Automatic Object Inlining in KESO

Bachelorarbeit im Fach Informatik

von

Christian Bay

Lehrstuhl für Informatik 4
Friedrich-Alexander Universität Erlangen-Nürnberg

Betreut durch:

Dipl.-Inf. Christoph Erhardt
Dipl.-Inf. Isabella Stilkerich

Beginn der Arbeit: 01. Dezember 2014
Ende der Arbeit: 31. Mai 2015

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 31.05.2015

Abstract

This thesis describes the design and implementation of co-allocation in KESO, a JVM for static configured embedded systems, as a first step to inline objects completely.
The realization of object inlining can serve many useful features like saving another level of indirection and reducing the fragmentation in memory.
Therefore the work presents how to determine suitable candidates for inlining and defines a new object layout that, for example, cooperates well with the polymorphic behavior of Java.
In the evaluation turns out that great improvements in runtime and size are not achieved yet. Nevertheless the work serves as a basis for complete object inlining.

Zusammenfassung

Diese Arbeit beschreibt einen Entwurf und eine Implementierung von Koallokation in KESO, eine virtuelle Maschine für Java für statisch konfigurierte eingebettete Systeme, als einen ersten Schritt um Objekte komplett zu inlinen.
Die Umsetzung von Objekt-Inlining kann viele nützliche Eigenschaften hervorbringen wie Dereferenzierungen einzusparen und Fragmentierung im Speicher zu reduzieren.
Dazu präsentiert die Arbeit wie man geeignete Kandidaten fürs inlinen bestimmt und definiert ein ein neues Objekt Layout, dass zum Beispiel mit der Polymorphie von Java gut kooperiert.
Die Auswertung der Arbeit ergab, dass bisher keine großen Verbesserungen in Laufzeit und Größe des Programms erzielt werden konnten. Nichtsdestotrotz dient diese Arbeit als gute Basis um das inlinen von Objekten zu vervollständigen.

Acknowledgments

I like to thank my advisers *Isabella Stilkerich* and *Christoph Erhardt* for their great support while writing this thesis.
Special thanks goes to Ulrich, feni, Karin, Oskar, my girlfriend Kess, mum, and finally my lovely cats, Farin and Susi.

Erlangen, May 2015

Contents

1 Introduction

Whenever the alarm clock rings in the morning, the coffee machine starts its grinder, the car tells us the current outdoor temperature and finally the watch informs us about lateness at work, we recognize that embedded systems play an important role in daily life.

Hardware, mostly microcontrollers, is built into every of the above mentioned devices where an embedded system serves tasks by using different hardware parts. The requirements on computer systems and a microcontroller differ a lot depending on their intended use. In contrast to commodity systems, microcontrollers tend to have processors with less power and smaller memory while on the other hand the power consumption of per-unit costs is comparatively low [Erh11]. The use of these systems is not limited to uncritical tasks but they are employed in safety-critical systems as well. Quite the contrary is the case because they often have to fulfill safety-critical applications.

The requirements for embedded systems and notably real-time systems initiated the development of *KESO*, a Java Virtual Machine designed for microcontrollers. The main idea behind KESO is to write applications for embedded systems in the type-safe language Java instead of *C* or *C++*, which are common in this area.

The use of Java in this environment is quite rare, but the decision to use it yields a set of features in contrast to other languages. Object-oriented programming can increase productivity and problems can be solved on a higher level. In addition to the design, also common programming mistakes that lead to buffer overflows within manipulating return addresses or references that point to a wrong position in memory are eliminated by Java's type and memory safety. These features prevent impacts of programming mistakes and encourage stable software.

A difficult problem in embedded systems is the efficient handling of resources. As a consequence it is essential to save memory with a smart management of data structures. Especially in Java, data information may be scattered in memory because the method of choice is by-reference semantics for field instances (see Figure 2), when information which belongs to each other is connected.

In contrast to Java, the programming language C++, which does not offer a strong type system concept, has generalized its object layout by leaving the choice of storing object information to the programmer of the application. The manual declaration of a field determines whether it is saved by-reference or by-value (see Figure 1).

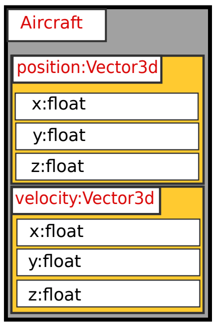

Figure 1: Fields `position` and `velocity` were stored by-value without a reference.

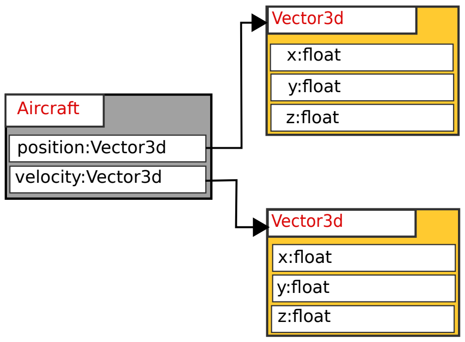

Figure 2: `Aircraft` has only by-reference fields containing the addresses of the corresponding objects.

Especially on embedded systems, it can make a big difference which concept is chosen. Due to the fact that references introduce another level of indirection, several aspects have to be pointed out. The access of a data structure via a reference takes more time than the access without one. Increasing amount of runtime costs causes the requirement for better and more expensive microcontroller.

Another underestimated aspect is the impact of bit flips in references on such devices. Bit flips are in charge for 90 percent of software errors [MIR$^+$11], for instance caused through cosmic radiation [Mak06]. Corrupted references can lead to fatal consequences and in the worst case the system might not be able to continue its tasks and stops proceeding.

One approach to address the above mentioned problems is *object inlining*, which tries to optimize the code by modifying the object layout through storing the content of referenced objects directly in the owner object (as depicted in Figure 1) if possible.

Implementing automated object inlining, which is leveraged by the type-safe language, can address the drawback of Java's by-reference design by applying the by-value approach. Discussing and implementing co-allocation in KESO, which is a first step to automated object inlining, is the topic of this work.

This thesis is structured as follows: Related work is discussed in Chapter 2. The composition of the KESO framework is presented in Chapter 3 focusing on its compiler (Section 3.1) and already implemented analysis (Section 3.2). It follows the implementation of object inlining in KESO (Chapter 4) and finally the evaluation (Chapter 5).

2 Related Work

Java is a well-known programming language with an object-oriented concept. Its design provides the opportunity to write complex software systems by offering, for instance, generic interfaces and a modular API.
The software layout is responsible for an increasing disposition of information which leads to a large number of small methods and objects. Data structures that logically belong together are distributed in memory and connected through references. Because indirections slow down programs at run-time, prior research projects advised that a reduction of references may reduce footprint and runtime overheads.
Object inlining can be described as merging multiple objects with a parent-child relationship. As a result a new object is created combining fields and primitive values.
A very notable work in this context was published by Chien and Dolby [JA00]. For their research they take a language mixing the syntax of C++ and semantics of Java, called *ICC++* [JASJ96]. They examined properties of field references to allow the semantically correct inlining. Their idea is to determine object pairs `(o1,o2)` where:

a) `o1` owns a field `f` pointing to `o2`.

b) Assignment `o1.f = o2` is always the initialization of `o1.f`.

In conclusion, no other addresses are assigned to `o1.f` through its lifetime. They named this pattern *one-to-one field*.
The fusion of the object pair implicates the creation of a new object containing all fields of `o1` and `o2`. The beneficial effect of inlining can be seen by the following valid substitutions that can be made in a program for any parent object `p`, corresponding inlined field `f` with the attribute `n`:

a) Field load: `v = p.f` → `v = p`

b) Attribute access via field load: `p.f.n` → `p.n`

As a result, one indirection layer can be omitted due to inlined objects.
When applying their analysis to several programs (e.g. xpdf, dict, otest) it turns out that in average about 30% of the fields are inlinable. Further measurements end up in the result that inlining has a great impact on runtime costs. On average, execution time is improved by 14%. For instance, 28% of the field reads and 58% of the object allocations are removed.
The work of Laud [P.01] describes object inlining in Java and uses the results of [JA00]. The conditions for a one-to-one field defined by Dolby and Chien

are changed a bit by Laud because of using different languages. A field `f` is allowed to point to more than only one child throughout its lifetime. On the other side each child is only referenced by `f`.

Due to this modification, Laud focuses on how fields can be inlined which can have different types at runtime. Hence different types of objects result in different sizes, a calculation of size is done for each type.

The largest size then determines location and size for the inlined child.

Another related work concerning object inlining in Java was published by Lhotak and Hendren [OL02]. They combined the ideas of [JA00] and [P.01] by specifying a set of properties for field references:

a) *contains-unique* The field `f` points to only one object during its whole lifetime. The lifetime begins with initialization of the field.

b) *unique-container-same-field* No other field `f` of any object points to the contained object. But it is possible that another object field `g` points to the contained object.

c) *unique-container-different-field* Every object referred to field `f` is never referred to another field diverse to `f` by any object.

d) *not-globally-reachable* None of the contained objects will ever be referenced, neither by an array nor by a static field.

In the research of [P.01] and [JA00] a constant object layout is applied. As a consequence a field reference has to fulfill all properties of that approach to become inlined. When a field violates any of those conditions it is marked as not inlinable. These restrictions are lowered by introducing different object layouts depending on the properties for field references defined by Lhotak/Hendren [OL02] (see itemization above). Precisely three different sets of properties are formed. When a field reference can be inlined by more than one strategy, the one with the most fulfilled requirements is chosen.

