A Practical Getaway: Applications of Escape Analysis in Embedded Real-Time Systems

Isabella Stilkerich Clemens Lang Christoph Erhardt Michael Stilkerich

Friedrich-Alexander University, Erlangen-Nuremberg, Germany

{isa, clemens.lang, erhardt, mike}@cs.fau.de

Abstract

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 optimization, and benefits from the properties of a type-safe language. In this paper, we explore the application of escape analysis in KESO [34], a Java ahead-of-time compiler targeting (deeply) 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 support for software-isolated applications, automated inference of immutable data or improved upper space and time bounds for worst-case estimations.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers; D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and Objects; D.4.7 [*Operating Systems*]: Organization and Design—Real-time Systems and Embedded Systems

General Terms Memory Management, Design, Languages

1. Introduction

Java is a relatively uncommon language in (deeply) embedded real-time systems, although it provides a series of advantages such as memory safety [2]. Numerous projects [19, 22, 23, 26, 27] 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 [33, 34] has demonstrated 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 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 paper.

Background. Embedded application software may be deployed on several kinds of microcontroller units (MCU) and the MCU landscape is rather heterogeneous in contrast to commodity systems: They may vary in their hardware-specific properties such as, for example, 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 analyses techniques to make the best possible use of the underlying hardware and operating-system (OS) features with respect to a specific application and its non-functional requirements on the system software on a particular embedded device. Escape analysis is one building block of the *cooperative memory management* (CMM) framework [32] 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, KESO 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 optimization back ends for (deeply) 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. Survivability: Support for the machine-independent space and time bounds analyses of memory management
5. Assisted explicit memory management
6. Resource-efficient mitigation of transient errors
7. Automated inference of immutable data
8. Object inlining

Permission to make digital or hard copies of all or part 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 bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

LCTES'15, June 18–19, 2015, Portland, OR, USA.
© ACM ISBN 978-1-4503-3257-6//…$15.00.
DOI: http://dx.doi.org/10.1145/2670529.2754961

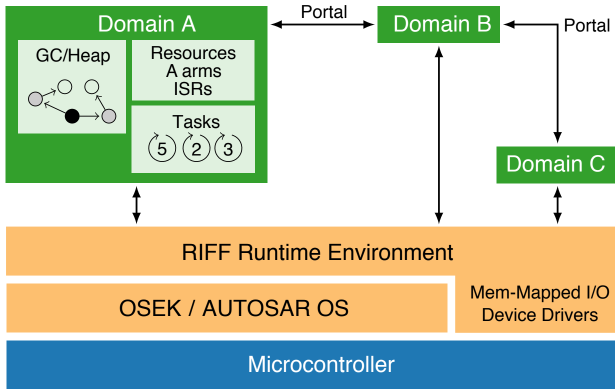

Figure 1: KESO's architecture

We also give an insight into our current work on *cycle-aware reference counting*.

Overview. Section 2 explains the relevant aspects of the embedded KESO JVM, in which we implemented and evaluated our approach. We quickly recap the outlines of Choi's original algorithm for escape analysis [10], which is the base for our own algorithm. Our EA back ends for deeply embedded systems are presented in Section 3. We evaluate the outcomes of our work in Section 4 on a microbenchmark and on the open-source real-time Java benchmark *Collision Detector* (CD$_x$). The paper wraps up with a discussion on related work in Section 5 and the conclusion as well as ideas for future work in Section 6.

2. KESO and Escape Analysis: An Outline

This section describes the characteristics of the KESO JVM and escape analysis. We only present those traits which are relevant for the implementation of our back ends.

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 (deeply) 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 safety even on low-end MCUs that lack dedicated hardware support by means of a memory-protection unit (MPU) or memory-management unit (MMU).

Moreover, 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 *type-unsafe* languages, this is usually performed manually by the programmer. In KESO, it is also possible to offload runtime checks to an unutilized MPU [33].

For communication, an RPC-like mechanism (called *Portal*) is available. To maintain software-based isolation, portals have to make sure that object references are not illegally propagated between domains: the portal mechanism initiates a deep copy of all objects passed as arguments. To make communication more efficient, we created a new optimization back end for our escape analysis (see Section 3.1). 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 synchronisation locks (called resources). KESO applications benefit from Java features like type safety, dynamic memory management and optionally a garbage collector. The 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 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 [3], 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, i.e. at well-known invocations of AUTOSAR OS's system call `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 the language's strong type system, it is possible for the JVM's compiler to automatically categorize objects in terms of their lifetime: The information collected by alias analysis and the computation of the references' reachability can be leveraged to determine if an object *escapes* a method, i.e. 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 [4, 5, 9, 10, 12, 21]:

- 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 (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 deeply embedded and safety-critical embedded systems, the information collected by escape analysis offers a lot more interesting optimization opportunities, which we address in this paper. We implemented an improved ahead-of-time version of Choi's (control-)flow-sensitive algorithm in the KESO JVM. Details on our modifications of the algorithm and on the implementation in KESO to speed up and improve the quality of results can be found in [16]. 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. For each analyzed method, the connection graph contains representations of local variables, static class members, dynamic instance variables, array indices and

objects. Variables of non-reference types are ignored since they do not contribute to alias information. A detailed description of the original escape analysis algorithm can be found in Choi's paper [10].

