3 Extended Escape Analysis

A standard pattern found in C programs is passing a buffer and its size to a function which will write a computed result into the given buffer. Since the calling function controls the location of the buffer, it can be allocated from stack memory. In Java, a method would instead allocate a new object from heap and return a reference to it to achieve the same. Using alias information and escape analysis, objects that escape their method of allocation into the caller but no further can be automatically identified. Allocating these objects on the caller's stack and passing a reference avoids the need for garbage collection for these objects and can improve worst and average case execution times of programs. In the KESO context, this is called *extended escape analysis* or *(variable) scope extension*. Es ist eher nur Scope Extension

Stack allocation is not the only possible optimization using these results, and may not be the best. Several other approaches such as thread-local heap regions or explicit deallocation operations come to mind. Unbounded usage of the optimizations discussed in the chapter may lead to problems and sub-par performance.

Section 3.1 outlines the general idea of the optimization implemented for this master's thesis and gives and example showing where, why, and how it can be applied. The following Section 3.2 discusses in detail which preconditions must be fulfilled, which constellations hinder or prevent optimization, and how these shortcomings could be addressed. Two different transformations using the analysis results and their advantages and drawbacks are presented in Section 3.3, before Section 3.4 concludes this chapter with a consideration of possible problems caused by excessive usage of extended escape analysis.

Da EEA die Grundlage für viele andere Optimierungen bietet

Da du dich im Folgenden doch stark auf die Stackallokation beziehst, kannst du hier ev. schreiben, dass du das tust.

Listing 3.1:Example containing a candidate for extended escape analysis

```
1  public class Factory {
2    class Builder {
3      // ...
4    }
5    protected Builder getBuilder() {
6      return new Builder();
7    }
8  }
9
10 class Simulation implements Runnable {
11   @Override
12   public void run() {
13     Factory f = new Factory();
14     while (true) {
15       Builder b = f.getBuilder();
16       for (Aircraft a : getAircrafts()) {
17         b.addPosition(a, getPositionForAircraft(a));
18       }
19       SimFrame frame = b.makeFrame(); // last reference of b
20       simulate(frame);
21     }
22   }
23   // ...
24 }
```

Example simplified from the CD_j benchmark from the CD_x family of benchmarks [KHP+09]. The object allocated in *Factory.getBuilder* does not escape *Simulation.run*. It can be allocated on the stack of *Simulation.run*.

3.1 Algorithmic Idea ?

Listing 3.1 shows an example adapted from the source code of the CD_j benchmark [KHP+09] where the technique can be applied. The *Builder* object allocated in *Factory.getBuilder* escapes its allocating method into *Simulation.run*, but is no longer referenced after line 19. It can be allocated in the stack frame of *Simulation.run* and automatically reclaimed after *Simulation.run* returns. A reference to the object would be passed to *Factory.getBuilder* using a new, artificial parameter. The allocation operation in the callee would then be replaced with a parameter read. The invocation of the constructor of *Builder* stays in *Factory.getBuilder*.

If the application can be interrupted between stack allocation in the caller and con-

unglücklich ausgedrückt, zumindestens in KESO kann das nur im Falle der on-demand-garbageCollection passieren
ob der GC hier potentiell laufen kann, ist berechenbar

structor call in the callee (e.g., by stop-the-world or on-demand garbage collection or a blocking method call) the referenced memory area must be in a defined state. Passing a reference to uninitialized memory (like in C) is not possible unless special precautions such as pointer tagging are used.

current

Returning to the running example, the *Builder* object can only be reclaimed by garbage collection without optimization. With optimization, the stack memory can be reused in later iterations of the loop for the same allocation, reducing the number of objects that need garbage collection at runtime.

The examples discussed so far all deal with objects escaping their method of creation via a return operation. Note that being returned is not the only way for an object to escape: storing references in a field of an object given as parameter will also increase the escape state. This case is omitted in all examples for simplicity, but always implied.

3.2 Analysis

extended ~~stack scope~~ region opt.

