FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

Clemens Lang

Compiler-Assisted Memory Management Using Escape Analysis in the KESO JVM

Masterarbeit im Fach Informatik

30. Juni 2014

Please cite as:
Clemens Lang, “Compiler-Assisted Memory Management Using Escape Analysis in the KESO JVM”, Master’s Thesis, University of Erlangen, Dept. of Computer Science, Jun 2014.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Verteilte Systeme und Betriebssysteme
Martensstr. 1 · 91058 Erlangen · Germany

Compiler-Assisted Memory Management Using Escape Analysis in the KESO JVM

Masterarbeit im Fach Informatik

von **Clemens Lang**
geboren am 9. August 1988 in Lichtenfels

Lehrstuhl für Informatik 4: Verteilte Systeme und Betriebssysteme
Friedrich-Alexander Universität Erlangen-Nürnberg

Betreut durch
Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat
Dipl.-Inf. Isabella Stilkerich
Dipl.-Inf. Christoph Erhardt

Beginn: 1. Januar 2014
Abgabe: 30. Juni 2014

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 30. Juni 2014

Abstract

Escape analysis can be used for automatic memory management in Java. Based on the work of Choi et al. in 2003 [CGS+03], this thesis improves the existing escape analysis in KESO, a Java virtual machine for deeply embedded systems. Enhancements implemented for this document include flow-sensitive analysis, modifications that reduce compile time and a fix for a conceptual flaw in the work of Choi et al. Further analysis explored the possibility of allocating objects in callers' stack frames or memory regions.

Based on the computed analysis results, two different optimization backends are presented and compared: The preexisting stack allocation and a new method using task-local heaps automatically managed in a stack-like fashion with precise overflow checks. Both methods have predictable allocation and deallocation behavior and can guarantee tight upper bounds of their runtime since they do not suffer from external fragmentation. This increases predictability, which is beneficial in an embedded real-time Java environment.

Besides memory management, the thesis discusses other usage possibilities of escape analysis results, such as optimizations in remote procedure calls, synchronization optimizations, and a new theoretical approach to cycle-aware reference counting.

Compiler-assisted memory management pays positively: In a real-time Java benchmark, object lifetime is automatically inferred at up to 43.7 % of all allocation sites. The optimizations reduce heap memory usage, in some cases to less than half of what it was without automatic memory management using escape analysis. Additionally, the benchmark's time requirements are cut short by up to 18.7 %.

Zusammenfassung

Fluchtanalyse kann in Java zur automatischen Speicherverwaltung genutzt werden. Basierend auf einer Arbeit von Choi et al. aus dem Jahr 2003 [CGS+03] verbessert diese Masterarbeit die existierende Fluchtanalyse in KESO, einer virtuellen Maschine für Java im Anwendungsfeld eingebetteter Systeme. Die Erweiterungen, die für dieses Dokument implementiert wurden, beinhalten Flusssensitivität, Modifikationen zur Reduzierung der Übersetzungszeit und die Korrektur eines inhaltlichen Fehlers in der Arbeit von Choi et al. Weitere Analysen untersuchten die Möglichkeit der Objektallokation in Stapelrahmen bzw. Speicherbereichen von Aufrufern.

Basierend auf den berechneten Analyseergebnissen werden zwei Optimierungsmöglichkeiten vorgestellt und verglichen: Die bereits existierende Stapelallokation und eine neue Methode, die stapelähnlich verwaltete aktivitätsträgerlokale Halden mit präzisen Überlaufprüfungen benutzt. Beide Methoden besitzen vorhersagbares Allokations- und Deallokationsverhalten und können enge obere Schranken ihrer Laufzeit garantieren, weil sie nicht von externer Fragmentierung betroffen sind. Dies erhöht die Vorhersagbarkeit, was in eingebettetem Echtzeit-Java von Vorteil ist.

Neben Speicherverwaltung beleuchtet diese Arbeit auch andere Nutzungsmöglichkeiten von Ergebnissen der Fluchtanalyse, wie Optimierungen in Fernaufrufen, Synchronisationsoptimierungen und einen neuen theoretischen Ansatz zur zyklengewahren Referenzzählung.

Übersetzerunterstützte automatische Speicherverwaltung lohnt sich: In einem Leistungsbewertungsprogramm für Echtzeit-Java konnten die Lebensdauern der erzeugten Objekte an bis zu 43.7 % aller Allokationsstellen automatisch bestimmt werden. Die Optimierungen reduzierten die Auslastung des Haldenspeichers im Vergleich zu Messungen ohne Speicherverwaltungstechniken, die auf Fluchtanalyse basieren, in einigen Fällen auf weniger als die Hälfte. Zusätzlich verbesserten sich die Ausführungszeiten des Leistungstests um bis zu 18.7 %.

Contents

1 Introduction

Embedded devices are on the rise. While internet-connected refrigerators, which have been journalists' favorite prediction for the future, are still a long way off, other Turing complete devices have found their way into daily life. Taking a look around an ordinary kitchen reveals devices like a fully automatic coffee machine, a digital kitchen scale, a dish washer, or even a WiFi-enabled bar code scanner to update shopping lists. All of those contain microcontrollers, each serving a special purpose. And it seems these are just the first vistas of an emerging trend: At the *embedded world* conference 2014 in Nuremberg, the *Internet of Things*, i.e. the interconnection of these embedded devices, was a very popular topic. It seems the future will bring loads of highly integrated and networked microcontroller-driven devices.

These single-purpose embedded devices are commonly programmed in C and the like. However, with increasing software complexity of these devices, Java has been emerging as an appropriate alternative for a number of reasons. Java's ease of use, large standard library and suitability for complex projects are increasingly called for even in embedded software. Its type safe design prevents memory faults that could be caused by out of bounds array accesses or mistakes in manual memory management, eliminating a whole class of potential bugs. Case studies found these improvements to be not only theoretical [Phi99].

Java's advantages used to come at the price of reduced performance and increased memory usage due to the need for runtime environment and garbage collection. New virtual machines and compilers try to overcome the performance penalty using new techniques. *Ahead-of-time* compilation, i.e., translation to native code before deployment, has been gaining popularity lately: Google revealed switching its Android platform to a new runtime environment, which is expected to "speed up apps

by around 100 %" [Ant13] using ahead-of-time compilation [Lin14]. Microsoft declared a similar intention for its .NET platform [Lar14]. Oracle and others have developed Java virtual machines targeted at embedded systems [Mer13, Max12].

1.1 The KESO Multi-JVM

The KESO Java virtual machine uses ahead-of-time compilation and assumes all application code is available for analysis at compile time. This "closed world" assumption makes aggressive optimizations possible. KESO systems are built on top of real-time operating systems (RTOSs) implementing either OSEK/VDX [OSE05] or AUTOSAR [AUT06], specifications used in operating systems for automotive embedded systems. KESO allows writing applications and even device drivers in Java and targets deeply embedded systems with real-time requirements [TSWSP10]. KESO's compiler *JINO* analyzes the application code, tailors the runtime environment (including the Java standard library) to the application's needs, and emits standards-compliant C code to be compiled for the target architecture.

Figure 1.1 gives an architectural overview of a KESO system. The runtime environment provides an abstraction layer on top of the OSEK/VDX or AUTOSAR RTOS and allows configurable access to specific memory addresses for memory-mapped I/O, e.g., in device drivers. KESO also provides a mechanism similar to the Java native interface (JNI) to execute native code. A KESO system can contain multiple protection realms (so-called *domains*), each of which can have a number of tasks, resources, alarms, and interrupt service routines (ISRs), its own heap region, and garbage collection mechanism. Domains communicate using a remote procedure call (RPC) mechanism called *portals*. The KESO runtime environment ensures objects passed through portals cannot interfere with other protection domains.

Because Java is a type safe language, KESO can employ a combination of compile-time and runtime checks to ensure that applications cannot modify memory outside their protection realm even in the absence of specialized hardware for this task, such as a memory protection unit (MPU) or memory management unit (MMU). Since KESO guarantees complete isolation even when one of the tasks misbehaves, multiple applications can be run on the same system, possibly further reducing required chip size, energy consumption and production costs. Due to the reduction of struc-

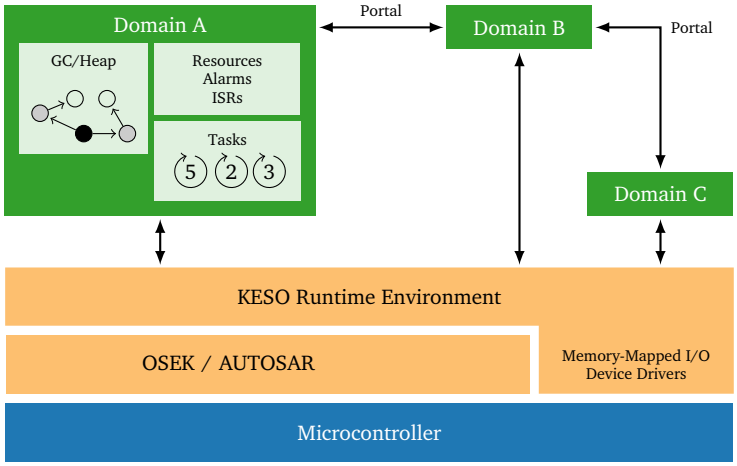


Figure 1.1: Schematic overview of a KESO system. An OSEK or AUTOSAR RTOS runs on a microcontroller. On top of the operating system, the KESO runtime environment provides services and abstractions used by the application, such as RPC primitives or device drivers. Multiple protection realms (*domains*) can contain multiple tasks each, have their own resources, heap, and garbage collector and communicate safely using *portals*.

ture sizes in modern computing chips, dealing with transient soft errors such as bit flips is mandatory for critical applications. Software-based mechanisms for isolation are at a disadvantage compared to microcontroller units (MCUs) with hardware-based memory protection such as MPUs and MMUs, which offer protection against errors caused by this problem class. Previous work on KESO attempts to compensate this [TSK+11, SSE+13].

1.2 Motivation

Manual memory management using library functions has been the de facto standard method of dealing with dynamic memory needs in C and C++. It provides fine-grained control over applications' memory allocation behavior, but comes with a downside: Programming mistakes can lead to leaks and dangling pointers, which in turn can lead to security vulnerabilities or crashes. As a consequence, developers need to be careful while writing code that uses manual memory management, in particular when used in safety-critical components.

In order to address these drawbacks, automatic memory management techniques, such as *garbage collection*, can be used. Instead of having the software developer deal with unused memory manually, slices of memory that are no longer referenced from the working data set are automatically identified and reclaimed, avoiding memory leaks and dangling pointers entirely. On the downside, unused memory is not reclaimed until the next garbage collection cycle, potentially reducing the predictability of an application's memory usage. Finding tight upper bounds for both runtime – the so-called worst-case execution time (WCET) – and memory usage is required to determine whether real-time constraints can be met. Compared to manual memory management, garbage collection is less error-prone at the cost of not reclaiming memory immediately, being less predictable and requiring additional computation. Both manual memory management and garbage collection need to deal with fragmentation caused by reclaiming in a sequence that differs from the allocation order (*external fragmentation*), further increasing the complexity of memory allocation and reclaiming.

Another alternative that exists in both manual and automatic variants are *region-based* approaches to managing memory. Each slice of memory is allocated in one of many memory regions (also called *pools*). Regions which are no longer used are reclaimed on the whole, avoiding external fragmentation completely. A pool will only be recycled if all of the objects allocated in its memory area are no longer referenced. A bad mapping from allocation to memory pool may thus prevent the recycling of unused memory; in the worst case a single object can hold a whole pool "hostage", preventing its re-use. On the other hand, region-based memory management does not suffer from external fragmentation because it does not reclaim slices from its pools. The time needed for allocation operations is thus easily predictable and allocation can be implemented in $\Theta(1)$. This is obvious when considering how allocation requests are handled and where the used memory resides: All previously allocated memory is still considered to be in use and occupies a single continuous block at the beginning of the memory pool. New requests can be served by extending this block, which takes a constant number of operations (moving a level marker). Constant runtime is an advantage over both manual memory management and garbage col-

lection, which do not always guarantee tight upper bounds for memory management operations[1].

Manual region-based methods require developers to map allocation operations to the regions that shall be used to satisfy the requests. The Real-Time Specification for Java supports manual region-based approaches to memory management with the subclasses of its `javax.realtime.ScopedMemory` class [BBG+00]. Manual methods allow fine-grained control over the application's behavior, but suffer from the same potential problems present in (non-region-based) manual memory management, such as dangling pointers.

Automatic region-based approaches infer regions and region assignments at compile time. Previously published techniques include semi-automatic methods, combinations of region inference and garbage collection to fully automatic region inference [GMJ+02, HET02, CCQR04]. Most of the work related to region inference is based on the work of Tofte and Talpin in 1994 originally aimed at functional languages [TT94]. Downsides of region inference in literature include its high algorithmic and computational complexity, its suboptimal results for larger programs and the assumption that object lifetimes follow a stack discipline. Small source code adjustments are sometimes necessary to achieve good memory performance. [HMN01]

A special case of manual region-based memory management is *stack allocation*. Each function call creates a new logical memory pool that can be used to allocate structures and objects on the stack. The pool is automatically reclaimed at the end of the method, so only objects whose lifetime is bounded by the runtime of their allocating function can be stored in stack memory. Since stack memory is automatically reclaimed, memory leaks cannot occur. Dangling pointers, however, are still possible, for example by returning a pointer to local stack memory. In type safe source languages such as Java, compilers can accurately compute which references point to the same memory locations (*alias analysis*) and whether any of these references remains live after the method that initially defined them terminates (*escape analysis*). Using escape analysis, dangling pointers can be detected at compile time and objects which will not escape their method of allocation can safely be allocated in stack

[1] This is not to say that they cannot, and there are a few algorithms that achieve good upper bounds for these operations by avoiding fragmentation or embracing it. See [Str14] for previous work in KESO about this problem and [PZM+10] for the work it is based on.

memory. In the context of real-time systems, automatic stack allocation can reduce the working set of the garbage collector and increase the system's predictability.

