Design and Implementation of an Escape Analysis in the Context of Safety-Critical Embedded Systems

CLEMENS LANG, Friedrich-Alexander University, Germany
ISABELLA STILKERICH, Schaeffler Technologies AG & Co. KG, Germany

The use of a managed, type-safe language such as Standard ML, Ada Ravenscar, or Java in hard real-time and embedded systems offers productivity, safety, and dependability benefits at a reasonable cost. Static software systems, that is systems in which all relevant resource entities such as threads and their priorities, for instance, and the entire source code are known ahead of time, are particularly interesting for the deployment in safety-critical embedded systems: Code verification is rather maintainable in contrast to dynamic systems. Additionally, static analyses can incorporate information from all software and system layers to assist compilers in emitting code that is well suited to an application on a particular hardware device. It was shown in the past that a program composed in type-safe Java in combination with a static system setup can be as efficient as one that is written in C [30], which is still the most widely used language in the embedded domain. Escape analysis (EA) is one of several static-analysis techniques. It supports, for instance, runtime efficiency by enabling automated stack allocation of objects. In addition, Stilkerich et al. [27, 28] have argued that EA enables further applications in safety-critical embedded systems such as the computation of memory classes stated in the *Real-Time Specification for Java* (RTSJ) [6]. EA can be applied to any programming language but the quality of its results greatly benefits from the properties of a type-safe language. Notably, embedded multicore devices can positively be affected by the use of EA. Thus, we explore an ahead-of-time (AOT) escape analysis in the context of the KESO JVM featuring a Java AOT compiler targeting (deeply) embedded (hard) real-time systems.

CCS Concepts: • **Computer systems organization → Embedded systems**; **Real-time systems**; *Reliability*; • **Software and its engineering → Compilers**; **Runtime environments**;

Additional Key Words and Phrases: Escape analysis, KESO, JVM, memory management, regional memory

ACM Reference format:
Clemens Lang and Isabella Stilkerich. 2020. Design and Implementation of an Escape Analysis in the Context of Safety-Critical Embedded Systems. *ACM Trans. Embed. Comput. Syst.* 19, 1, Article 6 (February 2020), 20 pages.
https://doi.org/10.1145/3372133

This work was funded within the project ARAMiS II by the German Federal Ministry for Education and Research withthe funding ID 01IS16025 and by the German Research Foundation (DFG) within the AORTA project with the funding ID SCHR 603/9-1. The responsibility for the content remains with the authors.
Authors' addresses: C. Lang, Friedrich-Alexander University Erlangen Nuremberg, Department of Computer Science 4 (Distributed Systems and Operating Systems), Martensstr. 1, 91058 Erlangen, Germany; email: clemens.lang@fau.de; I. Stilkerich, Schaeffler Technologies AG & Co. KG, Industriestr. 1-3, 91074 Herzogenaurach, Germany; email: isabella@stilkerich.eu.

1539-9087/2020/02-ART6 $15.00
https://doi.org/10.1145/3372133

1 INTRODUCTION

Java being a type-safe programming language provides a series of advantages such as memory safety [1]. The language is the foundation for comprehensive program analyses and runtime-system support. In this article, we investigate the design and implementation of EA which is often employed in commodity systems to perform, for example, stack allocation of objects. In the context of embedded systems, escape analysis provides even more interesting optimization opportunities.

Contribution. To make use of its optimizations, we implemented ahead-of-time, whole-program escape analysis based on the work of Choi et al. [11]. We were able to improve the algorithm in several ways and evaluated our version extensively. Our contribution is as follows:

(1) Fix of a conceptual flaw in Choi's algorithm.
(2) Reducing compile time by connection-graph compression.
(3) Improving runtime by limiting read-value propagation.
(4) Evaluation of escape analysis on top of the comprehensive real-time Java benchmark *Collision Detector*.

Overview. This article is structured as follows: Section 2 explains the relevant aspects of the RTSJ's memory model and the embedded KESO JVM, in which we implemented and evaluated our approach. Moreover, we give an overview over the application of escape analysis. We recap the outlines of Choi's original algorithm which are relevant to understand our modifications in Section 3. In Section 4, we present a fix for a conceptual flaw in the original algorithm which we term *double-return bug*. Afterwards, we introduce further interprocedural improvements. We evaluate the outcomes of our work in Section 5 on the open-source real-time Java benchmark *Collision Detector* (CD_x). The article wraps up with a discussion on related work in Section 6 and the conclusion in Section 7.

2 BACKGROUND INFORMATION

In the following, we take a look at memory management in embedded systems and introduce the memory-management model described in the RTSJ. Subsequently, we present our KESO framework [30] in which we implemented an ahead-of-time escape analysis to provide an automated solution to the RTSJ's memory management. At last, we give a short overview of escape analysis.

2.1 Manual Memory Management in Embedded Systems

Memory management is a typical example for one building block of the runtime system's resource handling. In the domain of embedded systems, memory management is usually performed manually, that is, the programmer decides over the appropriate data placement: On the one hand, the compiler may generate code to automatically handle memory given that it can determine the variable's scope. On the other hand, a memory service may provide data structures and programming abstractions to handle free and used memory.

While this manual approach has the potential to construct resource-efficient and predictable applications, the downside is that it is also prone to programming errors such as placing data in an unsuitable memory location or falsely releasing memory for variables that are still being used.

To address the problems of manual memory management, safer programming systems such as the safe C dialect *Cyclone* [18] or the *Real-Time Specification for Java (RTSJ)* [6] have been developed. In these programming language subsets, the programmer still manages particular memory regions themselves, however, rules are stated and have to be abided to ensure correct memory handling. Using the lexical (static) scopes provided, memory management is still pretty challenging and rises with the application's complexity. The developer has to deal with the proper placement

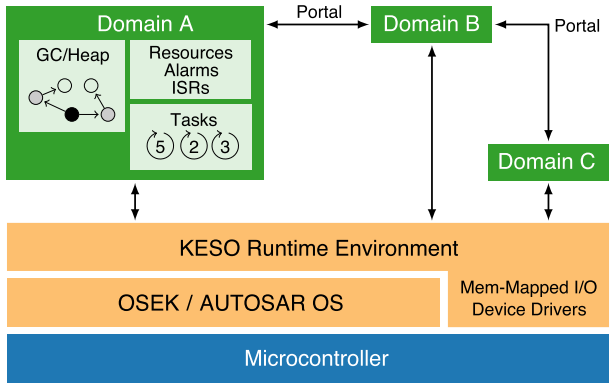


Fig. 1. The architecture of the KESO JVM.

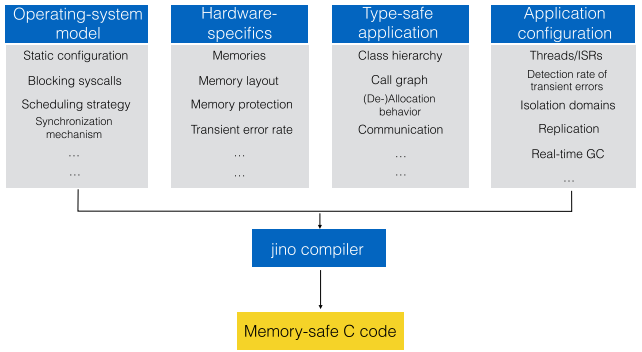


Fig. 2. The KESO compiler *jino*.

of data and the determination of the memory areas' sizes. Additional analyses (not described, for instance, in the RTSJ) are necessary to ensure correct memory management. To address this issue, we implemented an escape-analysis-based method to compute the memory regions in the context of the KESO JVM.

