Type-Safe System Services for the KESO Runtime Environment

Bachelorarbeit im Fach Informatik

vorgelegt von

Martin Hofmann

angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Dipl.-Inf. Isabella Stilkerich** **Dipl.-Inf. Christoph Ehrhardt**
Betreuender Hochschullehrer:	**Prof. Dr.-Ing. Wolfgang Schröder-Preikschat**
Beginn der Arbeit:	**1. Januar 2015**
Abgabe der Arbeit:	**31. Mai 2015**

Erklärung

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

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties.
I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Martin Hofmann)
Erlangen, 31. Mai 2015

Abstract

KESO is a Java runtime environment for embedded systems that is built on top of an OSEK operating system. It uses the Java programming language for its applications and is itself implemented in C.

This thesis evaluates the possibility of implementing KESO's garbage collection in Java. Several difficulties had to be overcome: Low-level access to system data structures had to be provided on Java's language level and constraints had to be placed on Java's object instantiation in the garbage collectors, as no heap memory is available from within a garbage collector's code.

KESO's *weavelet* mechanism, that allows to execute native statements from Java, could be adopted for the low-level system access. KESO's *stack allocation* feature could be used to ensure no objects are to allocated on the heap and KESO's compiler, JINO, was extended to automatically check this constraint.

Using the above techniques, an implementation of the *CoffeeBreak garbage collector* in Java was created that is statically allocated and manages its data structures via weavelets.

To circumvent the non-idiomatic code of the static CoffeeBreak, KESO's domain model was extended to support garbage collectors that can perform allocations on their own heap. By identifying long- and short-lived objects in the garbage collector, this heap may be cyclically reset to prevent memory exhaustion. This allows for a more complete Java feature set in the garbage collector's own code than available in related projects.

Kurzfassung

KESO ist eine Java Laufzeitumgebung für eingebette Systeme, die auf einem OSEK-Betriebssystem aufbaut. In ihr wird Java für Anwendungsprogramme benutzt, während die Laufzeitumgebung selbst in C implementiert ist.

Diese Arbeit evaluiert die Möglichkeit, KESOs Speicherverwaltung in Java zu implementieren. Mehrere Schwierigkeiten mussten dazu überwunden werden: Zugriff auf systeminterne Datenstrukturen müssen auf Javas Sprachniveau zur Verfügung gestellt werden und es gelten Einschränkungen für Javas Objektinstanziierung im Code der Speicherverwaltung, da dort kein Haldenspeicher verfügbar ist.

KESOs *Weavelet*-Mechanismus, der es erlaubt, native Instruktionen aus Java-Programmen heraus aufzurufen, konnte für den Zugriff auf systeminterne Datenstrukturen verwendet werden. KESOs *Stapel-Allokations-Verfahren* konnte, gemeinsam mit einem neu implementieren Überprüfungslauf in KESOs Übersetzer, JINO, verwendet werden, um Halden-Allokation von Objekten zu vermeiden.

Unter Verwendung der obigen Techniken konnte eine Implementierung des *CoffeeBreak*-Speicherverwalters erstellt werden, die selbst statisch allokiert ist und ihre Datenstrukturen mit Weavelets verwaltet.

Um den nicht-idiomatischen Sprachstil dieser statischen CoffeeBreak-Implementierung zu vermeiden wurde KESOs Domänenmodell erweitert, um einen Speicherverwalter zu ermöglichen der selbst eine Halde für Allokationen besitzt. Durch die Identifizierung von lang- und kurzlebigen Objekten im Speicherverwalter kann diese Halde zyklisch zurückgesetzt werden um Speichererschöpfung zu verhindern. Das erlaubt es, einen größeren Sprachumfang von Java im Quelltext des Speicherverwalters zu verwenden als dies in verwandten Projekten der Fall ist.

Contents

Chapter 1

Introduction

1.1 Motivation

Embedded systems become more and more widespread in today's world. From mobile sensors devices connected to the Internet of Things to embedded controllers for various systems in cars – many aspects of modern live contain small computers. Because of the limited functionality they must provide, these microcontrollers are often tailored specially to the purpose they are used for.

Usually, embedded systems are programmed in low-level programming languages such as C or even Assembler. Due to their increased use, which is boosted by sinking prices for devices that at the same time come with higher computing power, new challenges have arisen in regard to the way embedded systems are programmed.