Stack allocation using escape analysis has been implemented in my bachelor's thesis [Lan12]. The scope of this thesis is improving the existing analysis (covered in Chapter 2) and extending it to allow more objects to be managed automatically without garbage collection (in Chapter 3). To achieve this, an analysis pass identifying objects which will outlive their method of allocation but not the calling method, was developed. The results of this analysis were used in a transformation that extends the scope of such objects into the calling method, enabling their allocation in the caller's stack frame, similar to a pattern often encountered in C programs. This transformation can be beneficial in some Java APIs commonly used in generic software, e.g., when dealing with strings and `StringBuilders`. While embedded systems in a car environment typically do not deal with strings a lot, the optimization might still be useful when implementing network communication protocols. Although the *Internet of Things* is not one of KESO's target domains, the work done in this thesis should improve the memory allocation behavior of applications that deal with string-based protocols such as HTTP.

1.3 | Previous Work

Based on the work of Choi et al. published in 2003 [CGS+03], I implemented alias analysis and escape analysis in my bachelor's thesis. Different from the paper it was based on, KESO's implementation does not dynamically allocate memory on the stack. Instead, allocation sites whose objects' liveness regions overlap are allocated in garbage-collected heap memory, keeping the stack usage bounded in the absence of recursion. Since KESO targets embedded systems with real-time constraints, predictable stack usage is important.

Note that most of this work cannot be applied to languages with pointers easily. This is due to the fact that pointer arithmetic might not be predictable at compile time, increasing alias analysis complexity. Because conservative assumptions must often be made for code using pointers, determining whether an object does not escape its method of allocation is much harder [Hor97, HBCC99, Lan92, Ram94].

1.4 Document Structure

Chapter 2 describes the changes implemented in KESO's escape analysis for this thesis. It starts with a basic summary of the algorithms and data structures used and contains detailed descriptions of improvements and the fix for a conceptual flaw in the work of Choi et al. The following Chapter 3 outlines the idea of improving stack allocation using variable scope extension and explains how this was implemented and which challenges I encountered while doing that. The evaluation in Chapter 4 compares the results of my work to previous versions of KESO using static results – like the number of automatically managed allocations – and runtime results of a number of benchmarks. Chapter 5 gives an overview of related work and explains the differences to this thesis before Chapter 6 concludes, gives an overview over the work done for this thesis and discusses some ideas for future work.

2 Escape Analysis

KESO's compiler *JINO* uses alias and escape analysis to identify objects whose lifetime is bounded by the runtime of their allocating method. The algorithm was implemented in [Lan12] and is largely based on the work of Choi et al. in 2003 [CGS+03]. The following section contains a brief description of the implementation and highlights differences. For an in-depth explanation, please refer to [Lan12] and [CGS+03]. Section 2.2 lists and explains the improvements written for this thesis before Section 2.3 concludes with a few applications of the information computed in escape analysis.

2.1 Basics

The algorithm starts with alias analysis, which is separated into a method-local (also *intraprocedural*) analysis and a global (*interprocedural*) analysis. 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.

2.1.1 Intraprocedural Analysis

In intraprocedural analysis, each method in the call graph of an application is traversed and a CG representation is computed. It is a key contribution of Choi et al. that this representation is independent of the calling context. Since their allocation

site 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 as discussed in Section 1.3, summarizing a method's effect independently of aliasing relationships in the calling context is impossible [CGS+03, p. 886]. 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.

Nodes in the CG have different types: *Object nodes* are added for each encountered allocation site. Note that a single object node in the graph might represent multiple objects at runtime because an allocation might be executed multiple times (e.g., if it is inside a loop). Local variables, static class members, and member variables are represented using *local reference nodes*, *global reference nodes*, and *field reference nodes*, respectively. Array members are treated like fields and are thus also represented by field reference nodes. Each reference node can point to a series of object nodes and via so-called *deferred edges* to other reference nodes. Deferred edges are used to simplify updates of the CG while processing assignments. After intraprocedural analysis, these edges are removed by replacing all incoming deferred edges of a reference node with edges to its successors. Different from the work of Choi et al., reference nodes with incoming deferred but no outgoing edges are preserved without change. Section 2.2.2 gives the rationale underlying this difference. Finally, object nodes can point to field reference nodes, denoting that the pointed field exists inside the object where the edge originates.

Each node in the CG has one of three *escape states*, indicating whether a node will outlive its allocating method or thread. Among these states, a total order exists. *Local* is the lowest state. Nodes marked local do not escape the analyzed method. Next after local is *method*. Nodes that outlive a method by being returned or assigned to an object passed as parameter are tagged method-escaping. Objects with an escape state of *method* are only reachable from the thread that allocated them. The highest escape state is *global* and is given to objects and references that are assigned to static class members (which exist once in each KESO domain to ensure the isolation of the protection realms) or thrown as exceptions. While processing a method's instructions and building the CG, operations that cause the escape state of one of their operands to change trigger the appropriate change in the escape states recorded in the connection

Listing 2.1: A simple generic linked list in Java

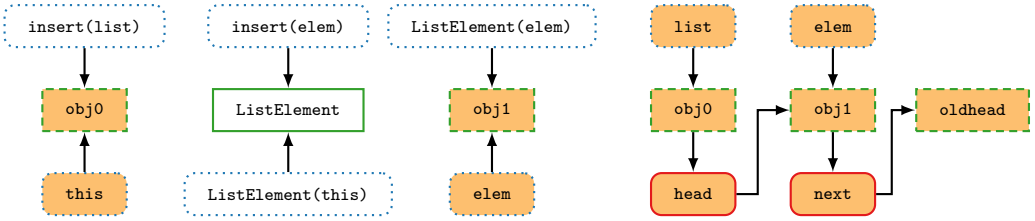
```
1  public class LinkedList<T> {
2    private static final class ListElement<U> {
3      ListElement<U> next;
4      U              elem;
5
6      public ListElement(U elem) {
7        this.elem = elem;
8      }
9    }
10
11   private ListElement<T> head;
12
13   public void addElement(T elem) {
14     insert(this, new ListElement<>(elem));
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 }
```

A simple generic linked list implementation in Java. Note that this example is more complex than it would have to be, especially due to the *insert* method, for demonstration purposes.

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.

graph. Allocations whose object node representation in the CG is tagged *local* are considered for stack allocation. Note that not all of these objects are converted into stack allocations. Allocations inside loops that create objects whose liveness periods overlap are not allocated on the stack, because that would require resizing the stack frame during the runtime of a method, which KESO avoids deliberately.

See Listing 2.1 for source code corresponding to the CGs to be explained in depth. The code 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 can not be allocated on the stack, because it must outlive the



(a) Connection graph for *addElement*. *insert(list)* and *insert(elem)* represent the parameters passed to the *insert* method at its invocation. *ListElement(this)* and *ListElement(elem)* do the same for the implicit invocation of *ListElement*'s constructor.

(b) 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.

Figure 2.1: The connection graphs for the *addElement* and *insert* methods given in Listing 2.1 after intraprocedural analysis. Vertices with rounded corners represent *reference nodes*, where *field reference nodes* have a red ■, other reference nodes a blue ■ border. Dotted borders mark artificial 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, orange ■, and red ■ represent *local*, *method*, and *global*, respectively.

method of its allocation *addElement*. Due to the structure of the example, intraprocedural analysis will not suffice to determine this. Global analysis will be necessary.

See Figure 2.1 for the connection graphs of the methods *insert* and *addElement* given in Listing 2.1. The *addElement* method has two parameters, but only the second one is visible in the code listing, because Java implicitly passes the *this* reference as first argument. These parameters are represented in Figure 2.1a by two reference nodes with dotted borders. Since they are reachable after the method returns, they are marked as *method*-escaping, denoted by the orange fill color. Because the allocation sites of the pointees of both *this* and *elem* are unknown, these objects are represented using *phantom nodes* (dashed green rectangles). Note that the escape state propagates along the edges from *this* into *obj0* and from *elem* into its pointee *obj1*. The first statement in the bytecode representation of *addElement* is the allocation of a new list element, which causes the creation of an object node (green rectangle) in the CG. The constructor of the newly created *ListElement* is called with two arguments: a reference to the object and a reference to the given parameter *elem*, represented in the connection graph by the *ListElement(this)* and *ListElement(elem)* reference nodes. The *this* parameter of the constructor invocation points to the allocated list element, denoted by a solid edge in the graph. The algorithm can deduce that *ListElement(elem)* points to the same object as *elem*. Adding a deferred edge

Listing 2.2: Example exposing the difference between flow-sensitive and flow-insensitive analysis.

```
1  public class FlowSensitivity implements Runnable {
2    static Object global;
3
4    public void run() {
5      Object a = new H();
6      Object b = new I();
7      Object c = new J();
8
9      a = b;
10     b = c;
11     c = a;
12
13     FlowSensitivity.global = c;
14   }
15 }
```

Simple Java example exposing the difference between flow-sensitive and flow-insensitive alias analysis. Flow-sensitive analysis will correctly determine that *c* will point to the *I* object at the end of the given method. Hence, escape analysis will determine that *H* and *J* do not escape *run*. Flow-insensitive analysis will list *H*, *I*, and *J* as possible pointees of *c*. As a consequence, all of the three allocations will be assigned a *global* escape state.

Adapted from [ALSU07, Sec. 12.4.3].

between the two encodes this situation (note that the deferred edge is not visible in the graph, because it has been compressed into an edge from *ListElement(this)* to *obj1*). Finally, processing the invocation of *insert* creates a similar set of nodes *insert(list)* and *insert(elem)* pointing to the *this* reference and the *ListElement* object, respectively.

The graph for *insert* is given in Figure 2.1b. The two parameters are again represented by a reference node and a phantom node each. The first few statements create the *next* field node below *obj1* and add a deferred edge to *head*. This edge is not shown in the graph, because the next statement in the code changes the value of *head*.

The next step of processing differs depending on whether the analysis is flow-sensitive or flow-insensitive. Since KESO's alias analysis was modified to be flow-sensitive for this thesis (see Section 2.2.1), the discussion of the current example will use the flow-sensitive approach. The difference between the approaches is best ex-

plained using the simple example given in Listing 2.2. Flow-sensitive analysis follows the control flow and computes the possible pointees of each variable after each statement. In the given example, the statement `a = b` removes the previous pointees of *a* (i.e., the *H* object) from the list of possible pointees of *a*. Flow-insensitive analysis on the other hand ignores the control flow. This means that statements are treated as if they could be executed in random order. Flow-sensitive analysis thus generates more accurate information at the cost of building a representation that is specific for a point in the control flow graph. The results generated by flow-insensitive analysis are valid for the entire method.

Returning to the current example, the flow-sensitive variant is used, which causes all incoming deferred edges of a reference node to be compressed and all outgoing edges to be removed before pointing to the new target. Compressing deferred edges requires the target of the deferred edge to have at least one pointee. Since *head* does not have any pointees at this point in the analysis, a phantom node representing the possible previous pointees of *head* is created. This node is denoted *oldhead* in Figure 2.1b. Using flow-insensitive analysis, the deferred edge from *next* to *head* would have been preserved and later compressed into an edge from *next* to *obj1*. Finally, the edge from *head* to *obj1* is added to represent the effect of the statement in line 20 in Listing 2.1, completing intraprocedural analysis.

2.1.2 | Interprocedural Analysis

Looking at the summary information generated for both methods given in Listing 2.1, the object node that represents the only allocation (*LinkedList* in Figure 2.1a) has an escape state of *local* at this point in the analysis. Recall that local nodes are considered for stack allocation, but entries of a linked list cannot be allocated on the stack. The results are thus not sound and further analysis is required. Looking at the example, the edge from *head* to *obj1* in the CG of *insert* is the edge that will prevent stack allocation of the list element in *addElement*. To determine this algorithmically, the edges found while processing *insert* need to be propagated to its caller *addElement*. Once the connection from *obj0* via a new field reference node *head* to the list element is established, the escape state of the *ListElement* object node increases to *method*, making the result sound.

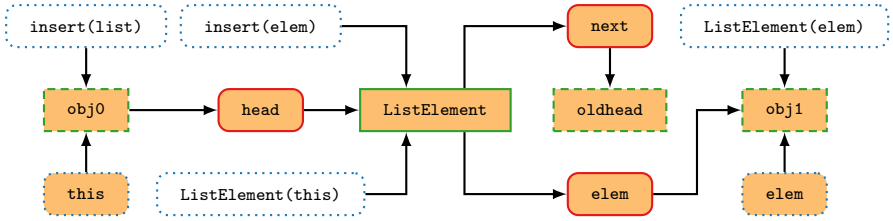


Figure 2.2: The connection graph for the *addElement* given in Listing 2.1 after interprocedural analysis. Colors and shapes cf. Figure 2.1.

A second analysis pass propagates information from the subgraph containing all non-*local* nodes of the CGs into the CGs of all calling methods. This *interprocedural* analysis modifies the summary information of the callers, which in turn require their callers to be updated again. To prevent unnecessary recalculation of information, this pass should use a bottom-up traversal of the call graph. In the absence of recursion, the call graph will not contain cycles, making this a simple problem. When recursion is used, identifying strongly connected components and iterating in each component until a fixed point is reached has proved to be effective. KESO's implementation uses Tarjan's algorithm to identify strongly connected components and a topological order among them in the call graph [Tar72].

