs. 2.2% on DaCapo, 7.4% vs. 10.4% on ScalaDaCapo, 5.4% vs. 8.7% on SPECjbb2005). However, it is hard to tell the difference between better Escape Analysis and the rest of the compiler performing better in the presence of Escape Analysis.

7. FUTURE WORK

The iteration that updates the allocation state could be run in parallel as soon as a control split is encountered. More fine-grained multithreading within the compiler will become more important as more cores are available for compilation.

Our algorithm currently relies on the scheduler to order the nodes, so that Partial Escape Analysis can process them in a valid order. By adding simple invariants to the Graal IR, such as limiting the maximum distance from nodes fixed in control flow to nodes affected by Partial Escape Analysis, the analysis could be performed without a schedule.

8. RELATED WORK

8.1 Java

Blanchet [3, 2] extends previous work on Escape Analysis to allow for precise treatment of assignments, and uses the

results of this control-flow-sensitive analysis for Stack Allocation. This work is one of the first to completely support the complete Java language and includes a formal proof of the correctness of their transformations.

Choi et al. [4] present both a control-flow-sensitive and a control-flow-insensitive Escape Analysis for Java. Even the control-flow-sensitive version, which has some similarities to our Partial Escape Analysis, is only used to make global decisions about escapability. In addition to the missing loop handling, it does not collect enough information to perform on-the-fly Scalar Replacement. The control-flow-insensitive version of this work is used to perform Scalar Replacement in the HotSpot server compiler [12] starting with version Java SE 6u23.

Kotzmann and Mössenböck [8, 9] introduce an implementation of Escape Analysis for the HotSpot client compiler that works on the compiler’s high-level intermediate representation (HIR). It uses the control-flow-insensitive *equi-escape sets* algorithm, and was the first to apply Escape Analysis in the presence of deoptimization. A similar algorithm is used by Molnar et al. [10] to perform stack allocation of objects in the Cacao VM.

Shankar et al. [15] present JOLT, a lightweight dynamic analysis which tries to reduce *object churn*, the excessive creation of short-lived objects. It does so by guiding inlining so that Escape Analysis is more effective. Their approach significantly increases the number of allocations amenable to Escape Analysis. Combining it with our approach of Partial Escape Analysis would be an interesting future work.

8.2 Other Languages

The LuaJIT compiler performs a so-called *Allocation Sinking* optimization [13], which is essentially an Escape Analysis tailored heavily towards trace compilation. It relies on Lua's alias analysis to remove all loads from non-escaping objects. The actual Escape Analysis consists of a backwards marking phase with iterative processing of PHI references, and a forward sweeping phase that tags non-marked instructions as removable, or "sunk". While the algorithm still takes a global decision, the trace-based nature improves its efficiency, because unlikely branches will reside in separate traces. An escaping reference in one of these side traces will not cause the object to escape in the main trace.

Current development versions of the v8 JavaScript engine [6] contain an implementation of Escape Analysis which performs a very local analysis (looking only at the usages of the allocation) to determine whether an allocation escapes or not. The simulation of the effects of operations for different allocations happens independently, which implies that complex cases cannot be handled.

9. CONCLUSIONS

In this paper, we presented a new approach to performing Escape Analysis, Scalar Replacement and Lock Elision in a more fine-grained way. Our analysis does not make a global decision about an object's escapability, but propagates the state of all allocations while iterating over control flow. It can thus perform optimizations like Scalar Replacement in one branch while an actual object is created in another branch.

While previous systems perform a control-flow-sensitive analysis step followed by a control-flow-insensitive optimization step, we combine both steps into a single control-flow-sensitive algorithm. This technique is an efficient way to implement Escape Analysis.

We implemented our algorithm for the open-source Graal compiler. In the DaCapo, ScalaDaCapo and SPECjbb2005 benchmarks, this Partial Escape Analysis can reduce memory allocated by up to 58.5% and shows an improvement in performance of up to 33%.

Acknowledgments

We thank all members of the Virtual Machine Research Group at Oracle Labs and the Institute for System Software at the Johannes Kepler University Linz for their support and contributions. The authors from Johannes Kepler University are funded in part by a research grant from Oracle.

10. REFERENCES

- [1] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190. ACM Press, 2006.
- [2] B. Blanchet. Escape analysis for object-oriented languages: application to Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 20–34. ACM Press, 1999.
- [3] B. Blanchet. Escape analysis for JavaTM: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, ACM Press, 2003.
- [4] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–19. ACM Press, 1999.
- [5] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck. Graal IR: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.
- [6] Google. The v8 JavaScript engine, 2013.
- [7] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–43. ACM Press, 1992.
- [8] 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. ACM Press, 2005.
- [9] T. Kotzmann and H. Mössenböck. Run-time support for optimizations based on escape analysis. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 49–60. IEEE Computer Society, 2007.
- [10] P. Molnar, A. Krall, and F. Brandner. Stack allocation of objects in the CACAO virtual machine. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 153–161. ACM Press, 2009.
- [11] OpenJDK Community. *Graal Project*, 2013.
- [12] M. Paleczny, C. Vick, and C. Click. The Java HotSpotTM server compiler. In *Proceedings of the Symposium on Java Virtual Machine Research and Technology*, pages 1–12. USENIX, 2001.
- [13] M. Pall. Allocation sinking optimization for the LuaJIT compiler, 2013.
- [14] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder. Da Capo con Scala: design and analysis of a scala benchmark suite for the java virtual machine. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 657–676. ACM Press, 2011.
- [15] A. Shankar, M. Arnold, and R. Bodik. Jolt: lightweight dynamic analysis and removal of object churn. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 127–142. ACM Press, 2008.