3. KESO's Back Ends for Escape Analysis

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

3.1 Extended Remote-Procedure-Call Support

```
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 ported version of the CD_x benchmark

Applications co-located 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 return 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 object tree outlive the portal call, that is, they have a global escape state in the callee's connection graph
2. Modification by the callee

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 object tree 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 case 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.[1] 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. While it is possible to compute which descendants of an object are modified or escape the callee, and only copy these parts rather than the entire object tree, the additional runtime support and runtime representation of the partial copy information made this a refinement not considered worthwhile.

The approach may be perceived as a variation of the *copy-on-write* technique. In addition to the copy-on-write trait, KESO's technique adds the *copy-on-escape* semantics and it is performed ahead-of-time.

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

Note that while only stack allocation 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 RTSJ 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 back end 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 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

[1] 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 identified at compile time by *jino* 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 CD_j.

return operation. Note that being returned is not the only way an object can escape: storing references in a field of an object given as parameter will also increase the escape state. This case is omitted in all examples for simplicity, but always implied.

Any object in the *method* escape state partition of a method's CG is a candidate for optimization. The escape state of the object's representation in the method's callers can be taken into account to decide whether the object should be allocated by the caller. Note that since there might be multiple callers and the optimization could be applied multiple times (moving allocations up multiple levels in the call hierarchy), considering the escape state of the object in the callers' CGs is not always a trivial task. For example, the object might escape further in some of the callers but not in others. When using stack allocation, even objects that are local in a calling method might still not be eligible for optimization due to overlapping liveness regions. KESO's stack-allocation transformation avoids possibly unbounded growth of stack usage by omitting the transformation into a stack allocation if multiple objects allocated at the same allocation site are in use simultaneously. 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, KESO's implementation does not take the escape state of object nodes' equivalents in the callers' CGs into account. For each run of the optimization pass, allocations are propagated at most a single level up against the direction of the call hierarchy. Therefore, running the pass multiple times will increase the maximum scope extension level. Note that it is not necessarily beneficial to run the pass often, since it may lead to undesirable results. The last part of this section also takes a closer look at the problems of scope extension.

Non-Virtual Calls. Non-virtual call sites, i.e., 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

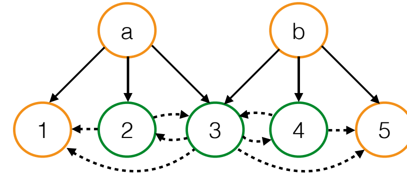

Figure 2: The call graph illustrates the complexity of scope extension for virtual method calls. Green vertices mark methods that contain allocations eligible for scope extension, orange vertices represent other methods. Solid lines are method invocations. Assume that both a and b contain a single virtual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

object node with a known allocation site (i.e., each non-phantom[2] object node) and an escape state of *method* will be optimized in KESO. When optimizing allocations of local objects using stack allocation, the allocation instruction must be moved into all callers. A reference to the allocated object is instead passed to the method on invocation, which uses this reference instead of the reference previously returned by the allocation instruction. Each allocation that is optimized using scope extension is copied into all callers and executed unconditionally, regardless of whether the allocation sites were in mutually exclusive control flow-paths before optimization, and hence could never be used at the same time. In some examples, this causes a large number of allocations and new method parameters even though only 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 2 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, KESO does not currently perform scope extension across virtual invocations. Note that some of the challenges are caused by properties of the applied optimization. Intermediate-code transformations that do not require changing a method's signature can simplify the problem. For example, instead of using stack memory, a separate thread-local heap section with a simple bump-pointer memory-management strategy could be used. 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.

[2] For the representation of objects with unknown allocation sites – which may have been passed as arguments, for example – special nodes called *phantom nodes* are created according to [10].

