The Perfect Getaway: Using Escape Analysis in Embedded Real-Time Systems

ISABELLA STILKERICH, CLEMENS LANG, CHRISTOPH ERHARDT, CHRISTIAN BAY, and MICHAEL STILKERICH, Friedrich-Alexander-University Erlangen-Nuremberg

The use of a managed, type-safe language such as Java in real-time and embedded systems offers productivity and, in particular, safety and dependability benefits at a reasonable cost. It has been shown for commodity systems that Escape Analysis (EA) enables a set of useful optimizations, and benefits from the properties of a type-safe language. In this article, we explore the application of escape analysis in KESO [Stilkerich et al. 2012], a Java ahead-of-time compiler targeting embedded real-time systems. We present specific applications of EA for embedded programs that go beyond the widely known stack-allocation and synchronization optimizations such as extended remote-procedure-call (RPC) support for software-isolated applications, automated inference of immutable data, or improved upper space and time bounds for worst-case estimations.

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

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

ACM Reference Format:
Isabella Stilkerich, Clemens Lang, Christoph Erhardt, Christian Bay, and Michael Stilkerich. 2017. The perfect getaway: Using escape analysis in embedded real-time systems. ACM Trans. Embed. Comput. Syst. 16, 4, Article 99 (May 2017), 30 pages.
DOI: http://dx.doi.org/10.1145/3035542

1. INTRODUCTION

Java is a relatively uncommon language in embedded real-time systems, although it provides a series of advantages such as memory safety [Aiken et al. 2006; Stilkerich 2012]. Numerous projects [Siebert 2007; Pizlo et al. 2010a, 2010b; Schoeberl et al. 2008, 2009] have exhibited that it is possible to employ Java in embedded (real-time) programming and that it is even feasible to write drivers in Java. Anyway, Java still has the reputation of being unsuited for this domain due to additional runtime overheads and increased code sizes. The KESO JVM [Stilkerich et al. 2012] has shown that static analyses on Java applications on top of a static system setup allow to generate memory-safe code that is competitive to native C programs in terms of runtime results and footprint. Being a type-safe programming language, Java provides the foundation

This article is based on an earlier conference contribution: A Practical Getaway: Applications of Escape Analysis in Embedded Real-Time Systems, in LCTES'15 Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2015, Copyright © 2015 ACM. http://doi.acm.org/10.1145/2670529.2754961. The work was partly supported by the German Research Foundation (Deutsche Forschungsgemeinschaft (DFG)) under grant no. SCHR 603/9-1 (AORTA). Correspondence to Isabella Stilkerich (email: isabella@stilkerich.eu).
Authors' addresses: I. Stilkerich; email: isabella@stilkerich.eu; C. Lang; email: neverpanic@gmail.com; C. Erhardt; email: erhardt@cs.fau.de; C. Bay; email: christian.bay@fau.de; M. Stilkerich; email: michael@stilkerich.eu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2017 ACM 1539-9087/2017/05-ART99 $15.00
DOI: http://dx.doi.org/10.1145/3035542

for comprehensive program analyses and runtime-system support, which can be very useful in embedded systems. One of these static program analyses is escape analysis, which is often employed in commodity systems for stack allocation of objects and for synchronization optimizations. For embedded software, the results of escape analysis open up a number of interesting optimization opportunities, which we will explore in this article.

Background. Embedded application software may be deployed on several kinds of Microcontroller Units (MCUs) and the MCU landscape is rather heterogeneous in contrast to commodity systems: They may vary in their hardware-specific properties such as the kinds of memories available, their respective memory layout, the existence of a memory protection unit (MPU) or the frequency of occurring random transient errors. Also, applications may differ in their need for certain non-functional properties such as the real-time capability of the program. Escape analysis is one of the essential analyses amongst other static analysis techniques to make the best possible use of the underlying hardware and operating-system (OS) features with respect to a specific application and its nonfunctional requirements on the system software on a particular embedded device. Escape analysis is one building block of the *cooperative memory management* (CMM) framework [Stilkerich et al. 2014b] provided by KESO: The application developer is *assisted* by the type-safe middleware—comprising compiler analyses and runtime support—while constructing safety-critical embedded systems. It is still up to the developer and the system integrator to decide which compiler back ends are most suitable under certain system requirements. To achieve a resource-efficient solution, CMM respects characteristics of the underlying operating system as well as the specification of the hardware device.

CMM provides the foundation to experiment with particular system configurations to generate system code that particularly fits to an application. We believe that in this way embedded developers can benefit from the use of a modern high-level language, automated memory management, and memory protection while still being able to directly influence system traits.

Contribution. Besides normal stack allocation, our implementation of escape analysis features a set of worthwhile escape-analysis applications for embedded safety-critical systems:

(1) Extended remote-procedure-call support for software-isolated applications
(2) Extended stack scopes
(3) Thread-local heaps and regional memory
(4) Automated inference of immutable data
(5) Object inlining
(6) Survivability: Support for the machine-independent space- and time-bounds analyses of memory management

An earlier publication [Stilkerich et al. 2015] presents further applications of escape analysis in the context of safety-critical embedded systems.

Overview. Section 2 explains the relevant aspects of the embedded KESO JVM, in which we implemented and evaluated our approach and the key features of escape analysis. In Section 3, we quickly recap the outlines of the original algorithm by Choi et al. [2003], which is the base for our own algorithm. That followed, we sketch the modifications to the EA's interprocedural analysis part in Section 4. Our EA applications for embedded systems are presented in Section 5. We evaluate the outcomes of our work in Section 6 on a microbenchmark and on the open-source real-time Java benchmark *Collision Detector* (CD_x). The article wraps up with the conclusion in Section 7.

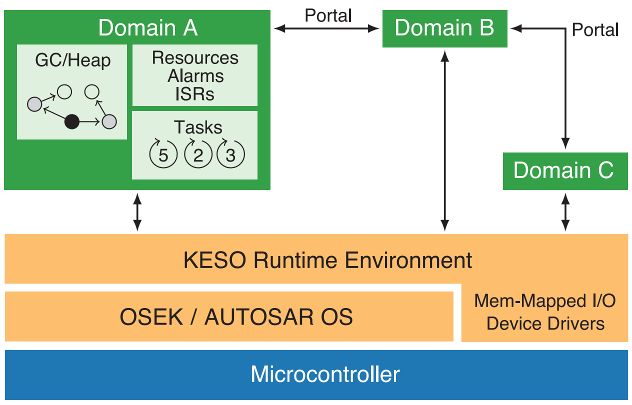


Fig. 1. KESO's architecture.

2. KESO AND ESCAPE ANALYSIS: AN OUTLINE

This section describes the characteristics of the KESO Java Virtual Machine (JVM) and introduces escape analysis. We only present those properties relevant for the implementation of our EA applications.

The KESO Runtime Environment. KESO is a JVM for statically configured embedded systems. In such systems, all relevant entities of the (type-safe) application as well as the system software are known ahead of time. Among others, these entities comprise the entire code base of the application, and operating-system objects such as threads and locks. Disallowing the application to dynamically load new code or to create threads at runtime allows the creation of a slim and efficient runtime environment for Java applications in embedded systems. This approach seems restrictive at the first sight, however, static applications cover most traditional embedded applications from the electronic control units found in appliances to safety-critical tasks such as the Electronic Stability Program (ESP) and many other electronic functions found in modern cars.

The architecture of KESO is shown in Figure 1. KESO is a multi-JVM, that is, applications can be isolated from each other by placing them in so-called protection *domains*. Spatial isolation is ensured constructively by the type-safe programming language and the strict logical separation of all global data (e.g., heap, static class fields)—providing memory protection even on low-end MCUs that lack dedicated hardware support by means of an MPU or memory-management unit (MMU). In contrast to hardware-based protection, software-based memory protection provided by KESO is ensured at the granularity of objects and KESO programs are therefore memory-safe: Memory safety is a program characteristic that guarantees that the program does not use memory in any unsafe way [Aiken et al. 2006]. Basically, a pointer must refer to a valid object and it must not be possible to access memory outside an object's boundaries through a pointer. Otherwise, the pointer must not be dereferenced. Moreover, research projects engaging in memory safety commonly state that pointers or references to deallocated memory (also called dangling pointers or references) must not exist: The memory referred to by the dangling reference can be used by another object and an access through the invalid reference may break the soundness of the type system.

In addition to constructively ensured memory protection, KESO is able to support the OS to use available hardware-based protection mechanisms by means of its control-flow-sensitive reachability analysis, which physically groups the domain data (i.e., the physically separated heaps and static fields) in separate memory regions. When using *weakly typed* languages such as C, this is usually performed manually by the programmer [Danner et al. 2014]. In KESO, it is also possible to off-load runtime checks to an unutilized MPU [Stilkerich 2012].

For communication, an RPC-like mechanism (called *portal service*) is available. To maintain software-based isolation, portals have to make sure that object references are not illegally propagated between domains: the portal service initiates a deep copy of all objects passed as arguments. To make communication more efficient, we created a new optimization for the portal service using escape-analysis results (see Section 5.1). The runtime system provides control-flow abstractions such as threads (called *tasks* in AUTOSAR OS) and Interrupt Service Routines (ISRs) and their respective activation and synchronization mechanisms such as alarms and synchronization locks (called *resources*). KESO applications benefit from Java features like type safety, dynamic memory management, and optionally a garbage collector. KESO's ahead-of-time compiler *jino* generates ANSI-C code from the application's Java bytecode, plus a slim, tailored runtime environment for that application.

While most of the code directly translates to plain C code, the Java thread Application Programming Interface (API) is mapped onto the thread abstraction layer of an underlying OS. In the case of KESO, that abstraction layer is normally provided by AUTOSAR OS [AUTOSAR 2009], however, KESO's concepts can also be applied to any other static OS. KESO optionally provides slack-based garbage collection for applications that want to use it. The collectors are scheduled, whenever a task blocks, that is, at well-known invocations of AUTOSAR OS's system called `waitEvent()`.