Multiple tasks should be performed by just one system. While this requirement is well-known in the realm of larger computers such as desktop- or even server-sized devices, it is relatively new for embedded devices. Most of these miniature systems do not use an actual operating system which would provide the means to manage multiple applications on one processor. As traditional operating systems go, they need considerable resources themselves – supplying them may not be feasible on small-scale computers.

From a developers perspective, writing software for embedded systems has been challenging: Writing in low-level languages such as the ones noted above is error-prone, as little support to avoid common mistakes at development time is provided by both the language and accompanying tools. The problems increase when the applications become more complex.

More modern languages facilitate the creation of large software systems by design. The Java programming language is such a language. It supports an imperative, object-oriented programing paradigm and enforces memory- as well as type-safety upon programs.

KESO is a Java runtime environment for embedded systems. It runs on top of an OSEK or AUTOSAR operation system, that provides low-level abstractions. Applications for KESO are written in Java and then translated into executable code using ahead-of-time compilation: Java code is compiled by KESO's own compiler, JINO, to C. In this step, JINO performs ahead-of-time optimizations and introduces integrity checks into the generated C code that guarantee Java's constraints to be held at runtime. However, not only applications can be written in Java: KESO also supports writing hardware drivers in Java by providing low-level access to hardware on Java's language level.

Nonetheless, many of KESO's internal system components are still written in C. The most prominent among them are KESO's garbage collectors. They benefit from the easily available low-level hardware access in C. However, their code can neither be checked by JINO at compile-time nor can runtime checks be introduced.

In this thesis, two of KESO's memory management techniques – the *Restricted Domain Scope* (RDS) allocator and the *CoffeeBreak garbage collector* – shall be ported to Java. This shall not only allow type-safety and static checking of the code at compile-time by JINO, but also increased extensibility by adhering to object-oriented design in the implementation.

1.2 Overview over KESO

This section is intended to give an overview over key aspects of KESO that will be important in the context of this thesis. First, the general structure of a KESO system will be shown. After that, an introduction to KESO's object model will be given. In closing, KESO's compiler, JINO, and its pass model will be presented.

1.2.1 The KESO System

Figure 1.1 shows the architecture of a KESO system. From top to bottom, the following elements of the system are visible: The *domains* house the Java applications of the system, the *KESO Runtime Environment* forms an abstraction layer on top of the *Microcontroller*.

KESO is statically configured. This means that all system objects are known at compile time. The system is created by parsing a *KESO configuration file* (.kcl file) that contains the feature set needed for a certain system. Only features required here will be included in the final system build. This allows to tailor the system to its use case and results in small binary sizes – a feature very welcome for embedded systems.

Java applications also benefit from KESO's static configuration. Relying on a closed-world assumption (all aspects of the system are known at compile-time), the

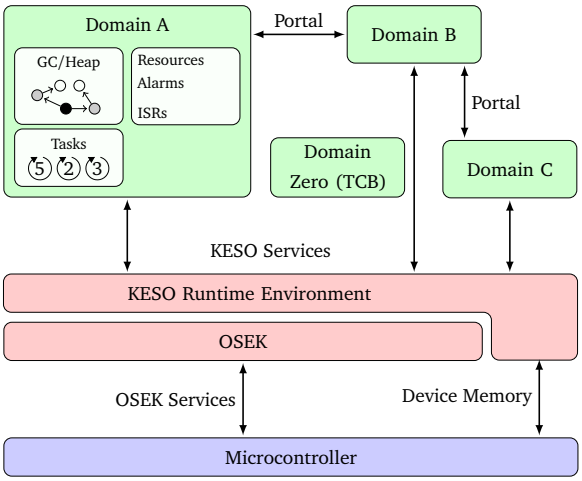


Figure 1.1 – The architecture of KESO (source: [1]).

applications can be optimized extensively, more drastically than in conventional Java compilers. However, features that are normally available in standard desktop JVMs, such as dynamic class loading, are not available.

As visible in Figure 1.1, there are multiple domain objects in a KESO system. Each domain object contains its own application. KESO is hence called a Multi-JVM – multiple applications can run within one system.

Domains function as a security mechanism. Each domain manages its own heap for the application running within it. Applications can contain multiple tasks that are implemented using Java's well-known `Runnable` interface. From an application, it is not possible to reference objects in the memory area of another domain. This way, KESO enforces *memory safety*. However, applications running in different domains can communicate with each other using the so-called *portal service*. Using this service, copies of objects can be passed amongst domains.