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: 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 [28], which are already implemented in KESO to remove unnecessary copies of variables in SSA deconstruction, but are not used in scope extension yet. Since consolidated memory areas might be used for objects of different types and sizes, garbage collectors would have to support uninitialized chunks of memory as method arguments. Summarizing so far, scope extension can increase memory usage due to the allocation of unused objects and it can cause sub-par performance when a large number of allocations are optimized because of the increased overhead of the modified method invocation.

3.3 Thread-Local Heaps and Regional Memory

Depending on the circumstances, turning heap allocations into stack allocations for automatic memory management is not necessarily the best solution. Especially in safety-critical embedded systems, allocating objects and arrays on the stack could lead to increased worst-case stack usage. Since the stack space needs to be reserved for each thread even if it is not going to be used simultaneously, the overall memory requirement can increase compared to a system without escape analysis. This situation occurs when the sum of upper bounds is larger than the upper bound of the sum. Furthermore, to keep stack usage limited and simplify finding an upper bound of stack usage, KESO does not turn allocations whose liveness regions overlap into stack allocations. In order to address these shortcomings, an alternative to stack memory is necessary. A special region can be used for all objects that can be automatically managed by the compiler. To provide a runtime advantage over the normal heap, this region must be exempt from garbage-collector sweeps. There should be one logical region for each method, while empty regions (i.e., regions corresponding to methods without local objects) can be omitted. At the end of the method, its associated region can be reclaimed as a whole. To retain the semantics of stack allocations and reclaim-on-return, these logical regions should be organized in a stack-like manner. One possible implementation of these constraints are small specialized heap regions local to each thread. The idea has been proposed in prior work [11, 17]. Given these local heaps, the logical regions are implemented similar to a stack in KESO: Each thread-local heap has a fill marker and a maximum fill level. At method entry, the fill marker is saved and necessary objects are allocated by moving the fill marker. At method exit, the fill marker is reset to its previous value. Saving the fill marker on the stack can be avoided if the amount of memory that will be allocated in a function and all alignment offcuts are known at compile time, because this knowledge can be used to calculate the value at method entry. The approach does not require any synchronization for allocations, which constitutes another advantage over usual heap allocation. Since object allocation on stack no longer occurs with this method of region-based memory management enabled, finding a tight upper bound for stack usage is simplified. With precise and quick checks preventing thread-local heap overflows in place, liveness-interference avoidance can be disabled, further reducing garbage-collector load (possibly exceeding amounts proportional to the number of affected allocations due to the use in loops). The necessary size of these local heaps can be statically configured using results from manual worst-case memory-usage analysis. Future work could automate the process of determining the size of thread-local heaps.

3.4 Assisted Explicit Memory Management

In the complete absence of a garbage collector, the region information computed by escape analysis can be used to check manual memory management for potential mistakes. In each location where KESO would automatically place a reclaim operation, it can check whether the programmer did write the expected instruction. If the object in question is not explicitly returned into the memory pool, leakage occurs and *jino* can print a warning. For this application, running the scope extension pass multiple times can be beneficial, because due to the lack of modifications, the code size does not increase, but the number of objects whose lifetime can be inferred by the compiler grows. This application of escape analysis can be seen as a compiler-assisted approach to manual memory management used in unmanaged languages, in which the compiler may advise the programmer of possible mistakes. The approach supports a stepwise migration from manual to automatic memory management in legacy systems that rely on established code verification tools such as *Polyspace* [24].

3.5 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 – which can be handled by *Absint's aiT* tool [1], for example – 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 deeply embedded real-time systems, dynamic memory management is still rarely used. In combination with type-safe languages, however, fragmentation-tolerant garbage collection as proposed by Pizlo et al. [23], 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. [31] 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 analysis and call graph analysis to `WaitEvent()`, i.e. the incorporation 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.

3.6 Resource-efficient Mitigation of Soft Errors

Soft errors (also called transient errors) happen as a consequence of shrinking structures sizes in CPUs [7], low supply voltages, aging or radiation effects [35] and manifest themselves as bit flips. Such errors can be handled via hardware-based, software-based fault detection/tolerance techniques or a combination of both. In the following, we focus on a software-based measure that is supported by escape analysis, however, it should be noted that the KESO framework also allows the combination of both techniques.