The three different layouts are the following:

a) A *simply one-to-one field* fulfills all conditions which are listed in the itemization above. The field reference is eliminated and the two objects are merged into one (see Figure 3).

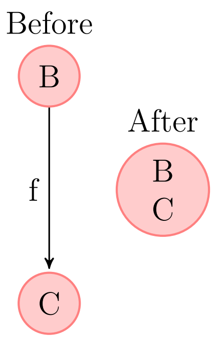

Figure 3: After the inlining procedure a new object is created, which contains all information about `B` and `C`.

b) A so-called *field specific one-to-one field* does not fulfill the *unique-container-different-field* constraint (see itemization above). As a consequence the new valid layout is a product out of two inline procedures (see Figure 4).

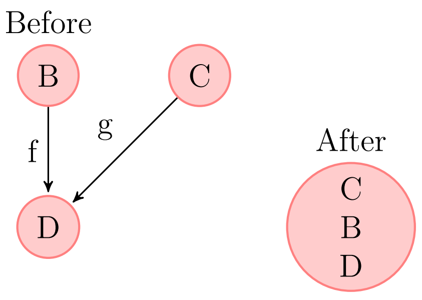

Figure 4: Object `D` is referenced by two objects different fields. The new object is created by first inlining `D` into `B` and afterwards into `C`.

c) The last presented case for object inlining occurs when an inlinable field `f` points to multiple objects during its lifetime and therefore does not fulfill the *contains-unique* constraint. The resulting layout keeps all involved parties alive. The container object inlines several objects during its lifetime. Whenever a new object gets assigned to `f`, each object information that belongs to the former object is deleted, while the new object gets inlined (see Figure 5).

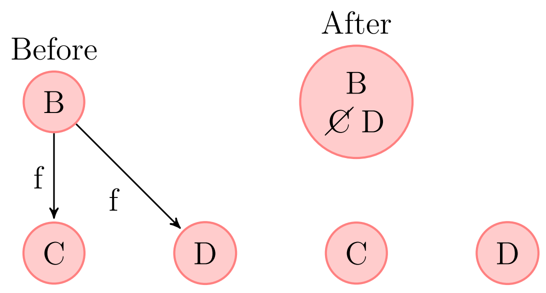

Figure 5: The field reference `f` points to several objects during its lifetime. The inlining procedure keeps all objects and inlines information only temporarily.

Hence it is possible for Lhotak/Hendren to inline the union of candidates provided by [P.01] and [JA00].

In contrast to the above presented related work, this thesis differs in some points. The chosen object layout, which is explained in Section 4.1, can be described as a co-allocation. The inlined object still exists as a object with its header and can be treated as a usual object. Furthermore the object inlining procedure is implemented in a framework, called KESO (introduced in Chapter 3). While the related works only change bytecode of the language own compiler, it is necessary to produce a compatible backend code in KESO's compiler jino (see Section 3.1.3, 4.3.2 and 4.6).

3 KESO – A Multi-JVM

The KESO system is a multi-JVM for statically configured embedded systems. The main idea behind KESO is to write applications for embedded systems in Java. As a consequence of the type-safe language and the logical separation of the software components' global data, memory protection can be ensured constructively. Dedicated hardware support provided by the microcontroller is not necessary [Waw09].
Moreover KESO does not use *just-in-time* (JIT) compilation. The procedure of a JIT compiler, also known as *dynamic translation*, can be very complex because it compiles program code during execution time. Higher latencies at runtime can be the consequence and are not recommended in embedded systems.
Instead of a JIT, KESO offers *ahead-of-time* (AOT) compilation [Lan12] by using the *closed-world assumption*, that the entire source code and configuration of the application and system components are known at compile time. Additionally the complete software configuration (e.g. size of the heap) is performed at compile time. Hence it is possible to create a slim and efficient runtime environment for Java applications. Thus, the compilation will be performed only one time before execution. Therefore it is not possible to load code dynamically or to use Java's reflection mechanism.

3.1 Structure of JINO

Since KESO has a pass structure it is reasonable to implement object inlining as a pass, too. As a consequence the environment of passes in KESO are explained. Before KESO's own compiler jino starts its work, the Java compiler *javac* translates Java sources into common Java byte code.
After that jino starts the translation from Java bytecode to C. As mentioned in Erhardt's work [Erh11] jino's architecture is inspired by the design of state-of-the-art compilers and comprises a frontend, intermediate code passes and backends. These three modules are explained next.

3.1.1 Frontend

The frontend operates on Java bytecode. Depending on the resulting bytecode an intermediate code representation will be created. Therefore every `.class` file gets parsed to extract its contents and finally saving it in a provided class repository. Precisely each method is stored and is divided into *basic blocks*. A basic block is a sequence of instructions without jump or conditional instructions.

Each instruction type has a corresponding intermediate class type within its properties and inheritance. For each occurrence of an instruction type, a new object in the intermediate representation is generated.
Additionally, jino introduces stack slots for local variables. Each result of an instruction will be saved in such a stack slot. The design of the rebuilt byte code is a stack machine, where every assignment is saved into a syntax tree. The nodes represent instructions and edges serve as intermediate results for operands.

3.1.2 Middle End

The middle end in jino uses the intermediate code to analyze and transform the code by applying passes. The quality of the optimization results profits from high-level program information and the static nature of the system.
A few examples of passes already implemented and relevant for this thesis are the following:

- Dominance Analysis: Section 3.2.2
- Escape Analysis: Section 3.2.3
- Control-Flow-Sensitive Analysis: Section 3.2.4

Every pass is located in an own class and either enabled or not. A pass can be activated in the configuration or is automatically included due to dependencies from other passes.
All passes are registered in the pass manager, which is inspired by the design pattern of the Low-Level Virtual Machine (`http://llvm.org`) [Erh11].
The manager schedules an execution order of all passes. The execution order is important for the following reasons [Erh11]:

- A pass often has dependencies to others, e.g. data structures or results will be reused. As a consequence, these passes should be evaluated before, e.g. the Escape analysis will need the Domination Tree analysis.
- An instance might run again when another pass transforms code in any manner or, when an analysis yields new optimization potential. This can be imagined as a fix-point iteration.

3.1.3 Backend

As the last step, jino translates the intermediate code to C and creates a configuration file for building a custom kernel based on the chosen backend. Producing C code allows to use existing C compilers to generate machine code. Hence, KESO supports all machine architectures for which a compiler exists by design. As a consequence jino creates memory-safe C code that contains the entire runtime system suited for the translated Java application. The concrete translation to C code begins with an iteration over all expression trees in every basic block. Every node in that tree has a class type in the intermediate code.
Therefore a specific `translate` method exist which emits suitable C code. Depending on the options (enabled passes) the produced code will change.

3.1.4 Summary

For realizing the Object Inline analysis it is necessary to change code in the middle end and the backend:

Middle end: New analysis pass has to be added for collecting necessary information to determine when it is even possible to inline objects and mark respective allocations in code.

Backend: An extension of the object layout is necessary and the code which translates allocation instructions has to change.

3.2 Relevant Analyses Provided by KESO

KESO's pass model is already introduced in Section 3.1.2. The mentioned passes have a certain execution order because of existing dependencies to each other.
This section points out analyses which affect the Object Inline analysis. Either they offer reusable results or change the intermediate code in a certain manner that affects the analysis in some way.

3.2.1 Method Inlining

As described in [Erh11], method inlining is a substitution of a method invocation with its body. From a performance point of view it provides a speedup by omitting the need of a method call overhead which results from writing return address and parameters on the stack and the following cleanup. In addition, method inlining supports other optimizations: When parameters

can be analyzed at compile time, for example by the use of constants, some basic blocks that depend on a specific value of a given parameter cannot be reached and are therefore removed. However, it is not always recommended to inline methods. A naive realization would lead to a blowup in code size especially if a method invocation appears in many code locations. Therefore a heuristic decides when inlining is a suitable option. As a consequence of method inlining whole classes can be inlined. Creating very few instances of a class down to one makes constructor inlining more probable.
The Object Inline analysis has to deal with the behavior of method inlining, which implicates a movement of methods and could furthermore cancel whole classes in the backend representation. The consequences of method inlining will be discussed in detail in Section 4.2.

3.2.2 Dominance Analysis

Section 3.1.1 mentioned that methods are logically divided into basic blocks by jino. A characteristic of a basic block is that it has only one entry and one leaving point and in turn each basic block is either traversed completely or just not entered. In some cases it is useful to know whether a certain basic block `A` always gets traversed before another one `B`. If this evaluates to `true`, a *dominance* is existent and thus `A` dominates `B`. Whenever the basic block `B` is traversed, the basic block `A` has to be traversed to a former point either. Depending on the flow graph of a method, the algorithm see [TE79] of this analysis can assert whether a dominance is existent or not.
In Section 4.6 the question about the order of allocation is answered with the help of the domination tree. The domination tree is characterized by mapping all dominance relations between the basic blocks into a tree structure.

3.2.3 Escape Analysis

A very important pass the Object Inline analysis depends on is the *Escape Analysis*, implemented by Clemens Lang in his bachelor thesis [Lan12]. It determines if an object can be allocated on the stack. Because (de-)allocations can be done much faster by stack allocations, its implementation is helpful. A tracing garbage collector for heap objects that are not referenced by the application anymore causes a higher runtime overhead.
Since Java does not provide any keyword for explicit stack allocation[1], it is necessary to figure out when a pointer, received via an allocation operation, leaves the method context. For each object, that exists only in a method context, stack allocation is done.