As each domain acts as a JVM and contains its own heap, KESO implements different memory management techniques. Each domain can employ a different memory management service. Currently, there is one allocation-only memory manager and three garbage collector implementations available in KESO. Whereas the CoffeeBreak *garbage collector* (GC), that will be examined closer in this thesis, is a conventional marking garbage collector, the *Idle Round Robin* and the *fragmentation-tolerant garbage collector* are more suitable for real-time applications.

As visible in the figure, garbage collectors do not run in the context of the domain whose memory they manage. Instead, the *Domain Zero* is a domain containing the privileged *Trusted Code Base* (TCB) of the garbage collectors. Garbage collection services within the Domain Zero are written in C instead of Java. This provides the benefit of easily accessing the underlying structures of the Java objects within

KESO's runtime environment. The next section will introduce KESO's object model in greater detail.

1.2.2 KESO's Object System

JINO translates Java classes to C code. This means, that Java's class structures must be represented in C `structs`. To be able to run KESO on embedded systems that possess only limited system memory, it is crucial to keep the size of the structures minimal. At the same time, efficient scanning of references that are stored within an object is necessary to facilitate garbage collection.

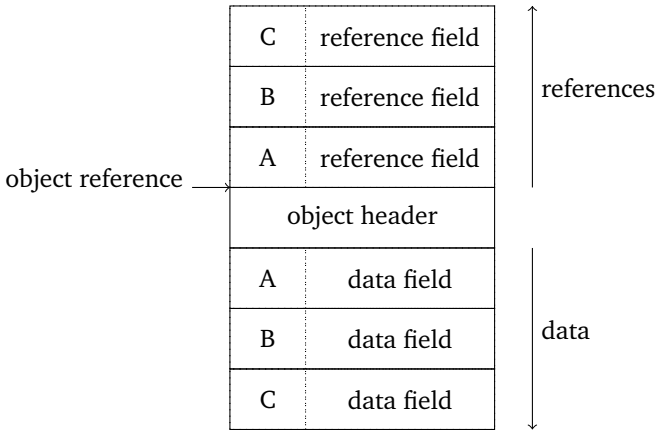


Figure 1.2 – Structure of an object in KESO [2, p. 84]. The object of class C inherits from the classes A and B, hence their fields are included in the object structure.

To achieve the above goals, KESO employs a *bi-directional object layout* that can be seen in Figure 1.2. Within KESO, an object reference points to the object header of an object. Using the bi-directional layout, data and reference fields of the object – containing fields inherited from parent classes – are laid out in adjacent directions starting from the object header.

By constructing the object structure in this way, a coherent array of references is achieved. This enables the garbage collector to treat the references of an object as an array.

The *object header*'s composition depends on the hardware KESO was compiled for. The details of its layout will not be discussed here and are available in [2]. Generally, it contains the class identifier, the size of an array (if the object is an array) and an area for memory management information. The latter is used, for example by the CoffeeBreak garbage collector, to store whether an object has already been scanned in this run of the GC.

Within the garbage collectors, it is necessary to determine the number of references stored in an object of a given type, as all references must be scanned by the collector. This number is stored in the so called *class store*. The class store is a property of each KESO system that is determined at compile-time, when all classes in the system are known. Within KESO, the class store is implemented as an array which can be accessed by using the class identifier of an object as index.

1.2.3 JINO

JINO is KESO's custom compiler that generates C code suitable for compilation with standard C compilers from Java class files. It is implemented in Java using a three stages approach: In the first stage, *parsing* of both the system configuration and Java class files into an intermediate representation of the KESO system occurs. After that, various *analysis and optimization passes* can be performed on the program. The final stage *emits* C code that can then be processed by a C compiler. The indirection over C code was introduced to avoid writing custom backends for each target architecture: A C compiler can be obtained for virtually any machine. It can also perform low-level optimization that is not possible in JINO.

The second step of the above compilation process is configurable: Some optimizations are only performed when the corresponding option was enabled in the configuration. It is also highly modular. Adding additional analysis or optimization passes is as easy as implementing and including a Java class – a benefit that will be exploited in Section 3.1 of this thesis.

JINO also implements KESO's *weavelet* mechanism. Weavelets allow arbitrary Java methods to be replaced by C code. Replacement can occur at the call-side of the method call or the method body can be replaced. For this purpose, JINO internally manages a list of possible weavelets – each of them is implemented in a custom class.