Isolated Replication: We extended KESO to provide automated homogeneous redundancy for applications that need it. Triple-modular redundancy (TMR) of critical software parts is a common software-based technique, where three instances of that software are executed with identical input data and a majority voter detects sane and faulty replicas. While this technique has been subject to numerous research projects, the KESO's multi-JVM architecture provides the infrastructure to deal with a replication in a fully automated way: The isolated protection domain with fully separated data, the portal mechanism and clearly defined external interfaces forge a suitable unit for replication and recovery which is facilitated by Java: The type-safe programming language simplifies the inspection of runtime structural information. However, replication is costly in space and time. A scrutinizing compile-time inspection of the application helps to keep the overhead reasonably low. The size of a domain is partly attributed to the size of the heap and escape analysis can significantly lower heap pressure. The extent of this decrease depends on the application. As an example, EA diminishes the heap usage of the CD_j benchmark by around 43%, which contributes to a considerable reduction of the replication domain's heap size.

KESO's portal mechanism is used as a transition point between the single and replicated execution. Usually, the object including its transitive closure passed into the portal is copied for each portal handler, which happens three times assuming TMR, for instance. With the special remote-procedure-call support provided by our EA implementation, the third copy can be omitted if required by the system configuration.

Improved Reference Checking: Closely related to the aforementioned topic is the revised support of reference validation by means of load-reference checks (LRC) and dereference checks (DRC), which have been proposed by [30]. In a nutshell, these checks are employed to confine the forging of wild references anticipating soft errors, that is the reference integrity can be verified, whenever it is either loaded from memory (LRC) or on dereference operations (DRC). In this way, software-based isolation is retained in the presence of soft errors. The overhead of protecting garbage collection against those errors is reduced significantly as – in addition to heap references – only the local references present in the stacks of blocked tasks need to be safeguarded during the GC execution.

3.7 Handling Immutable Data

Java has insufficient support for immutable data: There are situations in which Java's `final` keyword is not expressive enough to be mark certain data as truly constant or unmodifiable during runtime:

- It simply not possible to mark objects such as fields constant. Regarding `final` references or arrays, the content of the referenced object can still be altered, only the object or array to which they are referring must not be changed.
- Final fields not being tagged accordingly by an oblivious programmer may result in missed optimizations
- In case of configurable software for embedded control systems, there is usually specialized configuration for a particular code variant deployed on the microcontroller. The variability points may only be present at the program's design level or the initialization phase of the program (e.g. constructors), but are fix during the actual execution. Such data cannot be tagged accordingly in ordinary Java systems.

In KESO we use the alias information collected by our escape analysis to address the aforementioned issues and compute the program's immortal objects and effectively-final fields:

Immortal Objects: Regarding `final` references or arrays, the contents of the referenced object can still be manipulated, only the reference itself must not change. As a consequence, the code size, runtime as well runtime memory footprint are unnecessarily high. Also, the use of read-only memory (ROM) such as flash memory for the placement of constant data is very important, as a ROM unit is significantly cheaper than SRAM and should not be underused, if it is available. Such automatically derived constant data is similar to manual `ImmortalMemory` specified in the *Real-Time Specification for Java* (RTSJ) [6].

Effectively-`final` Fields: Fields with `final` properties allow the compiler to perform optimizations more aggressively than on regular fields: The fields are initialized and in case of a reference field, the compiler can elide the `null`-check without endangering the memory safety of the program. As the field is only written once, the initialization value can be propagated to all reads and in case the propagated values can be folded, this possibly supports other optimizations.

3.8 Object Inlining

Java usually does not store field instances by value but rather by reference in contrast to the C++ language that always embeds such instances and only uses the by-reference semantics if requested by the programmer. This can result in higher overheads at runtime imposed by pointer indirections, allocation code and the need to store both the field and object header [18]. The results computed by escape and effectively-final analysis can be employed to identify candidates for object inlining: Such objects 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 KESO. 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.

Embedded objects also do not need to be scanned by the GC and the memory of the embedded and surrounding object can be reclaimed as a whole. A moderate increase in memory usage due to deferred memory reclamation may also acceptable in soft-error-prone systems using LRC and DRC to protect references. 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 per-case base.