Any object in the *method* escape state partition of a method's CG is a candidate for optimization. The escape state of the object's representation in the method's callers can be taken into account to decide whether the object should be allocated in the caller. Note that since there might be multiple callers and the optimization could be applied multiple times (moving allocations up multiple levels in the call hierarchy) considering the escape state of the object in the callers' CGs is not always a trivial task. For example, the object might escape further in some of the callers but not in others. Even objects with a *local* escape state might still not be stack allocated due to overlapping liveness regions to avoid unbounded stack growth (see Section 2.1.2 and [Lan12, Sec. 3.3] for details). Virtual method invocations need to be handled with special care to avoid breaking the signature of these methods: all candidates for a virtual method invocation need to share the same signature before and after optimizing. See Section 3.2.2 for a detailed discussion of virtual method invocations in the context of extended escape analysis.

KESO's implementation does not take the escape state of an object node's equivalents in the callers' CGs into account. For each run of the analysis and optimization pass, allocations are propagated at most a single level up in the call hierarchy.

Therefore, running the pass multiple times will increase the maximum scope extension level. Note that is is not necessarily beneficial to run the pass often, since it may lead to undesirable results (see Section 3.4).

3.2.1 Nonvirtual Calls

Nonvirtual call sites, i.e., those where the invoked method is unique and known at compile time, constitute the simple cases of the analysis. The KESO compiler tries to increase the number of non-ambiguous invocations by devirtualizing method invocations where a single candidate can be deduced using static analysis [ESLSP11, Sec. 3.4].

Each object node with a known allocation site (i.e., each non-phantom object node) and an escape state of *method* will be optimized in KESO. Interference information for the allocated objects is not computed. This causes multiple allocations to be moved into calling methods even if they are allocated in mutually exclusive control flow paths. In some examples, this causes a large number of allocations and new method parameters even though only a few are used simultaneously. See Section 6.2 for possible ways to avoid this problem and a discussion of the challenges in solving it.

3.2.2 Virtual Calls

Virtual method invocations further complicate the decision whether to move an allocation into calling methods or not. Because all candidates of a virtual method invocation must share the same signature (i.e., the same parameter and return types), a method cannot be optimized individually without considering other invocations possibly calling this method and these call site's possible callees. Figure 3.1 contains a graphical representation of this problem. Interdependencies between methods cause methods to form up into groups sharing the same signature. Extended escape analysis, however, depends only on the code of the methods in these groups, which is in general unrelated. A single method in such a group could cause the optimization to add a number of arguments to all its invocations, which in turn would require that the same parameters be added to all other candidates for these invocation sites. These parameters would be unused in all other methods and cause overhead at run-

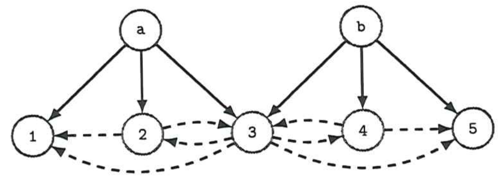

Figure 3.1: Call graph showing the complexity of extended escape allocation for virtual method calls. Green ■ vertices mark methods that contain allocations eligible for scope extensions, blue ■ vertices represent other methods. Solid lines are method invocations. Assume that both *a* and *b* contain a single virtual method invocation each, i.e., the possible callees are *1–3* for *a* and *3–5* for *b*. Dashed lines point from methods eligible for extended escape analysis to methods that must share their signature. Since this relation is transitive, nodes *1* through *5* and their invocation sites must be adjusted for each optimization in *2*, *3*, and *4*.

time as well as the allocation of unused memory.

Because of the overhead and the complexity inherent to applying this optimization correctly in the presence of virtual method calls, KESO does not currently perform scope extension across virtual method calls. Note that some of the challenges are caused by the way the results of alias and escape analysis are used to optimize allocations. Different intermediate code transformations that do not require changing a method's signature could simplify the problem.

For example, instead of using stack memory, a separate thread-local heap section with a simple bump pointer memory management strategy could be used (see Section 3.3.2 where this is discussed). Memory could be allocated in the section corresponding to a calling method in these thread-local heaps during a method's prologue before creating a new method frame. Memory allocated in this way would automatically be reclaimed after the calling method terminates.

A different approach to solve the same problem could be not changing the allocations at all (i.e., allocating objects that escape only a single level in the call graph in heap memory) and modifying the caller to explicitly reclaim the objects that are no longer used. However, in the presence of a garbage collector this does not necessarily reduce the memory management overhead: Marking a section as free does neither reduce the time required for the mark phase (the unreferenced section will not be marked, whether it was explicitly freed or not), nor the sweep phase (the task is the same, it will either by run by the garbage collector, or by the explicit statement). As a consequence, the only possible improvement achieved by explicit deallocations could be a reduction in the number of garbage collector runs. By keeping a list of

Manuelle Speicherverwaltung unterstützen memory areas that can be re-used immediately, the runtime system could avoid a garbage collector run entirely if enough memory can be made available using the known unused areas.