Escape Analysis. From a conceptual point of view, all Java objects are allocated in heap memory. There is no dedicated language support for the application developer to manually mark objects for allocation on the stack, because this would have the potential to break the soundness of the type system. As an example, allocating an object that lives longer than its method of creation on the stack of that method will lead to dangling references. However, due to the language's strong type system, it is possible for the JVM's compiler to automatically categorize objects in terms of their scope: The information collected by alias analysis and the computation of the references' reachability can be leveraged to determine if an object *escapes* a method, that is, if its lifetime exceeds that of the scope it was created in. As a consequence, non-escaping objects can be allocated on the stack and are not subject to the overhead entailed by heap management. Stack allocation implicates a series of advantages as has been presented by prior work [Blanchet 1998, 2003; Choi et al. 1999, 2003; Goldberg and Park 1990; Park and Goldberg 1992]:

—Allocation and deallocation are performed by moving the stack pointer. These are low-cost and time-predictable operations on a CPU register.
—Due to a reduction of the number of heap objects, the overhead of the heap-management strategy, such as a Garbage Collector (GC), is reduced. Incremental collectors do not need to synchronize with the mutator, that is, the application, whenever a local stack object is allocated or deallocated.
—Programs do not need to synchronize on objects known to be thread-local, which contributes to lock elision.

Especially in the context of embedded and safety-critical embedded systems, the information collected by escape analysis offers a lot more interesting optimization opportunities and we leverage them in the context of CMM. We will present applications of escape analysis in Section 5. In addition, an amended whole-program, ahead-of-time (AOT) version of Choi's (control-)flow-sensitive algorithm [Choi et al. 1999, 2003] has been implemented to pursue cooperative memory management. In the following, we

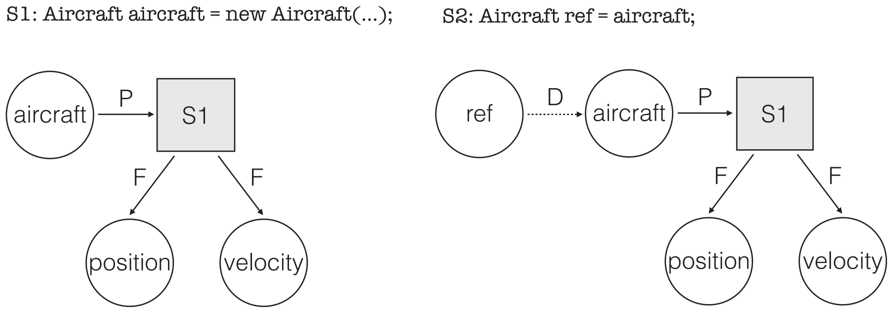

S1: Aircraft aircraft = new Aircraft(...);
S2: Aircraft ref = aircraft;
aircraft
P
S1
F
F
position
velocity
ref
D
aircraft
P
S1
F
F
position
velocity

Fig. 2. Example for a simple connection graph as used by Choi et al. [2003] applied to a code fragment occurring in the Collision Detector benchmark: Object nodes are denoted by boxes, whereas reference nodes are illustrated through circles. A box can be referred to by a points-to (P) edge (solid arrow). Dashed arrows indicate deferred (D) edges and solid arrows from boxes to circles denote field (F) edges.

will explain the algorithm CMM's escape analysis is based upon from a practical point of view. Afterward, we will describe the changes necessary to use this escape analysis in the context of CMM and KESO.

3. THE ESCAPE ANALYSIS OF CHOI ET AL. IN THE CONTEXT OF CMM

In the following, we present the conceptual design and basic idea of Choi's escape analysis since the actual implementation is rather complex. Detailed explanations and a formally correct description of the algorithm can be found in separate publications [Choi et al. 2003; Lang 2012].

3.1. Basics

In a nutshell, alias information is gathered from the application. The alias algorithm is divided into a method-local, intraprocedural and a global, interprocedural analysis. A dedicated data structure called *Connection Graph* (CG) is built up to hold alias information. Figure 2 shows a straightforward example for a CG. A connection graph consists of vertices and edges. A vertex is termed *node* and represents either an object or a reference. Nodes are interconnected via *edges* and a *points-to* edge from a reference node to an object node illustrates a possible aliasing relationship that might occur at some point in the program. *Deferred* edges are employed whenever a reference node points to another reference node. Deferred edges have the potential to enhance the analysis's efficiency through delayed evaluation since only a smaller share of references has to be updated. Deferred edges are removed by applying a bypass function called `ByPass(p)`, where `p` represents a reference node, and the function causes `p`'s incoming edges to point to its successor nodes. Delayed graph updates have to be handled with precaution as they influence the precision of escape information: The decision when to apply `ByPass(p)` poses a trade-off between efficiency and precision.

Connection graphs contain representations of local variables, static class members, dynamic instance variables (represented by local reference nodes, global reference nodes, and field reference nodes in a CG, respectively), array indices (represented by field reference nodes), and objects (represented as object nodes) for each processed method. Non-reference variables are discarded since they do not influence alias information. CGs are similar to points-to graphs employed in pointer alias analysis and are basically static abstractions of dynamic data structures referred to by pointers/references. However, points-to graphs are primarily used to compute if a set of references can point to the same memory location at runtime. Two references pointing to the same memory location result in the same node within a points-to graph. In contrast, a connection graph can have different nodes for two references that possibly point to the same memory location while still remaining correct. Simply put, CGs do not have the uniqueness property of points-to graphs since dynamic objects can be mapped to multiple static objects. Therefore, connection graphs can be constructed independent of their calling context, which is impossible for points-to graphs [Choi et al. 1993, 2003; Landi and Ryder 1992; Emami et al. 1994; Wilson and Lam 1995; Ghiya and Hendren 1996; Chatterjee et al. 1999].

Escape States. Three different object escape states exist within CGs and they are totally ordered: If the object does not escape its thread of creation, its state is *local* and in case the object escapes the method scope, that is, it is returned or assigned to a reference being passed as a method argument, but not its thread of creation, its state changes to *method*. If an object leaves the scope reachable by the control flow in which it became existent, it is assigned the *global* escape state. The escape state is determined through reachability analysis on the connection graph. Accordingly, local objects[1] can be allocated on the stack and method-escaping objects do not need to be guarded by synchronization locks.

Flow Sensitivity. Choi et al. present a flow-insensitive and a flow-sensitive analysis, whereat the flow-sensitive variant is able to compute more accurate aliasing relationships. Aho et al. [1986] elaborate on this difference in their book's *Flow Insensitivity* section. In flow-insensitive analysis, the control flow is ignored completely so that statements are effectively processed as they were executed in any order. Consequently, the algorithm computes *all* possible target locations for pointers. On the contrary, flow-sensitive analysis naturally takes a program's statement order into account and traces which statements kill points-to relations by making new assignments. As a result, the escape state information is more precise but the algorithm converges slower. For the sake of precision, CMM provides flow-sensitive escape analysis. The flow-insensitive algorithm applies *weak updates* whereas *strong updates* are necessary to achieve flow sensitivity (explained in the next sections).

3.2. Connection-Graph Construction

The connection graphs are created for each individual method by traversing instructions in *jino*'s intermediate code representation in Static Single Assignment (SSA) form. SSA causes each variable to be assigned exactly once and eases connection-graph construction by avoiding complex graph modifications imposed by multiple reassignments at, for instance, control-flow join points. Two analysis steps, that is, method-local and global analyses, are performed to determine the objects' escape states.

Method-Local Analysis. This analysis step computes the intraprocedural connection graph by processing the program call graph's methods. The algorithm starts at the method's formal reference parameters by adding an actual local reference node for each formal reference parameter and connects them via deferred edges. In case a method returns a reference, an actual reference node is added to the CG and it is marked *method-escaping*.

Connection-graph construction proceeds by identifying four basic statements in each method, which are assignments to either local or global (reference) variables:

(1) Plain assignments: `p = q;`
(2) Allocations: `p = new Object();`
(3) Write access to a field: `p.f = q;`
(4) Read access to a field: `p = q.f;`

[1]We do not allocate local objects with overlapping liveness regions on the stack to keep the stack usage bounded.

Any Java-source-to-bytecode compiler ensures that the relevant statements are compiled to this desired shape. Regular objects and array objects are handled in a likewise manner. For assignments to local variables (1), local reference nodes are created and added to the connection graph. Assignments to global variables (1) are handled by adding global reference nodes, accordingly. The true nature of the statement's right-hand side (`q`) is not relevant at this point and has to be handled individually. The algorithm makes sure that only one connection-graph representation for each variable exists in the CG. Assignments of the form (1) result in deferred edges for flow-insensitive analysis (weak updates) during the construction of the connection graph. Strong updates are performed by applying `ByPass(p)` before adding the edge between the two nodes.

For each allocation (2), a new object node is created if not yet existent, `ByPass(p)` is executed to enforce strong updates, and a points-to edge between `p` and the allocated object is added. The assignment between object node and reference node is handled analogously to (1).

Statements of the forms (3) and (4) contain field accesses. Field reference nodes[2] for `p` and all (reference) descendants of `p` (reachable via deferred edges) as well as the respective edges are added to the CG if the field reference nodes are not yet part of the CG. Assignments to static fields have to be considered separately. Objects pointed to by static fields are marked *global-escaping*. For both cases (3) and (4), there are situations in which a proper points-to relation cannot be identified for `p`: the reference `p` can be invalid, that is, `null`, or the object `p` refers to was created outside the method's scope. Such objects with an unknown allocation site have to be handled accordingly. For the representation of such objects that may have been passed as arguments, for instance, special nodes termed *phantom nodes* are created within a CG. Creating phantom nodes for `null` references does not affect the connection graph's correctness, which is why a phantom node is created for this case, too. Field reference nodes attached to object nodes reachable from `p` via points-to and deferred edges have to be present in the connection graph for each field that is accessed. In case a field is written (3), `q` is attached to the field reference node while read access (4) causes the field reference node to be attached to `q`.