3.9 Cycle-Aware Reference Counting

Automatic reference counting as a method of compiler-assisted memory management that does not require tracing garbage collection has been gaining popularity lately. For example, Apple's Objective-C used on iOS and OS X employs compiler-generated reference

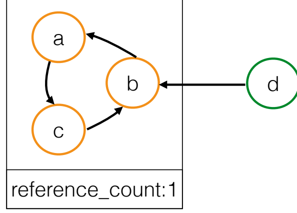

Figure 3: Self-referential data structure (*a*-*c*) using a cycle descriptor.

counting. Unfortunately, it is a well-known limitation of reference counting that it cannot automatically reclaim self-referential (i.e., cyclic) data structures. Cyclic data structures either require the additional use of a tracing garbage collector, or the use of special pointer types called weak pointers that do not increase the reference count of objects to break the cyclic structure. Using the connection graphs constructed in alias and escape analysis, cyclic structures, their components and corresponding allocation sites can be computed at compile time. Using this information, cycles could be automatically managed by assigning a cycle descriptor to each object that is part of a cyclic structure and keeping a reference count for the whole structure in the cycle descriptor. Once this reference count drops to zero due to the release of an object that references the structure, the whole cycle can be reclaimed.

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

Unfortunately the representation of the alias information in *jino* makes it difficult to apply this idea, because the CG of a method is independent of its calling contexts (which is one of the significant contributions of Choi et al. in [10]). Cycles in connection graphs can be identified, but are useless when containing phantom representations of nodes passed into a method from a caller. Because a single static representation exists for multiple instances of cycles, the approach is unlikely to perform well: For example, when used on KESO's current alias analysis results, all objects stored in a doubly linked list (which is a cyclic data structure) can only ever be reclaimed on the whole, even if the lists are completely separate. The approach might work reasonably well given a global points-to graph, but this implementation is currently in progress.

4. Evaluation

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

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

Due to space limitation we cannot present the results for all implemented back ends proposed in this paper. 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. For KESO, we use CD_j in the *onthegoFrame* variant, deployed on the Infineon TriCore TC1796 device (150 MHz CPU clock, 75 MHz system clock, 1 MiB external SRAM, 2 MiB internal flash). The application is translated to ANSI-C code using KESO. 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 600 KiB that is managed by a mark-and-sweep garbage collector. Section 4.1 presents the results for portal services assisted by escape analysis, while Section 4.2 introduces the CD_j benchmark and the evaluation of the latter two back ends 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.

Call Type	Execution Time
portal call	3.76 μs
regular virtual method call	3.09 μs
AUTOSAR non-trusted function	32.91 μs
portal call (2 int params)	4.17 μs
portal call (3 int params)	4.70 μs
portal call (1 element linked list)	31.47 μs
portal call (2 element linked list)	56.08 μs
portal call (3 element linked list)	84.37 μs
portal call (escape analysis, 1 element linked list)	3.99 μs
portal call (escape analysis, 2 element linked list)	4.19 μs
portal call (escape analysis, 3 element linked list)	4.65 μs

Figure 4: Execution Time for Portals

4.1 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). Figure 4 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 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 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 3.1, 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. To fortify the validity of this evaluation, we enriched the method bodies of the target methods with code, which does not modify, modifies or causes the portal parameters to escape, confirms the effectiveness of legally removing the copy of portal parameters by escape analysis while still retaining software-based memory protection.

4.2 The CD$_x$ Benchmark

The core of the CD$_x$ application is a periodic thread that detects potential aircraft collisions from simulated radar frames. A collision is assumed whenever the distance between two aircraft is below a configured proximity radius. The detection is performed in two stages: In the first stage (reducer phase), suspected collisions are identified in the 2D space ignoring the z-coordinate (altitude) to reduce the complexity for the second stage (detector phase), in which a full 3D collision detection is performed (detected collisions). A detailed description of the benchmark is available in a separate paper [15]. 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.

4.2.1 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 back ends.

Allocation Sites. The share of optimized allocations can be used as a compile-time criterion of KESO'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[3] callers on average. The number of stack allocations increases by 32 from 44 (30.1%) to 76 (43.7%). Note that these are statically determined numbers, i.e., the actual number of objects allocated at runtime does not change despite the increase in allocation instructions. The number of allocations not converted into stack allocations due to overlapping liveness regions of the allocated objects stays the same. Consequently, the number of allocations using task-local heaps stays at the same margin to stack-allocated ones in comparison to the measurement without scope extension.