[1]Nevertheless stack allocation is sometimes done by the compiler itself.

For realizing this behavior, a data structure called the *Connection Graph* is introduced. For each method, it stores *alias information* about references, where alias information means memorizing all targets of references through their lifetime. The internal data structure is a directed graph with the following attributes:

Vertices:

- An object node is illustrated by a vertex and represents an instance of a class. It is very important to keep in mind that the Escape analysis is a static and not a dynamic analysis. Therefore an object node is created only for each allocation statement in the code instead of every allocation made at runtime. For instance, if the same allocation is called multiple times, the analysis does not create more than one object node. An important subtype is the *phantom node*. That node is usually created when a reference points to an object where the analysis cannot determine where it is created. This happens mostly when it is created out of the currently analyzed method. Ergo reference nodes with edges to phantom nodes are not suitable for stack allocations.
- Reference nodes connect object nodes and represent the different kinds of references:
  - **Local References**: For instance, references saved in slots in Java byte-code.
  - **Field References**: For each non-primitive member variable a field reference node is created. They are connected with a so-called **field edge**. The field reference itself has an edge of type **points-to edge** to another object node.

Edges:

Edges have also different types depending on their source and destination node:

node type	object node	field reference node	reference node
object node	-	field edge	-
reference node	points-to edge	deferred edge	deferred edge

Table 1: Edge types depending on source (downwards) and destination (sidewards). This table is taken from [Lan12].

The results computed by Escape analysis are beneficial for object inlining, since it is easy to figure out any kind of references with corresponding target. With the connection graph it is possible to identify field references that are stack-allocatable by traversing the connection graph of each method.

The concept of connection graphs is demonstrated by using the following small example:

```
public class Test implements Runnable {
        private E e;

        public Test() {
                e = new E();
        }

        public void run() { ...  }


        class E {
                private final F f;

                public E(){
                        this.f = new F();
                }
        }

        class F {
                ...
        }
}
```

Listing 1: Simple code example for connection graph

Relevant in Listing 1 are the two classes `E`, `F` and in particular the field reference of `E` to `F`. This is actually an example where inlining the field `e.f` is a good choice. The example is reused and explained in Section 4.2. A fragment which skips relevant parts of the internal connection graph of the constructor `Test`'s can be seen in Figure 6.

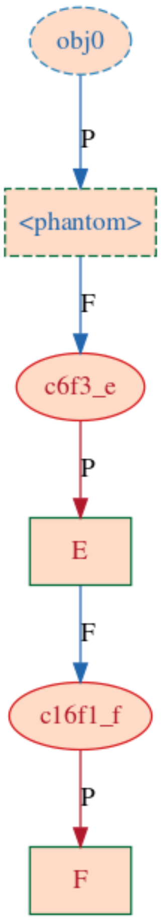

Figure 6: Connection graph corresponding to the constructor of class `Test` in Listing 1.

Each edge has an index which officiates as a shortcut for the corresponding edge type listed in Table 1. An edge marked with `F` stands for the existence of a field reference which belongs to the source object node. Similarly `P` is a shortcut for a *points-to edge* that illustrates the assigned object to the field reference.
All field references of class `E` are registered, namely `c16f1_f`. Every *points-to edge* emerged from a field reference node sticks to an assignment operation to this field. These edges point to an object node which represents the dynamic class type.
Additionally, it is worth mentioning the sense of a phantom node. It is usually created whenever a reference is transferred by a parameter and the referenced object is created before the method invocation. Besides those occurrences

each graph has at least one phantom node. It represents a class instance which holds the current object. The node `obj0` holds the corresponding `this` reference.

3.2.4 Control-Flow-Sensitive Analysis

KESO's Control-Flow-Sensitive Analysis is explained in [Erh11]. It gathers a lot of information about the intermediate code and tweaks it with several transformations, e.g.:

Constant Folding:
Tries to find variables that behave like a constant and substitute the variable for that constant value. The *folding* expression bases upon the fact that one substitution could cause other ones.

Virtual Method Invocation:
Whenever an object that invokes a method can have multiple types[2] at compile time, a virtual call will be made by determining the object type at runtime. This transformation tries to find out the concrete type at compile time to convert virtual calls into non-virtual ones.

Dead Code Elimination:
Eliminates unused code pieces, e.g. by evaluating conditions of if-else clauses and deleting branches which are never entered.

Analyses modify and optimize code in a continuous manner. For this reason the final state of the Control-Flow-Sensitive analysis is determined either by a maximum iteration time or by a fixed-point analysis.
By executing these transformations a lot of information is gathered about every instruction node in the intermediate code. Besides value ranges of primitive types every dynamic type for each field reference is determined as specifically as possible.
In Section 4.5 the analysis helps to determine the basic block of an instruction stored in the intermediate code.

[2]Typically named polymorphism.

4 Object Inline Analysis

In this section an algorithm to inline objects in KESO is presented. Therefore, at first a new object layout (Section 4.1) is established, followed by the task to evaluate the new size of those objects (Section 4.3) correctly. Later on, it is shown that the object layout decision affects the allocation order of inlined and containered objects. As a consequence it is necessary to take care of the allocation order (Section 4.5).
The field properties to fulfill to become a candidate for inlining are presented (Section 4.2) as well as the decision to leave mutually exclusive objects untouched (Section 4.4).
To complete the analysis, the translation of allocations is presented as the last step in Section 4.6. Besides final solutions, neglected ideas are also mentioned in this chapter, because they belong to the process of writing the implementation.

Definition: Inline (field) reference
Each reference field that points to an inlinable object is called inline field reference in this thesis.

4.1 Layout

As seen in Section 2, a part of the object inlining process is re-engineering the object layout. It is important to change it in a manner to keep variable code constructions compatible.
Therefore, KESO's current object layout is scrutinized. In the aftermath its representation in C and diversification of both levels, in case of inlining a field, are examined.

4.1.1 KESO Object Layout

The current structure of a KESO object, which is depicted in Figure 7, is classified into three different sections:

- **Object header**: Contains a class id and further object-specific information, and separates field references from the primitive values.
- **Field references**: Pointers to other objects.
- **Primitive values**: Those types are no objects and are stored directly without any indirection (e.g. float, int, char . . .).

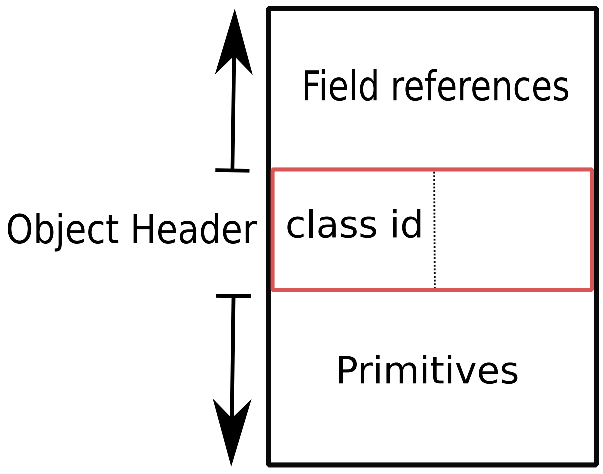

Figure 7: KESO's object layout separates all field references from the primitive values while the object header is stored in between.

The set of object members is split into field references and primitive values. Those two fractions are placed in the order of their class membership, descending in the class hierarchy. As a result, members which belong to upper classes are always placed nearer to the object header in comparison to subclass members. Such a decision enables the possibility to use the same object header in an upper class context.
Pointers always direct to an object header instead of its de facto beginning. For holding type information of an object, its header owns a unique type id, which is also called class id. The id is used as an index in a global *class store*, which holds information about each class's object size, interface and number of field references which are important to know to access the beginning of an object.

4.1.2 Object-Inline Layout

At first the properties for a new object layout have to be determined so that it works fine in the KESO environment.
The layout should still work harmonically with the usual layout and not violate the partition schema of field references, object header and primitives.
On the other hand, the object layout should consider Java-specific features such as inheritance especially if an inlined field could have multiple dynamic types.
Every modification of the object layout needs an equal representation in the generated C code.
As a consequence, the decision was made to keep the inlined objects' header

rather than to erase the header and to embed only the object's content (see Figure 8).