During translation, method signatures are matched against the signatures found in the weavelet classes. If a custom class implements a replacement for a certain method, it can emit the appropriate C code. As it also has access to some of the properties of the intermediate representation used in JINO, this step is not restricted to reproducing a template. For example, it can be checked whether a parameter given to a method is actually a constant value, which can be directly included in the C code of the weavelet instead of accessing it at runtime.

1.3 Summary

This chapter showed the basic system architecture of both KESO and JINO. Key points that are especially useful for this thesis, such as the bi-directional object layout

or JINO's weavelets, were highlighted. In the next chapter, the task of porting some of KESO's garbage collectors to Java will be analyzed in greater detail, and the challenges that arise during this task will be discussed.

Chapter 2

Analysis

This chapter is intended to examine the problem of porting system services of a *runtime environment* (RE)to the language the RE itself is written in.

To do so, first of all an overview over existing works in the same area will be given. As an example, the Jalapeño/Jikes RVM, Singularity OS and Squawk VM projects will be discussed and their implementation will be presented in closer detail. Common problems identified by the above projects are also relevant for the implementation of KESO's type-safe services in chapter 3.

Secondly, the current implementation of memory management in KESO will be introduced. The functionality of both the Restricted Domain Scope Allocator and the CoffeeBreak Garbage Collector will be explained. For both systems, a detailed summary of data structures that perform low-level data manipulation and serve as an interface to KESO will be given as these pose a challenge for the implementation of the services in Java.

In closing, the differences between C and Java will be presented. There are several distinctions in the feature set of both languages that must be kept in mind to successfully create a Java implementation of KESO's existing C code.

2.1 Related Work

Several other systems, including their system services (e.g. garbage collection), have already been implemented in a higher-level language like Java. This section is intended to give an overview over those systems and point out similarities and differences compared to KESO.

First, the Jalapeño/Jikes RVM system by IBM will be examined, the first *Java Virtual Machine* (JVM) written in Java. The second project being discussed is Singularity OS by Microsoft, a research operating system written in a dialect of C#. The

last system presented in this section is the Squawk VM by Sun, a system tailored especially towards embedded systems.

2.1.1 Jalapeño/Jikes RVM

The Jalapeño project by IBM, which was later continued under the name Jikes RVM, is a research project to evaluate the feasibility of a JVM written in Java. It is the first virtual machine written in Java that does not run on top of an already existing JVM, but uses a bootstrapping mechanism to set up its own runtime environment from an executable [3].

Being the first JVM written in Java, Jalapeño was also the first project to discover the difficulties coming with the task. Concerning garbage collection, which this thesis focuses upon, there are two main difficulties that were encountered by the developers of Jalapeño:

1. Low-level access is required to investigate system properties like marking bits in object headers or to access all references stored in an object (public references as well as private ones).

2. The garbage collector cannot make use of functionality it has to provide. This means that for example the `new` keyword is not available in the GC's own source code.

To circumvent the 1st problem, Jalapeño implements the so called `Magic` class. By using this class, an adapter between high-level Java code and low-level system manipulation is created: A method call in the Java code is replaced by the compiler with its actual functionality written in a low-level language like C or even assembler. While the coarsely grained structure of the `Magic` class has been criticized [4], as it does not allow a way to restrict memory access to the subset actually needed by the garbage collector, similar approaches were also used by the two other projects presented in the chapter. KESO's weavelets are a also a implementation of this concept.

The 2nd problem is solved by not using heap allocation during the running phase of the GC. Within Jalapeño this means that if a garbage collector needs any storage for internal data structures, the garbage collector instance is created in an immortal, uncollected memory region. In case of a copying garbage collector[1], that could potentially move the GC instance, special handling is needed to copy the GC memory before actually running the collection [5].

To enforce restrictions upon the system's source code, Jalapeño/Jikes RVM introduces the concept of "semantic regimes" [4]. Regimes allow the compiler

[1] A copying garbage collector moves all found objects at the start of the heap. This decreases fragmentation of heap space but does come at the cost of having to copy all objects and update the references pointing to them.

to check for forbidden operations in source code areas that need to obey certain limitations.

Closely tied to the project is the development of *MMTk*, the Memory Management Toolkit for and in Java [5]. It was created to replace the monolithic collectors of the original Jalapeño VM by a system using a more object-oriented design adhering to modern software engineering principles. For this purpose, object-oriented design patterns were used for the composition of the garbage collectors and a common interface to the Jikes RVM used by all their implementations was created.