To update the callers' connection graphs, pairs of corresponding nodes in the graph on the caller and the callee side are identified as starting points and added to a work list. Originating at these anchors, further nodes and their counterparts are found and again added to the work list. While the work list is not empty, processing continues and builds a relation of object nodes in the callee and the caller CG called *mapsToObj*. This step is called *updateNodes*. A simplified form of the procedure is given in Algorithm 2.1. It uses a dual work list approach to avoid creating spurious phantom nodes because some of the relationships might not be known until the algorithm completes. See [Lan12, Sec. 3.2.1] for a detailed explanation of the problem that causes unneeded phantom nodes to be added and slows down the analysis. *UpdateNodes* ensures that all object nodes used in a callee are represented in its caller. It also adds field reference nodes present in the callee CG but missing from the caller's graph and marks the counterparts of globally escaping nodes in the callee's CG as globally escaping in the caller CG.

Algorithm 2.1: The *updateNodes* procedure [Lan12, Alg. 2]

Input : xs: method parameters, ys: invocation arguments
Result: mapsToObj relation between caller and callee nodes

```
1  updateNodes (xs, ys)
2  begin
3      workList = {(x, y) | x ← xs | y ← ys };
4      needsPointee = ∅;
5      mapsToObj = ∅;
6      while workList ≠ ∅ or needsPointee ≠ ∅ do
7          while workList ≠ ∅ do
8              (mParam, iArg) = pop(workList);
9              // Mark iArg globally escaping if mParam is.
10             updateEscapeState(iArg, mParam);
11             // Find pairs of descendant object nodes
12             xPointees = pointees(mParam);
13             yPointees = pointees(iArg);
14             if xPointees ≡ ∅ or yPointees ≠ ∅ then
15                 // Find pairs of field reference nodes
16                 foreach (xd, yd) ∈ {(x, y) | x ← xPointees, y ← yPointees } do
17                     mapsToObj(xd) ∪= yd;
18                     foreach calleeField ∈ fields(xd) do
19                         callerField = getField(yd, calleeField);
20                         workList ∪= (calleeField, callerField);
21             else
22                 // The callee node is not represented in the caller node
23                 needsPointee ∪= (mParam, iArg);
24         while workList ≡ ∅ and needsPointee ≠ ∅ do
25             foreach (mParam, iArg) ∈ needsPointee do
26                 if pointees(iArg) ≠ ∅ then
27                     // mParam's pointees are represented (happens with recursion)
28                 else
29                     // Check for other representatives of pointees of mParam
30                     callerObjs = {x | x ← mapsToObj(y) | y ← pointees(mParam) };
31                     if callerObjs ≠ ∅ then
32                         addEdges(iArg, callerObjs);
33                     else
34                         // Delay adding phantom nodes
35                         continue;
36                 needsPointee \= (mParam, iArg);
37                 workList ∪= (mParam, iArg);
38             if workList ≡ ∅ then
39                 // workList is still empty, no pairs found. Add a phantom node.
40                 (mParam, iArg) = pop(needsPointee);
41                 addEdges(iArg, createPhantom(mParam));
42                 workList ∪= (mParam, iArg);
```

The example given in Listing 2.1 and Figure 2.1a is shown after interprocedural analysis (for both invocations) in Figure 2.2. In this case, interprocedural analysis adds field reference nodes *head* and *next* below *obj0* and *ListElement*, respectively. Furthermore, a phantom node is added to represent *oldhead*. Note that the *head* reference node does not yet point to the list element object node. The next step of the algorithm called *updateEdges* adds the missing connection. It takes the same parameters as *updateNodes* from Algorithm 2.1 and adds all missing edges. See Algorithm 2.2 for a pseudocode listing of the procedure. It identifies pairs of corresponding object nodes using the *mapsToObj* relation computed in *updateNodes*. For each possible pair it follows outgoing edges to any field reference nodes in the callee graph and finds the corresponding field reference node in the caller's CG. Next, the newly found correspondence pair is added to the work list and all outgoing edges to object nodes in the callee graph are added to the caller graph. Note that code to prevent endless loops in cyclic data structures has been left out for simplicity, but can be easily added.

Algorithm 2.2: The *updateEdges* procedure [Lan12, Alg. 3]

Input : xs: method parameters, ys: invocation arguments

```
1 updateEdges (xs, ys)
2 begin
3     workList = xs;
4     while workList ≠ ∅ do
5         mParam = pop(workList);
6         foreach (x, y) ∈ {(x, y) | y ← mapsToObj(x) | x ← pointees(mParam) } do
7             foreach (i, j) ∈ {(field, getField(y, field)) | field ← fields(x) } do
8                 workList ∪= (i, j);
9                 addEdges(j, {mapsToObj(k) | k ← pointees(i) });
```

After interprocedural analysis, nodes marked *local* 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 because the amount of memory used by these allocations might be unbounded, e.g., if the allocation is inside a loop. See [Lan12, Sec. 3.3] for detailed rationale and a description of the implementation.

2.2 Improvements

The algorithm implemented in [Lan12], which is based on [CGS+03], was improved for this thesis in a number of ways. Among these improvements was flow-sensitivity, which was proposed by Choi et al. in 2003, but not implemented in my bachelor's thesis. The changes required to achieve flow-sensitivity are outlined in the following Section 2.2.1. Furthermore, a problem producing possibly incorrect results was discovered in the algorithm given in [CGS+03]. Section 2.2.2 gives an example and explains where the incorrect analysis results occur and how they were fixed in KESO's implementation.

Last but not least, the algorithm's runtime on large examples amounted to several minutes and was deemed unsatisfactory. In particular, recursive and virtual method invocations significantly increased the size of the generated CGs, raising the runtime of interprocedural analysis. Section 2.2.3 deals with modifications implemented to reduce the runtime of the alias analysis.

2.2.1 Flow-Sensitivity

For flow-sensitive alias analysis a standard forward data flow analysis [ALSU07, Sec. 9.2] is used. The set of operations needed for data flow analysis is

$$C_o^b = f_b\left(C_i^b\right) \tag{2.1}$$

$$C_i^b = \bigwedge_{x \in \text{pred}(b)} C_o^x \tag{2.2}$$

where C_i^b and C_o^b are input and output data flow information for the basic block b, f_b in Equation (2.1) is called the *transfer function* for the basic block b, and $\wedge_{x \in \text{pred}(b)}$ in Equation (2.2) is the *meet* operation. The data flow transfer function modifies the data flow information (i.e., the connection graph) according to the statements in the basic block b. The meet operation combines the output information of all predecessors of a basic block into the input information of a given basic block b. The basics of the transfer function are explained in [Lan12, CGS+03]. In theory, all C^b are distinct. In practice, this would manifold the memory requirements for the CGs. Instead of copying the graph in each invocation of the transfer and meet

operations, KESO's implementation uses a single representation. This idea is also present in [CGS+03, Sec. 3], but is not explained very well. In specific, it is not clear to the author of this thesis what Choi et al. meant when they wrote "in the flow-sensitive version, we only kill local variables" [CGS+03, p. 885].

The KESO compiler achieves flow-sensitivity while retaining a single representation of the CG by tagging all reference nodes with the basic block for which they are valid. Object nodes are not modified for flow-sensitive analysis. For each reference node used in at least one predecessor, the meet operation creates a representation of the reference in the current basic block and subsumes all outgoing edges present in the predecessors. For each assignment operation encountered by the transfer function, *ByPass(p)* is called on the reference to be written. *ByPass(p)* (as explained in [CGS+03]) redirects all incoming deferred edges of p to its successors and removes any outgoing edges (i.e., it ensures that strong updates are performed).

After implementing this improvement, a fixed point iteration used to reduce the number of unnecessary phantom nodes in intraprocedural analysis did no longer terminate for some inputs. This happened because the iteration tracked changes to the CG rather than comparing the graph against an older copy. Due to the use of *ByPass(p)*, the graph was modified in every loop, but further processing returned to the previous state again. Switching to a comparison against an old copy of the CG rather than tracking of modifications fixed this particular problem. To efficiently implement comparisons against older versions of the same graph, the connection graph's nodes were extended with the ability to store a copy of a single older state of outgoing edges.

2.2.2 | Fixing Incorrect Results: The Double Return Bug

KESO's implementation of escape analysis produced incorrect results given inputs similar to those generated by the idea outlined in Section 3.1. Further analysis suggests this is a conceptual flaw in the work of Choi et al. See Listing 2.3 for an example triggering this bug. The *getObject* method allocates two objects and passes them to *chooseOne*, which selects one of them at random and returns it. The return value of *chooseOne* is then returned from *getObject*. Because either of the two objects allocated in *getObject* might escape, both allocations must not use stack memory.

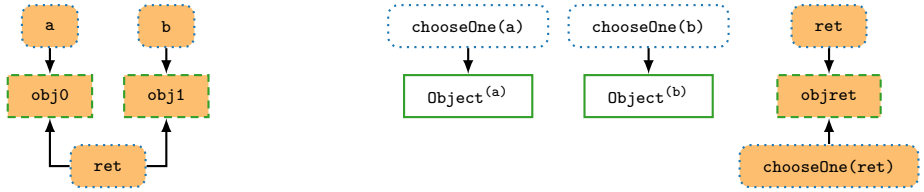
Listing 2.3: Example exposing the double return flaw

```
1  public class ChooseOne implements Runnable {
2    public void run() {
3      Object a = getObject();
4    }
5
6    private static Object getObject() {
7      return chooseOne(new Object(), new Object()); // bug occurs here
8    }
9
10   private static Object chooseOne(Object a, Object b) {
11     if (Math.random() < 0.5)
12       return a;
13     return b;
14   }
15 }
```

A simplified example exposing the double return bug in the escape analysis by Choi et al. One of the objects allocated in *getObject* is returned from its allocating method, but escape analysis did not detect this due to the use of phantom nodes to represent return values.

However, the CG constructed according to [CGS+03, Sec. 4] does not correctly identify the two objects as method-escaping. The connection graph for *chooseOne* is straightforward and given in Figure 2.3a. For the CG of *getObject*, Sections 4.3 "The Connection Graph Immediately Before a Method Invocation" and 4.4 "The Connection Graph Immediately After a Method Invocation" are the relevant parts of [CGS+03]. According to the first section a new *actual reference node* is created for each argument of the invocation and an assignment $\hat{a}_i = u_i$ is processed. $\hat{a}_i$ denotes the actual reference nodes, u_i are the corresponding invocation arguments. The statement causes the creation of deferred edges from the $\hat{a}_i$'s to the u_i's, which will later be compressed.

The handling of return values is not explicitly explained in this section, but the next section mentions them as "$\hat{a}_i$'s (representing actual arguments and return value) of the caller's CG," [CGS+03, p. 891] suggesting that an *actual reference node* for the return value is added in the caller's CG. Figure 2.3b shows this connection graph: The rounded rectangles with blue dotted borders are said *actual reference nodes*. Both *chooseOne(a)* and *chooseOne(b)* initially have a single outgoing edge pointing to a local variable, which in turn points to the allocated objects. This indirection is omit-



(a) The CG for *chooseOne*.

(b) The CG for *getObject* according to [CGS+03, Sec. 4].

Figure 2.3: The CGs for *getObject* and *chooseOne* from Listing 2.3, exhibiting the double return flaw. The object nodes in *getObject* should be pointed-to by *ret*, which would raise their escape state to *method*, but these edges are missing. Colors and shapes cf. Figure 2.1.

chooseOne	getObject
obj0	Object(a), objret
obj1	Object(b), objret

(a) The *mapsToObj* relation constructed using *updateNodes* as given in [CGS+03]. Note that this is the same even after KESO's modifications.

chooseOne	getObject
a	chooseOne(a)
b	chooseOne(b)
ret	chooseOne(ret)

(b) The *mapsToRef* relation constructed using KESO's modified *updateNodes* algorithm.

Table 2.1: The *mapsToObj* and *mapsToRef* relations for the call of *chooseOne* from *getObject* as given in Listing 2.3.

ted from the graph in Figure 2.3b for simplicity. The return value of *getObject* is modeled using assignments to a special "phantom" variable called *return* (in KESO's implementation: *ret*). Since the result of the call to *chooseOne* is returned from *getObject*, a deferred edge from the phantom return variable to the *actual reference node* representing *chooseOne*'s return value is added. After completing intraprocedural escape analysis all deferred edges are removed from the graph according to Choi et al., adding phantom nodes where necessary. This leads to the creation of the phantom node denoted *objret* in Figure 2.3b. The following path compression removes the deferred edges from *ret* to *chooseOne(ret)* and adds a *points-to* edge to *ret*. The *method* escape state of *chooseOne(ret)* is retained, but is not relevant for the further problem description.

The analysis ends with the *UpdateCaller* routine. It consists of *UpdateNodes* and *UpdateEdges*. The former computes equivalence pairs of object nodes in the callee's and caller's CGs (the so-called *mapsToObj* relation) and adds phantom nodes for objects that have no equivalence in the caller yet. *UpdateEdges* ensures all relevant edges present in the callee's CG are propagated into the caller's graph. The *mapsToObj* relation for the call from *getObject* to *chooseOne* is given in Table 2.1a. If *chooseOne(ret)*

did not yet have any pointees at this point of the analysis, statement 7 in *UpdateNodes* as displayed in [CGS+03, Fig. 7] would have created it as a phantom node, leading to the same problem. No new edges are inserted in interprocedural analysis for this example, because no structure of the form $p \rightarrow f_p \rightarrow q$ (where p, q are object nodes and f_p is a field reference node) exists in the CG of *chooseOne*.

Note that both phantom nodes in *chooseOne*'s CG map to both their respective object node and the *objret* phantom node, but the equivalence of $\textit{Object}^{(x)}\ \forall x \in \{a, b\}$ and *objret* is not represented in *getObject*'s CG, leading to incorrect escape states for the two allocated objects.

To work around this problem, KESO's alias analysis (outlined in Algorithms 2.1 and 2.2) was extended to not only track equivalences between object nodes in *mapsToObj*, but also between reference nodes in a new relation called *mapsToRef*. This data is used in a modified version of *updateEdges* to add the missing edges in the caller's CG. On each occasion of $p \rightarrow o$ in the callee's CG where p is a reference node that does not represent a parameter and o is an object node, an edge $x \rightarrow y$ is added in the caller's CG for each $x \in \textit{mapsToRef}(p)$ and $y \in \textit{mapsToObj}(o)$, if no such edge exists yet. Edges outgoing from parameters must be ignored in this step because Java has call-by-value semantics which means that the arguments given at a method invocation will always remain unchanged. All references reachable via other edges below the arguments can be modified, however, as can the return value.