2.2 The KESO Runtime Environment

KESO (see Figure 1) is a JVM for statically configured embedded systems and runs on top of AUTOSAR OS [2]. In such systems, all relevant entities of the (type-safe) application as well as the system software are known ahead of time. Among others, these entities comprise the entire code base of the application, and operating-system objects such as threads (also called *tasks* in the context of AUTOSAR OS) and locks. Disallowing the application to dynamically load new code or to create threads at runtime enables the creation of a slim and efficient runtime environment for Java applications in (deeply[1]) embedded systems. After a number of different optimization passes such as method inlining, KESO's ahead-of-time compiler *jino* (see Figure 2) generates ANSI C code from the application's Java bytecode, plus a slim, tailored runtime environment. By means of the type-safe

[1]On the low end, an example for such a platform is the Atmel 8-bit AVR architecture, a line of microprocessors whose derivates scale very fine-grained in the cost/resources tradeoff. The smallest derivate that we have so far run KESO on is the ATmega8535 device that is equipped with 8KiB of Flash ROM and 544 bytes internal SRAM. Resource-awareness was therefore a crucial factor in many of the design decisions that we took. For more demanding applications, KESO can also be used on more powerful 32-bit platforms such as Infineon's Tricore architecture.

Table 1. Rules for `MemoryAreas` as Stated in the RTSJ

	Reference to Heap	Reference to Immortal	Reference to Scoped
Heap	Yes	Yes	No
Immortal	Yes	Yes	No
Scoped	Yes	Yes	Yes, if same, outer, or shared scope
Local Var	Yes	Yes	Yes, if same, outer, or shared scope

programming language and the separation of all global data, applications located in protection realms called *domains* are isolated from each other without the need for hardware-based memory protection such as a memory-protection unit (MPU). If an MPU is available on the microcontroller, analyses on the type-safe (i.e., memory-safe) KESO code can be used to automatically group data and code by means of reachability analyses, thereby supporting embedded developers in the non-trivial memory-mapping job. Memory safety and memory management are closely intertwined: To ensure memory safety, that is, the soundness of the type system, and consequently being able to establish domains, semantic program analyses are applied on the type-safe program. These analyses enable automated, memory-safe compile-time and runtime memory management. In contrast to programs written in weakly typed languages, static pointer analyses on type-safe code deliver more accurate lifetime information for objects, which is beneficial for the RTSJ's memory model described in the next section.

2.3 Memory Management in the Real-Time Specification for Java

The RTSJ respects the distinct nature of applications by providing several memory partitions called `MemoryAreas` (see Table 1) that can be handled by choice of proper candidates out of the pool of available storage-control strategies. The specified `MemoryAreas` constitute RTSJ's memory model and are defined as follows:

`ScopedMemory` defines a region that exists for a certain time span, that is as long as there are threads that access objects residing in `ScopedMemory`. Lifetimes are determined by lexical scoping. `LTMemory` and `VTMemory` are specialized variants, where `LTMemory` guarantees linear allocation times and `VTMemory` variable allocation times, respectively.

`HeapMemory` is used for objects with an undefined lifetime. Only one single instance of `HeapMemory` can exist.

`ImmortalMemory` is alive for the duration of the program's lifetime. This memory area is similar to static memory reserved ahead-of-runtime.

`Local Var` is used for the local variables, respectively.

The RTSJ states that data can be stored in these memory areas. Recall, that the developer of an application has to deal with the proper placement of data and the determination of the memory areas' sizes. Additional measures (not described in the specification) are therefore necessary to ensure correct memory management.

2.4 Implementation of the RTSJ's Memory Model in KESO

KESO adopts the concept of logical memory areas specified as by the RTSJ via ahead-of-time escape analysis, that is, all objects are assigned to memory regions before runtime by analyzing the entire program. Our algorithm is based on the work of Choi et al. [11]. Their algorithm is applied in

the context of just-in-time compilation and thus has to be modified for our purposes.[2] With our implementation in KESO, we significantly improve compilation and runtimes for the analyzed program. Moreover, we provide a fix for a bug present in the original algorithm. We will describe in this article how these issues are addressed by our approach. To guide the reader, we will start with a coarse presentation of the topic of escape analysis.

2.5 Escape Analysis

Escape analysis is a special lifetime analysis that determines the dynamic scope of pointers. Thus, this static analysis can be employed to find out if an object can safely be allocated on the stack or in an extended scope. Additionally, EA can be used to find out if synchronization operations between threads are needed to preserve data consistency. Escape analysis has been of particular interest in managed languages such as ML and Java as manual stack allocation is not possible: Being a safe programming language, Java cannot rely on the programmer to deallocate unused objects as this would have the potential to break the soundness of the type system. As an example, allocating an object that lives longer than its method of creation on the stack of that method will lead to dangling references. From a conceptual point of view, all data is therefore allocated in heap memory. However, due to the language's strong type system, it is possible for the JVM's compiler to automatically and precisely categorize objects in terms of their lifetime: The information collected by alias analysis and the computation of the references' reachability can be leveraged to determine if an object *escapes* a method, that is, if its lifetime exceeds that of the scope in which it was created. As a consequence, non-escaping objects can be allocated on the stack and are not subject to the overhead entailed by heap management. Most of the research regarding escape analysis [3, 5, 10, 11, 17, 24] has been performed in the context of safe languages to facilitate compiler-guided stack allocation, although the concept of escape analysis can also be applied to unsafe languages such as (unsubsetted) C. However, the quality of results depends on the programming language being analyzed and the type of compilation being used. The analysis results are less accurate in unsafe languages due to the weaker type system, which requires conservative assumptions to preserve soundness. When using dynamic compilation (that is, just-in-time (JIT) compilation), there will always be a tradeoff between compilation time and the accuracy of the escape analysis' results, while static, whole-program ahead-of-time (AOT) compilation can produce more precise escape information at the expense of slower compilation times.

Ahead-of-Time Escape Analysis in Safety-Critical Embedded Systems. Automated stack allocation via escape analysis implicates a series of advantages such as efficient and time-predictable (de-) allocation, lock elision and reducing the overhead of incremental garbage collectors. Especially in the context of deeply embedded (safety-critical) systems, however, the information collected by escape analysis offers a lot more interesting optimization opportunities as shown by us [27, 28]:

(1) Efficient remote-procedure calls for isolated programs
(2) Extended stack scopes
(3) Thread-local heaps and regional memory
(4) Support for the machine-independent space and time bounds analyses of memory management
(5) Efficient mitigation of transient hardware errors
(6) Automated inference of immutable data
(7) Automated object inlining

[2]It should be noted, that the entire approach is specifically interesting for embedded systems, as it is used for memory management in static embedded systems. In desktop systems, dynamic programs are more relevant and they may benefit more from a dynamic memory model.

To benefit from these applications, it is necessary to implement an escape analysis. Due to our inquiry of established EAs (see Section 6), we found a suitable one to implement in our ahead-of-time compiler *jino* and selected the method proposed by [11]: Choi's EA is on the one hand suitable for the managed and imperative language Java and on the other hand it computes escape information, which is more accurate than other solutions for Java such as Blanchet's approach [4] at the expense of slower compilation times. As KESO was designed for static embedded (real-time) systems that do not need dynamic code loading and as KESO features an ahead-of-time Java compiler that is based on the closed-world assumption, the preciseness of escape analysis is more important than compilation times. Therefore, we decided to evolve Choi's method for safety-critical embedded systems.

3 ANALYZING THE BASE ALGORITHM

In the following, we explain how the algorithm as published by Choi et al. works. For a detailed discussion of the original escape analysis algorithm, please refer to [11].

The algorithm starts with alias analysis, which consists of method-local (also intraprocedural) and global (interprocedural) steps. To compute and store alias information, a specialized data structure called *connection graph (CG)* is used. For each analyzed method, this graph contains representations of local variables, static class members, dynamic instance variables, array indices, and objects. Variables of non-reference type are ignored because they do not contribute to alias information.