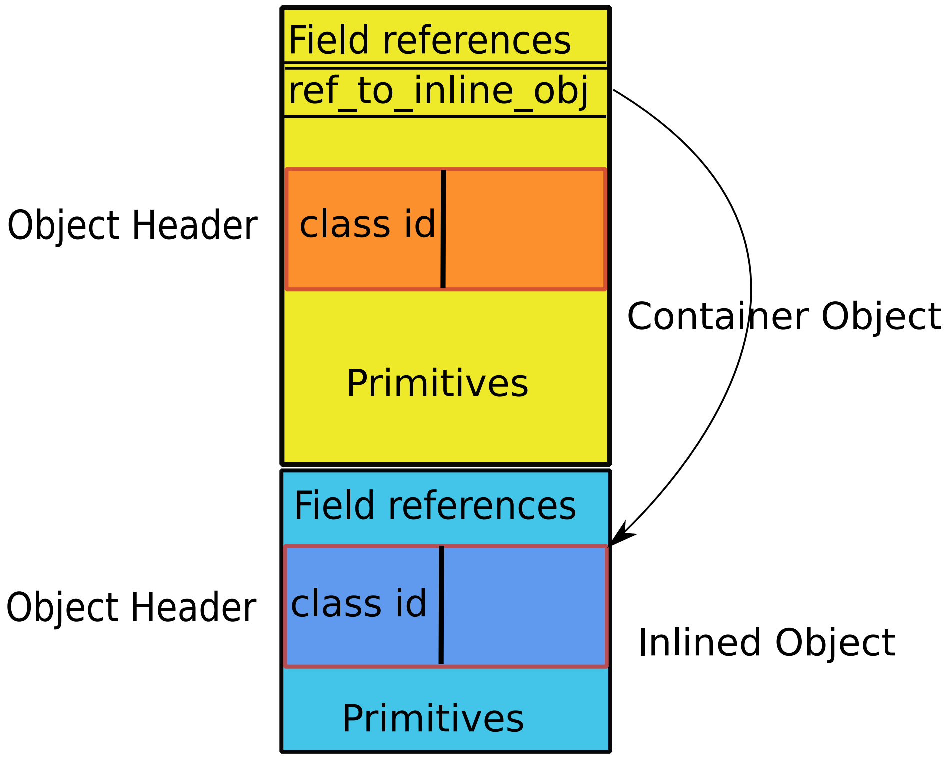

Figure 8: The layout of objects with inlined fields changes neither the size of the container nor of the inlined object. Instead, the inlined object is placed directly behind its container object, whose inlined field stores the address.

It is more like gluing objects together than actually inlining into each other. The inlined object with all of its members is placed behind the container object while keeping its structure. Over the field reference the inlined object is still reachable. Compared to other inlining strategies the container object still has the same size thus only one field reference address has changed. The preservation of the header spares out the task to create new class ids.
There are a few more reasons why this way was favored.
As mentioned earlier, jino is an ahead-of-time Java to C compiler. Jino optimizes code on a high level, while the low-level compilation task is done by a C compiler and therefore jino's last translation step produces ordinary C Code (see Section 3.1) where a Java class is represented as a C structure. By keeping the child object alive, it is not necessary to create new class ids for the container object. An id for a class variant with and without inlining would be necessary instead.

Furthermore the copy-in approach increases the object size for the complete amount of its lifetime, because it might be difficult to determine how long the child object was referenced before. Usually the garbage collector recognizes when no reference to an object exists anymore, but when inlining the object completely and removing its header this opportunity is missing. Since the child object in our approach still exists collecting it is not difficult.

4.1.3 Layout in C Code

Object instances, including inlined ones, are translated to equivalent C code.

For each class a header file is created including a C structure with its members. In Listing 2 such a `struct` construct is displayed. The original class in Java has three attributes including two pointers and one primitive value. In Java, the class definition works as a stencil for each object, which is exactly the same behavior represented by the resulting backend `struct` in C.

```
typedef struct {
    object_pointer c2b1_b;
    object_pointer c3c1_c;
    OBJECT_HEADER
    jfloat c1a1;
} c1_A_t;
```

Listing 2: Common layout produced by KESO's backend. Tripartite design by splitting references, object header and primitives. The macro `OBJECT_HEADER` defines the header.

According to the object header presented in Figure 8, the new concept for the backend has to be constructed by holding information about both objects, that is the container and the inlined object.

A little example with two classes is displayed in Figure 9. Class `Circle` has an inlinable field reference to an object of class `Point` representing the midpoint of the circle.

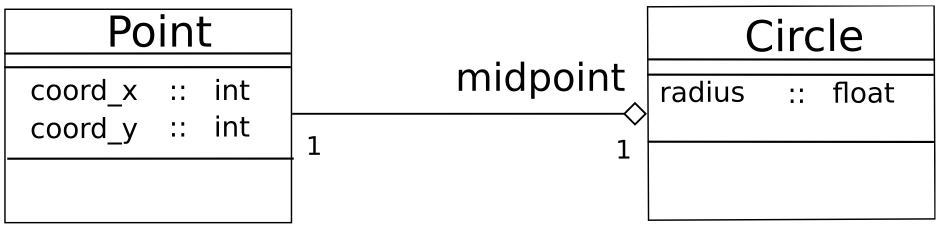

Figure 9: A simple class diagram which demonstrates a typical use case for inlining. The `Circle` class acts as a container object because its field reference `radius` holds a unique object.

The expanded layout in Listing 3 first itemizes the `struct` of both classes[3]. Then the final layout is created which contains an attribute of each `struct`. This placement implicates the desired order of the former declared object layout.

```
typedef struct {
   object_pointer midpoint;
   OBJECT_HEADER
   jfloat radius;
} circle_t;

typedef struct {
   OBJECT_HEADER
   jint coord_x;
   jint coord_y;
} point_t;

typedef struct {
   circle_t circle;
   point_t point;
} circle_inlines_point_t;
```

Listing 3: The layout represented in the backend code when inlining was done referring to the class diagram from Figure 9. Therefore a new `struct` is introduced that holds the `struct` of `Circle` and `Point` and serves as a new draft for the `Circle` class.

[3]Since `Point` has no field references, the first entry in its C structure is the object header.

4.1.4 Conclusion

The chosen layout refuses to remove the object header of the inlined object and inserting its values in the container object. Instead a co-allocation approach is done by ordering, whenever inlining is possible, the object pairs successive in memory.

4.2 Candidates for Inlining

It is still unanswered how the concrete candidates for inlining get identified and how the information will be saved in a sensible manner.
Firstly, the properties of field references for inlining have to be formulated. An important requirement on a field reference, whose object is a possible candidate for inlining is, that it must not point to any other object than the inlined object. Otherwise object inlining would make no sense since the reference points to various objects during its lifetime. That would corrupt the complete design.
Fortunately, Java offers the keyword `final` in its syntax. Marking a reference `final` causes the reference to point to the same object forever. Therefore it is a condition for any reference, which is about to be inlined, that it is declared as `final`. The collected final references pose as the initial set to which the analysis is applied.
It does not matter if multiple references point to an inlined object as long as multiple inlining does not cause problems, as can be seen in Section 4.4.

4.2.1 Relevant Methods

Next, it is necessary to find all methods that contain assignments to inline references. This knowledge is essential for the evaluation of the corresponding connection graphs as can be seen in Section 4.3.2.1.
The precondition that references are marked as `final` facilitates the search. Those references become one-time initialized in the constructor and never change throughout the program's lifetime. Method inlining complicates the search (see Section 3.2.1) because methods can inline constructors and consequently hold the assignments to inline references, which are originally stored in the constructor method. This behavior forces to expand the search in each method that invokes a constructor. As a consequence every method of every class has to be visited.
For filtering out each method that assigns an address to an inline reference it is necessary to traverse all methods with their basic blocks and expression trees. The storage instruction which stores a value into an inline field

reference serves as an indicator for a successful search.
In addition a relevant method must hold at least one allocation operation whose result must be assigned to a candidate. As a consequence the connection graph referring to this method must not include a phantom, that has a field reference node, which is also a candidate as predecessor.
Some candidates (see Figure 10) point to a phantom node and this causes disqualification for inlining. Whenever a field node has a phantom node as successor it indicates that the assigned object is already created in an existing method context.

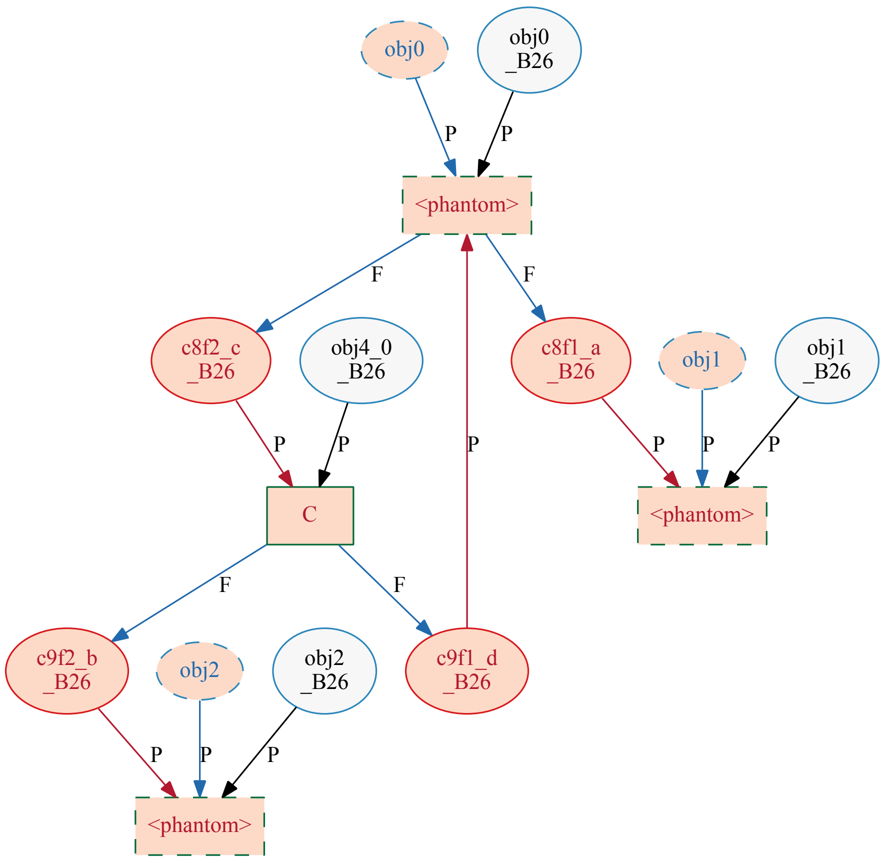

Figure 10: The connection graph of a method holding two field references that point to a phantom node instead of a normal object node.