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)

[3] ((174-(146-12))/12)

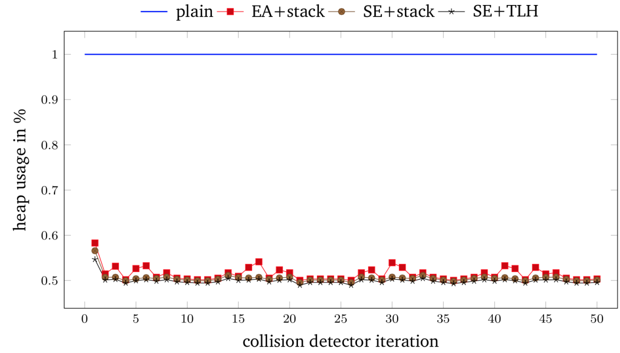

Figure 5: Heap memory usage of the on-the-go variant of the CDj benchmark with (a) scope extension and stack allocation (SE+stack), and (b) scope extension with task-local heaps (SE+TLH) relative to a run without escape analysis-based optimizations (plain). For comparison, heap memory usage for escape analysis and stack allocation (EA+stack) is also shown.

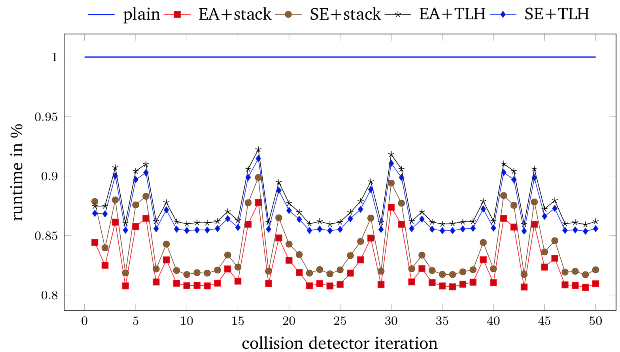

Figure 6: Runtime of the on-the-go variant of the CDj benchmark using (a) escape analysis with stack allocation (EA+stack), (b) escape analysis with task-local heaps (EA+TLH), (c) scope extension with stack allo- cation (SE+stack), and (d) scope extension with task-local heaps (SE+TLH) relative to a 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.

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 5 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.

Execution Time. In terms of execution time (illustrated in Figure 6), 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 faster than the native CD_c[4] 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 [34]. 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 3.2.

Conclusion Overall, enabling escape analysis considerably improves performance and reduces the heap memory requirements. Thus, it is recommended to enable escape analysis and one of its optimization back ends for all applications. A similar suggestion can, however, not be given for scope extension. While it does reduce heap memory usage a little and reduces the variance between the collision-detector iterations, this optimization comes at the price of slower execution speeds in some configurations. Some of the examples tested expose at least some of the erratic behavior predicted in Section 3.2, 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.

4.2.2 Automated Inference of Immutable Data

The effectively-`final` analysis reduces the data segment size by 44% and the text segment by 14%. This is attributed to constant folding and 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 CDx application by 10% due to dead code removal and runtime check elimination. Afterwards, 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 1 KiB of string constants, which can be moved to flash memory. Hence, the influence of the immortal object analysis is marginal for this benchmark.

[4] deployed under the same setup mentioned in Section 4

5. Related Work

The approach used in KESO's alias and escape analysis is based on the work of Choi [10]. Thus, behavior, results, and features of the analysis for stack allocation are similar. Unlike their work, *jino* avoids resizing a method's stack frame at runtime and provides a series optimization back ends such as improved remote procedure call support for software-isolated components or the determination of object *survivability* for real-time systems in addition to stack allocation. KESO's alias-analysis features a modification that considerably reduces compile times for large specimen by merging sibling nodes. This compression technique is inspired by ideas from Steensgaard's almost-linear-time points-to analysis [29]. Different from Steensgaard's work, KESO's analysis does not necessarily compress all sibling nodes pointed to by a common ancestor, but only merges nodes with the same escape state to avoid deteriorating the quality of escape-analysis results. Object nodes that represent an allocation site are not compressed either to preserve the one-to-one mapping between allocation instructions in the intermediate code and their corresponding object nodes in the connection graphs.

Except for automated memory management and synchronization, we are not aware of any related work of alternative applications of escape analysis results. Using escape analysis for automatic memory management solves the same problem as region inference. First published by Tofte et al. [36], region inference has seen widespread adaption in later work: Different from Grossman [13], KESO's escape analysis is fully automatic and does not require source-code modifications or developer interaction. Similar to Hallenberg [14], the system implemented in this thesis co-exists with garbage collection. The work of Chin [8] only supports a subset of Java called Core-Java for their analysis, while KESO does not impose limitations of the source-language features. Experimental results provided by Chin et al. are small: The largest example has only 170 lines of source code. CACAO JVM [20] proposes stack allocation using escape analysis and uses a Steensgaard-based method for escape analysis. Different from KESO's design, their virtual machine uses just-in-time compilation, and escape analysis is done at runtime. They do not support stack allocation of arrays and do not have a generic method to encode the escape information of objects passed to native methods.

6. Conclusion and Future Work

In this paper, we presented our application back ends for escape analysis in (deeply) 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 back ends 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 competitive to C programs in terms of 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.

Particularly for escape analysis, there are more application opportunities in our domain and we would like to present two of them for future work:

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

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

Also for multiprocessor support, the synchronization optimization might be worthwhile as multiprocessor priority ceiling protocols make use of locking mechanisms and lock elision can contribute to an overall better throughput of parallel programs.