Intraprocedural Analysis. In intraprocedural analysis, a representation of the method's effects in the form of a CG is computed by iterating over the control flow graph and the statements in the basic blocks in the control flow graph. It is a key contribution of Choi et al. that this representation is independent of the calling context. Since their allocation sites might be unknown for some objects (e.g., if they have been passed as argument), a special type of placeholder called *phantom node* is used to represent these objects. For pointer analysis, summarizing a method's effect independently of aliasing relationships in the calling context is impossible [8, 9, 11, 14, 16, 20, 32]. For each allocation, assignment, field or array access, return statement, method invocation, and exception throw, the CG is modified appropriately, ensuring possible alias relations are represented accurately.

A CG is a directed graph $CG = (N, E)$ where N is the set of nodes and E the set of edges. There are two major categories of nodes: N_o is the set of object nodes, which represent objects. N_r is the set of reference nodes, which represents all references to objects in the program. References to non-objects do not affect escape analysis and are ignored. There are several subtypes of reference nodes:

- *local reference nodes* represent local variables referencing objects,
- *actual reference nodes* represent method parameters, invocation arguments and return values,
- *field reference nodes* are created for object members of reference type, and finally
- *global reference nodes* are the CG representation of static members, that is, global variables.

Note that object nodes in the CG may represent multiple objects in memory of the program at runtime. This happens, for example, if an allocation is inside a loop.

The set of edges E in the CG also has multiple subtypes. Edges from a field reference node to an object node are called *points-to* edges. Formally, $p \rightarrow q$ is a *points-to* edge if $p \in N_r$ and $q \in N_o$. Edges from object nodes to reference nodes only exist to field reference nodes, so $o \rightarrow f$ with $o \in N_o$ and $f \in N_r$ means that the object o has a field f. Finally, the CG allows edges between

reference nodes to simplify processing assignments. Such edges $p \rightarrow q$ with $p, q \in N_r$ are called *deferred* edges.

A deferred edge $p \rightarrow q$ can be removed from the graph by replacing it with edges to the pointees of q. So, for each deferred edge $q \rightarrow r \in N_r$, add a new deferred edge $p \rightarrow r$. For each points-to edge $q \rightarrow o \in N_o$, add a new points-to edge $p \rightarrow o$. If q has no pointees, create a phantom object node $n \in N_o$ and add a points-to edge $p \rightarrow n$, then proceed as above. This operation is called *ByPass(q)*.

Each node in the CG is associated with one of three *escape states*. These states indicate whether a node in the graph (i) does not outlive the method of its creation (*NoEscape*), (ii) outlives the method of its creation (*ArgEscape*), or (iii) outlives the thread of its creation (*GlobalEscape*). Nodes that are reachable from method arguments or returned from a method are marked *ArgEscape*. Nodes reachable from objects and references assigned to *global reference nodes* or thrown as exceptions are marked *GlobalEscape*. The escape state propagates along edges where *ArgEscape* overrules *NoEscape* and *GlobalEscape* overrules *ArgEscape*.

After escape analysis, object nodes with an escape state of *NoEscape* correspond to allocations that can use stack memory. All other nodes constitute the *non-local subgraph* of a method's CG, i.e., the partition of the graph that represents a method's effect on its callers.

To construct a method's CG, its control flow graph is processed in-order, iterating over cycles until the CG converges. If convergence does not occur after a limited number of cycles, the algorithm conservatively assumes that all objects in this method's CG are *GlobalEscape*. In practice, we did not encounter this situation. The CG at the entry of a basic block can be computed from the union of the CGs of its predecessors, where escape states of common nodes are computed using the precedence rules outlined above.

Starting with the graph at the entry to a block, the graph at the end of the block can be computed by applying the *data flow transfer function* to each statement in the basic block in order. This function has four non-trivial cases:

(a) `p = new O()`. Create a new object node for O. Apply *ByPass(p)* and add a new points-to edge from p to the object node.
(b) `p = q`. Apply *ByPass(p)* and create a new deferred edge $p \rightarrow q$.
(c) `p.f = q` (or `p[f] = q`). If no object nodes are reachable from p via points-to and deferred edges only, create a new phantom object node and add a points-to edge from p. Then, for each of these reachable object nodes $o \in O$, ensure a field reference node for f exists and add the field reference edge $o \rightarrow f$. Finally, add deferred edges $f \rightarrow q$ for all such f.
    If $p.f$ is a global variable, set q to *GlobalEscape*.
(d) `p = q.f` (or `p = q[f]`). Apply *ByPass(p)*. Then, as in the previous case, if q has no pointees, create a phantom node and add a points-to edge from q. Again, find (or create if missing) all field reference nodes $f \in F$ below all objects reachable via points-to and deferred edges from q. Finish by adding deferred edges $p \rightarrow f \; \forall f \in F$.

Interprocedural Analysis. In addition to the cases discussed so far, which cover processing the control flow graph inside a method, the following three cases are required for completeness: (i) Creating the CG to be used as input for the initial basic block at method entry, (ii) computing the summary information for a method at the end of the last basic block, and (iii) processing method invocations within a basic block.

At method entry, process an assignment in the form of `f = a` for each parameter of reference type (including the implicit `this` parameter), where f is a local reference node for the method's parameter and a an actual reference node that represents the passed value. Since Java has call-by-value semantics for parameters of reference types, the local variable f can be changed later.

In contrast, a serves as an immutable representation of the parameter value on invocation. Actual reference nodes are initialized with an escape state of *ArgEscape*.

At method exit, return values are processed as assignments to an actual reference node representing the return value. Then, *ByPass* is used to remove all deferred edges. This creates phantom nodes for reference nodes that do not yet point to an object node. Unlike in intraprocedural analysis, Choi et al. retain the points-to edge from such leaf reference nodes to the newly created phantom nodes, which would usually be removed by *ByPass*. At this point, the partition of the CG that is reachable from a node with *ArgEscape* or *GlobalEscape* escape state forms the so-called *non-local subgraph* and represents the method's effect on its callers.

Immediately before method invocations, consider arguments of reference type passed to the callee to be assignments to actual reference nodes, i.e., process an assignment `â = p` where $\hat{a}$ is the actual reference node on the caller side that will later be correlated with the actual reference node in the callee CG created on method entry and p is the argument passed for this parameter.

Immediately after method invocations, the effect of the called method needs to be mapped back into the caller. To do this, find mappings between nodes in the callee CG and the caller CG starting at the actual reference nodes a_i representing the method parameters on the callee side and their corresponding actual reference nodes $\hat{a}_i$ created for the given arguments at the invocation site in the caller's CG. We can recursively define the mapping $\mapsto$ from this base case:

$$
\begin{aligned}
a_i &\mapsto \hat{a}_i \\
o \in \text{PointsTo}(p) &\mapsto \hat{o} \in \text{PointsTo}(\hat{p}) && \text{if } p \mapsto \hat{p} \\
o.f &\mapsto \hat{o}.g && \text{if } o \mapsto \hat{o} \wedge f \hat{=} g
\end{aligned}
$$

where PointsTo(p) denotes the set of all object nodes reachable via deferred or points-to edges from p. Choi et al. proposed the `UpdateNodes` procedure to compute this mapping for all object nodes. This function creates phantom nodes for object nodes that are present in the callee but missing in the caller's CG. Additionally, it ensures all field reference nodes used in the callee's summary CG are present in the caller. See [11, Section 4.4.1] for a formal notation of `UpdateNodes`.

As the final step of processing a method invocation, edges in the callee's CG that are not present in the caller need to be propagated. Choi et al. add edges $\hat{o} \rightarrow \hat{f} \rightarrow \hat{g}$ in the caller for constructs of the form $o \rightarrow f \rightarrow g$ in the callee CG where:

$$
\begin{aligned}
& o, g, \hat{o}, \hat{g} \in N_o \\
& q \mapsto \hat{q} \ \wedge \ p \mapsto \hat{p} \\
& f \hat{=} \hat{f} \quad \text{i.e., the fields are equivalent}
\end{aligned}
$$

This effectively copies the effects of the non-local subgraph of the callee into the caller's CG, which ensures that objects that escape due to a statement in the callee are correctly marked as escaping in the caller.