Using this extension for the running example generates the *mapsToRef* relation as given in Table 2.1b. To avoid the superfluous phantom node *objret* that would be added because of the removal of deferred edges before interprocedural analysis, KESO's alias analysis does not attach phantom nodes to nodes that have incoming deferred but no outgoing edges. This required modifying the analysis to be able to deal with deferred edges in interprocedural analysis. After the modified interprocedural analysis finishes, the CG of *getObject* (depicted in Figure 2.4) contains the missing edges.

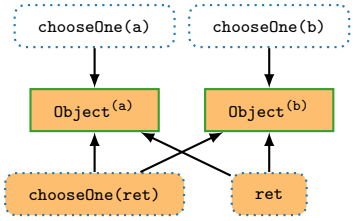


Figure 2.4: The CG for *getObject* as generated by KESO's modified algorithm to fix the double return flaw. The two object nodes are correctly marked *method*-escaping. Colors and shapes cf. Figure 2.1.

2.2.3 | Interprocedural Analysis Optimizations

Some of the larger applications (up to $28.3\ k$SLOC[1]) 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.

2.2.3.1 | No Propagation of Read Operations

Analyzing the generated CGs after interprocedural analysis 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 [Tar72] finds all cycle-free paths from the method's

[1] generated using David A. Wheeler's "SLOCCount"

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: For example, the *java.util.Hashtable* class from the GNU classpath project implements *equals* by comparing the entry sets of the two hash tables. This entry set is eagerly 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 Chapter 4 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.

2.2.3.2 | No Reprocessing of Unchanged Invocations

Strongly connected components in an application's call graph (i.e., recursive methods) are handled in interprocedural analysis by iterating until a fixed point has been reached. During this process, invocations might be reprocessed even though their callees' CGs have not changed since the last iteration. This happens for all call graph edges leaving the strongly connected component. See Figure 2.5 for a graphical representation of this situation.

KESO's interprocedural analysis only re-runs the *updateNodes* and *updateEdges* steps if a callee's CG changed since the last iteration, avoiding unnecessary overhead. Unfortunately, the savings from this optimization are marginal.

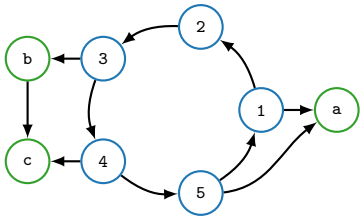


Figure 2.5: Call graph with a strongly connected component (blue vertices) and a few dependent nodes (green vertices). While methods inside the strongly connected component might have to be visited multiple times in interprocedural analysis, the summary information from `a`–`c` only needs to be propagated into their callers once.

2.2.3.3 | Connection Graph Compression

While the improvements in Sections 2.2.3.1 and 2.2.3.2 reduced the compile time of large applications, the savings were still not enough to enable escape analysis by default in KESO without having a noticeable effect during development. Connection graph sizes would still surpass 10000 vertices on large inputs with these optimizations enabled, slowing down further steps of the analysis. Most of these nodes were created in interprocedural analysis as phantom nodes to represent objects allocated in callees of the current function and often had siblings that would represent the same objects. To reduce the size of the connection graphs, a graph compression transformation inspired by Steensgaard's almost linear time points-to analysis [Ste96] was implemented.

Starting at each entry point into a method's CG (i.e., every method parameter and the 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. Since this might create new graph constellations that can be compressed, the color bit possibly marking the descendant field reference nodes as visited is reset.

Since these modifications always preserve object nodes and do not unify subgraphs with different escape states, the effect on the results is negligible. However, the compile time required for alias analysis has improved by an order of magnitude.

2.3 | Applications

Besides stack allocation, the results of KESO's escape analysis can be used for other applications. This section presents some of the analyses and optimizations that use the connection graphs generated by alias and escape analysis. Note that some of the ideas outlined have proved to be difficult or impossible to implement using KESO's representation of the alias information, and some are possible but have not been implemented yet due to time constraints.

2.3.1 | Removing Unneeded Copies in Portal Calls

To support communication between protection realms (*domains*), KESO offers an RPC implementation called *portals* (see also Figure 1.1). To ensure complete isolation of objects passed through portals, the KESO runtime environment creates deep copies of these objects in the target domain's heap. This ensures that modifications of the object in the target domain do not affect the object in the source domain. Especially for large trees of objects, this is very expensive. Results from escape and alias analysis can be used to determine whether the copy can be omitted. This optimization pass is called *superfluous portal copy removal*.

Objects must be copied if they, or any object reachable from one of their fields, (a) live longer than the runtime of the methods handling the portal call on the destination side, i.e., if they have a global escape state in the callee's CG, or (b) are modified by the callee. Whether (a) applies can be determined using the connection graph for the method handling the portal call. For each argument passed into a portal, the corresponding representation on the callee side and its descendant nodes

must not have a *global* escape state. A simple work list algorithm suffices to check for globally escaping objects.

Avoiding the removal of copies that are required because they are modified according to (b) is more complicated. KESO's implementation starts by constructing a mapping between the CG representation of objects and the index of the portal call argument that initially brought them into the portal handler's protection domain, if any. In a pass over all code reachable from the portal handler, each write operation's operands are checked for the presence of this mapping. If no mapping exists, the modified object was not passed through the portal but originated in the portal handler's domain. As such, it can be freely modified. On the other hand, if a mapping exists, the currently processed instruction modifies an object passed through a portal, i.e., the instruction is a witness stating that the parameter must be copied. Hence, the argument, whose index is obtained from the mapping, is marked as *must-copy*. If the code traversal encounters method invocations that reference any of the objects in question, the mapping is extended and the method's code is visited recursively. If no witnesses against the copy removal are found after the traversal is complete, the copy operation is omitted.

KESO's portal mechanism supports replication and voting on the results as an approach to software-based fault tolerance. By default, each object tree passed into a replicated portal is copied three times, once for each replicated portal handler. Superfluous portal copy removal is used to avoid the third copy, if copying is not necessary.

Note that KESO's implementation either completely copies an object and its dependents, or it does not copy any part of the object. While it is possible to compute which descendants of an object are modified or escape the callee and only copy these parts rather than the whole tree, the additional runtime support and runtime representation of the partial copy information made this a refinement not considered worthwhile by the KESO developers.

2.3.2 | Synchronization Optimizations

The information computed in escape analysis and stored in the CGs can be used to remove unneeded synchronization operations. Objects that are only reachable

from a single thread (or task in the KESO context) and are used for synchronization will never have to wait for a lock. Blocking can only occur when a different thread currently holds the lock of the object, but this is not possible if the object is not reachable from within more than one thread. The connection graph can be used to determine whether objects are local in a thread using the escape state. An escape state of *method* or lower implies that the object does not escape its thread of creation.

Since KESO does not currently synchronize at objects but uses OSEK's *resource* abstraction for mutual exclusion, this optimization's potential for performance improvement and code size reduction is likely to be low, and it has not been implemented in KESO. However, other situations that require synchronization in KESO could still benefit from the information. On target platforms with small word sizes, such as AVR microcontrollers, KESO ensures multi-word data writes are not interrupted by disabling interrupts for the duration of the write operation. If the written objects are task-local, this safety precaution is not necessary, because the modified object cannot be read in an inconsistent state by other tasks. Once execution of the interrupted task resumes, the write operation will continue and bring the multi-word data field into a consistent state.

2.3.3 | Cycle-Aware Reference Counting

Automatic reference counting as a method of automatic, compiler-assisted memory management that does not require garbage collection has been gaining popularity lately. For example, Apple's Objective-C used on iOS and OS X employs compiler-generated reference counting. Unfortunately, it is a well-known limitation of reference counting that it cannot automatically reclaim self-referential (i.e., cyclic) data structures. Cyclic data structures either require the additional use of a garbage collector, or the use of special pointer types called *weak pointers* that do not increase the reference count of objects to break the cyclic structure.

Using the connection graphs constructed in alias and escape analysis, cyclic structures, their components and corresponding allocation sites can be computed at compile time. Using this information, cycles could be automatically managed by assigning a cycle descriptor to each object that is part of a cyclic structure and keeping a reference count for the whole structure in the cycle descriptor. Once this reference

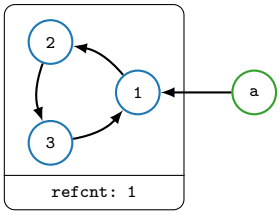


Figure 2.6: Cycle-aware reference counting using cycle descriptors. Blue ■ vertices are part of a cyclic data structure, green ■ ones reference cyclic structures. Nodes 1–3 share a cycle descriptor. Adding a reference to an object inside the cycle looks up its cycle descriptor and increases the cycle's reference count.

count drops to zero due to the release of an object that references the structure, the whole cycle could be reclaimed.

Figure 2.6 has a graphical representation of the approach. The structure consisting of the three nodes 1–3 is identified as self-referential at compile time. A cycle descriptor is allocated with the first object of the cycle, and each object's header references the cycle descriptor. Adding the reference from a to 1 increases the cycle's reference count. When the cycle reference count drops to zero, the complete cycle must be unreferenced and can be reclaimed.

Unfortunately the representation of the alias information in JINO makes it difficult to apply this idea, because the CG of a method is independent of its calling contexts (which is one of the significant contributions of Choi et al. in [CGS+03]). Cycles in connection graphs can be identified, but are useless when containing phantom representations of nodes passed into a method from a caller. Because a single static representation exists for multiple instances of cycles, the approach is unlikely to perform well: For example, when used on KESO's current alias analysis results, all objects stored in a doubly linked list (which is a cyclic data structure) can only ever be reclaimed on the whole, even if the lists are completely separate. The approach might work reasonably well given a global *points-to* graph, but this has not been tested in KESO.

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 location of the buffer is controlled by the calling function, it can be allocated in stack memory. In Java, a method would instead allocate a new object on the heap and return a reference to it to achieve the same. Using information from alias and escape analysis, objects that escape their method of allocation into the caller but no further can be automatically identified. Since the lifetime of these objects can be statically determined, the need for garbage collection can be avoided and the memory can be automatically managed using compiler-generated code (for example, but not limited to, using stack allocation). This further reduces the load of the garbage collection mechanism and can improve worst- and average-case execution times of applications. This optimization is called *scope extension* in the following.

Note that while only stack allocation has been discussed as optimization to manage objects with statically computed and bounded lifetimes, it is not the only possibility, and may not be the best. Several other approaches such as region-based methods or explicit deallocation operations come to mind. Depending on the nature of the optimization used, their unbounded application may lead to problems and can in fact worsen an application's performance. Nonetheless, stack allocation will serve as the default backend in the code and the following description of the algorithms. Steps in the optimization that are induced by the properties of stack allocations are marked as such.

In the remainder of this chapter, 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

Listing 3.1: Example containing a candidate for scope extension

```
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 : getAircraft()) {
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*.

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 use of scope extension.

3.1 | Algorithmic Idea

Listing 3.1 shows an example adapted from the source code of the CD_j benchmark [KHP+09] where scope extension 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. The runtime of *Simulation.run* is thus an upper bound for the lifetime of the object. Consequently, KESO does not have to rely on garbage collection to reclaim the memory used by the object, but can instead automatically manage the object's memory using one of the techniques from Section 3.3.

All examples discussed so far deal with objects escaping their method of creation via a return operation. Note that being returned is not the only way an object can 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

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 by 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. When using stack allocation, even objects that are *local* in a calling method might still not be eligible for optimization due to overlapping liveness regions. KESO's stack allocation transformation avoids possibly unbounded growth of stack usage by omitting the transformation into a stack allocation if multiple objects allocated at the same allocation site are in use simultaneously (see Section 2.1.2 and [Lan12, Sec. 3.3] for details).

When the optimization backend used requires additional parameter passing across invocations, virtual method calls need to be handled with special care to avoid breaking their signatures: 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 scope extension.

Because of the complexity involved in doing so, KESO's implementation does not take the escape state of object nodes' equivalents in the callers' CGs into account. For each run of the optimization pass, allocations are propagated at most a single level

up against the direction of the call hierarchy. Therefore, running the pass multiple times will increase the maximum scope extension level. Note that it is not necessarily beneficial to run the pass often, since it may lead to undesirable results. See Section 3.4 for a discussion of the limitations and problems of scope extension.

3.2.1 | Non-Virtual Calls

Non-virtual call sites, i.e., those where the invoked method is unique and known at compile time, constitute the simple cases of the analysis. Fortunately, 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], increasing the number of non-virtual method calls.

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. When optimizing allocations of local objects using stack allocation, the allocation instruction must be moved into all callers. A reference to the allocated object is instead passed to the method on invocation, which uses this reference instead of the reference previously returned by the allocation instruction. Each allocation that is optimized using scope extension is copied into all callers and executed unconditionally, regardless of whether the allocation sites were in mutually exclusive control flow paths before optimization, and hence could never be used at the same time. In some examples, this causes a large number of allocations and new method parameters even though only a few are used simultaneously. See Section 3.4 for a detailed analysis of the problem, possible ways to avoid it, and a discussion of the challenges in solving it.

3.2.2 | Virtual Calls

Virtual method invocations further complicate the decision whether to apply scope extension to an allocation site. Since 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 its siblings when the optimization requires adjusting a method's signature (as is the case when using the default stack allocation backend). Figure 3.1 contains a graphical representation of this problem. Interdependencies between methods cause them to form up into groups

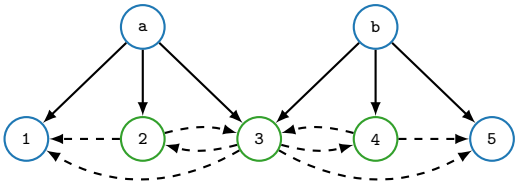


Figure 3.1: Call graph showing the complexity of scope extension for virtual method calls. Green vertices mark methods that contain allocations eligible for scope extension, blue vertices represent other methods. Solid lines are method invocations. Assume that both *a* and *b* contain a single virtual 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 scope extension 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*.

sharing the same signature. Scope extension, 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 modification of its invocations' argument lists, which in turn requires the same changes to all other candidates for the modified method calls. Since the modifications are in general unnecessary in all other methods than the one causing them, this increases the runtime overhead and possibly allocates memory that is unused in most candidates of a virtual invocation.

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 invocations. Note that some of the challenges are caused by properties of the applied optimization. Intermediate code transformations that do not require changing a method's signature can 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. Objects allocated in this region would remain valid until the calling method terminates.

A different approach to solve the same problem could be not to change the allocations at all (i.e., allocating in garbage-collected heap memory) and to modify all callers 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 reduces neither 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 work done in the sweep step is the same, whether it is run by the garbage collector or 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 memory areas that can be re-used immediately, the runtime system could avoid the scan phase entirely if enough memory can be reclaimed 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.

In the complete absence of a garbage collector, the scope extension could be used to check manual memory management for potential mistakes. In each location where KESO would automatically place a reclaim operation, it can check whether the programmer did write the expected instruction. If the object in question is not explicitly returned into the memory pool, leakage occurs and the compiler can print a warning. For this application, running the scope extension pass multiple times can be beneficial, because due to the lack of modifications, the code size does not increase, but the number of objects whose lifetime can be inferred by the compiler grows.

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 program and moves eligible allocations into a method's callers. To preserve soundness, a reference to the allocated object is instead passed as argument at the method's invocation. The newly created allocation can potentially use stack memory subsequently. The second new transformation applies in KESO's backend where C code is emitted. It is described in Section 3.3.2 and introduces a different concept to avoid potential problems with excessive stack usage.

Listing 3.2: Java bytecode of a complete object allocation