Naturally, it is not sensible to inline such a field, because the referenced object is already stored in memory.
The yet performed steps to identify inlinable objects summarized: At the beginning of the Object Inline analysis all final references are collected. Af-

terwards all methods with an assignment to these references are searched for. For each method a corresponding connection graph exists. With these connection graphs it is possible to determine all dynamic types of the references. Finally the gathered information is saved in the owner class of the reference. In more detail every instance of `IMClass` saves its inlinable fields in a map. Every field maps to a set of classes. Each class possibly poses a dynamic type that is determined by the connection graph. Algorithm 1 displays the coarse-grained procedure.

Algorithm 1: Gather information about inlinable field references

```
Input : All classes of a program containing final references
Result: Storage of inline candidates with possible types in class
1  findInlineRefWithType($clazzes$ :
   All classes used in program, $finalRefs$ :
   All references marked as final)
2  begin
3      // Find for every field reference corresponding method with
          assignment
4      $Map < IMFieldReference, IMMethod > fieldsAssigned =$
       findAllAssigningMethods($finalRefs$);
5      foreach $entry \in fieldsAssigned$ do
6          $field$ = entry.getKey();
7          $method$ = entry.getValue();
8          // Get corresponding connection graph
9          $cg$ = method.getCG();
10         // Traverse connection graph for dynamic types
11         $types$ = cg.findDynTypesInCG($field$);
12         // Add field with its type to holding class
13         field.getClass().addInlinedFieldWithType($field, types$);
```

The next Chapter 4.3 deals in detail with the question of how to determine the size of an inlined object while considering its polymorphic behavior/characteristics.

4.3 Object size

Listing 3 points out that inlining works by concatenating two or more structures to a single new one. This requires determining the concrete type of the inlined object. One of Java's strengths is the inheriting principle. Class hierarchies are built from generic to ever more specialized classes. Thus it is common to declare more general types of references because any subtype can be assigned to it.

The class diagram depicted in Figure 9 illustrates that whenever an additional subclass is included it is necessary to determine the correct dynamic type as is illustrated in Figure 11.

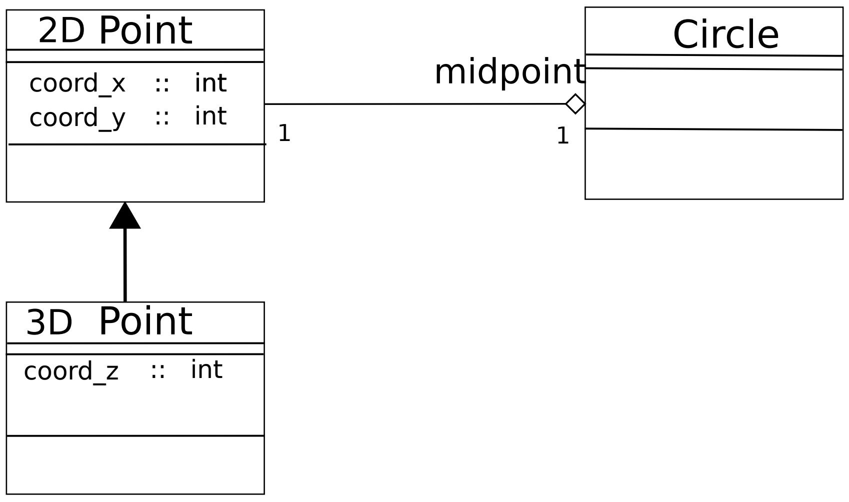

Figure 11: Expansion of the former class diagram (depicted in Figure 9) by an additional subclass of the `Point` class. Thus the inlined field reference can point to different types.

The problem arises when considering a typical initialization procedure via a constructor: Static analysis cannot figure out which object type the field will have. For situations such as illustrated in Listing 4 it is quite difficult to obtain all necessary information to know the right instruction path and consequently the correct size of the inlined object. Subclasses can be larger than their corresponding upper class tree, because of holding additional fields and primitive values.

```java
class Circle{
    private Point midpoint;

    public Circle(boolean is2D){
        if(is2D){
            midpoint = new Point2D(0,0);
        } else {
            midpoint = new Point3D(0,0,0);
        }
    }
}
```

Listing 4: Upper class has multiple constructor paths. In an AOT compiler it is mostly hard to determine which path will be chosen, except the value of the condition can be figured out at compile time, e.g. when it is a constant value.

To still allow such a situation and not decrease the number of possible candidates for inlining it is a suitable idea to handle the unsafe size by determining the maximum size of an object.
It is trivial to determine the static type of an inlinable field but much more difficult to figure out the dynamic type and its size.

4.3.1 Largest Dynamic Type

A naive approach is to simply compute the size of all dynamic types by traversing the type of the field and its subclasses. But this idea is not recommended because it is very hard for jino to figure out the correct object size at compile time. The GCC compiler could figure it out because of its just-in-time compilation procedure.
The following two arguments emphasize this rating:

1. The size of the object header is not necessarily fixed at compile time. Many components can increase its size, e.g. when more bits for class ids are necessary.

2. Even object inlining itself is an uncertain factor. When any (sub)class owns a field which can be inlined, the size of the field cannot be determined. In a gradual manner fix-point iteration would help.

This algorithm will allocate mostly too much space and the implementation seems very complicated.

4.3.2 Object Size With Union

Instead of summing up the different sizes, it is possible to make use of the `union` construct in C. Unions can store multiple values with different types in one field where only one of them will actually be saved. The compiler determines which field is the largest one and allocates that amount of memory. For the object size purpose it is the perfect feature to create for every inlined object a `union` construct which lists all different possible types (see Listing 5).

```
typedef struct {
   object_pointer c1c1_B;
   object_pointer c1c1_C;
OBJECT_HEADER
   jfloat c1a1;
} c1_A_t;

typedef struct {
   c1_A_t c1_A;
   union {
     c3_C_t c2_B;
     c4_D_t c4_D;
   } c1c1_C_inline;
} c1_A_inline_t;
```

Listing 5: Layout of `structs` holding an inlined object. A new `struct` is introduced storing the `struct` of the container object and the `union` construct with set of dynamic class types.

The backend must be adjusted in `IMClass` and `DefaultObjectLayout`. `IMClass` stores class-specific information, for example methods, fields and primitive values but also a `translate` method which emits C code. The `DefaultObjectLayout` defines the outgoing layout of objects in C depending on interactions of given options and is the place to implement the new `struct`. Possible options are for instance a modification of the object header or swapping fields to the stack.
When the backend translates a class, it adds in the header file additional `include` instructions for each possible dynamic type of the classes' inline objects.
While the `union` takes care of the correct size of each dynamic type it does not fix the problem that too much space is reserved. Referring to Listing 5, still the largest possible dynamic type will be chosen even if no object instance of that type is assigned to the field. To fix this issue it is necessary to register which object types are actually assigned to the field.

4.3.2.1 Dynamic types

To examine the problem of different types and corresponding object sizes a little class diagram is introduced in Figure 12 which holds a typical initialization pattern (see Listing 6).

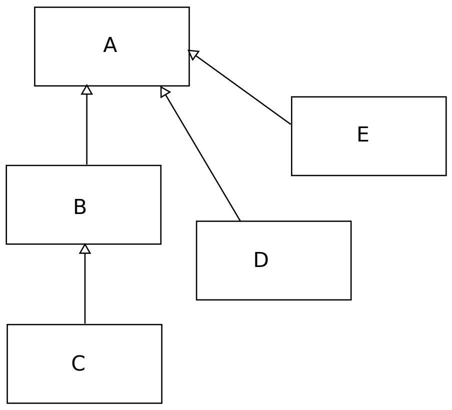

Figure 12: The figure shows a generic class hierarchy. References of type `A` can obtain an assignment of every type listed in the hierarchy.

```
...
final A a;
public Class(){
    if(condition){
        a = new B();
    } else {
        a = new E();
    }
}
...
```

Listing 6: Displaying a part of a constructor method. The field reference `a` can have two dynamic types.

The Control-Flow-Sensitive analysis preserves dynamic type information for each field. In detail it saves one dynamic type for each field that fits for all possible assignments. All objects of a dynamic type could successfully perform an `instanceof` operation to the type evaluated by the Control-Flow-Sensitive analysis.
Referring to the example given in Listing 6 the analysis notices two assignments to the field reference `a` and deduces the most general type: `A`.

As a consequence this proceeding is still not satisfying. In some cases it will perform well unless more than one type will be assigned to the target field. The concrete dynamic types cannot be figured out (see Figure 13).

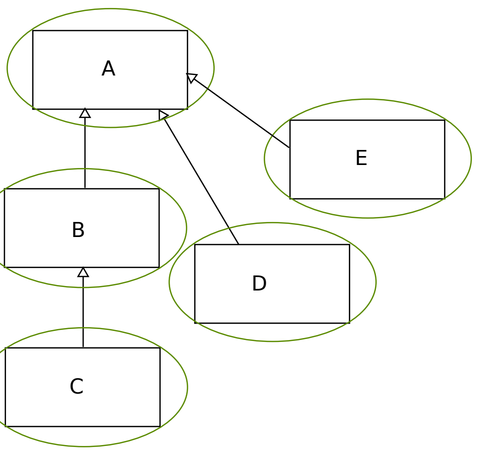

Figure 13: Possible types for field reference `a` (see Listing 6) based on Control-Flow-Sensitive analysis are marked with green ovals.