Method calls and return statements have to be respected during the CG creation as well. Method calls (such as `p = method(x)`) are processed by creating actual reference nodes for each formal reference parameter (including the `this` reference of the method) at the caller side. These nodes are connected to the actual parameters at the invocation site. If a reference is returned, an actual reference node is created and `p` and this node are connected via an edge. Return statements containing references (e.g., `return p`) are processed by connecting the node representing the return value and `p` via a deferred edge.

During the construction of the connection graph, reachability analysis is applied incrementally to find out if an object can be allocated on the stack: Nodes reachable by a node marked *global* are also *global*. Nodes marked *method* and *local* are handled in the same manner.

Global Analysis. Intraprocedural analysis is not sufficient to calculate an object's escape state. Objects could exist that currently have the *local* escape state but in fact escape the method context[3] making the analysis unsound and inducing memory-safety issues. A subsequent, global analysis is needed to determine this. Another use case of global analysis poses the computation of thread-local objects.

[2]The variables' scope determines whether local or global reference nodes are added.
[3]Exemplarily, members of a linked list are marked *local* by intraprocedural analysis.

CG information computed in intraprocedural analysis is propagated into the callers' connection graphs in the interprocedural analysis step. The call graph is processed in reversed typological order to avoid unnecessary computations. Recursion has to be handled separately to ensure the algorithm's halting. CMM achieves this through Tarjan's solution [Tarjan 1972] to find Strongly Connected Components (SCCs) and an iteration over the call graph's SCC until the dataflow solution converges.

Equivalence pairs of object nodes in callees' and callers' connection graphs are determined via a *mapsToObj* relation. Further nodes are found by means of these pairs using a work list algorithm. Additional details on the algorithm can be found in Choi et al. [2003] and Lang [2012, 2014]. Nodes marked as *local* after the completion of global analysis can be allocated on the stack. Objects tagged *method-escaping* but not *global* are determined to be thread-local. Local and thread-local objects can benefit from compiler-assisted lock elision.

4. EVOLVING CHOI'S ESCAPE ANALYSIS FOR CMM

Compile times for ahead-of-time, whole-program alias and escape analyses following the proposed method are unacceptably high for larger applications[4] and are caused by the interprocedural part of alias analysis. To solve this issue, the interprocedural algorithm is optimized in various ways: The compile time is reduced down to a few seconds and the runtime behavior is revised.

Handling of Read Operations. Virtual methods containing calls to the same set of virtual methods cause connection graphs to grow very large during interprocedural analysis. Methods exist that do not add new aliases to the connection graph since they do not impose modifications due to write operations. Therefore, CMM's escape analysis only pursues a connection graph's edges that were added due to write accesses during intraprocedural analysis. The connection graphs' nodes are enriched with an additional property that indicates if a node is accessed by a write operation or if it is the operand of a write operation. This analysis step is followed by applying a modified version of Tarjan's algorithm [Tarjan 1972] to identify SCCs and it computes cycle-free paths from a method's formal parameters to *relevant* edges added due to write operations. All other edges are ignored.[5]

Connection-Graph Compression. For the Collision Detector benchmark used in Section 6, connection graphs constructed according to the method of Choi et al. have thousands of nodes which considerably affects the duration of the analysis. An extensive amount of vertices originates from global analysis in which they were added to the connection graphs as phantom nodes. Recall that phantom nodes stand for objects with an unknown allocation site, which is the case for objects allocated in callees of the current function. These phantom nodes often have siblings that represent the same objects. To decrease the number of nodes in connection graphs, CMM implements a graph compression algorithm imbued by Steensgard's points-to analysis [Steensgaard 1996] which ameliorates the application of escape analysis within our framework.

The algorithm processes each reference node in a connection graph. It collects information about nodes pointed to by each individual reference node and stores them in lists partitioned according to their escape states. Merging only those nodes with the

[4]Processing the Collision Detector benchmark used in Section 6 (around 29,000 lines of code) takes several minutes.

[5]It should be noted that in Java, it is not possible to mark a method *constant*. Constant methods such as offered, for instance, by C++, do not allow the modification of the value of member variables. The alias analysis part of the escape algorithm could be simplified if Java had constant methods. Consequently, CMM's escape-analysis results could be more accurate for such tagged methods.

```
public void runDetectorInScope(final DScopeEntry d) {
  // ...
  FService srv =
        (FService) PortalService.lookup("FrameService");
  srv.setFrame(f);
  // ...
}
```

Listing. 1. A simplified example for portal communication in KESO's ported version of the CD$_x$ benchmark.

same escape state prevents the deterioration of analysis results. Lists having more than one member and a least one phantom node are condensed by removing all phantom nodes. Edges to these nodes are turned to the phantom nodes' object nodes. Field reference nodes referenced by phantom nodes are put below the object nodes in the compression set and the relocated nodes' outgoing edges are moved, accordingly.

5. APPLICATIONS OF ESCAPE ANALYSIS

The escape analysis implemented in CMM supports ordinary stack allocation, which will not be described. This section presents CMM's additional EA applications based on the analysis results.

5.1. Extended Remote-Procedure-Call Support

Applications colocated on a single microcontroller often do not execute completely independently from each other and need a way to communicate. For this, KESO offers an RPC-like mechanism referred to as portals, which offers service protection and maintains spatial isolation.

Service Protection. Service protection inhibits the invocation of services by unexpected client domains at runtime. A name service is used in the client domain for retrieving the portal object and it returns a `null` reference in domains that did not import the service (i.e., ahead of time by static configuration). In the service domain, the name service will directly return the actual service object. Listing 1 gives an example for acquiring the portal for the *FrameService*. The `lookup` method needs to be provided with a `String` constant and this string does not exist at runtime. The actual lookup is compiled into an array lookup.

Parameter Passing. In Java, primitive parameters of a method invocation are passed *by-value*, whereas objects are passed *by-reference*. This scheme could violate protection-domain boundaries: Write operations of callees to object references of the caller, for example, must never happen. As a consequence, references must usually not be propagated to other protection domains. For this, a deep copy of the objects has to be created in the callee domain's heap. For larger objects, this procedure can be very expensive in space and time. Escape and alias analyses can help to determine if the deep copy is actually needed, which is the case in two situations:

(1) The object itself or any member of its transitive closure, that is, all objects transitively reachable from an object's fields, escape the portal call. In this case, they have a global escape state in the callee's connection graph.
(2) The object or members of its transitive closure are modified by the portal call.

To determine the escape state for method parameters in the first scenario, the connection graph is used: Each parameter passed to the portal call has a complement representation in the callee's domain. The complement of the regarded object and the members of its transitive closure must not have a global escape state. This is computed by using a work-list algorithm. The second condition is somewhat more difficult to prove. The mapping between the connection-graph representation of objects and the index of the portal-call argument that brought them into the portal handler's protection domain is constructed. In all code reachable from the portal handler, the operands of write operations are examined for the existence of this mapping. In a case in which a mapping does not exist, the modified object was not brought into the domain by a portal but originated from the portal handler's domain. As a consequence, the object may freely be modified. If a mapping is present and the currently processed instruction modifies an object passed through a portal, the parameter has to be copied: 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 candidate objects, the mapping is extended and the method's code is visited recursively.[6] If no indications against copy removal are found after the traversal is complete, the copy operation is omitted. Our implementation either completely copies an object and its transitive closure or it does not copy any part of the regarded portal object. It is possible to compute which descendants of an object are modified or escape the callee, and to only copy these parts rather than the entire object's transitive closure. However, this approach requires additional runtime support to maintain the runtime representation of the partial copy information and this circumstance made this a refinement not considered worthwhile.

Our mechanism may be perceived as a variation of the *copy-on-write* technique. In addition, CMM's approach adds the *copy-on-escape* semantics and it is performed ahead-of-time. Our lightweight communication technique based on escape analysis is not only interesting in the context of KESO but also for the *Exotask* model implemented by Auerbach et al. [2009]: Exotasks can exchange data through strongly typed data connections, which are defined in the form of an Exotask graph. Data passed through such a connection is deep copied to the receiving Exotask.

5.2. Stack-Scope Extension

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, however, the callee would instead allocate a new object on the heap and return a reference to it to achieve the same. Using information maintained in connection graphs, objects that escape their method of allocation into the caller but no further can be automatically identified. Since the scope 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 (e.g., 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 is 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, the automated application of `ScopedMemory` specified in the *Real-Time Specification for*

[6]As recursive calls in the application complicate the prediction of the stack size, it is not recommended to make use of such calls in real-time systems. Recursive invocations can be found at compile time by *jino* through Tarjan's algorithm that identifies strongly connected components [Tarjan 1972] and the programmer is advised to check the real-time capability of the application code. In case recursion shall be allowed, it must be bounded as the algorithm for the revised RPC support may enter an infinite loop otherwise. The existence of such upper bounds can also be determined by compile-time analyses.

```
01 public class Factory {
02   class Builder {
03    // ...
04   }
05   protected Builder getBuilder() {
06     return new Builder();
07   }
08 }
09 class Simulation implements Runnable {
10   public void run() {
11     Factory f=new Factory();
12     while (true) {
13       Builder b=f.getBuilder();
14       for (Aircraft a : getAircraft()) {
15         b.addPosition(a, getPositionForAircraft(a));
16       }
17       // b's last reference
18       SimFrame frame=b.makeFrame();
19       simulate(frame);
20    }
21   }
22  // ...
23 }
```

Listing. 2. A simplified example taken from the Collision Detector benchmark CD_j.

Java (RTSJ) [Bollella et al. 2000], 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 EA application in the code and the following description of the algorithms.

Listing 2 shows an example adapted from the source code of the CD_j benchmark [Kalibera et al. 2009] 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 20. 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. 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 a parameter will also increase the escape state. This case is omitted in all examples for simplicity, but always implied.