Example. See Listing 1 for source code corresponding to the CGs to be explained in the following paragraphs. The example is a simple generic linked list. Using common sense we can deduce that, in the absence of a removal operation, all list elements will be reachable until the list itself has reached the end of its lifetime. Consequently, the only allocation in the given example cannot be allocated on the stack, because it must outlive the method of its allocation `addElement`. Due to the structure of the example, intraprocedural analysis will not suffice to determine this.

See Figure 3(a) for the summary CG of `insert`. Analysis of `insert` starts at method entry by creating both a local and an actual reference node for the `this` and `elem` parameters and adding a deferred edge from the local reference node to the actual reference node. The actual reference

```
1   public class LinkedList<T> {
2    private static final class ListElement<U> {
3      ListElement<U> next;
4      U              val;
5
6      public ListElement(U newval) {
7        this.val = newval;
8      }
9    }
10
11   private ListElement<T> head;
12
13   public void addElement(T value) {
14     insert(this, new ListElement<>(value));
15   }
16
17   private static <V> void insert(LinkedList<V> list,
18                                  ListElement<V> elem) {
19     elem.next = list.head;
20     list.head = elem; // make elem the first entry
21   }
22  }
```

Listing 1. A simple generic linked list implementation in Java. Note that this example is – for demonstration purposes – more complex than it would have to be, especially due to the `insert` method. An inner class is used to wrap the list entries with references to their successor. The `addElement` method allows insertion of new entries. Internally, `addElement` uses insert, which enqueues the given new element at the start of a list.

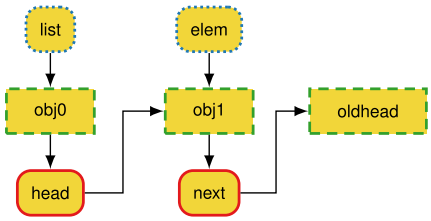
nodes are set to *ArgEscape*. Since the local reference nodes have *NoEscape* escape state they are not part of the non-local subgraph and not shown in the summary information.

We then continue processing the only basic block in `insert` according to our data flow transfer function as outlined above. The first statement `elem.next = list.head` is broken down into two instructions by a Java compiler. We thus first consider `t1 = list.head` to read the list head and then discuss writing `elem.next = t1` where t_1 is a temporary. For the read operation, apply the `p = q.f` case of the data flow transfer function. To do this, we would apply *ByPass* to our temporary. Since it does not have incoming or outgoing edges at this point, this is a no-op. We then create a new phantom node `obj0` since `list` does not yet point to an object node and add the points-to edge `list` → `obj0`. Next, create the field reference node `head` and add the field edge `obj0` → `head`. For the second part of the statement `elem.next = t1`, we use the `p.f = q` case. No object nodes are reachable from `elem`, so a new phantom node `obj1` is added. The new field reference node `next` is created as a successor of `obj1` and a deferred edge `next` → `head` represents the assignment. For the last statement `list.head = elem`, the same case applies, except in this case *ByPass(*`head`*)* is not a no-op and will create a new phantom node `oldhead` to represent the previous pointees of `head` before redirecting its incoming deferred edge from `next` to it. Then, a deferred edge `head` → `elem` is added, concluding the processing for this basic block.

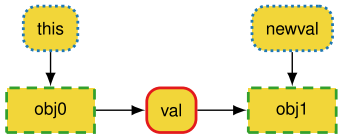
At method exit of `insert`, *ByPass* removes all remaining deferred edges in the graph. The non-local subgraph in this state is shown in Figure 3(a).

The same analysis steps are run for the constructor of the `LinkedList` element. Its non-local subgraph of the connection graph is given in Figure 3(b).

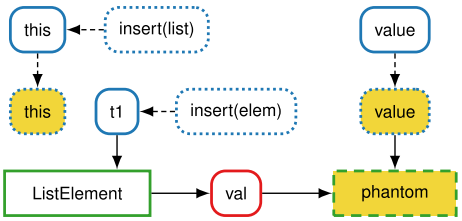
For `addElement`, analysis again starts at method entry and creates both a local and an actual reference node for the two parameters `this` and `value`. The first instruction in the only basic block is the allocation of a new `ListElement` object, whose result is assigned to a temporary `t1`. This is case `p = new O()` in the data flow transfer function, which causes the nodes `t1` and `ListElement`



(a) Summary connection graph for `insert`.



(b) Summary connection graph for the `ListElement` constructor.



(c) Connection graph for `addElement` before invoking `insert`.

Fig. 3. The summary connection graphs for `insert` and `ListElement`'s constructor, and the connection graph for `addElement` immediately before invoking `insert`. See Listing 1 for the corresponding source code. Vertices with rounded corners represent reference nodes, where field reference nodes have a red, other reference nodes a blue border. Dotted borders mark actual reference nodes representing a method's parameter or return value. Rectangles with green borders are object nodes. If the border is dashed, the node is a phantom node. The escape state of nodes is encoded in the fill color. White, yellow, and red represent *NoEscape*, *ArgEscape*, and *GlobalEscape*, respectively. (a) illustrates the connection graph for `insert`. The edge from head to `obj1` is the one that prevents stack allocation of the list element in `addElement`. Interprocedural analysis needs to propagate this edge into the caller context to determine this. (b) depicts the connection graph for `ListElement`'s constructor, which adds the edge from the list element to its value. (c) shows the connection graph for `addElement` before invoking `insert`. `insert(list)` and `insert(elem)` represent the parameters passed to the `insert`.

to be added to the CG. We do not discuss the constructor invocation of the list element in detail in this example. Its processing adds the `val` field below `ListElement`, creates a phantom node to represent the pointees of `value` and adds the points-to edge `val` → `phantom`. See Figure 3(c) for the connection graph in this state.

The final statement in `addElement`'s basic block is the invocation of `insert`. Right before its invocation, new actual reference nodes are added for the arguments passed to insert by processing an assignment according to the data flow transfer function. For `insert`'s two parameters, this results in the `insert(list)` and `insert(elem)` reference nodes and their outgoing deferred edges. Starting from these nodes and their counterparts in the CG for `insert` given in Figure 3(a), the

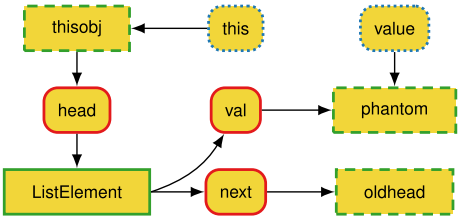


Fig. 4. The non-local subgraph of the CG for `addElement` as given in Listing 1. Colors and shapes cf. Figure 3.

algorithm finds mappings from callee nodes to caller nodes and creates missing field reference nodes. Since `obj0.head` from `insert` is not represented in the caller CG, a new phantom node `thisobj` with this field will be created below `this`. `obj1` is mapped to `ListElement` and `obj1`'s `next` field is created as a successor of `ListElement`. Finally, `next`'s pointee `oldhead` causes the creation of a phantom node in `addElement` so its incoming edge can be represented in the caller.

After creating all necessary nodes, missing edges are propagated from the callee to the caller. There are two chains of the structure $o \rightarrow f \rightarrow g;\ o, g \in N_o$ in `insert`'s CG: `obj0` → `head` →`obj1` and `obj1` → `next` → `oldhead`. Using the mapping discovered in the last step, these edges are added to the caller's CG as `thisobj` → `head` → `ListElement` and `ListElement` → `next` → `oldhead`. The first of these chains causes the *ArgEscape* escape state of the `this` actual reference node to spread to `ListElement`, which makes the analysis sound.