Selective Use of Memories. With the evolving technology for hardware platforms even in deeply embedded system, virtual memory becomes increasingly interesting. As escape analysis can be leveraged to determine the objects' lifetimes, we propose to employ it in combination with KESO's cooperative memory management approach to improve the handling and efficiency of virtual memory. Upcoming non-volatile memories have a higher density and are more energy-efficient while some of them have, for example, high write latencies and only allow for a limited amount of write operations. Traditional virtual memory is unaware of the characteristics of these new memory technologies. Hybrid solutions exist to address the problem, however, they require special hardware and induce significant overheads. Roy et al. [25] presented a software-based virtual memory design for new embedded memory architectures, which does not rely on specialized hardware. They provide a static analysis to identify and allocate data with certain read and write affinities and hence reduces write operations to non-volatile memory. Moreover, the programmer's interface is extended and the programmer manually annotates read and write intensive heap memory by using these functions.

Due to its whole-program analyses on type-safe code being able to incorporate operating-system- and hardware-specific information, KESO is aware of all read and write operations and data lifetimes by design. From a conceptual point of view, all objects are allocated on the heap, the programmer does not have to have a particular memory interface. *jino* is able to determine the placement of the data: By means of the memory specification, collected read and write behavior and the results of our flow-sensitive escape analysis, which provides the knowledge of the objects' lifetimes and which is aware of their sizes, the memory hierarchy can be used optimally. Statically-allocated data (i.e., the needed space is reserved ahead-of-runtime), and dynamic short-living objects in either stack memory or smaller regions can automatically be clustered due to the read and write affinity and arranged in the memory hierarchy as proposed by Roy. For dynamically-allocated heap memory, we apply the concept of *survivability* for real-time systems from Section 3.5, that is we know at compile-time, which objects will be dead after the garbage collector's sweep phase and which portion will contribute to write operations in case of generational collection. Bounded loops and fixed allocation rates[5] are beneficial in real-time systems anyway. However, it is possible annotate such sensitivity points to allow for a broader field of application. Annotation points, that is code locations at which the programmer has to provide further information about upper boundaries, are reported by *jino* to the programmer for assistance purposes. In contrast to Roy, the programmer is supported by the runtime system and does not *have to* be aware of the actual data location in either stack, regional or heap memory. However, the programmer *can* acquire the information for analysis purposes. Paging algorithms may also profit from the computed program information and may thus be applied with acceptable overhead while being able to support a series of virtual address spaces in small embedded systems.

Acknowledgments

The authors would like to thank Christian Wawersich, Michael Strotz and Wolfgang Schröder-Preikschat for their valuable hints and ideas for this project. This work was partly supported by the German Research Foundation (Deutsche Forschungsgemeinschaft (DFG)) under grants no. SCHR 603/9-1 (AORTA), SFB/TR 89, Project C1 (Transregional Collaborative Research Centre *Invasive Computing* and LO 1719/1-1 (DanceOS)).

References

[1] Absint aiT. Worst-Case Execution Time Prediction by Static Program Analysis. URL `http://www.absint.com/aiT_WCET.pdf`.

[2] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus. Deconstructing process isolation. In *MSPC '06: 2006 W'shop on Memory System Performance and Correctness*, pages 1–10, New York, NY, USA, 2006. ACM. ISBN 1-59593-578-9. .

[3] AUTOSAR. Specification of operating system (version 4.0.0). Technical report, Automotive Open System Architecture GbR, Dec. 2009.

[4] B. Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *POPL '98: 25th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages*, pages 25–37, 1998. .

[5] B. Blanchet. Escape analysis for Java: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, Nov. 2003. ISSN 0164-0925. . URL `http://doi.acm.org/10.1145/945885.945886`.

[6] G. Bollella, B. Brosgol, J. Gosling, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. AW, 1st edition, Jan. 2000.

[7] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6): 10–16, November 2005. ISSN 0272-1732. .

[8] W.-N. Chin, F. Craciun, S. Qin, and M. Rinard. Region inference for an object-oriented language. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 243–254, New York, NY, USA, 2004. ACM. ISBN 1-58113-807-5. . URL `http://doi.acm.org/10.1145/996841.996871`.

[9] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *14th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '99)*, pages 1–19, New York, NY, USA, 1999. ACM. .

[10] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for Java using

[5] The allocation rate is composed of the minimum inter-arrival time of events and the size and number of objects allocated in the wake of this event. The execution of a periodic task can also be seen as an event.

escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, Nov. 2003. ISSN 0164-0925. . URL http://doi.acm.org/10.1145/945885.945892.