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. CMM's stack-allocation transformation avoids possibly unbounded growth of stack usage by

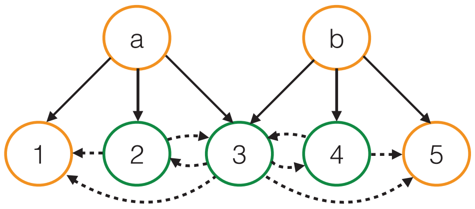


Fig. 3. The call graph illustrates the complexity of scope extension for virtual method calls. Green vertices mark methods that contain allocations eligible for scope extension; orange vertices represent other methods. Solid lines are method invocations. Assume that both a and b contain a single virtual method invocation each, that is, 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 1–5 and their invocation sites must be adjusted for each optimization in 2, 3, and 4.

omitting the transformation into a stack allocation if multiple objects allocated at the same allocation site are in use simultaneously. When the optimization back end 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. Because of the complexity involved in doing so, CMM'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. The last part of this section also takes a closer look at the problems of scope extension.

Non-virtual Calls. Non-virtual call sites, that is, call sites where the invoked method is known at compile time, constitute the simple cases of the analysis. Devirtualization performed by the KESO compiler increases the number of such non-ambiguous invocations where a single candidate can be deduced using static analysis. 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 CMM. 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 few of them are used simultaneously.

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 back end). Figure 3 contains a graphical representation of this problem. Interdependencies between methods cause them to form up into groups 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 methods other 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, CMM 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. 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.

Problems and Limitations. Applying scope extension to all candidates does not yield a better program in all cases. A number of situations can actually decrease 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[7]: The method that shows the undesirable behavior consists of a large distinction of cases where each case allocates and returns an object. Applying scope extension creates a new parameter for each object and adds the corresponding allocation to all callers. Besides the overhead caused by passing a lot of parameters, this example also exhibits two further problems. Firstly, 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. Thus, memory usage is 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 [Sreedhar et al. 1999], 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 subpar performance when a large number of allocations are optimized because of the increased overhead of the modified method invocation.

5.3. Thread-Local Heaps and Regional Memory

The RTSJ [Bollella et al. 2000] is one example for a specification that promotes *hybrid* memory management using static and dynamic memory management: The RTSJ respects the distinct nature of applications by providing several memory partitions called `MemoryAreas` that can be handled by choice of proper candidates out of the pool of available storage-control strategies. The specified `MemoryAreas` constitute RTSJ's memory model and are defined as follows:

[7]The default CD_j-program variant uses a prerecorded, encoded, large set of radar frames denoting the airplanes' positions ensuring deterministic program inputs thereby allowing one to evaluate the impact of various optimizations provided by a JVM compiler.

Table I. Rules for Memory Areas as Stated in the RTSJ

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

—`ScopedMemory` defines a region that exists for a certain time span, that is, as long as there are threads that access objects residing in `ScopedMemory`. Lifetimes are determined by lexical scoping. `LTMemory` and `VTMemory` are specialized variants, where `LTMemory` guarantees linear allocation times and `VTMemory` variable allocation times, respectively.
—`HeapMemory` is used for objects with an undefined lifetime. Only one single instance of `HeapMemory` can exist.
—`ImmortalMemory` is alive for the duration of the program's lifetime. This memory area is similar to static memory reserved ahead-of-runtime.

An implementation of `ScopedMemory` provides semi-explicit memory management, as the allocation and deallocation as well as the selection of the memory areas' sizes and the placement of the data is performed manually by the application programmer. Special Application Programming Interface (API) methods to handle the regions are defined by RTSJ. `ScopedMemory` is used by setting its total size and by associating one or several real-time threads for its instantiation. The application programmer activates the region by invoking its `enter()` method, which runs the desired thread. A reference counter is used to keep track of active threads. When the last thread leaves the context of the `enter()` method, the reference counter drops to zero. As a consequence, all objects in the region are finalized and their memory is reclaimed, no matter if any references to these objects still exist. To avoid the forging of dangling references and to ensure the program's type safety, RTSJ defines rules as shown in Table I for handling the memory areas. An implementation of the RTSJ has to check for the compliance of these rules (at compile time or runtime): RTSJ states that the checks *must* be performed on each assignment before the statement is executed. In particular, the specification suggests *that* static analyses could be used for that purpose. However, *which* analyses should be used and *how* these analyses operate is not described any further. `ImmortalMemory` is also handled manually by reserving space for a number of instances of a particular type in that region.

To provide an automated solution for RTSJ's regions described in Table I, escape-analysis results can be used. Turning heap allocations into stack allocations is not always an appropriate solution for the reasons discussed in Section 5.2. Depending on an application's memory-usage behavior, a memory-management strategy completely avoiding heap allocations can result in a system with higher memory consumption in contrast to a setup that incorporates a heap. This happens if the amount of memory reserved for all worst-case stack sizes totaled is larger than the amount of memory needed for all objects. A program's efficiency can be ameliorated by allocating objects, which are not shared across threads in a heap not handled by a tracing GC. Hence, CMM puts objects determined to be thread-local by our escape analysis in a separate heap on request and this special heap region can coexist with garbage collection. To enhance the runtime behavior of an optionally employed GC-managed heap, it is important that the thread-local objects are located in a distinct memory area so that the GC does not have to mark and sweep those objects.

We provide a logical region for each method[8] and consequently, upon a method's end its region can be reclaimed at once. The regions are implemented in a stack-oriented manner to keep the semantics of stack allocations. A particular implementation satisfying these criterions is thread-local heaps, which have been proposed by prior work by Domani et al. [2002] and Lee et al. [2007]. In contrast to our approach, these solutions make use of profiling techniques rather than fully compiler-assisted region control, and our implementation of thread-local heaps can be used to provide an automated solution to RTSJ's `ScopedMemory`. CMM implements thread-local heaps similar to a stack or regional memory: Each heap has a region pointer and a maximum fill level. The region pointer is saved upon method beginning and thread-local objects are allocated by moving the region pointer. Respectively, upon method exit the region pointer is reset to its earlier value. Saving the region pointer to stack memory can be optimized out given that the memory needed within a function and all alignment cutoffs can be derived at compile time.

It should be noted that only allocations reachable from a thread's constructor or entry function can use thread-local heaps and CMM ensures this thereby preventing runtime errors. In the current implementation, interrupt handlers do not use thread-local heaps but they could employ their own heap or the local heap of the thread that they preempt. For both approaches, the minimum interarrival time of interrupts could be respected to keep the memory usage bounded.

5.4. Handling Immutable Data

Static memory can comprise mutable and immutable data. The handling of immutable data depends on the programming language being used and can be complicated. For CMM, we developed an automated solution by leveraging alias information collected in the connection-graph representation to compute compile-time-constant and runtime-constant, that is, immutable, data. Recall that we employ escape analysis to determine the share of immortal, *mutable* objects (see Section 5.3 that described escape-analysis-based regional memory). Hence, handling immutable data completes the realization of RTSJ's `ImmortalMemory`. However, CMM's version of it differs from RTSJ's proposal by intention: `ImmortalMemory` handles both mutable and immutable data but does not distinguish between compile-time-constant and runtime-constant data, thus it does not take the possible use of Read-Only Memories (ROM) into consideration. In contrast, CMM's handling of immutable data is twofold: The share of static memory occupied by truly constant data, that is, data computable at compile time, can be placed in read-only memory. Runtime-constant data can be allocated by means of pseudo-static allocation (that is, the memory is allocated during program initialization) or at any time during the program's execution and is stored in RAM. Thus, the semantics of CMM's implementation do not directly map to RTSJ's `ImmortalMemory`, however, we believe that this solution respects the microcontroller's memory characteristics in a better way.

The knowledge of compile-time-constant data is very precious as this data can be placed in ROM; for instance, flash memories: A ROM unit is significantly cheaper than RAM, which is a scarce resource in embedded systems. More importantly, flash memories are much less susceptible to soft errors [Cellere et al. 2009]. Stored flash data does usually not need to be protected against transient faults. Other advantages are similar to the ones mentioned with escape analysis such as a reduced overhead during garbage collection. The Hardware-near Virtual Machine (HVM) [Korsholm 2011] also features a particular handling of compile-time-constant data and provides a manual annotation-based solution, whereas CMM derives both types of immutable data

[8]Empty regions without local objects can be discarded.

through a compiler-based approach. A developer can make use of the `final` keyword to tag compile-time constant data, but it is not required.

In addition to the benefits stated so far, the knowledge of runtime-constant fields is also useful since *jino* can employ the information to optimize the program in a better way: As the compiler knows that the field is initialized, that is, the field is not `null`, it does not need to emit `null` checks for accesses to this field. Moreover, since the field is written exactly once, the initialization value can be propagated to all reads of this field. This may in turn encourage folding and other optimizations that benefit from folding such as turning conditional branches into unconditional ones, further supporting dead code elimination.

In the following, we *briefly* present our procedure of how to identify immutable data. A detailed explanation of the algorithm and static allocation in the context of the *Von Neumann* and the *Havard* architectures[9] is given in a separate paper [Erhardt et al. 2014].

Finding Immutable Data. The analysis is termed *effectively final* analysis and operates on SSA and makes use of aliasing relationships maintained by connection graphs computed during escape analysis. It processes further information computed by *jino*'s data-flow analysis that is based on Wegman-Zadeck's *Sparse Conditional Constant Propagation* [Wegman and Zadeck 1991]. Moreover, results computed by *jino*'s dominator analysis are needed to derive immutable data. Dominator analysis calculates the *dominance* relationships for each basic block[10]: A control-flow graph consists of basic blocks. A basic block B1 dominates another basic block B2 if every path from the starting node in a control-flow graph to B2 goes through B1. Consequently, a statement S1 dominates another statement S2 if the basic block to which S1 belongs dominates the basic block containing S2 or in case S1 and S2 are part of the same basic block, S1 is situated before S2.

To declare a field *final*, three conditions must be derived: First of all, the field is written once in its associated object's class constructor. Secondly, the write statement associated with that field must dominate the constructor's exit. Thirdly, any read statements on that field within the constructor must be dominated by the write statement. To make sure that the determination of these three conditions is sufficient to find static final fields, the KESO runtime system has to invoke each protection domains' class constructors one after another before program execution so that no class is used before it has been initialized. In addition, constructors containing potential final fields must never block, that is, they do not contain any `WaitEvent()` system calls or methods that invoke this system call. Finding constant arrays is similar to finding effectively final fields, that is, the analysis has to identify arrays whose members are constant. In addition, the members have to be initialized with compile-time constants.