A better means to solve the problem is the connection graph derived from Escape analysis (compare Section 3.2.3). Whenever a `new` instruction appears in code and the created object is assigned to a field reference, the connection graph adds an edge from the field to an object node with the dynamic type. The intermediate code representation of allocations holds the type information. As shown in Figure 14 only the concrete types are extracted.

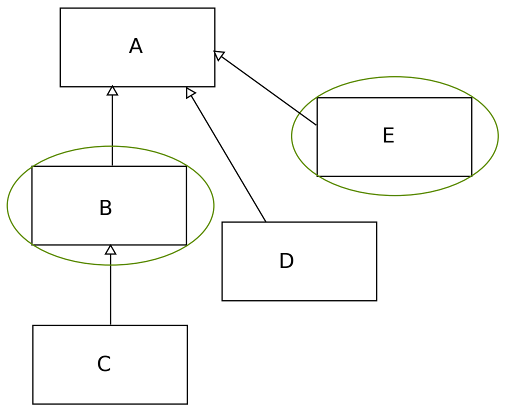

Figure 14: Possible types for field reference `a` (see Listing 6) based on a connection graph are marked with green ovals.

The connection graphs containing the alias information are already known. Each graph belongs to a method which is represented by it. Therefore it is necessary to make all relevant methods available that make an assignment to an inline reference. In Section 4.2 this procedure has been presented.

Since every possible dynamic type is known, only their header files are included. Furthermore the members of each `union` can be reduced by containing only these types.

4.4 Cycles in Inline Graph

Definition: Inline graph

Describes a directed graph whose edges represent inline references. The vertices are classes which either hold the reference (source of edge) or can be inlined (destination of edge). Outgoing edges mean that the source class has at least one reference with the dynamic type of the destination class.

A possible problem are cycles within the inlined object references. Cycles are created by the object layout and the permission to share inlined object references. The scan for phantom nodes in the connection graph should avoid cycles except for the case when method inlining falsifies the graph.

Hence through unfortunate constructions, classes could cyclically inline each other. An example for cyclic inlining is represented in Figure 15. Without any adjustment member objects of such a circle would allocate endless space at runtime, because the new object layout leads to a co-allocation of inlined objects. When every inlined object has itself as an inlined field reference, it results in an endless memory requirement.

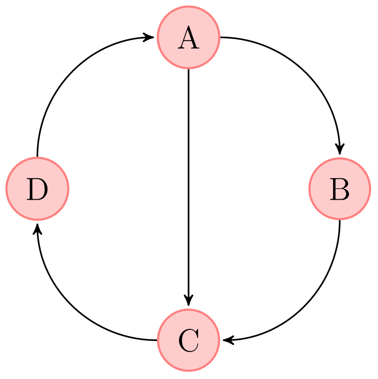

Figure 15: The directed graph represents the dependencies between classes based on the references that can be inlined. Every directed edge symbols that the source holds a reference to the destination with the ability to be inlined.

To prevent false inlining, all cycles must be dissolved by declaring specific references as not inlinable. Therefore an inline graph is calculated as in Figure 15. The results of Algorithm 1 can be used to create this inline graph.
To create the inline graph it is necessary to check each class's inline references and add an edge from the container class, which holds this reference, to each possible dynamic type class. This procedure is illustrated in Algorithm 2.

Algorithm 2: The procedure of building directed graph for inline references

```
Input  : Class storage that holds all classes
Result: Directed graph for cycle finding

1 buildDG($classStore$ :
  Repository that holds all classes of program, $dg$ :
  Directed graph instance)
2 begin
3     foreach $clazz \in classStore$ do
4         foreach $inlineRef \in$ clazz. getInlineRefs() do
5             $dynTypes =$ clazz. getDynTypes($inlineRef$);
6             foreach $dType \in dynTypes$ do
7                 dg. addEdge($clazz, dType$);
```

Before concrete cyclic problems can be solved, all *strongly connected components* must be extracted. All vertices in a strongly connected component are reachable from each other vertex. An algorithm invented by Tarjan [Rob74] is used to filter out all components.

With this set of components in hand, an algorithm can be formulated to remove edges from the graph wisely. Cycles are detected with a *depth first search* (DFS). Whenever an edge will close a circle, it will be removed in the graph. Thus all references of the class which are responsible for the edge, become marked as not inlinable.

Depending on the selected starting position, the results of the algorithm will vary. When, as illustrated in Figure 16, `C` is chosen as a consequence two edges are removed from the graph.

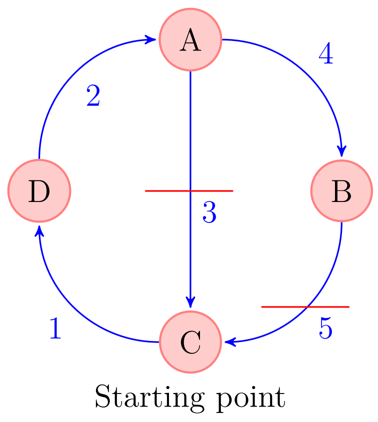

Figure 16: Choosing the starting point randomly implicates a non-optimal result. Then, the search with DFS deletes edges that causes cycles. As a consequence two edges get removed: A→C and B→C.

To minimize the number of edges to be removed, a heuristic is introduced. As a starting point the vertex with the most outgoing edges is selected. This heuristic is applied on the same example and delivers a correct graph by deleting one edge (see Figure 17) instead of two.

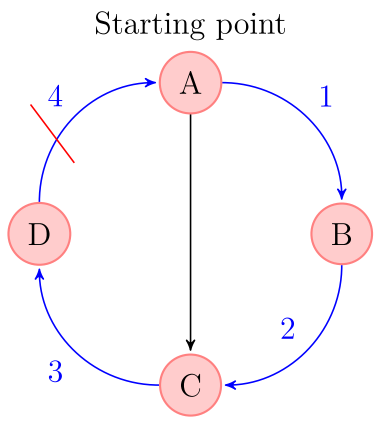

Figure 17: Choosing the starting point depending on the vertex which has the most outgoing edges. The search traverses the graph again with DFS and delete edges that close cycles. Here only one edge, D→A, is removed.

4.5 Allocation Order

Method inlining bothers the evolving of cycles as well as it is in charge when a connection graph does not notice that the assigned object to an inline field references is created in outer context.
As a consequence, the connection graph creates a regular object node instead of a phantom node. In any case it is not possible to inline an object when it is created before its container object. This is reasoned by the nature of the object layout. At first the container object needs to be created with enough place for the inlined objects. Later on, the inlined objects are placed in memory behind the container object. Therefore it is essential to verify the correct order of allocations of the container and inlined object.
The examination of the allocation order begins by searching all allocations of objects in charge for creating the container object holding the inline field reference and the object referenced by the inlined reference.
These allocations are represented in the intermediate code through appropriate objects including the basic block they live in.
It can be determined if a basic block *dominates* another one by using the Dominance analysis that is explained in Section 3.2.2. Thus the correct allocation order can be verified by checking if the basic block of then container object allocation dominates the other one.
To find each relevant allocation pair it helps to rescan the set of connection graphs explained in Section 4.2.1. The characterizations of these graphs are that their corresponding methods initialize any reference marked as inlinable.

Therefore each connection graph holds at least one pattern such as depicted in Figure 18.

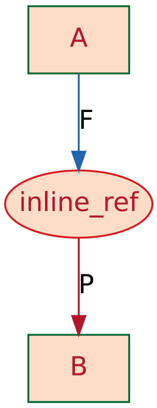

Figure 18: A part out of a connection graph. Class `A` holds a field reference which points to an object of type `B`. Objects created by the allocation behind `A` will inline an object of type `B`.

The inline field references of each graph are already identified (see Section 4.2.1 as well). Next, the preceding and succeeding object node is collected. The object nodes are very helpful because each one represents a required `new` operation, which was already pointed out in Section 3.2.3.
To determine the basic block of any instruction the Control-Flow-Sensitive analysis can be used again because it stores for each instruction the corresponding basic block.
Finally the Dominance analysis validates to *true* if a domination between these allocations exists. In the corner case of receiving two times, the same basic blocks the instructions must be traversed in serial order until the first allocation occurs. Therefore it is not necessary to use the Dominance analysis, because the same basic blocks dominates always itself.
Through introducing the constraint that allocation pairs must be allocated in the right order, it is not essential to look for inlining circles. Circles can only occur when at some point in the code the order gets violated. This is quite reasonable because for closing such a circle it is necessary that at least one inline field reference points to an object from outer context.

4.6 Translate Allocations

Until now each field reference with the option to be inlined is known. In addition each allocation responsible for container and inlined object is identified.

At next a translation for the backend must be formulated that produces equivalent C code. When object allocations are translated they look usually as displayed in Listing 7. A macro hides the concrete allocation process and returns the address of the object header.

This result is then saved into a variable respectively a stack slot. Definitions for these macros are placed in the header files of the corresponding class type. The macro statements themselves are emitted by the `translate` method of the allocation instruction. In order to change the behavior for object inlining purposes it is, on the one hand, essential to modify the backend code provided by translate methods, and on the other hand, to introduce new macros that fit for the requirements of object inlining.

```
obj0 = KESO_ALLOC_CXY_A(); // Allocate object of type A
```

Listing 7: This Listing exemplifies the common procedure of allocations in the backend code.