[11] T. Domani, G. Goldshtein, E. K. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald. Thread-local heaps for java. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, pages 76–87, New York, NY, USA, 2002. ACM. ISBN 1-58113-539-4. . URL http://doi.acm.org/10.1145/512429.512439.

[12] B. Goldberg and Y. G. Park. Higher order escape analysis: Optimizing stack allocation in functional program implementations. In N. D. Jones, editor, *ESOP*, volume 432 of *LNCS*, pages 152–160. Springer, 1990. ISBN 3-540-52592-0.

[13] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '02)*, pages 282–293, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. .

[14] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 141–152, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. . URL http://doi.acm.org/10.1145/512529.512547.

[15] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek. CD_x: A family of real-time Java benchmarks. In *JTRES '09: 7th Int. W'shop on Java Technologies for real-time & embedded systems*, pages 41–50, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-732-5. .

[16] C. Lang. Compiler-Assisted Memory Management Using Escape Analysis in the KESO JVM, June 2014.

[17] K. Lee, X. Fang, and S. P. Midkiff. Practical escape analyses: How good are they? In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 180–190, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-630-1. . URL http://doi.acm.org/10.1145/1254810.1254836.

[18] O. Lhoták and L. Hendren. Run-time evaluation of opportunities for object inlining in Java. In *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande*, JGI '02, pages 175–184, New York, NY, USA, 2002. ACM. ISBN 1-58113-599-8. . URL http://doi.acm.org/10.1145/583810.583830.

[19] T. K. Martin Schoeberl, Stephan Korsholm and A. P. Ravn. A hardware abstraction layer in Java. In *ACM Transactions on Embedded Computing Systems*, volume 5, pages 1–42. ACM, Sept. 2009.

[20] P. Molnar, A. Krall, and F. Brandner. Stack allocation of objects in the cacao virtual machine. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 153–161, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-598-7. . URL http://doi.acm.org/10.1145/1596655.1596680.

[21] Y. G. Park and B. Goldberg. Escape analysis on lists. *SIGPLAN Notices*, 27(7):116–127, 1992.

[22] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. Vitek. High-level programming of embedded hard real-time devices. In *ACM SIGOPS/EuroSys Eur. Conf. on Computer Systems 2010 (EuroSys '10)*, pages 69–82, New York, NY, USA, Apr. 2010. ACM. ISBN 978-1-60558-577-2. .

[23] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: Fragmentation-tolerant real-time garbage collection. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '10)*, pages 146–159, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. .

[24] Polyspace. Comprehensive Static Analysis Using Polyspace Products, 2013. URL http://www.mathworks.com/tagteam/77340_91517v02_30211_Polyspace-WhitePaper_final.pdf.

[25] P. Roy, M. Manoharan, and W. F. Wong. Envm: Virtual memory design for new memory architectures. In *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '14, pages 12:1–12:10, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3050-3. . URL http://doi.acm.org/10.1145/2656106.2656121.

[26] M. Schoeberl, S. Korsholm, C. Thalinger, and A. P. Ravn. Hardware objects for Java. In *11th IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '08)*, pages 445–452, Washington, DC, USA, 2008. IEEE. ISBN 978-0-7695-3132-8. .

[27] F. Siebert. Realtime garbage collection in the JamaicaVM 3.0. In *JTRES '07: 5th Int. W'shop on Java Technologies for real-time & embedded systems*, pages 94–103, New York, NY, USA, 2007. ACM. ISBN 978-59593-813-8. .

[28] V. C. Sreedhar, R. D.-C. Ju, D. M. Gillies, and V. Santhanam. Translating out of static single assignment form. In *Proceedings of the 6th International Symposium on Static Analysis*, SAS '99, pages 194–210, Heidelberg, Germany, 1999. Springer. ISBN 3-540-66459-9.

[29] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3. . URL http://doi.acm.org/10.1145/237721.237727.

[30] I. Stilkerich, M. Strotz, C. Erhardt, M. Hoffmann, D. Lohmann, F. Scheler, and W. Schröder-Preikschat. A JVM for soft-error-prone embedded systems. In *2013 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES '13)*, pages 21–32, New York, NY, USA, June 2013. ACM. ISBN 978-1-4503-2085-6. .

[31] I. Stilkerich, M. Strotz, C. Erhardt, and M. Stilkerich. RT-LAGC: Fragmentation-Tolerant Real-Time Memory Management Revisited. In *JTRES '14: 12th Int. W'shop on Java Technologies for real-time & embedded systems*, pages 87–96, New York, NY, USA, Oct. 2014. ACM. ISBN 978-1-4503-2813-5/14/10. .

[32] I. Stilkerich, P. Taffner, C. Erhardt, C. Dietrich, C. Wawersich, and M. Stilkerich. Team up: Cooperative memory management in embedded systems. In *2014 Int. Conf. on Compilers, Architectures, and Synthesis for Embedded Systems (CASES '14)*, page Art. No. 10, New York, NY, USA, Oct. 2014. ACM. ISBN 978-1-4503-3050-3/14/10. .

[33] M. Stilkerich. *Memory Protection at Option - Application-Tailored Memory Safety in Safety-Critical Embedded Systems*. PhD thesis, Friedrich-Alexander University Erlangen-Nuremberg, 2012.

[34] M. Stilkerich, I. Thomm, C. Wawersich, and W. Schröder-Preikschat. Tailor-made JVMs for statically configured embedded systems. *Concurrency and Computation: Practice and Experience*, 24(8):789–812, 2012. ISSN 1532-0634. .

[35] A. Taber and E. Normand. Single event upset in avionics. *IEEE Transactions on Nuclear Science*, 40(2):120–126, Apr. 1993. ISSN 0018-9499. .

[36] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lamba-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 188–201, New York, NY, USA, 1994. ACM. ISBN 0-89791-636-0. . URL http://doi.acm.org/10.1145/174675.177855.