5.5. Object Inlining

Java usually does not store field instances by value but rather by reference in contrast to the C/C++ language that always *embeds* (or *inlines*) such unique instances and only uses the by-reference semantics if requested by the programmer during class definition. Hence, C/C++ promotes co-allocation of member data and possibly allows the data to be placed in the same cache line. Figure 4 gives an example for the difference between by-value and by-reference semantics. Both approaches exhibit pros and cons.

[9]The Von Neumann architecture has a unified address space, whereas the Harvard architecture has multiple address spaces. Some low-end microcontrollers use multiple address spaces. The peculiarities of the respective platforms have to be respected during code generation in *jino*.
[10]A basic block contains neither jumps nor labels.

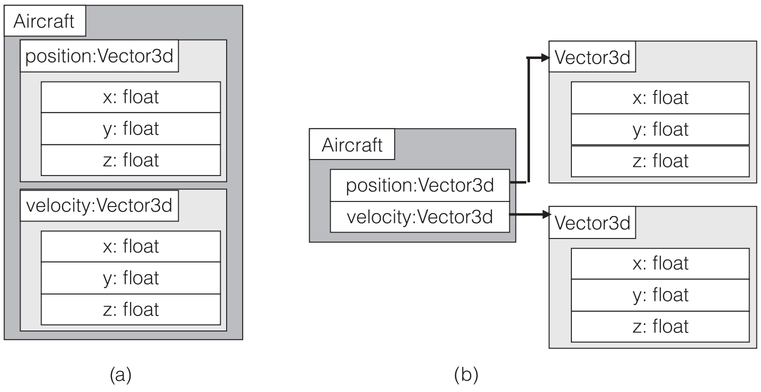


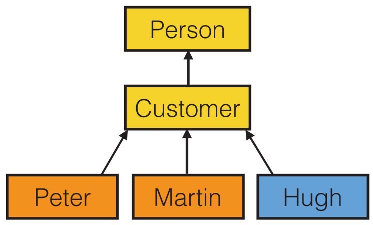
Fig. 4. Value fields (a) and reference fields (b): (a) shows objects (`Vector3d position, Vector3d velocity`) embedded in `Aircraft`. This object placement can be enforced manually by the C/C++ developer, whereas in Java no such means exists; (b) depicts the normal Java approach, where `position` and `velocity` are put into two separate objects that are referenced by `Aircraft`.

The designers of the Java language decided against the facility to manually inline class members. Allowing for programmer-controlled embedded objects imposes several issues in a strongly typed development environment such as the following: Additional static and dynamic checks are necessary to ensure that the object to be stored is type-compatible to avoid endangering the type system's integrity. If the compiler cannot statically determine a unique type for the object to be inlined, the storage for the largest possible subtype has to be provided. The possibility to dynamically load new code even aggravates the problem. Also, storing objects by reference rather than by value is beneficial from a software-design perspective as the view on and handling of all objects and their memory representation is uniform and transparent to the programmer.

In contrast to the by-value semantics, the by-reference semantics results in higher runtime overheads imposed by pointer indirections, allocation code, and the need to store both the field and object header. These additional runtime costs can be problematic in embedded systems. Several research groups engaged with an automated technique to embed objects, which is often referred to as *object inlining* [Dolby 1997; Dolby and Chien 1998, 2000; Ghemawat et al. 2000; Laud 2001; Lhoták and Hendren 2002] to allow modern, managed languages to benefit from embedded objects. Their work was done in the context of dynamic systems and they trade preciseness of analysis results off against the translation time of just-in-time compilers. Lhoták and Hendren, for example, perform a dynamic trace to identify inlinable fields. In the context of KESO, we can apply ahead-of-time escape analysis leveraging the properties of the language's strong type system and the closed-world-assumption to mimic object inlining known from C/C++ to improve performance. Due to these constraints, our analysis focuses on preciseness rather than compilation times. Due to our modified version of escape analysis in which we apply connection-graph compression, the compilation times are acceptable.[11]

A compiler-based approach also exhibits several advantages over a manual solution: Using C/C++, manual object inlining is not as frictionless as it might appear at first sight. The liveness properties of objects have to be analyzed by the developer, as embedded objects and their parents are basically a form of regional memory. An object's by-value data is alive as long the object is in scope. This instance might lead to increased memory consumption in unfortunate scenarios in which long-lived and short-lived data

[11]It takes only a few seconds to compile a CD$_j$ benchmark variant that consists of approximately 30kLoC.



```
class Person {
        private: Address address;
        // ...
};

class Customer: public Person {
        private: Account account;
        // ...
};
```

Listing. 3. An example in the context of C++: Adapting member criteria complicates uniform software development as objects might have to change their memory representation due to new requirements.

is inherent to the same object. Furthermore, embedded objects have an ownership, that is, they belong to an object in which the membership is defined. If the class properties change, it might sometimes be necessary to place the member object in another memory location. Listing 3 visualizes an example that suffers from different memory representations from a software engineering point of view. The developer decided that each `Customer` should have their own account. In that case, the instance variable `account` can be embedded in the `Customer` objects `Hugh` and `Martin`. However, the software requirements change, so that common accounts are also allowed, so that `Peter` and `Martin` can share an (unembedded) account but `Hugh` still has his own instance that could be embedded. Technically, to provide a uniform access, the by-reference semantics can be used. In this use case, all access operations applied to the member objects would have to be adapted. But the by-value semantics would still be useful for `Hugh`. To provide efficiency for `Hugh` to access his account and shared account access for `Martin` and `Peter`, a new account type with the same functional properties but a different memory representation would have to be introduced and that requires the developer to adapt the class hierachy. From the composability and extensibility point of view, manual object inlining is therefore problematic. Using by-reference semantics solves this issue but voids all the benefits known from by-value storage. Even if the system design is stable, the adaptability aspect of the memory representation of objects can be of particular interest in embedded systems as several non-functional properties influence each other. Optimizations based on manual approaches or static analyses might be useful to increase the performance of software, but may decrease the reliability of the program at the same time. A proper balance between the quality attributes has to be found. While the code in Listing 3 is trivial and both extensibility and variability could be imitated through (user-unfriendly) wrapper code,[12] imagine the following use case: Increased memory consumption and attack surface to soft errors imposed by deferred memory reclamation of object inlining, decreased performance (memory and execution time) of by-reference semantics due to introduced indirections, and consequently, the risen vulnerability of the system against transient hardware faults have to be evaluted on a per-case basis to find the best possible trade-off. A generative technique that is able to automatically inline objects supports this approach as the code base does not have to be adapted by the developer.

Automated object inlining is more accurate in type-safe than in weakly typed languages due to a better analyzability of pointer aliasing relationships. The compiler-based solution is particularly supported by KESO's ahead-of-time approach of gathering as much static information as possible and the closed-world assumption allows one

[12]Aspect-oriented code weavers could also be a solution, but this source-to-source translation approach is less powerful than using data- and control-flow information collected by whole-system compilation.

to have object inlining as known from C/C++. It further introduces additional interesting properties: First of all, memory-safety bugs known from C/C++ are not an issue. Moreover, embedded objects do not need to be scanned by the garbage collector and the memory of the embedded and surrounding object can be reclaimed as a whole. The co-allocation of objects that belong together semantically also improves data locality. A moderate increase in memory usage due to deferred memory reclamation may also be acceptable in soft-error-prone systems using Load-Reference checks (LRCs) and Dereference checks (DRCs) to detect bit flips in references [Stilkerich et al. 2013, 2014b]. As embedded objects are not referenced, the performance of such systems is better. The best possible trade-off is application-specific and has to be selected on a per-case basis. Due to the automated approach, an easy static and dynamic evaluation of the trade-off is possible. Compiler-assisted object inlining is a worthwhile optimization for some candidates exposing particular properties while preserving a uniform view on the objects no matter if they are inlined or not. In this way, software design and other non-functional properties such as efficiency and reliability can be addressed at the same time.

Candidates for Object Inlining. The results computed by escape and effectively final analysis can be employed to identify candidates for object inlining: A reference to an object eligible for inlining must only point to that one object to be embedded. Effectively `final` and programmer-tagged `final` references are possible candidates for the optimization. In the following, we refer to those references as *final* references no matter if this condition was inferred by our analysis or if the marking was performed by the developer. Final references are initialized once[13] and do not change during the program's lifetime. Objects pointed to by these references do not grow and do not exceed the lifetime of their parent object. Hence, increased memory consumption by inlining applied too aggressively is not an issue in CMM. The information about the embedded object is cross-checked with the statically configured threads' stack sizes to prohibit stack overflows. Runtime stack checks are thus not necessary.

All methods are visited, searched for assignments to final references, and tagged as *relevant* methods, accordingly. A relevant method has to contain an allocation operation belonging to a final reference. Hence, the method's connection graph is useful to identify inline candidates: If a field-variable node points to a phantom node, that is, a node with an unknown allocation site, it indicates that the object has already been created in another method context and consequently, that the object cannot be inlined. Thus, the necessary one-to-one mapping property between reference and the object to be embedded can be ensured. We term embedded objects *inlined objects* and references to those objects *inline rereferences* in the remainder of this section.

Determining the Largest Dynamic Type. To inline an object meeting the aforementioned properties, the type of that object has to be calculated, as the object's type determines the storage space occupied by the inlined object. To guarantee memory safety, our analysis has to identify the largest dynamic type from the inlined object's inheritance hierarchy. The concrete dynamic type of an object can sometimes not be derived at compile time as exemplified in Listing 4 and therefore the space for the largest subtype (class `B` in the example) has to be allocated. Several ways exist to compute the size of the memory space needed. Traversing the entire class hierarchy and summing up all sizes is tedious, as subclasses can in turn contain inlined objects. A fix-point iteration can help to solve the issue, but the algorithm is not trivial and also might allocate space that is not actually needed.