```
new $1           # 1 is constant pool entry for the object type
dup
invokespecial $2 # 2 is constant pool entry for the object's constructor
astore_0         # stores a reference to the object in a local variable
```

Java bytecode of a complete object allocation. The *new* instruction allocates new memory. The following *dup* operation is required because of the JVM's stack machine semantics. Before *astore_0* stores a reference to the allocated object in the variable 0, *invokespecial* calls the object's constructor and passes a reference to the memory area as first parameter.

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. See Listing 3.2 for the sequence of Java bytecode instructions generated for each allocation.

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. If the application can be interrupted between stack allocation in the caller and constructor 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.

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 opti-

mization 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 method parameter, where the parameter type equals the type of the object. Replacing the *new* operation with the instruction reading the parameter (one of the *aload* instructions in Java bytecode) 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].

Adding a new parameter changes 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 optimizing allocations of *local* objects.

3.3.2 | Task-Local 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. 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 use heap memory for these allocations.

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 (i.e., those corresponding to 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 local to each task. Given these local heaps, the logical regions are implemented similar to a stack in KESO: Each task-local heap has a fill marker and a maximum fill level. At method entry, the fill marker is saved and necessary objects are allocated by moving the fill marker. At method exit, the fill marker is reset to its previous value. Saving the fill marker on the stack can be avoided if the amount of memory that will be allocated in a function and all alignment offcuts are known at compile time, because this knowledge can be used to calculate the value at method entry. KESO's implementation does not currently support this optimization. The approach does not require any synchronization for allocations, which constitutes another advantage over heap allocation.

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 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 (possibly exceeding amounts proportional to the number of 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 Chapter 6) could automate this process and determine the size of task-local heaps automatically.

It is worth noting that this optimization can only be applied to code that will always be called in a task context. Class initializers and methods called from class initializers may not use a task-local heap because they are called during startup and not in a specific task. Since tasks and their code are implementations of Java's *Runnable* interface in KESO, they have a constructor. While this constructor will also be called from outside any task context, task-local allocations can still be enabled by using the task-local heap corresponding to the task that is being constructed.

KESO's implementation uses static analysis to determine which tasks reach a specific method. If only a single task uses an allocation site, the dynamic lookup of the task descriptor (which contains the task-local heap descriptor) is replaced with a simple address-of operation that can be resolved to a memory address by the compiler and linker. If multiple tasks use an allocation, a dynamic lookup is required and preserved. If an allocation is not reachable from any task constructor or entry point, it must not use local heaps. KESO's compiler will detect this situation and generate an error message to prevent runtime errors. ISRs do not use task-local heaps in the current implementation. They could use their own task-local heaps or use the task-local heap of the current active task they interrupt. The latter solution would, however, increase the worst-case memory usage of all tasks that can be interrupted by an ISR. Portal handler methods use the task-local heap of their caller, which is in a suspended state until the portal call finishes.

3.4 Potential Problems and Limitations

Applying scope extension 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 allo-

cation 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 scope extension, however, all objects are allocated by the caller methods and references are passed for each one, even though only one of the arguments is actually used. 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 in scope extension yet. Since consolidated memory areas might be used for objects of different types and sizes, garbage collectors would have to support uninitialized chunks of memory as method arguments.

Summarizing so far, scope extension can increase memory usage due to the allocation of unused objects and it can cause sub par performance when a large number of allocations are 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 has exactly one 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 an *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 Equation (3.2) shows, the change in code size is dominated by the number of callers c. Note that

Listing 3.3: Scope extension example

```
1  public class ScopeExtExample implements Runnable {
2    public StringBuilder buildString() {
3      StringBuilder sb = new StringBuilder();
4
5      sb.append("Ground control to Major Tom\n");
6      sb.append("Ground control to Major Tom\n");
7      sb.append("Lock your Soyuz hatch and put your helmet on\n");
8      sb.append("Ground control to Major Tom\n");
9      sb.append("Commencing countdown, engines on\n");
10     sb.append("Detach from Station, and may God's love be with you\n");
11     // ...
12
13     return sb;
14   }
15
16   public void run() {
17     StringBuilder sb = buildString();
18     System.out.println(sb.toString());
19   }
20 }
```

Example for scope extension. The *StringBuilder* object allocated in *buildString* is returned into the *run* method. Extending its scope will make it a local object in *run* and subject to optimization.

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. For interference analysis and consolidation, garbage collectors must be adjusted to tolerate chunks of uninitialized memory in method arguments. 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.

3.5 Example

Consider the example given in Listing 3.3 and its Java bytecode representation in Listing 3.4. The *StringBuilder* object allocated in *buildString* escapes its method of creation into *run*. Consequently, its CG object node representation has a *method* escape state due to its incoming edge from the node representing *buildString*'s return value. In the connection graph for *run*, the *StringBuilder* object is is represented as a phantom node. This phantom node has a *local* escape state because it does not escape *run* and is not assigned to any global variables inside *toString*.

When scope extension encounters the allocation, it removes the *new* instruction at bytecode position 0 in *buildString* and creates an equivalent instruction between bytecode positions 0 and 1 in *run*. It also adds an additional parameter to the invocation of *buildString*. See Listing 3.5 for bytecode excerpts from both methods after scope extension. The allocation has been moved into *run*, where its representation in the CG after another run of alias and escape analysis will be *local*. The previous *new* instruction has been replaced with an operation that reads the new parameter. SSA deconstruction removes the unnecessary copy that would occur if the *new* operation was naively replaced with the appropriate *aload* instruction due to the sequence *aload*, *dup*, constructor invocation, *astore*.

The remaining allocation can be turned into a stack allocation or use a task-local heap of the *ScopeExtExample* task. Garbage collection is no longer required for the object previously allocated in *buildString*. Note that the *toString* method of the *StringBuilder* class contains the allocation of a *String* object that escapes its method of creation into *run*. This allocation would also be moved into its calling method, despite being part of the standard library. This behavior is not limited to scope extension, but is a general feature of KESO. All Java library code used in an application is analyzed and tailored just like application code.

Listing 3.4: Bytecode for the scope extension example

```
1  public class ScopeExtExample implements java.lang.Runnable {
2    // ...
3    public java.lang.StringBuilder buildString();
4       0: new           $2  // StringBuilder
5       3: dup
6       4: invokespecial $3  // StringBuilder."<init>":()V
7       7: astore_1
8       8: aload_1
9       9: ldc           $4  // Ground control to Major Tom\n
10     11: invokevirtual $5  // StringBuilder.append:(LString;)LStringBuilder;
11     14: pop
12     15: aload_1
13     16: ldc           $4  // Ground control to Major Tom\n
14     18: invokevirtual $5  // StringBuilder.append:(LString;)LStringBuilder;
15     21: pop
16         // ...
17     50: aload_1
18     51: areturn
19
20   public void run();
21      0: aload_0
22      1: invokevirtual $9   // buildString:()LStringBuilder;
23      4: astore_1
24      5: getstatic     $10  // System.out:Ljava/io/PrintStream;
25      8: aload_1
26      9: invokevirtual $11  // StringBuilder.toString:()LString;
27     12: invokevirtual $12  // java/io/PrintStream.println:(LString;)V
28     15: return
29 }
```

Excerpts from the Java bytecode compiled from the source given in Listing 3.3, generated using `javac 1.7.0_55` from OpenJDK 7.

Listing 3.5: Bytecode after scope extension

```
public class ScopeExtExample implements java.lang.Runnable {
  // ...
  public java.lang.StringBuilder buildString(java.lang.StringBuilder);
     0: aload_1
     3: invokespecial $3  // StringBuilder."<init>":()V
     6: aload_1
        // ...