2.1.2 Singularity OS

Singularity OS is a research operating system created by Microsoft Research with the goal to create a more dependable system than existing ones. To reach this goal, Singularity employs logical separation for each process much in the way KESO does with domains. Singularity separates its components into *SIP*s (Software Isolated Processes), closed environments in which the code is executed. Similar to KESO's domains, object references may not cross SIP borders and messages may only be passed using special channels.

Garbage collection in Singularity is implemented in Sing#, a dialect of C# that was created for the project. The system also has the ability to use different garbage collection strategies for each SIP within one system. A total of five garbage collectors has been implemented for Singularity.

To interface with underlying object data structures, the GC code is enhanced with the use of a `Magic` class, similar to that found in Jalapeño. This class provides access to properties normally not known in C#, such as the memory address of an object. Calls to methods of the `magic` class are replaced by Singularity's Bartok compiler with native code. This is not the only way Singularity uses low-level memory access. Contrary to Java, there are two modes for programming in C#: The safe mode and the unsafe mode. Code regions marked as unsafe may use pointers to objects. This mechanism is also heavily used within Singularity.

Similar to the mechanism implemented in this thesis (see section 3.1), Singularity's garbage collectors acknowledge the need that some methods may not perform heap allocations and use a method attribute to mark those methods.

2.1.3 SQUAWK Virtual Machine

The Squawk Virtual Machine is intended for wireless sensor devices. It was developed by Sun Microsystems. The goal of its development was the creation of a system that allows easy prototyping of sensor network applications by providing simpler tools than normally used in embedded systems programming [6].

Contrary to KESO, the Squawk VM is running on the bare metal without any underlying operating system. It is tailored to run on the Sun SPOT sensor device, and features device drivers and a network stack for that platform, both written in Java. There are two modes for running the Squawk VM: The hosted mode, which runs on development systems within a conventional JVM, and the on-device mode for using the VM in the field on its target hardware.

Similar to KESO, Squawk VM provides an isolation mechanism to separate multiple applications running on one device from each other. However, there is an actual, interpreting virtual machine running on the device. The on-device VM is written in Java and converted to C, which in turn is compiled into a binary. This VM cannot execute standard Java bytecode: A preprocessing step is required to convert Java classes to Squawk's custom bytecode, that is optimized for size.

To enhance configurability, the Squawk project implemented a custom preprocessor for Java. The preprocessor instructions come in the form of Java comments. This ensures compatibility with already existing tools and IDEs. Using the preprocessor, it is possible to inject compile-time constants into the code or conditionally alter which type of object is being instantiated in a line of code. This mechanism is, for example, used to choose the garbage collector algorithm in the system setup routine.

Native manipulation of memory is performed by means of the `NativeUnsafe` class [7]. The public methods of this class will be replaced by their native implementation during compilation, a mechanism similar to KESO's weavelet or the Magic classes in Jalapeño/Jikes RVM and Singularity OS. However, the `NativeUnsafe` class also provides a Java implementation of the methods for use in the hosted mode mentioned above.

Memory management is implemented in a subset of Java [6]. To obtain increased performance, its code is not interpreted by the VM on-device. Instead, it is translated into C code and then compiled. The memory management algorithms in the two available garbage collectors are not backed by a heap during their execution. The Java code of the garbage collection can therefore not perform any object instantiations during runtime. The only new object creations of the GCs are performed in their constructors. As Squawk VM uses only one heap for the whole system, the objects created by the garbage collector reside in the permanent address space that is located at the start of the heap. This part of the memory is reserved for system data, effectively being static allocation. The restriction to one heap differs from KESO, where there are multiple heaps for the domains in the system

It is interesting to note that Squawk VM's memory management implementations heavily rely on unsafe access to memory, even for their own data structures. This is a result of the limitations described in the preceding paragraph. The implementation of the marking stack[2] of the `Lisp2Collector` [7] may serve as an example. Its stack

[2]For explanation of the purpose of a marking stack, see section 2.2.2.

implementation is assigned fixed begin and end memory addresses, both of which are located in the permanent system memory region. Slots on this fixed size memory area are then subsequently accessed via an index. Object references are then stored on the marking stack using `NativeUnsafe`'s methods.

All in all, the Squawk VM provides a comprehensive implementation of a virtual machine for embedded devices in sensor networks. Its garbage collection, while implemented in Java, is, however, limited in the way it can provide type-safety on a language level as it relies on unsafe memory access for its operation.