[13]Note that the initialization of final references does not necessarily have to be performed in the constructor due to possible (method) inlining of the constructor itself or due to a deferred initialization associated with the effectively final semantics.

```
class A {                              class Foo {
  int a=3;                               public A vector;
  int b=4;                               public void boo() {
};                                         int x = new Random().nextInt();
                                           if(x>0) {
class B extends A {                          vector=new A();
  float x = 7.1f;                          } else {
  float y = 8.1f;                            vector=new B();
};                                         }
                                         }
                                       };
```

Listing. 4. Static analysis cannot identify the concrete dynamic type of `vector`.

Therefore, we decided to address the problem in a different way. First of all, we offload work to the underlying C compiler. *jino* translates Java bytecode to C and we can make use of the `union` constructs offered by C. Unions can contain values with different types in one field and only one of the values is actually stored. The C compiler determines the largest field and allocates the appropriate amount of memory. This approach preserves memory safety. In embedded systems, however, memory constraints have to be taken into consideration and too much space might be allocated by just respecting the class hierarchy. Hence, we incorporate assignment statements during code analysis which determines the fields actually added to the union. By combining the approach with a control-flow-sensitive analysis that preserves dynamic type information for each field, the results are still too inaccurate. Taking Listing 4 as an example, the analysis registers two assigments to the `vector` field and derives the most general type `A`. This is sufficient in some cases but is inaccurate if more than one type is assigned to the target reference. A more suitable and general solution is offered by the connection graphs computed by escape analysis. Whenever a `new` instruction is found in the code and the created object is referred to by a reference field, escape analysis adds an edge from the field to an object node with the dynamic type in the connection graph. The intermediate code representation of allocations holds the needed type information.

Avoiding Cycles in the Inline Graph. A problem that can arise is cycles within inline references. Through particular constructions, classes could cyclically inline each other and therefore trigger continuous allocations. Thus, each inlined object that contains other inline objects could cause a memory-consumption explosion. To address this problem, cycles have to be broken and for this, we construct an *inline graph*. An inline graph describes a directed graph whose edges represent inline references. The vertices are classes that either hold the reference (source of edge) or can be inlined (destination of edge). Outgoing edges express that the source class has at least one reference with the dynamic type of the destination class. To create the inline graph, it is necessary to check each class's inline references and add an edge from the surrounding container class, which holds this reference, to each possible dynamic type class. Before dissolving cycles, *strongly connected components* are identified using Tarjan's algorithm [Tarjan 1972]: All vertices in a strongly connected component are reachable from each other's vertices. The information about strongly connected components is used to remove edges from the inline graph. Cycles are detected by means of a Depth First Search (DFS). Edges constructing a cycle are removed from the graph. As a consequence, all class's references responsible for such edges are marked as not inlinable. To minimize the number of edges being removed, a heuristic is employed. As a starting point, the vertex

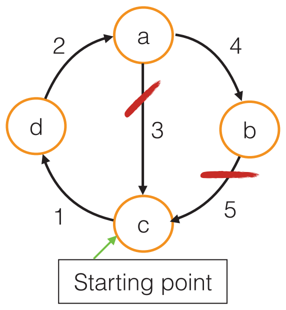


Fig. 5. Choosing the starting point randomly causes a non-optimal solution. The search with DFS deletes edges that produce cycles. Consequently, two edges are removed: A→C and B→C.

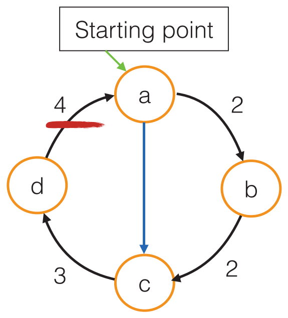


Fig. 6. Choosing the starting point depending on the vertex that has the most outgoing edges works better. The search traverses the graph again with DFS and deletes edges that produce cycles. Here, only one edge is removed: D→A.

with the most outgoing edges is selected. Figures 5 and 6 illustrate the procedure and explain why this heuristic is sensible.

5.6. Survivability: Machine-Independent Space- and Time-Bounds Analyses

In real-time systems, the memory consumption and runtime has to be predictable. Usually, the worst-case memory-consumption and execution-time analyses consist of a machine-dependent and a machine-independent part. In the machine-dependent part, hardware-specific information such as the execution times of basic blocks containing the processor's instructions, caching, and pipelining effects are respected. The machine-independent part is performed by static program analysis. In embedded real-time systems, dynamic heap memory management is still rarely used. In combination with type-safe languages, however, fragmentation-tolerant garbage collection as proposed by Pizlo et al. [2010b] and Stilkerich [2016], for example, can efficiently be employed when supported by escape analysis. Besides the overall better throughput of garbage collection, EA can help to improve the predicted upper space and time bounds for real-time garbage collection as discussed by Stilkerich et al. [2014a] since the overhead imposed by a GC depends on the number of *surviving* objects: The objects are put into categorization classes with respect to their *survivability*, that is, if they are able or unable to outlive a GC run. As *jino* performs whole-program analyses on type-safe code, those categories and the objects belonging to them can automatically be determined: On the one hand, objects may completely be extracted from garbage collection according to their categorization and will never contribute to the fragmentation issue. On the other hand, due to the liveliness criterion and the knowledge of the points in time the GC is scheduled, we derive the objects' survivability by means of a combination of escape and call-graph analysis to the `waitEvent()` system call (i.e., use of operating-system knowledge). Thus, we can determine upper space bounds that are considerably lower than those derived by commercial tools without escape analysis on type-safe code and system-specific knowledge at compile time. Furthermore, the derived information can be used to assist machine-dependent analysis tools that determine the exact time bounds. In the following, we give a *short* insight into machine-independent reasoning about space and time bounds for allocations on a fragmentation-tolerant heap as implemented by CMM's *RT-LAGC* [Stilkerich et al. 2014a]. More information about real-time garbage collection in the context of KESO can be found in Stilkerich [2016].

5.6.1. Method-Local and Region-Local Objects. In case stack and region allocation is used, local data can be administrated in special memory areas. Stack memory is handled by moving the stack pointer and likewise, thread-local heaps are managed by moving the region pointer. All operations are of constant complexity on the TriCore TC1796 device we used for the evaluation.

Method-local and region-local objects that do not reside in blocking methods do not survive a GC cycle and the space estimation can be adapted, respectively. In Section 6.2.1, we give insight into the impact of such data in the context of the CD_j benchmark.

5.6.2. Allocations on RT-LAGC's Heap. Objects that are not managed in stacks or regions are allocated on the RT-LAGC heap. Allocations of higher-priority threads can interrupt allocations of lower-priority threads. RT-LAGC features an interrupt-friendly implementation of the free-memory list [Stilkerich 2006; Stilkerich et al. 2014a]. All critical sections are short and of constant complexity and constitute the time by which a higher-priority application can be delayed by an allocation of a lower-priority thread. In the following, we outline how the bounds for the allocation operation can be derived.

Space Bounds. Heap objects consist of fragments. For regular objects, the number of fragments occupied by an object is constant for each class and computable at compile time. Array objects can be contiguous or fragmented. The use of contiguous arrays is more time- and space-efficient, but for the worst-case estimation, fragmented arrays have to be considered. Fragmented arrays of statically known size consist of a derivable number of fragments and spines, which maintain array metadata: The spine size and the number of required fragments is dependent on array size and the size of the array members. If *jino* cannot derive this information, it prints a warning for the developer. The spine space needed contributes to space requirements: Exemplary for handling of spines, the amount of memory needed for array spines multiplied by 2 is required to handle array metadata. In addition, the number of objects that survive a GC cycle has to be respected. This approach allows one to estimate the upper space bounds for both arrays and regular objects.

Time Bounds. To allow an efficient solution for array objects, RT-LAGC optionally tries to allocate contiguous arrays. An upper time bound for allocation is given by limiting the number of tries to allocate contiguous fragments. The current GC configuration limits the contiguous allocation to three tries and this constitutes the upper time bound for contiguous array allocation. Spines needed for fragmented allocation are allocated by incrementing an index. This operation is independent of the spine size and is of constant complexity.

In the worst case, each fragment—regardless of whether the fragment belongs to a regular object or array—has to be taken from a different free-memory block, thereby constituting an upper time bound for object allocation.

If the allocation fails due to memory exhaustion, the error is signaled and can by handled by application-specific code. The overall timing requirements have to ensure the exception handling function can be completed in time.

6. EVALUATION

To illustrate the effectiveness of our optimizations for escape analysis, we selected the following EA applications for evaluation:

—Extended remote-procedure-call support
—Stack-scope extension and thread-local heaps
—Automated inference of immutable data

Table II. Execution Time for Portals

Call Type	Execution Time
portal call	3.76μs
regular virtual method call	3.09μs
AUTOSAR nontrusted function	32.91μs
portal call (two int params)	4.17μs
portal call (three int params)	4.70μs
portal call (one element linked list)	31.47μs
portal call (two element linked list)	56.08μs
portal call (three element linked list)	84.37μs
portal call (escape analysis, one element linked list)	3.99μs
portal call (escape analysis, two element linked list)	4.19μs
portal call (escape analysis, three element linked list)	4.65μs

Due to space limitation, we cannot present the results for all implemented EA applications proposed in this article. We performed a microbenchmark for KESO's portal mechanism. Furthermore, we examine the footprint, runtime, and heap usage of selected KESO configurations. For this, we employ the Java version of the real-time Collision Detector (CD_x) benchmark, which is available in a C (CD_c) and a Java (CD_j) version.

CD_j can be configured in different ways to adapt to the runtime environment being used. We employ the CD_j benchmark in the *onthegoFrame* variant [Stilkerich et al. 2012; Stilkerich 2012] using six airplanes. This configuration variant features a simplified radar-frame calculation. The measurements for the CD_j configurations were performed by processing 10,000 frames. The input values for the employed CD_j configuration are fixed and reproducible results are obtained during the evaluation.