To conclude analysis of `addElement`, the steps for method exit are applied, i.e., *ByPass* is applied to remove all remaining deferred edges. See Figure 4 for the non-local subgraph of `addElement` at method exit.

Optimization. To obtain results for a program, the steps outlined above must be run for every method. Due to the propagation of information from callees to callers, iteration in a bottom-up manner through the program call graph is advisable. Recursion causes strongly connected components in the call graph, which can be solved by iterating until a fixpoint is reached or assuming every node in there methods has an escape state of *GlobalEscape*. KESO's implementation uses Tarjan's algorithm to identify strongly connected components and a topological order among them [31].

After analysis, nodes marked *NoEscape* in the CG can be allocated on the stack. Note that KESO does not convert all allocations that fulfill this criterion into stack allocations. Instead, variable liveness information is used to compute whether multiple objects allocated at the same allocation site are needed at the same time. Objects with overlapping liveness regions are not allocated on the stack. Optimizing such allocations would require finding an upper boundary of objects in use at the same time or resizing the method's stack frame at runtime, which KESO avoids deliberately for predictability reasons.

4 ADAPTION OF GENERAL-PURPOSE ESCAPE ANALYSIS FOR STATIC EMBEDDED JAVA

This section describes several improvements we applied to the escape analysis proposed by Choi et al. First of all, we discuss a couple of findings regarding the creation of phantom nodes from our experience. Afterwards, we present a conceptual flaw in the original analysis and a way to fix this issue. To conclude, we show how interprocedural analysis can be modified to enhance both ahead-of-time escape analysis and the runtime behavior. With respect to the CD_j benchmark used in the evaluation, we were able to improve the execution time of that program by up to 9.5%.

4.1 Avoiding Phantom Nodes

In our implementation, we have found that avoiding the creation of phantom nodes where they are not necessary is important to keep analysis runtimes low. In our experience, a main contributing factor to the creation of phantom nodes is the `UpdateNodes` procedure used in interprocedural analysis to handle method invocations. Given pairs of actual reference nodes describing parameters on the callee side and arguments on the caller side, it recursively computes equivalences between object nodes in the callee's CG and object nodes representing them in the caller's CG. `UpdateNodes` will create new phantom nodes in the caller's CG if no equivalence for the pointees of a reference node in the callee's CG can be found.

KESO's escape analysis delays creating these phantom nodes until it has no other option to discover new equivalences. This is achieved by converting the `UpdateNodes` algorithm as given in [11, Figure 7] to a worklist-based approach. The worklist is initialized with the base cases of equivalence, the pairs for actual reference nodes on caller and calle side. When the creation of a phantom node would become necessary, it is delayed by enqueing the currently processed pair in a second `needsPhantomNode` worklist. Only when the first worklist is empty, a phantom node is created for an equivalence pair in `needsPhantomNode` and the pair is re-added to the worklist. See [21, Section 3.2.1] for a detailed explanation of this problem.

Additionally, we do not create phantom nodes for nodes with incoming deferred edges but no outgoing edges on method exit while applying *ByPass*. Instead, incoming deferred edges for such leaf nodes are left unmodified. This change requires propagation of *GlobalEscape* escape states to callers for reference nodes instead of just propagating this for object nodes.

4.2 Fixing the Double Return Bug

KESO's implementation of escape analysis produced incorrect results given inputs similar to those generated by stack scope extension outlined in [28, Section 3.2]. Further analysis suggests this is a conceptual flaw in the work of Choi et al. See Listing 2 for an example triggering this bug. The `get` method allocates two objects and passes them to `choose`, which selects one of them at random and returns it. The return value of `choose` is then returned from `get`. Because either of the two objects allocated in `get` might escape, both allocations must not use stack memory.

However, the CG constructed according to [11, Section 4] does not correctly identify the two objects as method-escaping. The connection graph for `choose` is given in Figure 5(a). The two phantom nodes `obj0` and `obj1` are created at method exit while applying *ByPass* to remove the remaining deferred edges from the return value `ret` to `a` and `b`.

```
1  public class ChooseOne implements Runnable {
2    public void run() {
3      Object a = get();
4    }
5    private static Object get() {
6      return choose(new Object(), new Object()); // Bug
7    }
8    private static Object choose(Object a, Object b) {
9      if (Math.random() < 0.5)
10       return a;
11     return b;
12   }
13 }
```

Listing 2. Example triggering the double return bug.

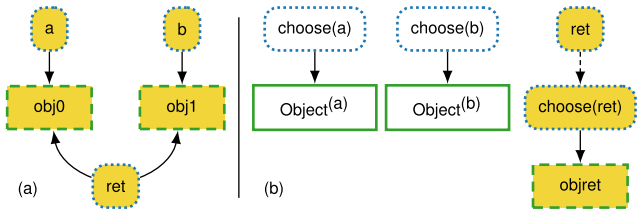


Fig. 5. The connection graphs according to Choi et al. for the methods (a) choose at method exit and (b) get right before method exit from Listing 2. Rounded dotted rectangles denote actual reference nodes. Rectangles indicate object nodes, whereas dashed rectangles indicate phantom nodes. *NoEscape* elements are white and *ArgEscape* ones are yellow.

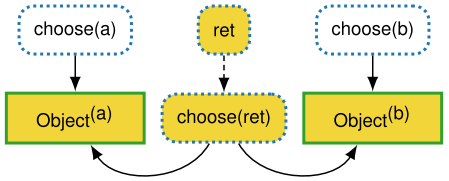


Fig. 6. The CG for get right after the invocation of choose as generated by KESO's modified algorithm to fix the double return flaw. The two object nodes are correctly marked method-escaping. Colors and shapes cf. Figure 3.

For the CG of get, consider Figure 5(b): Immediately after the invocation of choose in get, equivalence pairs between callee and caller CG are computed using the UpdateNodes procedure. This also causes the creation of objret to represent the pointees of ret in the CG for choose, because choose(ret) does not have any pointees at this point in the analysis. Then, the algorithm proceeds to propagate edges in the form of $o \rightarrow f \rightarrow g$ where o, g are object nodes and f are field reference nodes to the caller [11, Section 4.4.2]. It is simple to see that the CG for choose does not contain any field reference nodes, and no edges are propagated. This causes edges from choose(ret) to Object$^{(a)}$ and Object$^{(b)}$ to be missing from the graph of get. This becomes a problem when the return value of choose escapes get.

Two changes were necessary in KESO's implementation to work around this problem. First, when processing the CG at method exit *ByPass* is used to remove deferred edges. Contrary to Choi et al., we do not create phantom nodes for reference nodes with incoming deferred edges that do not yet point to an object node, but instead retain the deferred edges.

Second, instead of propagating $o \rightarrow f \rightarrow g$ edges only, KESO's alias analysis propagates all points-to edges $p \rightarrow o$ from reference nodes to object nodes into the equivalent nodes in the caller's CG, creating $\hat{p} \rightarrow \hat{o}$ for all $p \mapsto \hat{p}$ and $o \mapsto \hat{o}$. As an exception to this rule, outgoing edges of actual reference nodes representing a method parameter are not propagated. This is due to the fact that Java has call-by-value semantics, which means that the argument given to an invocation will always remain unchanged.

Using these changes as well as the measures to avoid spurious phantom nodes outlined in Section 4.1, KESO's implementation arrives at the CG given in Figure 6. The points-to edges from the actual reference node representing the return value to the phantom nodes representing the a and b parameters have been added to the CG of get. This propagates the *ArgEscape* escape state from ret through choose(ret) to both objects and fixes the bug.

This problem did occur because the summary information of a callee was only partially propagated into the caller, which caused the caller's CG to be unsound. By changing the propagation

rule from chains of the form $o \rightarrow f \rightarrow g$ to propagating all edges of the form $p \rightarrow o$ where p is not an actual reference node (which represent the value of parameters at method entry, are immutable and can thus not be changed), the entire non-local subgraph of a callee is represented in the caller, making the analysis sound again.