When object inlining can be adopted to an object pair the original code looks likewise to Listing 8. At first memory for the prospective container and inlined object are allocated. Afterwards a field reference simply connects both objects. The result of the whole procedure is illustrated in Figure 19.

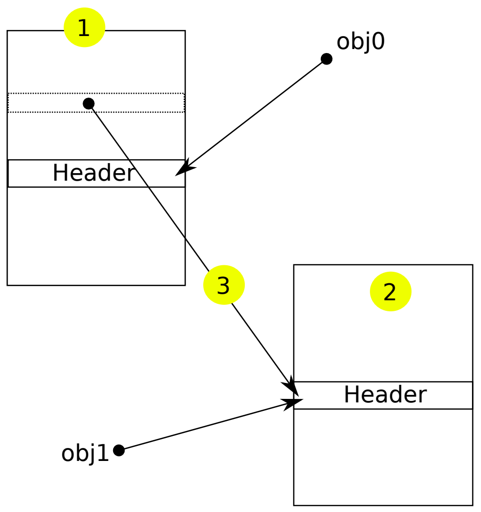

Figure 19: The results on memory level of Listing 8 are represented here. At first an instance of class `A` is allocated and assigned to field reference `obj0`. In an analogous manner an instance of `B` is assigned to `obj1`. In a third step a reference of `obj0` points to `obj1`.

```
// Step 1
obj0 = KESO_ALLOC_C1_A();
...
// Step 2
obj1 = KESO_ALLOC_C2_B();
...
// Step 3
(ACCFIELD_C1_A_Ref1(obj0)) =
    obj1;
```

Listing 8: Whenever inlining is possible, the original code pattern include these three instructions. Two objects get allocated (Step 1 and 2) and afterwards the second one is assigned to a field reference of the first object. Figure 19 visualizes the behavior.

a) **Container**: Each allocation for a container object can still use the `KESO_ALLOC` macro. Instead of allocation space for the container object only, the presented solution allocates memory for all inlinable objects as is pointed out in Section 4.3. As a consequence the new macro, which is called `KESO_ALLOC_INLINE` (see in Figure 20, Step 1), transfers the computed size to the `KESO_ALLOC` macro.

   Whenever this new macro is written into the backend, its slot name, which stores the returning address, is communicated to each inlined object via a map.

b) **Inlined object**: As shown in Listing 5 every new created `struct` for a container object has a (union) member for each inlined object. Depending on the member and slot name the address is transferred to the new macro `KESO_OBJECT_INLINE`. The macro itself evaluates merely the position of the object header and returns it (see in Figure 20 and Listing 9, Step 2).

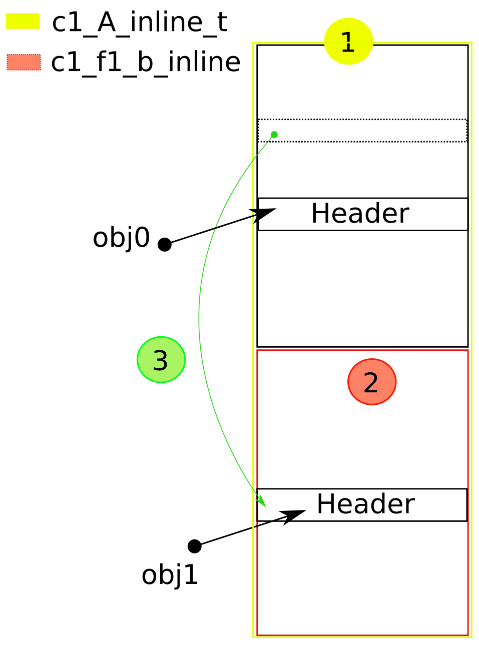

Figure 20: The figure illustrates the stepwise procedure of Listing 9 in case of object inlining. In step 1 (yellow shape) space get allocated for both objects. Next, the inlined object get stored in it (red shape) and the resulting reference address points on its header. The last step implies again the assignment of the inlined object to a field reference of the container one.

```
// Step 1
obj0 = KESO_ALLOC_INLINE_C1_A
    ();
// Step 2
obj1 =
    KESO_OBJECT_INLINE_C2_B(
    &( ( (c1_A_inline_t *)
        C1_A_OBJ(obj0) )->
        c1f1_b_inline)
);
// Step 3
(ACCFIELD_C1_A_C2_B(obj0)) =
    obj1;
```

Listing 9: This backend code shows the result of Listing 8 after modification for object inlining. The first step is similar except that the new macro allocates more space for both objects. In step 2 no new object is created, instead the associated place for it in the former object get assigned. Step 3 has not changed.

4.6.1 Combination of Stack Allocation and Object Inlining

The thesis of Lang [Lan12] is about allocating objects on the stack whenever possible instead of bothering the heap. There is no reason to omit any of the analysis in favor of the other one because they can be combined.
Therefore it must be verified that the container object is stack allocatable as well as the inlined objects. When the allocation statement that corresponds to container object is translated by jino, the compiler checks that for each inline statement if the corresponding object is stack-allocatable, too.
If every child object can allocated on the stack, the same translation procedure, which is described in Section 4.6 can be used. Instead of storing the container object on the heap, as is depicted in Listing 7, it is placed on top

of the stack. The only difference to the *normal* co-allocation is, that a stack address is saved in the corresponding slot.
The constraint that all members have to be stack-allocatable is essential. Otherwise it might happen that an inlined object tries to access an invalid stack address.

5 Evaluation

After presenting the implementation of object inlining, the evaluation of it follows. Therefore two benchmarks are compared with the analysis enabled and disabled. With these results at hand, it is possible to conclude how effective the analysis is in term of memory footprint and runtime costs.
At first the benchmarks used for evaluation are introduced.

5.1 Benchmarks

5.1.1 CDx

CDx is ”an open source application benchmark suite that targets different hard and soft real-time virtual machines.” [KHP+09]. The benchmark was developed for providing a Java application for tests on microcontrollers and embedded systems. It ”models a hard real-time aircraft collision detection application” [KHP+09]. In detail the program is divided in two parts. A *radar simulator* scans the aircraft traffic in certain time frames. These results are transferred to the *collision detector*. It figures out for each frame if a collision happens between any aircraft pair and displays the collision point in a 3-D vector space.

5.1.2 Snooker

The *Snooker* benchmark is developed for the purpose of testing object inlining. It generates a set of balls, settled in a 2-D vector space, with a certain velocity and direction. In each time frame it is checked for a collision between any of the balls plus determining the collision point.

Benchmark	Classes	Ref-Fields
CDx on-the-go	131	52
Snooker	8	38

Table 2: Class and field information.

Both benchmarks are tested on a TriCore TC1796 microprocessor running a CiAO operating system (`https://www4.cs.fau.de/Research/CiAO/`). The hard- and software configuration for the testing environment are listed in Table 3. The proportion of the number of classes and field references in the benchmarks is given in Table 2.

Components	**CDx on-the-go / Snooker**
CPU	Infineon TriCore TC1796 150 MHz CPU 75 MHz system
Memory	2 MiB Flash, 1 MiB SRAM
OS	CiAO Commit: 30df8c50453c14
Compiler	GCC 4.6.3
KESO	r4304
KESO's memory management	Restricted Domain Scope

Table 3: Hard- and software configuration used to run both benchmarks.

5.2 Measurements and Results

5.2.1 Static Evaluation

The number of field references that can be inlined in a benchmark acts as a first indicator for the effectiveness of the analysis.

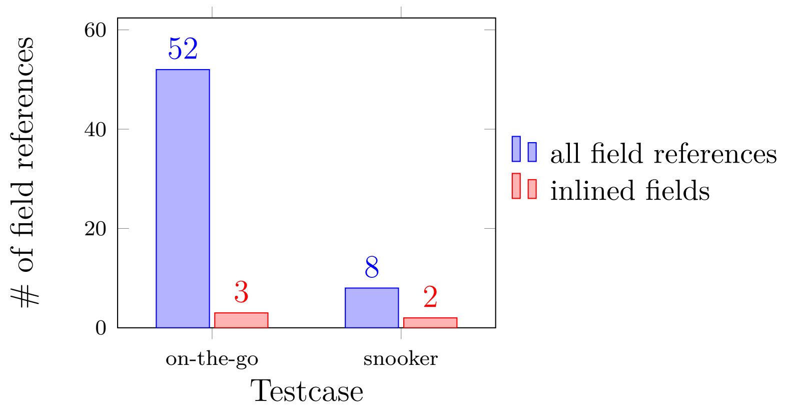

Figure 21: This figure displays the number of field references existing in a benchmark compared to the fields that can be inlined. Each inline field reference counts as field reference, too.

The proportion of field references to references, which can be inlined at least one time, is depicted in Figure 21. In the CDx benchmark only about $\approx 4\%$ of the field references can be inlined while the snooker benchmark has $\approx 8\%$. This result is reasoned by several aspects. On one side the most field references do not hold the `final` predicate even if a field would fulfill all necessary conditions. An analysis that examines static-fields for the qualification of being `final` exists already, the so-called *Slot Alias analysis* [CSIW14]. An extension for non-static is "currently not yet implemented in *jino*" [CSIW14]. On the other side, objects, which are assigned to such a `final` reference, are created before the container object itself. As a consequence the co-allocation is not possible, too.

It is also interesting how many times jino translates a co-allocation statement in comparison to stack and heap allocations. The statistic is illustrated in Figure 22.