  public void run();
     0: aload_0
        new           $2   // StringBuilder
     1: invokevirtual $9   // buildString:(LStringBuilder;)LStringBuilder;
     4: astore_1
        // ...
}
```

Excerpts from the Java bytecode compiled from the source given in Listing 3.3 after KESO's scope extension. The allocation has moved into *run*, where it is *local* and subject to optimizations such as stack allocation.

4 Evaluation

In order to determine whether the optimizations implemented in this thesis actually improve compiled programs, its results should be compared to systems generated without the optimizations. A series of criteria can be evaluated to find differences between different runs. Some of them can be determined without running the generated program, such as code size and the number of optimized allocations. Others require measurements at runtime, such as heap memory usage or execution speed.

Any measured application should be as close as possible to real-world usage to yield representative results. On the other hand, any benchmark should produce output that can be easily processed, compared and graphed. Rather than using a series of micro-benchmarks targeting a certain aspect of the system, previous work on KESO used an open source real-time Java benchmark family for embedded systems [Erh11, ESLSP11, STWSP12]. This benchmark is called Collision Detector (CD_x), its Java variant CD_j, and was published by Kalibera et al. in 2009 [KHP+09]. It consists of two main components: (a) an *air traffic simulator* that generates a stream of radar frames and passes them to (b) the *collision detector*, which scans the radar frames for potential aircraft collisions.

The KESO project uses two variants of this benchmark depending on the size restrictions of the target platform. The *on-the-go* variant generates the radar frames in the collision detector task and avoids the overhead of passing the frames between the two components. At the cost of less realism, this modification significantly shrinks the size of the generated binary and reduces the memory requirements. Due to the lower system requirements, this version of the benchmark fits and runs on an Infineon TriCore TC1796 board used for testing.

	TriCore system	Linux system
CPU	Infineon TriCore TC1796 150 MHz CPU, 75 MHz Bus	Intel Core i5 650, 3.20 GHz
Memory	2 MiB internal Flash 1 MiB external SRAM	7817 MiB DDR3 PC1333
OS	CiAO 4c19874	Ubuntu 13.10, Linux 3.11
Compiler	TriCore GCC 4.5.2, Binutils 2.20	GCC 4.8.1, Binutils 2.23.52
KESO	r4072	

Table 4.1: Hard- and software configurations for the benchmarks

The second, larger version of the CD_j benchmark used to test the KESO compiler and its optimization result is called the *simulated* variant. This type runs the air traffic simulator in a separate protection domain and passes the generated frames to the collision detector using a queue. When frames are generated faster than they can be processed (i.e., when deadlines are not met), frames are dropped. Due to heap size requirements and code size, this variant of the benchmark does not fit on the TriCore board. Since runtime measurements in a simulated OSEK or AUTOSAR OS environment are heavily affected by jitter, time-sensitive measurements are only conducted using the *on-the-go* variant.

The test setups consisting of relevant compiler and software versions and system specifications are given in Table 4.1. Measurements of the *on-the-go* benchmark always use the TriCore system, others are built and run on Linux using an OSEK emulation layer. These emulation layers are either JOSEK [SB10] or Trampoline [BBFT06]. All runtime measurements use KESO's *CoffeeBreak* stop-the-world garbage collector for heap memory management.

4.1 Static Results

The number and percentage share of optimized allocations can be used as a compile time criterion for the quality of KESO's optimizations. The higher the number and share of automatically managed objects, the lower the heap load, which possibly reduces garbage collector usage. Figure 4.1 lists the number of stack allocations, task-local heap allocations and the total number of allocations in the CD_j *on-the-go* benchmark. For the number of stack allocations without using scope extension,

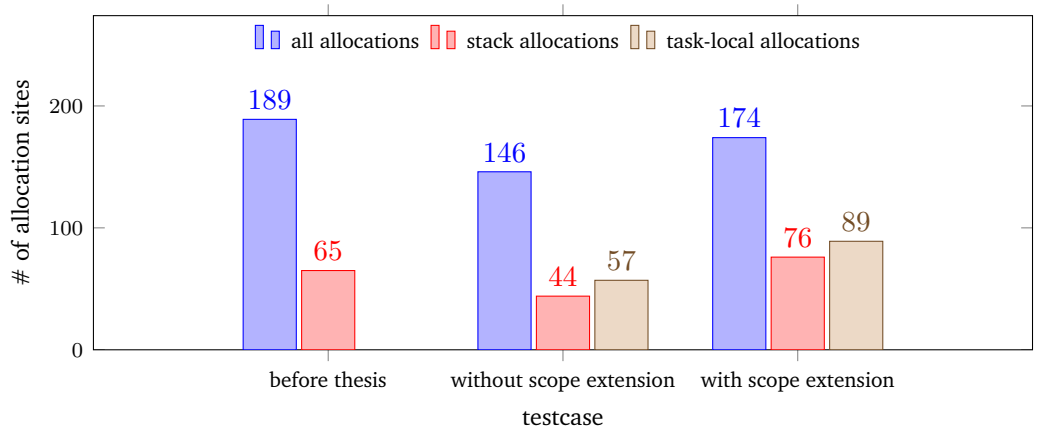


Figure 4.1: Number of stack and task-local allocations in the *on-the-go* CD_j before this thesis and after this thesis with and without scope extension.

the share of optimizations fell from 34.4 % before this thesis to 30.1 %. This drop is caused by the removal of 43 allocations likely due to improved removal of unused fields, which has been added to KESO between these measurements. Using task-local heaps instead of stack allocation increases the percentage of optimized allocation sites to 39.0 %. The 13 additional optimizations are local objects with overlapping liveness regions that are left unmodified in stack allocation to avoid unbounded growth of stack usage. Enabling scope extension in the same measurement adds another 28 allocations created by copying allocation bytecode instructions into multiple callers. This will likely also increase code size (see also Section 3.4). The 28 additional allocations are created instead of 12 allocation sites that are eligible for scope extension. Each of the dozen allocations is thus propagated into 3.33[1] callers on average. The number of stack allocations increases by 32 from 44 (30.1 %) to 76 (43.7 %). Note that these are statically determined numbers, i.e., the actual number of objects allocated at runtime does not change despite the increase in allocation instructions. The number of allocations not converted into stack allocations due to overlapping liveness regions of the allocated objects stays the same. Consequently, the number of allocations using task-local heaps stays at the same margin to stack-allocated ones in comparison to the measurement without scope extension.

[1] $\frac{174 - (146 - 12)}{12}$

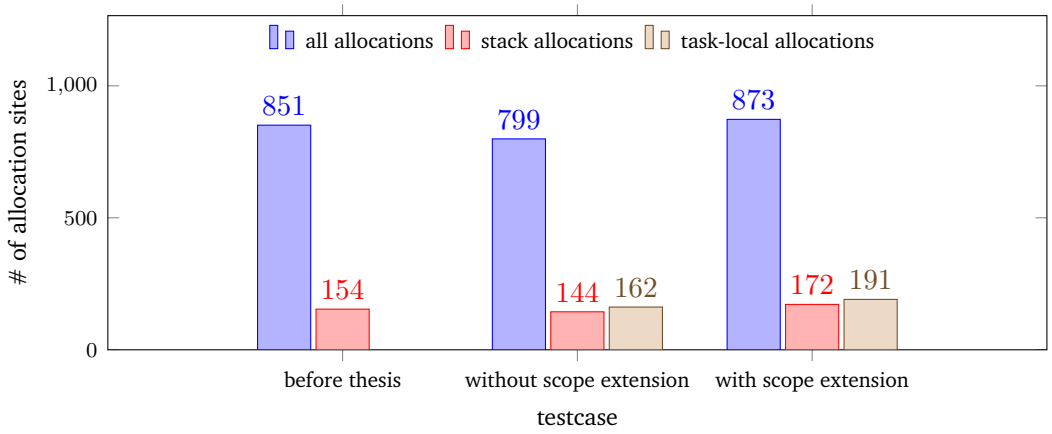


Figure 4.2: Number of stack and task-local allocations in the *simulated* CD_j before this thesis and after this thesis with and without scope extension.

Results for the *simulated* variant (given in Figure 4.2) show similar behavior albeit with lower percentages of optimized statements. These are caused by the much lower share of *local* objects relative to the total amount of allocations. Between the results from before this thesis and those without scope extension, the number of total allocations was reduced again, likely due to removal of unused fields. The percentage of allocations that use stack memory stayed roughly equal (18.1 % vs. 18.0 %). Scope extension increases the total number of allocation sites by 74 to 109.3 %. This should also result in a significant increase of the code size. Another 28 allocations are eligible for stack allocation after scope extension. Again, the number of objects with overlapping liveness regions hardly changes between the measurement with and without scope extension – in the *simulated* variant, it increases by 29 (one allocation more than stack allocation) from 20.3 % to 21.9 % of all allocations.

As expected, stack allocation and task-local heap allocation increase the size of the code. In the *on-the-go* variant shown in Figure 4.3a increases once escape analysis is enabled. This increase is caused by inlining the code that initializes an object's header data. Previously, this initialization was only present in a single place (the allocation function) in the binary. Because stack allocations have been added in multiple places, this initialization code gets replicated and increases the binary size. Additional runtime code further increases the code size. New runtime functions and the explicit creation and destruction of regions at entry and exit points of methods increase the text segment size when task-local heaps are used. As predicted in Sec-

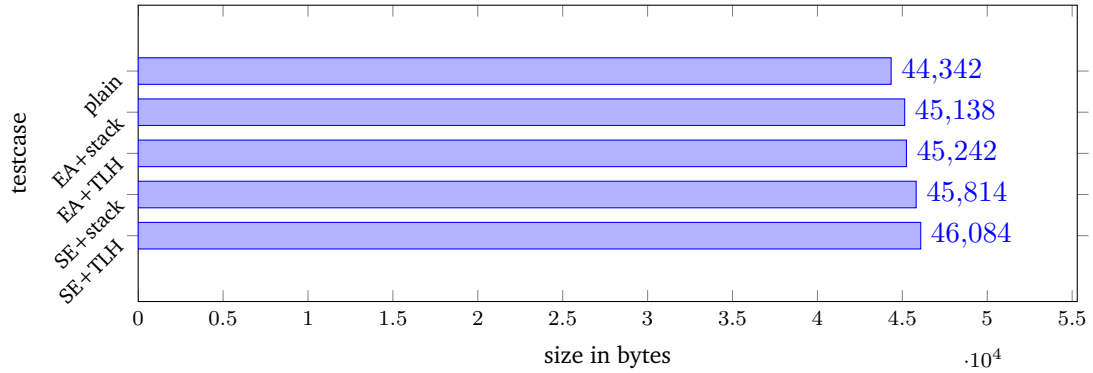
tion 3.4 scope extension further increases the size of the code unless methods with candidates for the optimization only have a single caller. Since the *on-the-go* variant extends variable scope into 3.33 callers on average, growth of the text segment is expected. Overall, the text segment's size increases only moderately to a maximum of 104.0 % compared to the smallest selection.

For the *simulated* CD_j benchmark in Figure 4.3b, code size behaves similar when enabling escape analysis, both with the stack and task-local heap allocation backends. Again, task-local heaps need a little more space, but the growth is small in comparison with the code size ($\leq$ 1.2 % relative to the variant with stack allocation). For the simulated benchmark, enabling scope extension significantly increases code size by up to 20.3 KiB or 9.3 %. Most of the additional allocations are created by a small number of methods (e.g., a generated parser) that, however, allocate a large number of escaping objects. Limiting the number of optimizations per method as suggested in Section 3.4 or other heuristic limits could stem this problem.

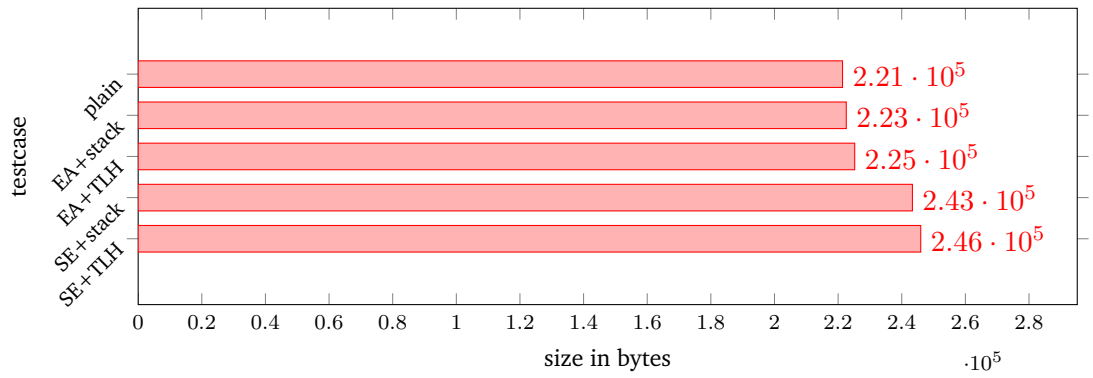
The data segment size does not change for stack allocation. When using task-local heaps, each configured task-local heap adds two additional pointers to the data segment. For the *on-the-go* variant, the size of the data section grows by 24 bytes (the size of two pointers on the 32-bit TriCore target architecture times three task-local heaps). The larger *simulated* variant uses four tasks – its data segment size increases by 32 bytes.

4.2 Runtime Results

While static analysis shows a significant share of allocations is optimized, it is not immediately obvious that the use of automatic memory management reduces the benchmark's runtime. While a reduction in execution time is not imperative because reducing heap memory usage (and with it garbage collector load) is a worthwhile end itself, a large runtime overhead of other automatic memory management method might still make garbage collection the method of choice in all but corner cases. Measurements at runtime will also help determine whether the optimized instructions are located in much frequented code paths or outside the standard control flow (e.g., in error handling code). Hence, runtime data should reveal whether the optimizations improve application behavior and are useful.



(a) *On-the-go* variant



(b) *Simulated* variant

Figure 4.3: Text segment sizes of the CD_j benchmark before optimization (plain), after escape analysis with stack allocation (EA+stack), after escape analysis with task-local heaps (EA+TLH), after scope extension with stack allocation (SE+stack), and after scope extension using task-local heaps (SE+TLH).

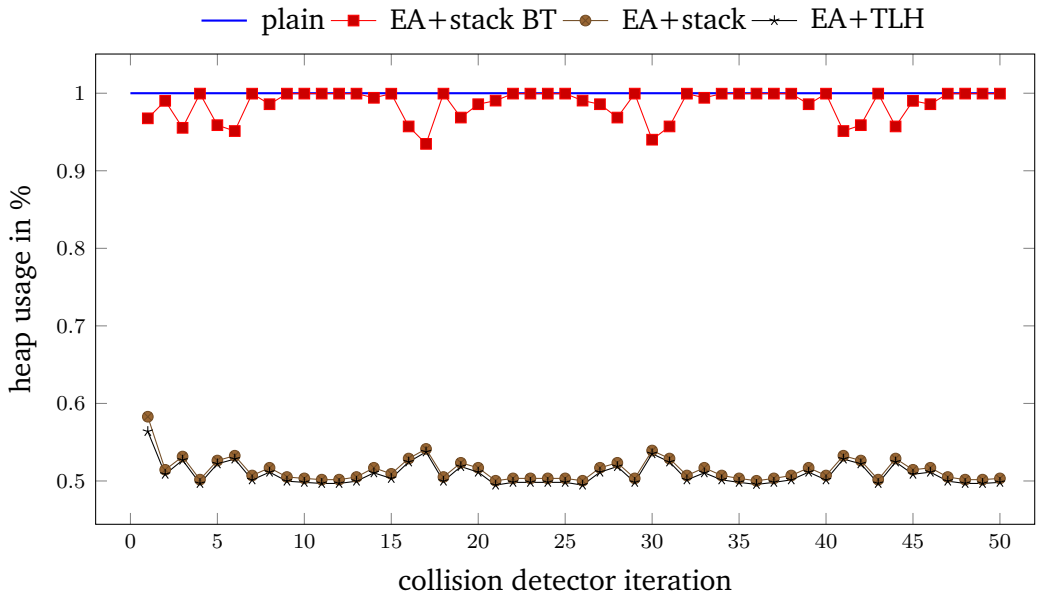


Figure 4.4: Heap memory usage of the *on-the-go* variant of the CD_j benchmark with (a) escape analysis and stack allocation before this thesis (EA+stack BT), (b) escape analysis and stack allocation (EA+stack), and (c) escape analysis and task-local heaps (EA+TLH) relative to a run without escape analysis-based optimizations (plain).

All measurements in this section were conducted on the TriCore system setup as described in Table 4.1. 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. For this reason, none of the plots use error bars – they would simply not be visible.

Figure 4.4 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 backend 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 improvements

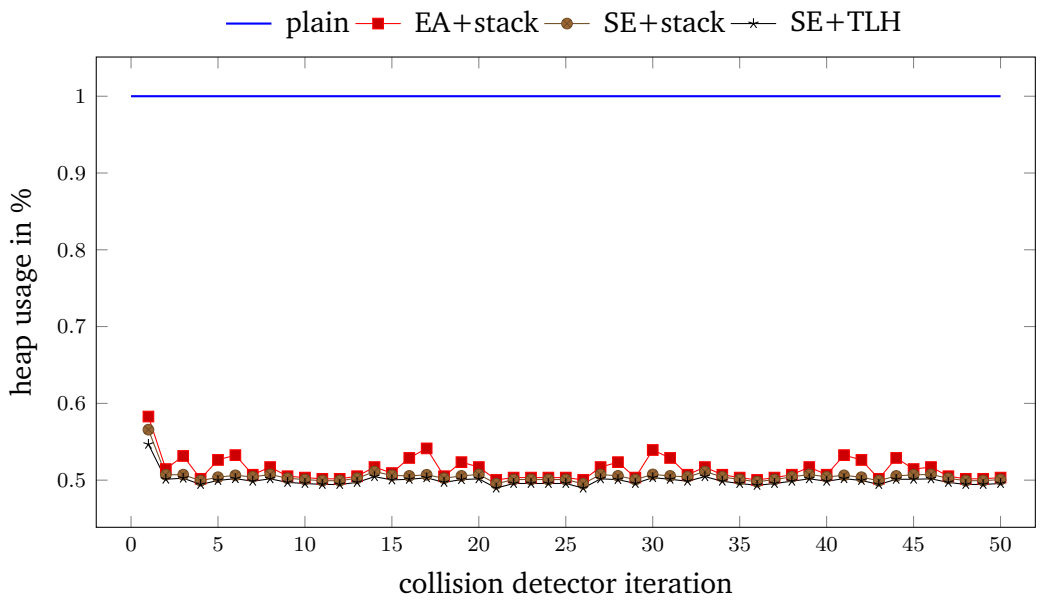


Figure 4.5: Heap memory usage of the *on-the-go* variant of the CD_j benchmark with (a) scope extension and stack allocation (SE+stack), and (b) scope extension with task-local heaps (SE+TLH) relative to a run without escape analysis-based optimizations (plain). For comparison, heap memory usage for escape analysis and stack allocation (EA+stack) as in Figure 4.4 is also shown.

implemented in this thesis (see Section 2.2) heap memory usage has been massively improved from the previous median usage of 99.6 % of the baseline[2].

When enabling scope extension, the fluctuations in heap memory usage present in the optimized variants given in Figure 4.4 are smoothed: In Figure 4.5, the points measured for scope extension have considerably less variation than those of a run with escape analysis only. The median heap usage is reduced to 50.4 % and its standard deviation is reduced from 1.51 percentage points for escape analysis with stack allocation to 0.94 percentage points. When using task-local heaps, the numbers are again similar to stack allocation but a little lower: The median heap memory usage is 49.8 %, the standard deviation decreases from 1.41 to 0.76 percentage points. The lower variance is probably caused by invocations that only occur in some of the collision detector iterations. The invoked methods allocate objects in heap memory.