4.3 Interprocedural Analysis Optimizations

Some of the larger applications (up to 28.3 kSLOC[3]) used in testing the KESO compiler took up to 19 minutes to compile with alias and escape analysis enabled. The compile times were dominated by the duration of alias analysis. To reduce this unacceptable overhead, a series of possible culprits were identified and modifications to the algorithm where implemented in order to improve compile times. Since the vast majority of the time was spent in interprocedural analysis, all optimizations described in the following sections apply to this part of alias analysis.

No Propagation of Read Operations. Analyzing the generated CGs revealed that virtual invocations of methods which in turn call the same set of virtual methods caused the size of the graphs to increase rapidly. This situation commonly occurs in Java with simultaneous use of the `equals` method and collections (whose `equals` implementations call `equals` once for each element in the collection). Since calling `equals` usually does not change any references reachable from its parameters, it does not add new aliases. Based on this observation, the intraprocedural analysis was extended to track all edges that were added to the CG due to a write operation. KESO's implementation uses a set of properties called `isWritten` and `isWriteOperand` available in each connection graph node to store this, because information cannot be easily attached to the edges themselves in KESO's adjacency list-based implementation of the CG. After intraprocedural analysis, a modified version of Tarjan's algorithm to find strongly connected components [31] finds all cycle-free paths from the method's formal parameters to edges created by write operations. All edges that compose this subgraph are called *important* and marked for later use. Note that the subgraph may contain cycles because while *important* edges alone will not cause cycles, an additional edge created by a write operation might. Furthermore, interprocedural analysis was extended to ignore all nodes and edges that have no role in a write operation and are not marked important (i.e., are not on a path from the method's entry points to a write operation edge).

In theory, these changes should have removed the effect of calls to `equals`, `hashCode` and similar methods completely. In practice, however, some implementations of `equals` may in fact contain write operations: The `java.util.Hashtable` class from the GNU classpath project, for example, implements `equals` by comparing the entry sets of the two hash tables. This entry set is lazily created and cached inside the hash table class. This write operation causes all edges leading up to it to be marked *important*. These edges are then propagated into all other invocations of `equals`, causing further edges to be considered *important*, nullifying the effect of the optimization for `equals`. Other implementations and functions might, however, still benefit from the improvement, and this is in fact the case for the CD_j benchmark used in the evaluation where the data gathered in this modification causes an allocation in a hot spot of the application to be optimized. If Java did have constant methods like C++ does, `equals` (and other methods that are marked constant and only have constant reference parameters) could be automatically ignored in alias analysis.

Connection-Graph Compression. While the improvements in the last paragraphs reduced the compile time of large applications, the savings were still not enough to achieve acceptable analysis times. Huge connection graphs mostly consisting of phantom nodes were created due to interprocedural analysis–often, these phantom nodes had siblings that would represent the same

[3]Generated using David A. Wheeler's *SLOCCount*.

objects. To reduce the size of the connection graphs, a graph compression transformation inspired by Steensgaard's almost linear time points-to analysis [26] was implemented.

Starting at each actual reference node representing a formal parameter or return value, the graph compression algorithm processes each reference node recursively but avoids loops using a color bit. For each reference node, lists of pointees segregated by escape state are collected. The separation into different escape states ensures that object nodes are only unified with nodes that have the same escape state. This avoids deterioration of the computed results up to this point. Each list that contains at least two object nodes, at least one of which must be a phantom node, is compressed by removing all phantom nodes. Note that any two non-phantom object nodes (i.e., any two nodes with a known allocation site) are not consolidated to preserve the one-to-one mapping between intermediate code allocation instruction and its CG representation.

Incoming edges pointing to the phantom nodes to be removed are redirected to the retained object nodes. Field reference nodes reachable from the phantom nodes are re-created below the object nodes in the compression set. Edges outgoing from the removed field reference nodes are moved to their equivalents below the retained object nodes. This might create new graph constellations eligible for compression, so the color bit of any successor field reference nodes is reset.

Since these modifications always preserve non-phantom object nodes and do not unify subgraphs with different escape states, the effect on the results is negligible.[4] However, the compile time required for alias analysis improved by an order of magnitude (see Section 5.4).

5 EVALUATION

To illustrate the effectiveness of KESO's escape analysis, we measure runtime and heap usage of selected KESO configurations. For this, we employ the real-time Collision Detector (CD_x) benchmark, which is available in a C (CD_c) and a Java (CD_j) version. For KESO, we use CD_j in the *on-the-go*-frame variant, deployed on the Infineon TriCore TC1796 device (150-MHz CPU clock, 75-MHz system clock, 1-MiB external SRAM, 2-MiB internal flash). The application is translated to ANSI-C code using KESO (version 4072). Method inlining, constant propagation, and dead code removal are enabled in all measured configurations. The generated code is bundled with an AUTOSAR OS implementation and compiled with GCC (version 4.5.2). For a detailed comparison, please refer to [22].

5.1 The CD_x Benchmark

The core of the CD_x application is a periodic thread that detects potential aircraft collisions from simulated radar frames. A collision is assumed whenever the distance between two aircraft is below a configured proximity radius. The detection is performed in two stages: In the first stage (reducer phase), suspected collisions are identified in the 2D space ignoring the z-coordinate (altitude) to reduce the complexity for the second stage (detector phase), in which a full 3D collision detection is performed (detected collisions). A detailed description of the benchmark is available in a separate paper [19]. Since CD_j allocates temporary objects and uses collection classes of the Java library, it requires the use of dynamic memory management. We selected a stop-the-world GC available in KESO. It should be noted that the Java benchmark selection in the area of embedded real-time systems is very restricted. We believe that the evaluation of escape analysis is best performed by choosing an established benchmark rather than our own programs we created for KESO on small embedded devices. A comparison of CD_x variants using different memory-management mechanisms is available [29]. In this earlier publication, we compared several garbage-collection techniques against the baseline escape analysis (that is, the original algorithm without any of the

[4]Precision can be affected in the presence of recursion if edges propagating a non-local escape state are added in later iterations over the strongly connected component in the call graph formed by recursion.

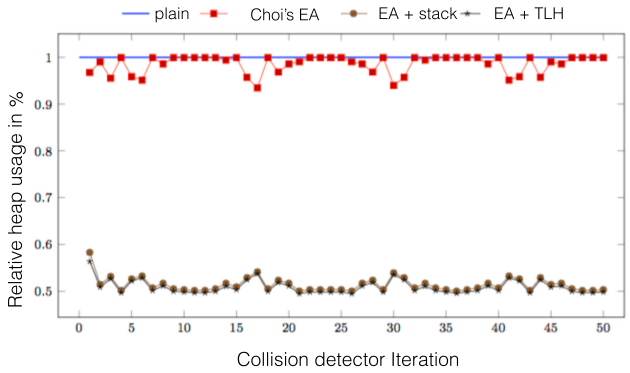


Fig. 7. Heap memory usage of the *on-the-go variant* of the CD_j benchmark with (a) Choi's escape analysis and stack allocation, (b) escape analysis and stack allocation (EA+stack), and (c) escape analysis and task-local heaps (EA+TLH) relative to a run without escape analysis-based optimizations (plain).

modifications described in this article). With the algorithm we presented in this article, we can further improve runtime efficiency and heap usage as shown in the following sections.

5.2 Region Inference via Escape Analysis

Due to the modifications we applied to the algorithm of Choi et al. [9–11], we are able to implement region inference via escape analysis at reasonable compile-time costs. One variation of regional memory we use in the following evaluation are *task-local heaps (TLH)* [13, 23]. Details on the implementation of region inference via our modified escape algorithm can be found in a separate publication [27]. In this way, we are able to provide an automated solution to `ScopedMemory` as specified in the *Real-Time Specification for Java (RTSJ)* [6].