2.2 Current Implementation of Memory Management

Currently, there are several options for memory management available in KESO. The simplest implementation, called the *RDS* allocator and the *CoffeeBreak* garbage collector will be examined closer in this thesis. Advanced memory management techniques, that are especially suitable for the use in real-time systems, are the Idle Round Robin and a dedicated fragmentation-tolerant garbage collector *FragGC*. For the sake of brevity, those two will not be subject to this thesis.

Theoretically, each domain in the KESO system can employ different memory management strategies. As an example, it is possible to use (for example) a domain with an RDS allocator and another domain with a Coffeebreak garbage collector. Each memory manager can store state information (counters, free memory lists and the like) in a special heap area of embedded in the domain descriptor of the domain it manages. Also, for each domain there is a fixed-size array that is used to provide memory for object allocations of tasks executed in the domain (the so called *heap array*). Its size may be configured by the user in the system configuration.

Presently, all of the above memory management techniques are implemented in C. This makes accessing KESO's system properties very simple. Configuration of the mode of garbage collection is done via preprocessor usage or conditional inclusion of source files by JINO.

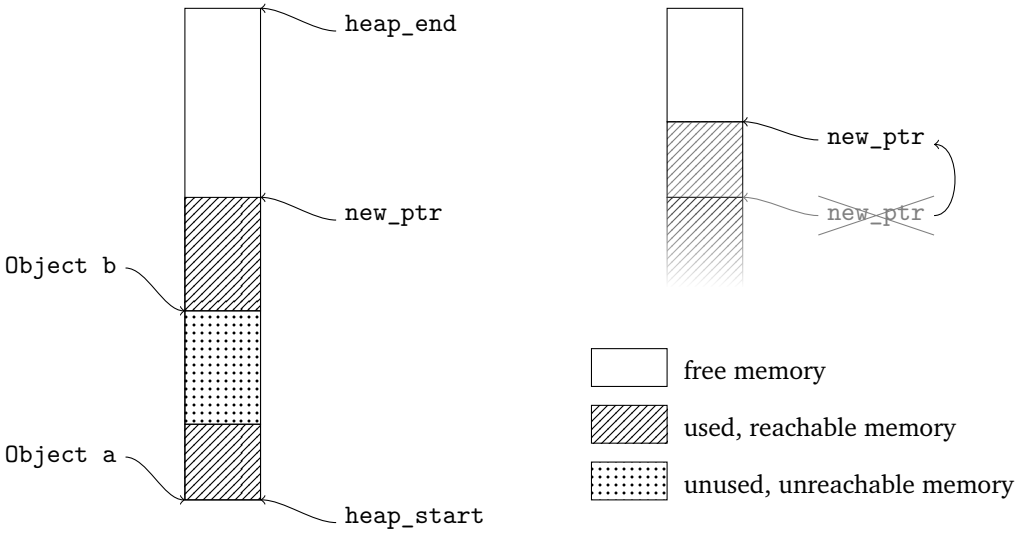
2.2.1 Restricted Domain Scope Allocator

The RDS allocator is the simplest memory management technique. It provides pseudo-static allocations without an actual garbage collection technique. The functionality of the RDS allocator will be discussed in this section.

2.2.1.1 Operation of the RDS Allocator

The *Restricted Domain Scope allocator* (RDS allocator) is an allocation-only memory management technique. It does not implement any garbage collection mechanisms

as allocations are performed statically. Its internal state consists of the heap array, a *new pointer* and a *heap end* pointer. The heap array is being used as a memory source to satisfy incoming requests for memory. The new pointer references a position in the heap array. At this position, a new object may be created. The heap end pointer points to the end of the heap array. It is used as a safety feature: Before every allocation, the memory request is checked against the heap end pointer. If the memory request would exceed the boundaries of the memory area managed by the RDS allocator, system execution is aborted.



(a) General structure of an RDS heap **(b)** Allocation on an RDS heap

Figure 2.1 – Structure of an RDS heap and schematic of an allocation

Figure 2.1a illustrates the general structure of an RDS heap. Several allocations have already occurred. For that reason, the new pointer has already been advanced toward the end pointer of the heap array. Figure 2.1b shows how an object allocation occurs. First, it is checked whether the memory request can be fulfilled. The memory address the new pointer points to is being used as the address of the newly created object. The new pointer is then increased by the size the object uses in the heap array.