[2]Measurements representing the state before this thesis were made using KESO r3092. The version of CiAO used was the same across all measurements. Since the baseline was generated using KESO r4072, the EA+stack BT values are not the percentage of improvements that would be measured if the baseline was generated using r3092. Do not use the line to rate the previous implementation. It can, however, serve as a reference to visualize the improvements in this thesis.

These allocations seem to be candidates for scope extension and are hence no longer allocated in the heap when the optimization is enabled.

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 4.6 contains the execution times of three configurations relative to the baseline without optimizations based on escape analysis. Before this thesis (measured using KESO r3092 and a current version of CiAO), the median runtime was 90.5 % of the reference with a standard deviation of 0.79 percentage points[3]. Due to the changes implemented, the same configuration with escape analysis and stack allocation now performs significantly better with a median of 81.1 %. 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 backend. The median runtime improvement for task-local heaps is 13.7 % compared to 18.7 % for stack allocation.

While enabling scope extension further reduces the heap memory usage, the same is not necessarily true for execution time, as Figure 4.7 shows. For the stack allocation backend, enabling scope extension slows down the median time needed by the collision detector by 1.14 percentage points to 82.3 %. The task-local heap backend on the other hand, speeds up with scope extension by 0.59 percentage points to a median value of 85.5 %. The increased time requirements with stack allocation might be another effect caused by over-optimization of pathologic examples as discussed in Section 3.4.

Overall, enabling escape analysis considerably improves the performance and reduces the heap memory requirements. In situations where a danger of stack overflows exists, task-local heaps provide a safe alternative is exchange for a small overhead until KESO supports stack overflow checks or automatic worst-case stack usage and WCET analysis. While the changes implemented for this thesis achieve a lower percentage of objects that are candidates for the optimizations, the runtime

[3] Note that the baseline was again generated with KESO r4072.

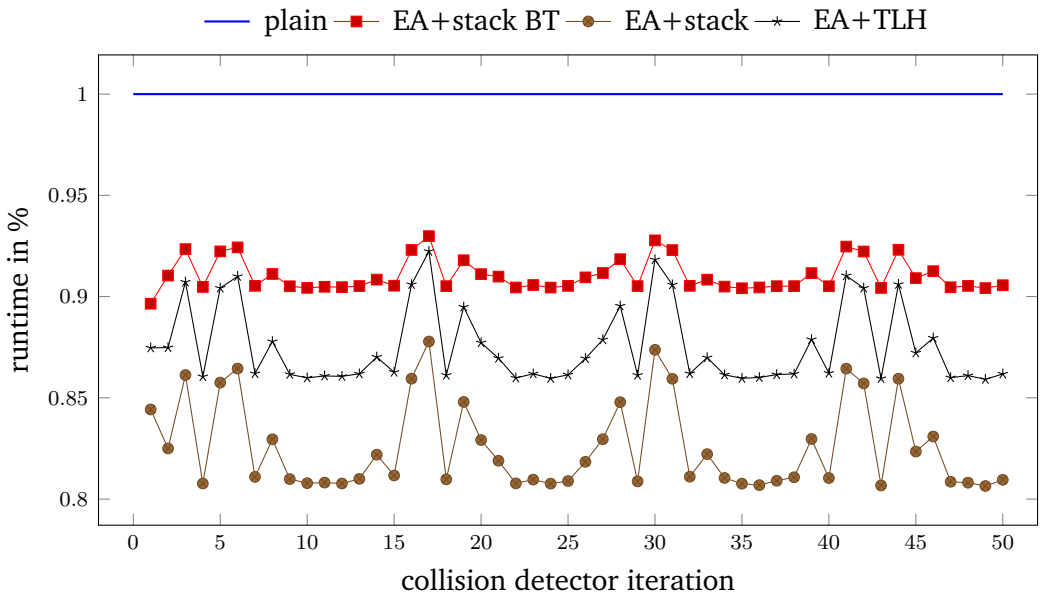


Figure 4.6: Runtime of the *on-the-go* variant of the CD_j benchmark using (a) escape analysis with stack allocation before this thesis (EA+stack BT), (b) escape analysis with stack allocation (EA+stack), and (c) escape analysis with task-local heaps (EA+TLH) relative to a run without escape analysis-based 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.

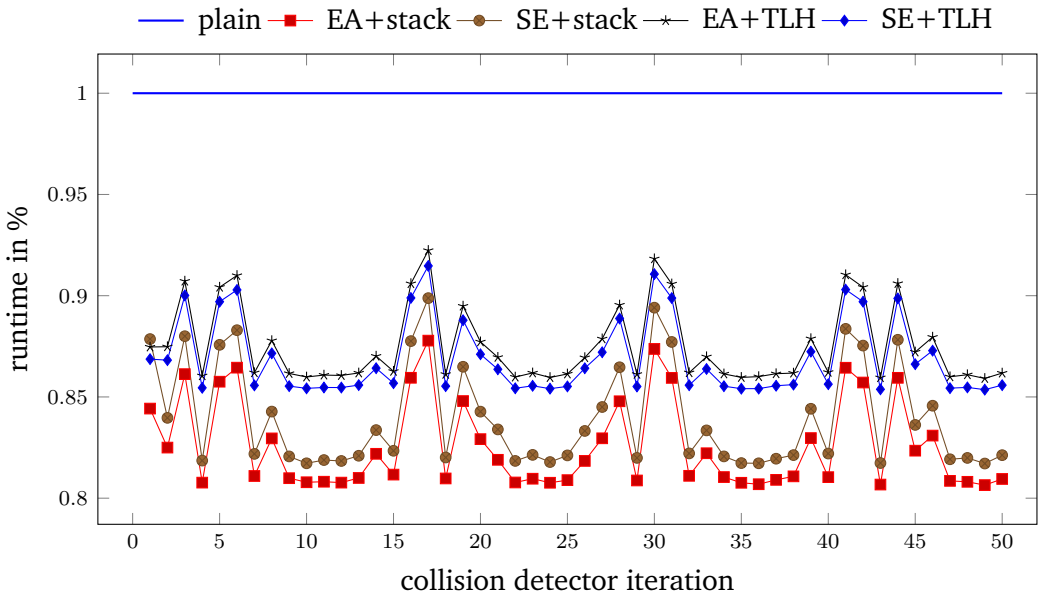


Figure 4.7: Runtime of the *on-the-go* variant of the CD_j benchmark using (a) escape analysis with stack allocation (EA+stack), (b) escape analysis with task-local heaps (EA+TLH), (c) scope extension with stack allocation (SE+stack), and (d) scope extension with task-local heaps (SE+TLH) relative to a run without escape analysis-based optimizations (plain). Time measurements as in Figure 4.6.

results are considerably improved: Execution time is reduced by up to 9.5 percentage points, heap memory usage in collision detector runs is more than halved. It is thus recommended to enable escape analysis and one of its optimization backends for all applications.

A similar suggestion can, however, not be given for scope extension. While it does reduce heap memory usage a little and reduces the variance between the collision detector iterations this optimization comes at the price of slower execution speeds in some configurations. Some of the examples tested expose at least some of the erratic behavior predicted in Section 3.4, for example by significantly increasing the code size. For some applications, the decreased variance that manifested in the *on-the-go* variant of the benchmark when scope extension was activated might increase the predictability of the application. In real-time systems, this might make the optimization worthwhile. Whether scope extension improves an application's behavior should be determined on a case-by-case basis.

5 Related Work

Since the algorithms used in KESO's alias and escape analysis are based on the work of Choi et al. published in 2003 [CGS+03], behavior, results, and features of the analysis are similar. However, different from their work, KESO's compiler avoids resizing a method's stack frame at runtime and offers task-local heaps as an alternative optimization backend to stack allocation. For this thesis, a conceptional flaw discussed in Section 2.2.2 present in Choi's original work was fixed. Several other performance improvements were implemented in Section 2.2.3. Further applications for escape analysis' results, such as removal of unneeded copies in RPCs across protection realm boundaries were added to KESO. Chapter 3 extended the algorithms published in [CGS+03] with allocation of objects in callers' stack frames.

Section 2.2.3.3 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 [Ste96]. Different from Steensgaards 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.

Using escape analysis for automatic memory management solves the same problem as region inference. First published by Tofte and Talpin in 1994 [TT94], region inference has seen widespread adaption in later work by Henglein [HMN01], Grossman [GMJ+02], Hallenberg [HET02], Chin [CCQR04], and Salagnac [SYG05, SRY07] et al. While the initial publication only applied to a call-by-value λ-calculus,

later publications have successfully used similar techniques for Standard ML [HMN01, HET02], safe dialects of C [GMJ+02] and Java [CCQR04, SYG05, SRY07].

Different from [GMJ+02, SRY07], KESO's escape analysis is fully automatic and does not require source code modifications or developer interaction. Salagnac's work attempts to overcome the problem of region size explosion (i.e., region inference placing a lot of objects in the same region that will then not be reclaimed for an extensive period of time) by developer interaction and review. While other region based approaches are also frequently affected by this problem, KESO's escape analysis-based optimizations do not suffer from it because they do not try to avoid garbage collection completely, but rather complement it. If the exact lifetime of an object cannot be determined, the object is allocated in a garbage-collected heap instead of risking region explosion. In fact, [SRY07] gives an example causing region explosion in their approach which would not be converted into stack allocations by KESO due to the overlapping liveness region analysis presented in Section 2.1.2.

Similar to [HET02], the system implemented in this thesis co-exists with garbage collection. Hallenberg's approach was implemented for Standard ML, whereas KESO exclusively uses Java. Both publications by Salagnac et al. [SYG05, SRY07] do not use garbage collectors, stating that garbage collection is generally unsuitable for real-time Java systems, a view the KESO project does not share.

Chin's [CCQR04] only supports a subset of Java called *Core-Java* for their analysis, while KESO does not impose limitations of the source language features[1]. Experimental results provided by Chin et al. are minimal: The largest example has only 170 lines of source code.

All of [CCQR04, SYG05, SRY07] are written with a context of Java usage in real-time systems, which brings them closer to this thesis than other work on region inference. In [SYG05] the goal is to avoid garbage collection entirely, even though the only measurements at runtime still use a garbage collector together with region inference. In this benchmark, the actual region-allocated memory is only about 5 %. The approach also only analyzes a subset of the code used by an application, whereas KESO's compiler always analyzes the complete application and all library methods used by it. Their algorithm is based on work by Gay and Steensgaard [GS00], while

[1] even though it does currently not support catching exceptions, but this is not a limitation caused by escape analysis, and escape analysis does in fact handle exceptions correctly

this thesis uses Choi et al. [CGS+03] as foundation. Salagnac's [SRY07] combines developer-supported and automatic methods and requires annotations to avoid region size explosion. Furthermore, it makes assumptions that have to be verified at runtime [SRY07, Sec. 3.1]. It is also flow-insensitive (even though it operates on code in SSA form, which makes this less of an issue) as opposed to KESO's fully flow-sensitive approach presented in Section 2.2.1.

Molnar et al. [MKB09] propose stack allocation in their CACAO Java virtual machine using escape analysis and use a Steensgaard-based method for escape analysis. Different from KESO's design, their virtual machine uses just-in-time compilation, and escape analysis is done at runtime. Their virtual machine also supports loading code at runtime, which KESO deliberately avoids to improve its optimization output. Molnar et al. handle local objects allocated in loops similar to KESO's overlapping liveness region approach and do not resize stack frames at runtime. They do not support stack allocation of arrays and do not have a generic method to encode the escape information of objects passed to native methods. Finalizers are handled by avoiding stack allocation – KESO currently ignores them, but this could easily be changed. Overall, their approach seems to perform better for some small examples, and on par for larger benchmarks, although a quantitative comparison has not been performed.

6 Conclusion

This thesis strove to improve alias and escape analysis in the KESO Java virtual machine for deeply embedded systems. The escape analysis implementation in KESO's compiler *JINO* – initially based on the work of Choi et al. in 2003 [CGS+03] – was improved to be flow-sensitive and run faster. A conceptual flaw that produced incorrect analysis results was discovered together with a possible solution in Section 2.2.2. Using the information computed by escape analysis, a number of optimizations such as removal of unneeded copy operations for method calls into different protection domains, synchronization optimizations and cycle-aware reference counting were discussed or implemented.

The core part of this thesis concentrates on the application of escape analysis results for automatic memory management. Objects whose lifetime is bounded by the runtime of the method they are allocated in are optimized using one of two mechanisms. Besides stack allocation, this thesis introduced small task-local heaps with very simple automatic memory management as a special case of region-based memory management similar to the *ScopedMemory* class in the Real-Time Specification for Java. Furthermore, allocation in a caller's stack frame or task-local heap region was explored in Chapter 3.

Measurements show that in a benchmark suite for real-time Java systems, up to 43.7 % of allocations are modified to automatically manage the allocated objects without garbage collection. At runtime, heap memory usage is more than halved in some configurations, a huge improvement from previous versions of this analysis. Additionally, execution is sped up by up to 18.7 % compared to the same configuration without optimizations based on escape analysis.

The automatic management of a considerable share of objects reduces garbage collector load. Since the lifetimes of optimized allocations are organized in a stack manner, external fragmentation can be (and is in KESO's implementation) completely avoided. In the context of recent work in KESO concerned with fragmentation-tolerant garbage collection [Str14], reducing the amount of memory that is potentially affected by fragmentation is a welcome side effect. Other work concerned with transient errors and software-based mechanisms to detect and correct them – especially in the garbage collector [Taf14] – also benefit from the optimizations in this thesis.

The alias analysis results in the form of connection graphs have also been used in attempts to improve Java's bad support for constant array data in KESO. Using the alias graph, constant objects have been identified and placed in read-only memory on target architectures that support it [Kuh14].

Future Work

Despite the good results outlined in Chapter 4, a number of further ideas could still improve results or usability of the optimizations. The effectiveness of scope extension is currently limited by virtual method calls, which are not modified. Heuristics that use scope extension in its current form when improvements can be expected and the additional overhead is low have the potential to further increase the share of objects managed without garbage collection. Optimization backends that allow allocation in the memory region corresponding to a calling method would lift the requirement of adjusting all call sites and solve the current problem with virtual method invocations outlined in Section 3.2.2.