5.3 Effectiveness of KESO's Escape Analysis

In the context of this evaluation, we focus on two possible back ends for escape analysis, that is *stack allocation* and *task-local heaps*. We have a look on the CD_j's heap usage and execution time while using Choi's algorithm and our modified version containing the interprocedural analyses improvements presented in Section 4.3. In all evaluated versions, the double return bug was fixed. The data always shows the average of five runs. The standard deviation of the measurement values was always lower than 0.06% for time measurements and equal to 0 for heap memory usage. Figure 7 graphs the relative heap memory usage of the collision detector CD_j after escape analysis. The median heap usage for escape analysis with the stack allocation optimization back end is only 50.7% relative to a run without optimizations based on escape analysis. When using task-local heaps instead of stack allocation, the median heap usage drops to 50.1% due to the added optimizations of allocations that create objects with overlapping liveness regions. Other than expected, the impact of those allocations is small, even though they can be executed multiple times because they are in loops. Compared to the state of escape analysis before the interprocedural improvements heap memory usage has been massively improved from the previous median usage of 99.6% of the baseline.

For runtime measurements, the CD_j benchmark internally reads values from a high resolution timer before and after a collision detector run. The difference (i.e., the duration) is stored in a global array and printed after the simulation completes. Again, escape analysis shows significant improvements: Figure 8 contains the execution times of three configurations relative to the baseline without optimizations based on escape analysis. In Choi's baseline algorithm, the median

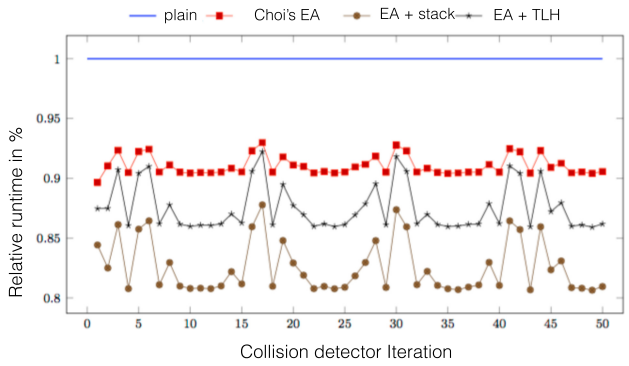


Fig. 8. Runtime of the *on-the-go* variant of the CD_j benchmark using (a) Choi's escape analysis with stack allocation, (b) KESO's escape analysis with stack allocation (EA+stack), and (c) escape analysis with task-local heaps (EA+TLH) relative to a run without escape analysis-based optimizations. Times are measured in the application by reading from a high-resolution timer before and after each collision detector run. The difference is computed and shown.

runtime was 90.5% of the reference with a standard deviation of 0.79 percentage points. Due to the changes implemented, the same configuration with escape analysis and stack allocation now performs significantly better with a median of 81.1%. We traced the improved runtime behavior back to an allocation in a hot spot of the benchmark, which was optimized after we implemented avoiding the propagation of read operations as explained in Section 4.3.

As the graph shows, some of the iterations have previously executed slower causing spikes in the graph. The impact of the spikes increases, which explains the higher standard deviation of 2.17 percentage points. As expected due to the additional instructions managing regions in task-local heaps on method entry and exit, stack allocation is faster than the code generated by the task-local heap allocation back end. The median runtime improvement for task-local heaps is 13.7% compared to 18.7% for stack allocation.

5.4 Effectiveness of CG Compression

In Section 4.3, we introduced connection graph compression to reduce compile times. We measured compile times of the *on-the-go* variant of the CD_j benchmark on Debian Stretch with OpenJDK 8u171-b11-1~deb9u1. The benchmark machine was a VirtualBox VM with 1024 MB of RAM and a single CPU. We measured time spent in the escape analysis before and after the commit implementing connection graph compression. The average compile time dropped from 16224ms with a standard deviation of 1341ms to 1140ms with a standard deviation of 113ms, a reduction of almost 93%.

6 RELATED WORK

General escape analysis is an application of abstract interpretation [12] to higher-order programming languages. A simple version of escape analysis, which was able to deal with complex data types and first-order functions was implemented in LISP compilers in the 1970s. Goldberg and Park [17] devised a more accurate higher-order escape analysis for an elementary language called *NML (not much of a language)*, which has a monomorphic type system. In a follow-up [24], the authors extended their approach to programs manipulating lists to allow cells of list spines to either be reclaimed promptly or reused without the overhead of garbage collection.

Blanchet [3] combines the work of Goldberg, Park, and Deutsch. He extended the correctness proof for escape analysis in the context of the functional and managed ML programming language and also presents implementation results. In contrast to Goldberg and Park, Blanchet extends

escape analysis to all recursive types (not only lists). Support for imperative operations and higher-order functions is included. In addition, escape analysis and method inlining is combined. As a consequence, the analysis complexity increases but inlining also promotes stack allocation, which can have a positive influence on the program's runtime behavior.

Park, Goldberg, and Blanchet assume that a value escapes when its value is stored in another value. This approach is sufficient for functional languages but inappropriate for imperative languages. In contrast to ML, Java is an imperative language and thus uses assignments, which complicates precise escape analysis considerably. Besides other research groups, Gay and Steensgard [15] came across this problem when they applied an escape analysis that followed the same procedure as applied in ML in the context of Java: They state that an object escapes as soon as the reference to it is stored in another object, which makes their analysis imprecise.

To address these shortcomings, Blanchet's later work [4, 5] applied escape analysis in the context of the object-oriented Java on a commodity system using Linux, JDK 1.1.5, and just-in-time compilation. Blanchet's implementation delivers much more accurate results in contrast to previous work and also measures the effects of lock elision and stack allocation in contrast to a heap managed by garbage collection. Blanchet's escape analysis allows us to shift stack-allocatable objects to the scope of method with which those objects are associated. In benchmark programs, the author demonstrates that his escape analysis is able to stack-allocate a great amount of data, which results in a considerable execution time speedup. The improvements are traced back to lock elision and the decrease of garbage collection, whereas his implementation in ML benefits from better data locality [7] due to a different garbage collection approach. Blanchet uses special integer values called *type heights* to encode one object's references to other objects and its subtyping relationships to other objects. His escape analysis is control-flow-insensitive and consists of a forward and a backward phase to compute escape information to facilitate lock elision and stack allocation. Blanchet's analysis targets just-in-time compilers and focuses on faster translation times.

Choi et al. [10, 11] implement an alternative, control-flow-sensitive escape analysis for Java. Their algorithm is predicated on connection graphs that resemble alias and points-to graphs. However, connection graphs are constructed independent of the calling context, which poses a major improvement over alias and points-to graphs: Connection graphs can easier be summarized since the escape state does not have to be recomputed in case the method is called from a different context. It is unfeasible for pointer analysis to summarize a method's effect independently of aliasing relationships in the calling context [8, 9, 11, 14, 16, 20, 32], which makes the connection graph representation attractive. In contrast to Blanchet's method, the analysis of Choi et al. is slower in terms of compilation times, however, their method delivers much more accurate escape information for objects.

Since KESO's alias and escape analyses are based on the work of Choi et al. [11], behavior, results, and features of the analyses are similar. However, unlike the work of Choi et al., *jino* avoids resizing a method's stack frame at runtime and offers thread-local heaps as an alternative optimization back end to stack allocation. Section 4.3 presents an alias-analysis modification that considerably reduces compile times for large specimen by merging sibling nodes. This compression technique is inspired by ideas from Steensgaard's almost-linear-time points-to analysis [26]. Unlike Steensgaard's work, KESO's analysis does not necessarily compress all sibling nodes pointed to by a common ancestor, but only merges nodes with the same escape state to avoid deteriorating the quality of escape-analysis results. Object nodes that represent an allocation site are not compressed either to preserve the one-to-one mapping between allocation instructions in the intermediate code and their corresponding object nodes in the connection graphs.