The used program variant contains around 29,000 lines of code, three configured threads, and is deployed on the Infineon TriCore TC1796 device equipped with a 150MHz CPU clock, a 75MHz system clock, 1MiB external SRAM, and 2MiB internal flash. The application is translated to ANSI-C code using the KESO compiler. The generated code is bundled with an AUTOSAR OS implementation and compiled with GCC (version 4.5.2). Code and constant data is located in internal flash. The heap size is set to 600KiB that is managed by a mark-and-sweep garbage collector.

Section 6.1 presents the results for portal services assisted by escape analysis, while Section 6.2 introduces the CD_j benchmark and the evaluation of the latter two EA applications on top of CD_j. A software-partitioned version of CD_j using portals exists, however, this fully fledged variant does not fit on the TriCore device (neither in the C version with hardware-based protection nor the Java versions (hardware- and software-based memory protection)). We also believe the microbenchmark scenario best visualizes the various types of inter-domain communication.

6.1. Extended Remote-Procedure-Call Support

An essential evaluation is that of inter-domain communication as it determines the extent to which developers will actually place software in different protection domains. We perform microbenchmarks on the cost of different variants of portal calls and compare them to the cost of a regular virtual method call (i.e., without spatially isolated components) and also a non-trusted function call in an AUTOSAR OS implementation (i.e., spatial isolation enforced by region-based hardware protection). Table II shows the results. The method bodies of all target methods are empty (i.e., portal parameters are not modified and do not escape). Hence, we only measure the cost of the protection-domain context switch that is performed on a portal call.

Types of Protection. For comparing the cost of a portal call to the cases of no spatial isolation and hardware-based spatial isolation, we use the simplest form of a function that does not take any parameters and does not return a value. The portal call introduces an overhead of 22% over the regular virtual method call. The overhead is attributed to service protection (i.e., the check, if the calling domain is a valid client to the service) and the change of the running task's effective domain. For comparison, we have also included the cost of a comparable non-trusted function call in an AUTOSAR operating system, which is comparable to a portal call but with domains isolated by hardware-based memory protection rather than constructive software-based memory protection. This measurement shows that the cost of a software-protection context switch is significantly less than that of an MPU reconfiguration that is needed in the case of AUTOSAR OS.

Portal Calls with Parameters. We also measured the time needed for portal calls with both primitive and reference parameters. In the case of primitive parameters, the added cost is the same as for any regular function that is expanded with parameters. The cost depends on the actions that the C compiler needs to take in order to prepare the parameters according to the Application Binary Interface (ABI). The measured overhead therefore mainly depends on the C compiler and the ABI and is not caused by the portal mechanism. For portal calls with reference parameters lacking the escape analysis support presented in Section 5.1, we passed the head pointer of a linked list of size 1–3. During the portal call, a complex routine that copies the referenced object and all transitively reachable objects to the heap of the target domain is invoked. The cost of the call is dominated by this operation and increased by an order of magnitude compared to the portal calls with only primitive parameters.

Activating the escape analysis support improves the portal mechanism significantly through the safe removal of parameter copies. To fortify the validity of this evaluation, we enriched the method bodies of the target methods with code, which does neither modify portal parameters nor causes the portal parameters to escape (Footnote: and likewise with code that modifies portal parameters or that causes the portal parameters to escape), thereby confirming the effectiveness of legally removing the copy of portal parameters by escape analysis while still retaining software-based memory protection provided by the KESO JVM (i.e., no memory-protection unit is used).

6.2. The CD$_x$ Benchmark

The core of the CD$_x$ application is a periodic thread that detects potential aircraft collisions (detector component) from simulated radar frames (simulator component). A collision is assumed whenever the distance between two aircraft is below a configured proximity radius. The detection is performed in two stages: In the first stage (reducer phase), suspected collisions are identified in the 2D space ignoring the z-coordinate (altitude) to reduce the complexity for the second stage (detector phase), in which a full 3D-collision detection is performed (detected collisions). A detailed description of the benchmark is available in a separate paper [Kalibera et al. 2009]. Since CD$_j$ allocates temporary objects and uses collection classes of the Java library, it requires the use of dynamic memory management. We evaluated escape analysis in combination with a throughput-optimized GC variant available in our KESO JVM (that is, the fragmentation-tolerant GC is not used).

6.2.1. Stack-Scope Extension and Thread-Local Heaps.

In this evaluation section, ordinary stack allocation is contrasted to scope extension and thread-local heaps to determine the impact of these EA applications.

Allocation Sites. The share of optimized allocations can be used as a compile-time criterion of CMM's optimizations. The higher the number of objects managed by compiler-assisted memory management, the lower the GC's heap load, which may reduce the garbage-collection workload. For the number of stack allocations without using scope extension, 44 of 146 (30.1%) allocations are eligible for stack allocation. 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. 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[14] 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, that is, 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.

Footprint. Stack allocation and thread-local heap allocation increase the size of the code. 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 is 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 thread-local heaps are used. Scope extension further increases the size of the code unless methods with candidates for the optimization only have a single caller. Since the *onthegoFrame* 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.

The data-segment size does not change for stack allocation. When using thread-local heaps, each configured thread-local heap adds two additional pointers to the data segment. The size of the data section grows by 24 bytes (the size of two pointers on the 32-bit TriCore target times three thread-local heaps).

Heap Usage. The median heap usage for escape analysis with the stack-allocation optimization back end is depicted in Figure 7 and is only 50.7% relative to a run without optimizations based on escape analysis. When using thread-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. When enabling scope extension, fluctuations in heap-memory usage present are smoothed in contrast to configurations without scope extension. The median heap usage is reduced to 50.4%. When using thread-local heaps, the number again is similar to stack allocation but a little lower: The median heap usage is 49.8%. The lower variance is caused by invocations that only occur in some of the collision-detector iterations. The invoked methods allocate objects in heap memory. These allocations seem to be candidates for scope extension and are hence no longer allocated in the heap.

[14]((174 − (146 − 12))/12).

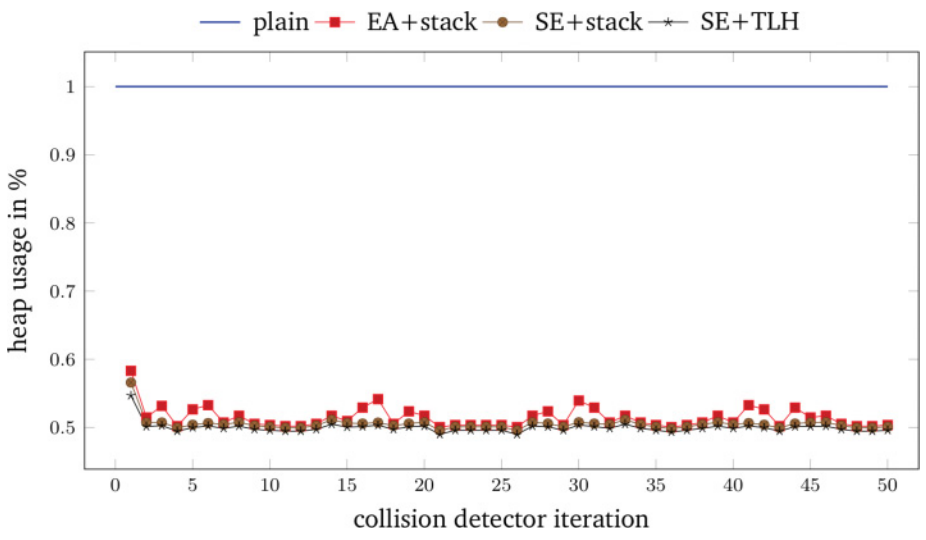


Fig. 7. Heap memory usage of the on-the-go variant of the CDj benchmark with (a) scope extension and stack allocation (SE+stack), and (b) scope extension with task-local heaps (SE+TLH) relative to a run without escape analysis-based optimizations (plain). For comparison, heap memory usage for escape analysis and stack allocation (EA+stack) is also shown.

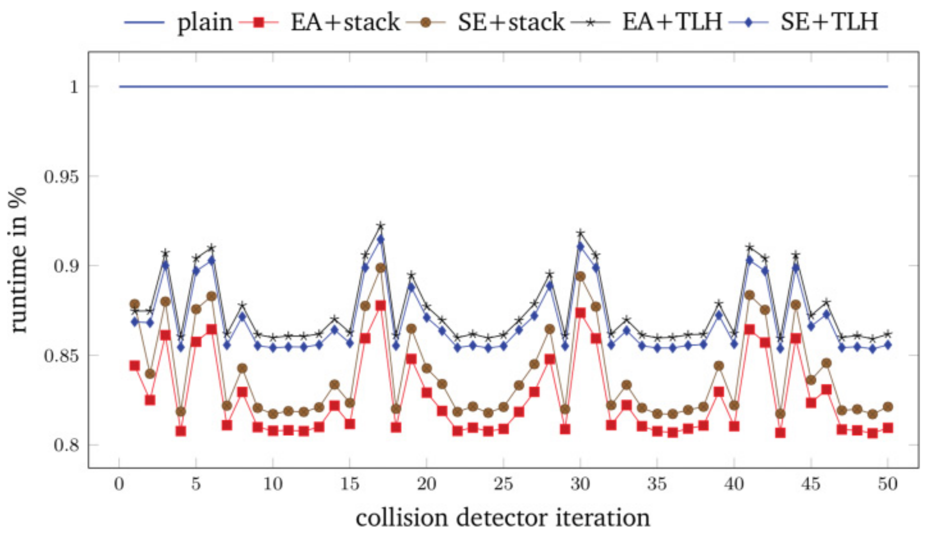


Fig. 8. Runtime of the on-the-go variant of the CDj benchmark using (a) escape analysis with stack allocation (EA+stack), (b) escape analysis with task-local heaps (EA+TLH), (c) scope extension with stack allocation (SE+stack), and (d) scope extension with task-local heaps (SE+TLH) relative to a run without escape analysis-based optimizations (plain). 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.