In Figure 2.1a, there are currently two objects on the heap that are still reachable by the user application. Also, there is a memory block in between the objects that is not filled with an object reachable by a reference in the application. This block has previously been used by the application. When the object that took up the chunk of memory went out of scope, the memory block is not reachable any more. This illustrates the drawback of the RDS allocator: As absolutely no garbage collection takes place, memory that was once allocated but is now not actively referenced by

the user application any more may not be reused. The RDS allocator is therefore not usable for more complex applications that repeatedly allocate and dispose objects at runtime. In such applications, memory requests could not be fulfilled any more after a certain amount of time. Albeit seeming very limited, the RDS allocator is suitable for many small embedded systems that only perform allocations at the start of their life-time.

On its positive site, it is very simple and provides a predictable allocation strategy with little to no chance for errors. It is useful especially for simple applications that allocate objects during their creation and use only this limited set of objects during their runtime.

2.2.1.2 Native Interface to the System

As seen above, the RDS allocator possesses only a minimal number of data structures. The only data structures stored in the allocator are the new and the heap end pointers. These pointers are embedded in the domain data structure which the allocator was configured for. Access to these pointers must therefore be managed by weavelets because the domain descriptor is not accessible in Java code. Access to the new pointer must be read/write whereas the heap end pointer should be read only.

For object creation, three different methods are necessary. First, a raw chunk of memory must be split from the domain heap array. Secondly, this chunk of memory has to be written over with zeros. This serves two purposes: Object references stored in the object are being reset to null, preventing them from pointing to arbitrary memory locations and potentially breaking memory access restrictions. A second benefit is that primitive fields in the object structure are initialized to 0, their default value according to Java language specifications.

In a last step, the raw memory block must be initialized as a valid object. This requires casting the raw memory block to an object structure and adding the reference offset (see Section 1.2.2) to the start address of the object. The latter step requires pointer arithmetics and may therefore not be done by Java. The class identification number is being stored in the object structure – requiring access to the raw object structure, as this field is not accessible by Java code.

2.2.2 CoffeeBreak Garbage Collector

2.2.2.1 Operation of the Garbage Collector

The CoffeeBreak garbage collector is a conventional blocking garbage collector following a mark-and-sweep principle. During its execution, all other tasks in the system must be paused or blocked. This constraint ensures that all data structures are in a consistent state and no non-garbage objects will be freed by the CoffeeBreak

GC. The collection phase of the garbage collector is running in its own task with the lowest priority within the KESO system. This GC task is logically located within the so-called Domain Zero (see Section 1.2.1) and is therefore located in the trusted code base of KESO.

Using the CoffeeBreak garbage collector, allocated but unreferenced objects can be found. By collecting them, their memory can be reused later to create fresh object instances. This feature enables applications with a more complex allocation and deallocation behavior, which are able to run in an endless loop while creating new objects in each cycle.

Internally, a free memory list is stored to keep track of the available free heap memory. An example heap of a CoffeeBreak-managed domain is visible in Figure 2.2. As can be seen in the figure, the free list is implemented as a simple linked list. In addition to a pointer to the next element in the list, each list element also stores the size of its memory block. It is also apparent that the list elements of the free memory list are stored in the heap areas they describe. This is done to avoid managing a dedicated memory area for the free memory list. This is done to avoid the principle of static allocation within the CoffeeBreak GC – a separate memory area for the free memory list would mean managing a kind of heap. The last list element's next pointer will point to the symbolic address `NULL`, thus marking the end of the list.

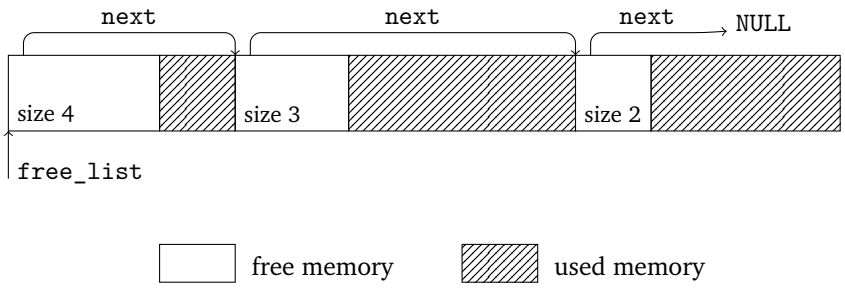


Figure 2.2 – Heap of a CoffeeBreak managed domain. The free list is embedded in the free memory blocks (source: own work).