As outlined in Section 3.4, scope extension will move allocations from mutually exclusive control flow paths to locations that are not mutually exclusive. Since this increases memory usage and potentially slows down applications, further analysis to identify these situations and avoid them could be useful. Interference analysis, possibly based on Sreedhar's SSA based coalescing using ϕ congruence classes [SJGS99] could identify non-interfering objects – empirical data already revealed this possibility due to a bug in KESO's SSA deconstruction triggered by non-interfering param-

eters created by scope extension. A direct method to allocate objects in a caller's memory region would be an alternative approach to solving this problem.

Objects allocated at one of the optimized sites do not support Java's *finalize* methods at the moment. Since finalizers could add new references to an object, supporting them would require special analysis support. Additional data structures to locate all objects on the stack or in task-local heaps would be required as well as new runtime support code that processes objects and calls the *finalize* methods.

Furthermore, alias analysis could use data flow information associated with references to increase the precision of the results without affecting legality. For example, references that are known to be *null* at a given point in alias analysis cannot be dereferenced. Any fields read from objects apparently pointed to be these references could be ignored – if the reference were actually dereferenced, the program would abort with an exception.

Finally, the minimum size requirements for stack allocation or task-local heaps could be automatically computed in most cases (e.g., in the absence of unbounded recursion or loops). Stack overflows could thus be avoided at compile time and programmers would not have to manually determine and configure task-local heap sizes. This idea could be implemented in parallel with a WCET analysis, because the principles used for both analyses would likely be similar in nature.

7 Appendix

7.1 About the Author

Clemens Lang was born in August 1988 in Lichtenfels, Germany. He finished his Abitur at *Clavius-Gymnasium Bamberg* in May 2007 and interned as web developer at *webDa Medien GmbH* from November 2011 to April 2008. With an awakened interest in computer science, he worked for *Upjers GmbH & Co. KG* before enrolling in computer science at the University of Erlangen-Nuremberg in October 2008.

During his studies, Clemens worked as teaching assistant, instructing undergraduate students in exercises on algorithms and data structures in Java, and system programming for Linux in C starting May 2009 until October 2012. In summer 2011, he participated in the Google Summer of Code program for the *MacPorts Project*, a package manager for open source software targeting OS X systems. Clemens is an active MacPorts contributor and enthusiast to this day.

After finishing his bachelor's thesis [Lan12] and earning the Bachelor of Science degree, he took a research assistant position at the KESO project of the *System Software Group at FAU*. Clemens' tasks included code refactoring, build automation, setting up and administrating continuous integration, gathering and graphing of statistics and implementing new optimizations. This thesis concludes the work on alias and escape analysis started in his bachelor's thesis.

During his time at the University of Erlangen-Nuremberg, Clemens held office as one of two student representatives in the computer science study commission for five semesters. Furthermore, he contributed to the foundation of a fabrication laboratory at the university.

Current contact details can be found on his website `https://neverpanic.de`.

7.2 Source Code Access

KESO is distributed under the terms of the *GNU Lesser General Public License*, version 3. The source code is published in irregular snapshots available for download from `https://www4.cs.fau.de/Research/KESO/#download`. The implementation described in this thesis is not available in any snapshots created earlier than 2014-06-23.

The website also has a documentation section at `https://www4.cs.fau.de/Research/KESO/doc/` that can be very helpful in starting to work with KESO. The "First steps" and "Toolchain" articles are a must-read to work with the analyses and optimizations outlined in this thesis.

Table 7.1 lists the flags for KESO's *JINO* compiler related to this thesis that can be set in the `$JINOFLAGS` environment variable. Some of the source files that have been written or modified for this thesis are given in Table 7.2.

Flag Name	Meaning
`escape_analysis`	Enables escape analysis and, unless `tasklocal_heaps` is set, stack allocation.
`production`	Enable production mode, omits debug strings, reduces binary size. Used for measurements.
`scope_extension`	Enable scope extension. See Chapter 3.
`stack_alloc_stats`	Print statistics about the number of stack allocations and objects' escape states.
`superfluous_portal_copy_removal`	Enable removal of unneeded copy operations in portal calls, see Section 2.3.1.
`tasklocal_alloc_stats`	Print statistics about the number of task-local heap allocations and object's escape states.
`tasklocal_heaps`	Use task-local heaps instead of stack allocation. Requires tasks to have the size of the task local heaps configured using the `LocalHeapSize` property in the KESO configuration file.
`tasklocal_heaps_avoid_overlap`	Avoid allocating objects with overlapping liveness regions from task-local heap memory like it is the default for stack allocation.

Table 7.1: JINO configuration flags used by escape analysis and extended escape analysis and their meaning.

File	Contents
`analysis/EscapeAnalysis.java`	Alias and escape analysis.
`analysis/SuperfluousPortalCopyOptimizaton.java`	Removal of unneeded copy operations in portal calls, see Section 2.3.1.
`transform/ScopeExtension.java`	Scope extension, see Chapter 3.
`transform/StackAllocation.java`	Stack allocation and liveness overlap analysis, see Section 2.1.2.
`transform/TaskLocalHeapAllocation.java`	Task-local heap allocation, see Section 3.3.2.

Table 7.2: Paths in the KESO source code that were written or modified for this thesis. All paths are relative to `keso/src/builder/keso/compiler` in the KESO source tree.

7.3 | Bibliography

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc., second pearson international edition, 2007.

[Ant13] Sebastian Anthony. Android ART: Google finally moves to replace Dalvik, to boost performance and battery life. Nov 2013. `http://www.extremetech.com/computing/170677-android-art-google-finally-moves-to-replace-dalvik-to-boost-performance-and-battery-life`, accessed 2014-05-19.

[AUT06] AUTOSAR. Specification of operating system (version 2.0.1). Technical report, Automotive Open System Architecture GbR, June 2006.

[BBFT06] Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou, and Yvon Trinquet. Trampoline: An OpenSource implementation of the OSEK/VDX RTOS specification. In *IEEE Conference on Emerging Technologies and Factory Automation, 2006. ETFA '06.*, page 62–69, September 2006.

[BBG+00] Greg Bollella, Benjamin Brosgol, James Gosling, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*, 1st edition, January 2000. `http://www.rtsj.org/specjavadoc/mem_overview-summary.html`, accessed 2014-05-21.

[CCQR04] Wei-Ngan Chin, Florin Craciun, Shengchao Qin, and Martin Rinard. Region inference for an object-oriented language. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, page 243–254, New York, NY, USA, 2004. ACM.

[CGS+03] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, November 2003.

[Erh11] Christoph Erhardt. A control-flow-sensitive analysis and optimization framework for the keso Multi-JVM, March 2011.

[ESLSP11] Christoph Erhardt, Michael Stilkerich, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Exploiting static application knowledge in a Java compiler for embedded systems: A case study. In *JTRES '11: 9th*, page 96–105, September 2011.

[GMJ+02] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *(PLDI '02)*, page 282–293, 2002.

[GS00] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *Compiler Construction*, pages 82–93. Springer, 2000.

[HBCC99] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, July 1999.

[HET02] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, page 141–152, New York, NY, USA, 2002. ACM.

[HMN01] Fritz Henglein, Henning Makholm, and Henning Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '01, page 175–186, New York, NY, USA, 2001. ACM.

[Hor97] Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, January 1997.

[KHP+09] Tomas Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek, Ben Titzer, and Jan Vitek. CD$_x$: A family of real-time java benchmarks. In *JTRES '09: 7th*, page 41–50, 2009.

[Kuh14] Simon Kuhnle. ROM allocation of constant data in a JVM for embedded systems. Diploma thesis, University of Erlangen-Nuremberg, Dept. of Computer Science, February 2014.

[Lan92] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, December 1992.

[Lan12] Clemens Lang. Improved stack allocation using escape analysis in the KESO multi-JVM. Bachelor's thesis, University of Erlangen-Nuremberg, Dept. of Computer Science, October 2012.

[Lar14] Frederic Lardinois. Microsoft updates Visual Studio with support for universal projects, TypeScript 1.0 and .NET native code compilation. Apr 2014. http://techcrunch.com/2014/04/02/microsoft-updates-visual-studio-with-support-for-universal-projects-typescript-1-0-and-net-native-code-compilation/, accessed 2014-05-19.

[Lin14] Brad Linder. Goodbye Dalvik? Android code commit makes ART runtime the default. Jan 2014. http://liliputing.com/2014/01/goodbye-dalvik-android-code-commit-makes-art-runtime-default.html, accessed 2014-05-19.

[Max12] Clive Maxfield. 28KB Java virtual machine supports 32-bit MCUs. *eetimes.com*, Nov 2012. http://www.eetimes.com/document.asp?doc_id=1317511, accessed 2014-05-19.

[Mer13] Rick Merritt. Oracle brews Java for the Internet of Things. *eetimes.com*, Sep 2013. http://www.eetimes.com/document.asp?doc_id=1319569, accessed 2014-05-19.

[MKB09] Peter Molnar, Andreas Krall, and Florian Brandner. Stack allocation of objects in the cacao virtual machine. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 153–161, New York, NY, USA, 2009. ACM.

[OSE05] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, February 2005. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, visited 2011-08-17.

[Phi99] Geoffrey Phipps. Comparing observed bug and productivity rates for Java and C++. 29(4):345–358, 1999.

[PZM+10] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: Fragmentation-tolerant real-time garbage collection. In *(PLDI '10)*, page 146–159, 2010.

[Ram94] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.

[SB10] Michael Stilkerich and Johannes Bauer. JOSEK - an open source implementation of the OSEK/VDX API, February 2010. `http://www4.cs.fau.de/Research/KESO/josek`.

[SJGS99] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *Proceedings of the 6th International Symposium on Static Analysis*, SAS '99, page 194–210, 1999.

[SRY07] G. Salagnac, C. Rippert, and S. Yovine. Semi-automatic region-based memory management for real-time java embedded systems. In *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pages 73–80, August 2007.

[SSE+13] Isabella Stilkerich, Michael Strotz, Christoph Erhardt, Martin Hoffmann, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. A JVM for soft-error-prone embedded systems. In *2013 ACM SIGPLAN/SIGBED (LCTES '13)*, page 21–32, June 2013.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[Str14] Michael Strotz. A fragmentation-tolerant real-time garbage collector for the KESO JVM. Master's thesis, University of Erlangen-Nuremberg, Dept. of Computer Science, March 2014.

[STWSP12] Michael Stilkerich, Isabella Thomm, Christian Wawersich, and Wolfgang Schröder-Preikschat. Tailor-made JVMs for statically configured embedded systems. *Concurrency and Computation: Practice and Experience*, 24(8):789–812, 2012.

[SYG05] Guillaume Salagnac, Sergio Yovine, and Diego Garbervetsky. Fast escape analysis for region based memory management. In *AIOOL 2005: Abstract Interpretation for Object-Oriented Languages*, volume 141 of *ENTCS*, page 99–110. Elseiver, January 2005.

[Taf14] Philip Taffner. Design and implementation of a fault tolerant garbage collector for the KESO JVM. Diploma thesis, University of Erlangen-Nuremberg, Dept. of Computer Science, February 2014.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[TSK+11] Isabella Thomm, Michael Stilkerich, Rüdiger Kapitza, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Automated application of fault tolerance mechanisms in a component-based system. In *JTRES '11: 9th*, page 87–95, September 2011.

[TSWSP10] Isabella Thomm, Michael Stilkerich, Christian Wawersich, and Wolfgang Schröder-Preikschat. KESO: An open-source Multi-JVM for deeply embedded systems. In *JTRES '10: 8th*, page 109–119, 2010.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, page 188–201, New York, NY, USA, 1994. ACM.

7.4 List of Figures

7.5 List of Tables

7.6 List of Listings

7.7 List of Algorithms