Stadler et al. [25] use a flow-sensitive escape analysis to compute branch-specific escape information. This prevents code in branches that are unlikely to be taken at runtime from impeding

optimizations in other parts of the same method. This can improve the average runtime and heap memory usage significantly but does not affect the worst case unless a branch is not going to be used. In KESO's target domain of embedded hard real-time systems, reductions in worst-case heap memory usage and runtime are very desirable. Nonetheless, Stadler's algorithm would be a potential improvement and a topic for future work.

7 CONCLUSION AND FUTURE WORK

During the implementation of EA, we found and corrected the *double-return bug* present in Choi's algorithm. In addition, we were able to significantly improve the compile-time and runtime behavior by avoiding the propagation of read operations and compressing the connection graph. We evaluated our enhancements in the context of the real-time CD_x benchmark by choosing the stack allocation and task-local heap back end of KESO's escape analysis. For future work, we plan to improve our analyses to make better use of hardware features of embedded multicore devices as they are becoming more relevant in the context of safety-critical control applications such as the ones that can be found in electrical-drive systems. We would like to deploy such an application having been optimized by escape analysis on the embedded multicore Infineon AURIX TC277 device. Particularly interesting is the use of escape-analysis results to optimize data placement and memory management with the joint usage of software-based isolation provided by KESO and hardware-based memory protection provided by TC277.

REFERENCES

[1] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. 2006. Deconstructing process isolation. In *Proceedings of the 2006 Symposium on Memory System Performance and Correctness (MSPC'06)*. 1–10. DOI: https://doi.org/10.1145/1178597.1178599

[2] AUTOSAR. 2010. *Specification of Operating System (Version 4.1.0)*. Technical Report. Automotive Open System Architecture GbR.

[3] Bruno Blanchet. 1998. Escape analysis: Correctness proof, implementation and experimental results. In *Proceedings of the 25th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'98)*. 25–37. DOI: https://doi.org/10.1145/268946.268949

[4] Bruno Blanchet. 1999. Escape analysis for object-oriented languages: Application to Java. *ACM SIGPLAN Notices* 34, 10 (1999), 20–34.

[5] Bruno Blanchet. 2003. Escape analysis for Java: Theory and practice. *ACM Trans. Program. Lang. Syst.* 25, 6 (Nov. 2003), 713–775. DOI: https://doi.org/10.1145/945885.945886

[6] Greg Bollella, Benjamin Brosgol, James Gosling, Peter Dibble, Steve Furr, and Mark Turnbull. 2000. *The Real-Time Specification for Java* (1st ed.).

[7] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. 1994. Compiler optimizations for improving data locality. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*. ACM, New York, 252–262. DOI: https://doi.org/10.1145/195473.195557

[8] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. 1999. Relevant context inference. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*. ACM, New York, 133–146. DOI: https://doi.org/10.1145/292540.292554

[9] Jong-Deok Choi, Michael Burke, and Paul Carini. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93)*. ACM, New York, 232–245. DOI: https://doi.org/10.1145/158511.158639

[10] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape analysis for Java. In *Proceedings of the 14th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*. ACM, New York, 1–19. DOI: https://doi.org/10.1145/320384.320386

[11] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. 2003. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Trans. Program. Lang. Syst.* 25, 6 (Nov. 2003), 876–910. DOI: https://doi.org/10.1145/945885.945892

[12] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium*

on Principles of Programming Languages (POPL'77). ACM, New York, 238–252. DOI: https://doi.org/10.1145/512950.512973

[13] Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. 2002. Thread-local heaps for Java. In *Proceedings of the 3rd International Symposium on Memory Management (ISMM'02)*. ACM, New York, 76–87. DOI: https://doi.org/10.1145/512429.512439

[14] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI'94)*. ACM, New York, 242–256. DOI: https://doi.org/10.1145/178243.178264

[15] David Gay and Bjarne Steensgaard. 1998. *Stack Allocating Objects in Java (Extended Abstract)*. Technical Report. University of California, Berkeley.

[16] Rakesh Ghiya and Laurie J. Hendren. 1996. Connection analysis: A practical interprocedural heap analysis for C. *Int. J. Parallel Program.* 24, 6 (Dec. 1996), 547–578.

[17] Benjamin Goldberg and Young Gil Park. 1990. Higher order escape analysis: Optimizing stack allocation in functional program implementations. In *Proceedings of the 3rd European Symposium on Programming (ESOP'90)*, Neil D. Jones (ed.), Vol. 432. 152–160.

[18] Jim Trevor, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A safe dialect of C. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference (ATEC'02)*. 275–288.

[19] Tomas Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek, Ben Titzer, and Jan Vitek. 2009. CD_x: A family of real-time Java benchmarks. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'09)*. 41–50. DOI: https://doi.org/10.1145/1620405.1620412

[20] William Landi and Barbara G. Ryder. 1992. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI'92)*. ACM, New York, 235–248. DOI: https://doi.org/10.1145/143095.143137

[21] Clemens Lang. 2012. *Improved Stack Allocation using Escape Analysis in the KESO Multi-JVM* (Bachelor Thesis). Friedrich-Alexander University Erlangen-Nuremberg, Germany.

[22] Clemens Lang. 2014. *Compiler-assisted memory management using escape analysis in the KESO JVM* (Master Thesis). Friedrich-Alexander University Erlangen-Nuremberg, Germany.

[23] Kyungwoo Lee, Xing Fang, and Samuel P. Midkiff. 2007. Practical escape analyses: How good are they?. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE'07)*. ACM, New York, 180–190. DOI: https://doi.org/10.1145/1254810.1254836

[24] Young Gil Park and Benjamin Goldberg. 1992. Escape analysis on lists. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference (PLDI'92)*. 116–127.

[25] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial escape analysis and scalar replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'14)*. ACM, New York, Article 165, 10 pages. DOI: https://doi.org/10.1145/2544137.2544157

[26] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*. ACM, New York, 32–41. DOI: https://doi.org/10.1145/237721.237727

[27] Isabella Stilkerich, Clemens Lang, Christoph Erhardt, Christian Bay, and Michael Stilkerich. 2017. The perfect getaway: Using escape analysis in embedded real-time systems. *ACM Transactions on Embedded Computing Systems*16, Article 99 (2017), 99:1–99:30. Issue 4. DOI: https://doi.org/10.1145/3035542

[28] Isabella Stilkerich, Clemens Lang, Christoph Erhardt, and Michael Stilkerich. 2015. A practical getaway: Applications of escape analysis in embedded real-time systems. In *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2015 CD-ROM (LCTES'15)*. ACM, New York, Article 4, 11 pages. DOI: https://doi.org/10.1145/2670529.2754961

[29] Isabella Stilkerich, Michael Strotz, Christoph Erhardt, and Michael Stilkerich. 2014. RT-LAGC: Fragmentation-tolerant real-time memory management revisited. In *Proceedings of the 12th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'14)*. 87–96. DOI: https://doi.org/10.1145/2661020.2661031

[30] Michael Stilkerich, Isabella Thomm, Christian Wawersich, and Wolfgang Schröder-Preikschat. 2012. Tailor-made JVMs for statically configured embedded systems. *Concurrency and Computation: Practice and Experience* 24, 8 (2012), 789–812. DOI: https://doi.org/10.1002/cpe.1755

[31] Robert Tarjan. 1972. Depth first search and linear graph algorithms. *SIAM J. Comput.* (1972), 146–160.

[32] Robert P. Wilson and Monica S. Lam. 1995. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI'95)*. ACM, New York, 1–12. DOI: https://doi.org/10.1145/207110.207111

Received January 2019; revised July 2019; accepted November 2019