Execution Time. In terms of execution time (illustrated in Figure 8), escape analysis and stack allocation perform significantly better with a median of 81.1% relative to the baseline stated by the same KESO configuration without this optimization. As expected due to the additional instructions managing regions in thread-local heaps on method entry and exit, stack allocation is faster than the code generated by the thread-local-heap allocation back end. The median runtime improvement for thread-local heaps is 13.7% compared to 18.7% for stack allocation. It should be noted that CD_j on top of KESO using a throughput-optimized GC variant, escape analysis and stack allocation is 15% faster than the native CD_c[15] in the median execution time, despite the overhead of runtime checks, virtual method calls, and overhead to maintain the runtime data structures of the runtime environment. A detailed evaluation of CD_c and CD_j on top of KESO can be found in a separate paper [Stilkerich et al. 2012] showing that the Java programs compiled by KESO are competitive to native C programs in terms of execution-time performance and memory footprint. While enabling scope extension further reduces the heap-memory usage, the same is not necessarily true for execution time. For the stack-allocation back end, enabling scope extension slows down the median time needed by the collision detector by 1.14 percentage points to 82.3%. The thread-local heap back end, 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 5.2.

Overall, using escape-analysis results considerably improves performance and reduces the heap memory requirements. Thus, it is recommended to enable escape analysis and one of its applications for all program variants. 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 5.2, for example by significantly increasing the code size. For some applications, the decreased variance 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.

6.2.2. Automated Inference of Immutable Data. The effectively `final` analysis reduces the data-segment size by 44% and the text-segment size by 14%. This is attributed to constant folding, folding of conditional branches and the resulting dead basic blocks. Also, lots of reference fields are known to be initialized and do not have to be checked upon access: The number of `null` pointer checks emitted has been reduced by 30%, which contributed to the code size reduction. The effectively `final` analysis reduced the execution time of the overall CD_x application by 10% due to dead code removal[16] and runtime-check elimination. Afterward, placing constant data in ROM instead of RAM increased the CD_j's runtime by 6% in turn due to higher access times to flash memory. Thus, the overall execution time with constant data placed in ROM is still better than a KESO variant without effectively `final` analysis.

CD_j does not contain constant arrays, but 1KiB of string constants, which can be moved to flash memory. Hence, the influence of the immortal object analysis is marginal for this benchmark.

7. CONCLUSION

In this article, we presented our applications for escape analysis in embedded real-time systems. We evaluated a fast RPC mechanism for software-isolated components with a microbenchmark and measured the effects of two further selected EA applications

[15]Deployed under the same setup mentioned in Section 6.

[16]Without the automated inference of immutable data, constant values not tagged `final` cannot be propagated and consequently, affected basic blocks are still reachable though they would not be needed. Thus, they cannot be removed.

in the context of the comprehensive real-time CD$_x$ benchmark. We believe that these optimizations open up new possibilities for Java in the safety-critical embedded domain. The characteristics of type-safe languages in combination with a static system setup and static analyses in general affect embedded development positively: Besides safety and security benefits as well as increased productivity, the evaluation shows that Java programs are efficient in terms of execution-time performance and memory footprint. Our design is not limited to the Java language, but can be applied in any system featuring a type-safe language incorporating system-specific knowledge that is available ahead-of-time.

REFERENCES

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. AW, Boston, MA.

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

Joshua Auerbach, David F. Bacon, Daniel Iercan, Christoph M. Kirsch, V. T. Rajan, Harald Röck, and Rainer Trummer. 2009. Low-latency time-portable real-time programming with exotasks. *ACM Trans. Embed. Comp. Syst.* 8, 2 (2009), 1–48. DOI:http://dx.doi.org/10.1145/1457255.1457262

AUTOSAR. 2009. *Specification of Operating System (Version 4.0.0)*. Technical Report. Automotive Open System Architecture GbR.

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

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

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

G. Cellere, et al. 2009. Neutron-induced soft errors in advanced flash memories. In *IEDM 2008*. IEEE.

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

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

Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape analysis for Java. In *14th ACM Conference on OOP, Systems, Languages, and Applications (OOPSLA'99)*. ACM, New York, 1–19. DOI:http://dx.doi.org/10.1145/320384.320386

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

Daniel Danner, Rainer Müller, Wolfgang Schröder-Preikschat, Wanja Hofer, and Daniel Lohmann. 2014. Safer sloth: Efficient, hardware-tailored memory protection. In *Proceedings of the 20th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS'14)*. IEEE, 37–47.

Julian Dolby. 1997. Automatic inline allocation of objects. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI'97)*. ACM, New York, 7–17. DOI:http://dx.doi.org/10.1145/258915.258918

Julian Dolby and Andrew Chien. 2000. An automatic object inlining optimization and its evaluation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'00)*. ACM, New York, 345–357. DOI:http://dx.doi.org/10.1145/349299.349344

Julian Dolby and Andrew A. Chien. 1998. An evaluation of automatic object inline allocation techniques. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*. ACM, New York, 1–20. DOI:http://dx.doi.org/10.1145/286936.286943

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

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

Christoph Erhardt, Simon Kuhnle, Isabella Stilkerich, and Wolfgang Schröder-Preikschat. 2014. The final Frontier: Coping with Immutable Data in a JVM for Embedded Real-Time Systems. In *Proceedings of the 12th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'14)*. ACM, New York, 97–106. DOI:http://dx.doi.org/10.1145/2661020.2661024

Sanjay Ghemawat, Keith H. Randall, and Daniel J. Scales. 2000. Field analysis: Getting useful and low-cost interprocedural information. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'00)*. ACM, New York, 334–344. DOI:http://dx.doi.org/10.1145/349299.349343

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

Benjamin Goldberg and Young Gil Park. 1990. Higher order escape analysis: Optimizing stack allocation in functional program implementations. In *ESOP*, Lecture Notes in Computer Science, Vol. 432, Neil D. Jones (Ed.). Springer, 152–160.

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

Stephan Korsholm. 2011. Flash memory in embedded Java programs. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'11)*. ACM, New York, 116–124. DOI:http://dx.doi.org/10.1145/2043910.2043930

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

Clemens Lang. 2012. Improved Stack Allocation Using Escape Analysis in the KESO Multi-JVM. (Oct. 2012).

Clemens Lang. 2014. Compiler-Assisted Memory Management Using Escape Analysis in the KESO JVM. (June 2014).

Peeter Laud. 2001. Analysis for object inlining in Java. In *Proceedings of the Joses Workshop*. 1–8.

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

Ondrej Lhoták and Laurie Hendren. 2002. Run-time evaluation of opportunities for object inlining in Java. In *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande (JGI'02)*. ACM, New York, 175–184. DOI:http://dx.doi.org/10.1145/583810.583830

Young Gil Park and Benjamin Goldberg. 1992. Escape analysis on lists. *SIGPLAN Not.* 27, 7 (1992), 116–127.

Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj, and Jan Vitek. 2010a. High-level programming of embedded hard real-time devices. In *Proceedings of the ACM SIGOPS/EuroSys Eur. Conference on Computer Systems 2010 (EuroSys'10)*. ACM, New York, 69–82. DOI:http://dx.doi.org/10.1145/1755913.1755922

Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. 2010b. Schism: Fragmentation-tolerant real-time garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. ACM, New York, 146–159. DOI:http://dx.doi.org/10.1145/1806596.1806615

Martin Schoeberl, Stephan Korsholm, Christian Thalinger, and Anders P. Ravn. 2008. Hardware objects for Java. In *Proceedings of the 11th IEEE International Symposium on OO Real-Time Distributed Computing (ISORC'08)*. IEEE, 445–452. DOI:http://dx.doi.org/10.1109/ISORC.2008.63

Martin Schoeberl, Stephan Korsholm, Kalibera Tomas, and Anders P. Ravn. 2009. A hardware abstraction layer in Java. In *ACM Trans. Embedd. Comput. Syst.*, Vol. 5. ACM, 1–42.

Fridtjof Siebert. 2007. Realtime garbage collection in the JamaicaVM 3.0. In *Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'07)*. ACM, New York, 94–103. DOI:http://dx.doi.org/10.1145/1288940.1288954

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

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

Isabella Stilkerich. 2016. *Cooperative Memory Management in Safety-Critical Embedded Systems*. Ph.D. Dissertation. Friedrich-Alexander University Erlangen-Nuremberg.

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

Isabella Stilkerich, Michael Strotz, Christoph Erhardt, Martin Hoffmann, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. 2013. A JVM for soft-error-prone embedded systems. In *Proceedings of the 2013 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'13)*. ACM, New York, 21–32. DOI:http://dx.doi.org/10.1145/2465554.2465571

Isabella Stilkerich, Michael Strotz, Christoph Erhardt, and Michael Stilkerich. 2014a. RT-LAGC: Fragmentation-Tolerant Real-Time Memory Management Revisited. In *Proceedings of the 12th International Workshop on Java Technologies for Real-Time and Embedded Systems*. ACM, New York, 87–96. DOI:http://dx.doi.org/10.1145/2661020.2661031

Isabella Stilkerich, Philip Taffner, Christoph Erhardt, Christian Dietrich, Christian Wawersich, and Michael Stilkerich. 2014b. Team Up: Cooperative memory management in embedded systems. In *Proceedings of the 2014 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES'14)*. ACM, New York, Article 10. DOI:http://dx.doi.org/10.1145/2656106.2656129

Michael Stilkerich. 2006. An OSEK Operating System Interface and Memory Management for Java. Diploma thesis (DA-I4-2006-14), University of Erlangen-Nuremberg, Germany. (Aug. 2006).

Michael Stilkerich. 2012. *Memory Protection at Option—Application-Tailored Memory Safety in Safety-Critical Embedded Systems*. Ph.D. dissertation. Friedrich-Alexander University Erlangen-Nuremberg.

Michael Stilkerich, Isabella Thomm, Christian Wawersich, and Wolfgang Schröder-Preikschat. 2012. Tailor-made JVMs for statically configured embedded systems. *Concurr. Comput.: Pract. Exper.* 24, 8 (2012), 789–812. DOI:http://dx.doi.org/10.1002/cpe.1755

Robert Tarjan. 1972. Depth first search and linear graph algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160.

Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (April 1991), 181–210. DOI:http://dx.doi.org/10.1145/103135.103136

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

Received August 2015; revised September 2016; accepted December 2016