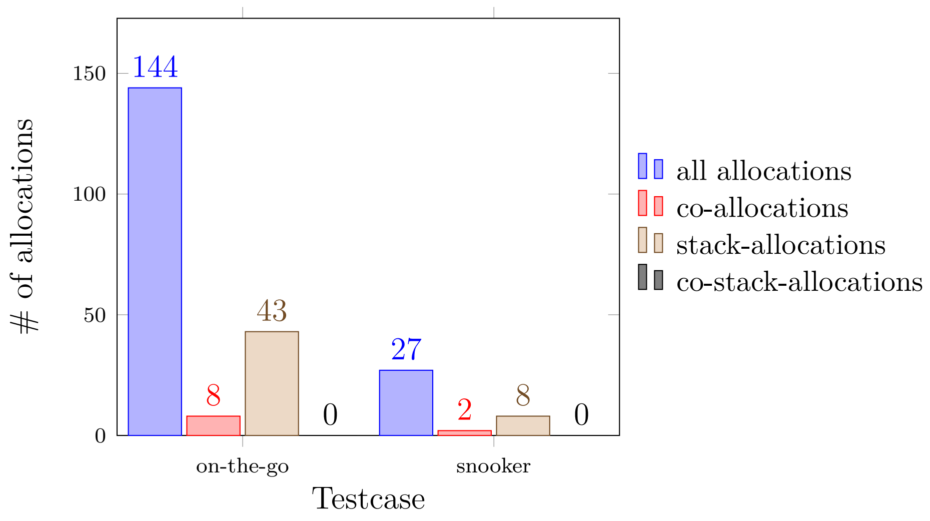

Figure 22: The graph displays how many times jino translates usual allocations, co-allocations, stack allocations and co-allocation that are stack-allocations too.

A matter of fact is that no stack-allocatable object can be inlined and vice versa. Such a combination is quite rare because the container object and inlined object needs to be stack-allocatable. The proportion of co-allocations is approximately as high as many field references are inlinable.

The code size of both benchmark differs not that far to their corresponding inline variant (see Table 4). The *Data* segment has still the same size while the *BSS* segment of the CDx benchmark is increased by 4 Byte. The size of the *Text* segment has also increased a bit.

Benchmark	Text	Data	BSS	Total size
CDx on-the-go normal	42071	1893	983419	1027383
CDx on-the-go inline	42199	1893	983423	1027515
Snooker normal	18379	493	702106	720978
Snooker inline	18395	493	702106	720994

Table 4: The Table lists the code size of each benchmark, with and without inlining, divided by its different data segments. The measurements are done with the *size* program of the GNU toolchain.

This result does not surprise in anyway because the analysis removes neither variables yet nor adds new ones. The additional space required in both *Text* segments is caused by the longer function names introduced for inlining. The additional 4 Byte in the CDx variant which has inlining enabled can be caused, for instance, by alignment.

5.2.2 Dynamic Evaluation

The gathered information about stack and heap usage are stored in Table 5. The variants of the CDx benchmark differs in the number of heap and stack allocations while the Snooker benchmark has merely changed in the number of heap allocations. In both cases the variant which uses object inlining has less stack and/or heap allocations because of replacing them through co-allocations.
It might be a bit surprising that object inlining concerns the amount of stack allocations. If an object, which is inlinable and stack-allocatable, is allocated on the stack depends on its corresponding container object. In case the container object is not stack-allocatable its inlinable objects are not stored on the stack either and instead inlined on the heap.
Apart from that the requested heap and stack size are nearly the same. In future work a better heap and stack requirement can be achieved, for instance, by removing the field reference in the container object and instead using a fixed offset which would spare out the space for a reference (4 byte on the TC1796 device). If it works out to inline the object completely, it is not essential to keep the header of the inlined object anymore with its size of 32bit [Sti12]. Summarized it might be possible to save about 64bit for each

inlined object.

Benchmark	Allocations on Stack	Requested Stack size in byte	Allocations on Heap	Requested Heap size in byte	Co-allocations
CDx on-the-go normal	13667	166046	12909	293688	-
CDx on-the-go inline	13637	165086	12673	294720	266
Snooker normal	15	180	18313	220512	-
Snooker inline	15	180	18213	220510	100

Table 5: The Table shows the total requirement of stack and heap size as well as the number of stack, heap and co-allocations at runtime.

The final measurement refers to the different runtimes (see Table 6).

Benchmark	Average Runtime in ns	Differential in ns
CDx normal	27002560	-790344
CDx inline	27792904	
Snooker normal	22358254477	22856361
Snooker normal	22335398116	

Table 6: The average runtime of each benchmark

The runtime of the CDx benchmark is almost the same. In several runs it points out that each variant is sometimes faster than the other one. In contrast the Snooker benchmark is continuously a little bit faster (about $\approx$ 0.1%). This behavior might be reasoned by the fact that the pair of container and inlined objects are fitting in a cache line or not. In the Snooker variant the container object has a size of 40 Byte and its both inlined objects of 12 Bytes. They fit perfectly in a cache line and the access is a bit faster. On the opposite in the CDx benchmark is the size of the container object, which is high frequently used, including its three inlined objects 96 Byte large.

Nevertheless the speedup is marginal until now. The effectiveness of the Object Inline analysis is highly application-specific as it depends on the number of final references being used. Combining the Object Inline analysis with an enhanced version of the Slot-Alias analysis [CSIW14] has the potential to improve the runtime behavior.

6 Conclusion and Future Work

For this thesis an initial version of an object-inlining analysis for KESO has been described and implemented. Therefore a co-allocation is implemented which stores the inlinable objects behind the container object in memory. Conditions defined for this co-allocation are for example:

a) The field references, which point to inlinable object(s), need to be marked as `final`. This constraint guarantees that such a field reference points to the same object during its lifetime.

b) The allocation order between the container and inlined object(s) is essential. The container object has to be allocated before its inlined objects.

The realization in KESO's backend is done by introducing a new object layout. It provides enough space for the container object and its set of inlined objects. When the container object is translated the compiler allocates the size of the new layout. Afterwards each inlinable object passes the allocation procedure and is stored in the container object. The polymorphic aspect has been solved by using the `union` construct of C.

In the evaluation it was figured that for two exemplary benchmarks just about $\approx 4 - 8\%$ of the fields can be inlined. This is caused by the lack of `final` references. To improve this result it would be helpful to extend the *Slot Alias Analysis* [CSIW14] by figuring out for non-static fields if they can be marked as `final` either.
Two suggested steps that can be done next to complete the inlining procedure are:

a) **Removing the reference of the container object**

   The address of each object can be obtained by a global map which saves the corresponding offset. This would save a field reference for each inlined object.

b) **Completely inline the objects**

   Instead of keeping the inlined objects alive it might be interesting to completely merge the inlinable object with its container object. This procedure would spare out the object header and field reference for each inlined object. In contrast the new object size is increased during its whole lifetime.

Both steps could save memory to runtime by erasing object headers and/or field references. In addition another level of dereferencing can be optimized out which causes lower runtime costs.

7 Appendix

References

[CSIW14] Erhardt C., Kuhnle S., Stilkerich I., and Schröder-Preikschat W. The final fronier: Coping with immutable data in a jvm. *ACM*, October 2014.

[Erh11] Christoph Erhardt. *A Control-Flow-Sensitive Analysis and Optimization Framework for the KESO Multi-JVM.* Diplomarbeit, Friedrich-Alexander University Erlangen-Nuremberg, March 2011.

[JA00] Dolby J. and Chien A. An automatic inlining optimization and its evaluation. *ACM*, pages 345–357, 2000. In Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation.

[JASJ96] Dolby J., Chien A., Reddy U. S., and Plevyak J. Icc++ - a c++ dialect for high performance parallel computing. pages 76–95, March 1996. In Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software.

[KHP$^+$09] Tomas Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek, Ben Titzer, and Jan Vitek. CD$_x$: a family of real-time java benchmarks. pages 41–50, 2009.

[Lan12] Clemens Lang. Improved stack allocation using escape analysis in the keso multi-jvm (keso/estackalloc). Bachelorarbeit, Friedrich-Alexander University Erlangen-Nuremberg, October 2012.

[Mak06] Dariusz Makowski. *The Impact of Radiation on Electronic Devices with the Special Consideration of Neutron and Gamma Radiation Monitoring.* Dissertation, Technical University of Lodz, 2006.

[MIR$^+$11] Stilkerich M., Thomm I., Kapitza R., Schröder-Preikschat W., and Lohmann D. Automated application of fault tolerance mechanisms in a component-based system. *ACM*, September 2011.

[OL02] Lhoták O. and Hendren L. Run-time evaluation of opportunities for object inlining in java. *ACM*, (10), November 2002.

[P.01] Laud P. Analysis for object inlining in java. 2001.

[Rob74] Tarjan Robert. Depth-first search and linear graph algorithms. *SIAM Journal on Computing 1*, pages 146–160, 1974.

[Sti12] Michael Stilkerich. *Memory Protection at Option - Application-Tailored Memory Safety in Safety-Critical Embedded Systems.* PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2012.

[TE79] Lengauer T. and Tarjan R. E. A fast algorithm for finding dominators in a flowgraph. *ACM*, pages 121–141, 1979.

[Waw09] Christian Walter Alois Wawerisch. *KESO: Konstruktiver Speicherschutz für Eingebettete Systeme*. PhD thesis, Friedrich-Alexander University Erlangen-Nuremberg, 2009.