Because KESO's current escape analysis summarizes a method's effect independent of any calling contexts, but both approaches outlined in the last paragraphs depend on the caller, further analyses would have to be implemented to use these ideas.

3.3 Optimization

Based on the results of alias and escape analysis and the decisions presented in Section 3.2, KESO's compiler can apply two new optimizing transformations. The first transformation operates on the intermediate code representation of the compiled program and moves certain allocations into the callers of the method containing the allocation. To preserve soundness, a reference to the allocated object is instead passed as argument to the method previously containing the allocation. The newly created allocation can potentially be served from stack memory subsequently, but Section 3.3.2 also introduces a different concept to avoid potential problems with excessive stack usage.

3.3.1 Extending Variable Scope

Extending the scope of variables constitutes the core part of extended escape analysis. The creation of an object that will be optimized by the transformation consists of two major instructions on the intermediate code level. Since *JINO*'s intermediate code is similar to Java bytecode, these instructions loosely correspond to the bytecode instructions generated by the Java compiler for allocations.

The first of two instructions allocates a new chunk of memory, initializes any internal data expected by the virtual machine (such as runtime type information) and sets the rest of the object's memory to zero to comply with Java's semantics. In Java bytecode, this operation is known as *new*. Note that this differs from the interpretation of the same keyword at the Java language level, which includes the call to the constructor.

The second instruction invokes the object's constructor. The first argument of this call is always a reference to the allocated object. Further arguments are passed, if the constructor has any.

This distinction is important, because the transformation will exclusively deal with the first part. The invocation of the constructor is unaffected and will not be moved since that would increase the complexity and reduce the number of possible optimization spots. Besides the instruction itself, the invocation's arguments would have to be replicated in different methods, which would in turn require copying computations and possibly further method calls while preserving Java call semantics.

Since the invocation of the constructor needs a reference to the allocated object, the allocation instruction is replaced with an operation reading a variable. The variable is a newly created parameter, where the parameter type equals the type of the object. Replacing the *new* operation with an *aaload* operation is simple but may lead to unnecessary copying. However, since *JINO* operates on code in static single assignment (SSA) form at this point, superfluous variable copies will automatically be consolidated in SSA deconstruction using Sreedhar's SSA based coalescing [SJGS99].

← ev. Fussnot weil nicht jeder auf Anhieb weiß ~~..~~ load Variante das i St

Adding the new parameter changed the method signature of the callee. This invalidates all existing invocations. As a consequence, all callers of an optimized method must be adjusted accordingly. This adjustment consists of copying the previously removed allocation instruction right before all invocations and passing the reference returned by this operation as new last argument.

After the pass finishes and all candidates for optimization have been processed, escape analysis is run again. This ensures that alias and escape information for the objects allocated at these new allocation sites is up to date when it is needed in a subsequent pass turning *local* heap allocations into stack allocations (see Chapter 2).

3.3.2 Local Task Heaps

Turning allocations into stack allocations for automatic memory management is not necessarily the best solution, depending on the circumstances. Especially in safety-critical embedded systems allocating objects and arrays on the stack could lead to increased worst-case stack usage estimations. Since the stack space needs to be reserved for each task even if it is not going to be used simultaneously, the overall memory requirement can increase compared to a system without escape analysis.

ev. auch Hinweis auf virtual calls

Keine Anpassung der Signaturen nötig, Verwendung der Referenz auf das Region-objekt

This situation occurs when the sum of upper bounds is larger than the upper bound of the sum. Furthermore, to keep stack usage limited and simplify finding an upper bound of stack usage, KESO does not turn allocations whose liveness regions overlap into stack allocations. Overlapping objects occur because they are allocated inside a loop and alive after the loop. This requires memory proportional to the number of loop runs, where an upper bound might be unknown. To avoid stack overflows, KESO will always serve these allocations from heap memory.

In order to address these shortcomings, an alternative to stack memory is necessary. A special region can be used for all objects that can be automatically managed by the compiler. To provide a runtime advantage over the normal heap, this region must be exempt from garbage collector sweeps. There should be one logical region for each method, while empty regions (from methods without local objects) can be omitted. At the end of the method, its associated region can be reclaimed as a whole. To retain the semantics of stack allocations and reclaim-on-return, these logical regions should be organized in a stack. One possible implementation of these constraints are small specialized heap regions associated with tasks – each task has its own local heap for these regions. The logical regions are implemented in KESO similar to a traditional stack in C: Each task-local heap has a start address, a size, and a fill marker. At method entry, the fill marker is saved and necessary objects are allocated by increasing the fill marker. At method exit, the fill marker is reset to its previous value. Storing the fill marker can be avoided if the amount of memory that will be allocated is known at compile time, since it can be calculated using offsets from the current fill marker. This implementation does not require any synchronization for allocations, which constitutes another advantage over ~~a~~ heap allocations.

Memory shortages can be detected by checking whether the next operation would move the fill marker above the maximum level, preventing unforeseen behavior in case of stack overflows. Since object allocation on stack no longer occurs with this method of region based memory management enabled, finding a tight upper bound for stack usage is simplified. With precise and quick checks preventing task-local heap overflows in place, liveness interference avoidance can be disabled, further reducing garbage collector load (likely exceeding amounts proportional to the number affected allocations due to the use in loops). The necessary size of these local heaps can be statically configured using results from manual worst-case memory usage analysis. Future work (see Section 6.2) could automate this process and determine the size of these local heaps automatically.

3.4 Potential Problems

Applying the optimization to all candidates does not yield a better program in all cases. A number of situations could actually decrease the performance. Heuristics are necessary to avoid these transformations.

For example, suboptimal results are generated for methods that allocate a large number of objects that are eligible for the optimization. A particular specimen exposing this behavior is a generated recursive descent parser used in the CD_j benchmark [KHP+09]: The method that shows the undesirable behavior consists of a large distinction of cases where each case allocates and returns an object. Applying scope extension creates a new parameter for each object and adds the corresponding allocation to all callers. Besides the overhead caused by passing a lot of parameters, this example also exhibits two further problems.

First, since the control flows in the *switch* statement of the optimized method are mutually exclusive, at most a single object is allocated and returned in the example. After extended escape analysis, however, all objects are allocated in the caller methods and references are passed for each one. Only one of the arguments is actually used, though. In this case, the memory usage is thus actually increased by the optimization. This problem could be avoided by consolidating memory areas (and the corresponding method parameters) that are used in mutually exclusive control flows. Interference analysis is needed to determine this information. Good results can probably be achieved using a modification of Sreedhar's ϕ congruence classes [SJGS99], which are already implemented in KESO to remove unnecessary copies of variables in SSA deconstruction, but are not used for this purpose yet. Summarizing so far, scope extension extended escape analysis can increase memory usage due to the allocation of unused objects and it can cause subpar performance when a large number of allocations is optimized because of the increased overhead of the modified method invocation.

The necessary modification of a method's call sites induces another set of potential problems. First and foremost, optimizing a method with more than one call site will increase code size. Because the allocation instruction is removed from the callee and replicated in all callers instead, the optimization is only neutral with respect to the code size if a method only has a single caller.

$$C_{\text{after}} = C_{\text{before}} + (r - a) + c \cdot (a + p) \tag{3.1}$$

$$C_{\text{after}} = C_{\text{before}} + \Theta(1) \quad + \Theta(c) \tag{3.2}$$

Equation (3.1) gives a relation between the code sizes before and after applying the optimization. In the equation, r denotes the code size of a *aload* (read from variable) operation, a is the size of an allocation, p the size of passing an argument to a method and c is the number of callees of the optimized method. As Eq. (3.2) shows, the change in code size is dominated by the number of callers c. Note that the number of objects allocated at runtime does not change even though the number of allocation instructions increases. This is obvious when considering the number of calls to the object's constructor, which is not touched by the transformation and hence stays the same.

The use of appropriate heuristics can prevent the potential problems with methods that have a lot of candidates for the optimization: To avoid allocating memory that is not actually used later, interference analysis can be implemented. The overhead of passing a lot of parameters can be countered by limiting the number of applications of the optimization per method. Code size explosion can be prevented by avoiding the optimization for methods whose number of callers is above a certain threshold. Techniques that do not require adjusting the calling context (see also Section 3.2.2) would completely remove the overhead of argument passing and prevent code size from increasing.

ev. Beschreibung des Algo der EEA implementiert? Bsp CG?

↓

Modifikationen zur einfachen EA