Operation of the CoffeeBreak garbage collector can roughly be divided into three steps: Allocation, a marking phase to find referenced objects and a sweep phase to reconquer memory occupied by objects which are not reachable from application code.

Allocation

The allocator method receives all necessary information to create a new object. This information consists of the size of the object, its class identifier and the reference offset (see Section 1.2.2 for details about the object layout) of the object. In the allocation method, the free memory list is being traversed to find a block of memory that is sufficiently large to contain an object of the needed size. If such a block is found, the iteration of the list is stopped. Depending on the relation between the size of the found memory block and the size of the requested memory chunk, there are now two possible cases:

- If the memory block size is exactly that of the request, the whole memory block is removed from the list and used for the object.
- If the found memory block is larger than the memory request, it is split in two. The first part is used for the actual object. The second part is enqueued into the free memory list as a new list element. However, this may only occur if the size of the remaining block is large enough to take up the list element structure needed for the new list entry.

In both cases the memory chunk reserved for the new object will be overwritten by zero, cast into an object structure and equipped with a class id and a reference offset. This last step is equal to that described for the RDS allocator in section 2.2.1.1.

Marking Phase

During the marking phase live objects on the heap must be found and their position must be memorized by the garbage collector. For that purpse, information about used and free areas of the heap is stored in a bitmap and will be used in the following sweep step. Each bit in the bitmap contains information whether a certain area of the heap is in use. To facilitate usage of the bitmap, the heap is logically divided into so-called *slots*. All slots are of the same size which is configurable by the user. Organizing the heap in slots reduces the size of the bitmaps. Instead of using one bitmap entry for each addressable memory unit (typically the size of a byte), a bitmap entry now corresponds to a larger block of memory. The bitmap is used in the sweep phase of the collector to find coherent free memory areas.

Internally, the marking phase uses a working stack to store found objects that need to be scanned for further object references. Marking continues as long as there are still objects on the working stack.

At the beginning, there are three sources of objects that form the so-called *root set*:

System Objects System objects that are created by JINO ahead-of-time, such as the instances of the `Thread` class, can contain reference fields.

Static Fields References in the static fields of objects are found by JINO at compile-time. All static reference fields are then placed in a shared array in the domain descriptor. As the CoffeeBreak GC is aware of the domain descriptor of the domain it is currently scanning, all objects in the root set can be found by traversing these arrays and pushing its entries onto the working stack.

Objects on the Stack(optional) Linked stack frames [8] are a technique employed by KESO to allow scanning local object references in tasks that are blocked. Stacks of tasks that are not blocked are empty, because such tasks are already finished: The CoffeeBreak task is running with the lowest priority of the system.

In conventional JVMs, it is difficult to scan a task's stack for references: At runtime, object references may not only be stored in local variables bound to the root set described above, but also in the registers of a processor or on the stack frame of a method. However, it is impossible to tell for a garbage collector whether a value found in a register is a primitive value or a reference. A pessimistic approach would therefore count every primitive value, that could by its value also be a reference, as a live object on the heap. This reference will then be added to the set of live objects, and its memory stays reserved. This could postpone collection of the referenced memory area that does not contain any actual objects.

KESO is eliminating this source of error by using an always up-to-date list of references in the current method frame and omitting registers in the reference scan. As a task can only be garbage collected if it's either finished or blocking (caused by a method call to the `WaitEvent` system call in OSEK), its linked stack frames are always in a consistent state, meaning that it is guaranteed that all its references that are not stored in fields are stored in any of the reachable stack frames.

Upon entry of a method that may in its call hierarchy call `WaitEvent`, an empty reference table is being created. Its size corresponds to the number of object references used in the method's code. At the beginning, the entries in the object table are empty. For every new object created in the method, its entry in the table is replaced with the reference to the object. Now, all references kept in a method are stored in the table and can be easily traversed by the garbage collector.

To find references that have been created on different levels of the call hierarchy, the reference tables of all methods in the hierarchy are linked. The first table can be found by accessing a special field in the task descriptor. The end of the list is marked by a magic value instead of a valid pointer. Upon creation of a new reference table, occurring when entering a method, it must be linked to its predecessor [2].

All in all, the linked list of local references allows the garbage collector to find object references that only exist in method local variables. It eliminates the need for the garbage collector to scan processor registers and allows to dependably identify only actual objects.

Sweep Phase

The sweep phase makes use of the information that was stored in the bitmap during the marking phase. At its start, the free memory list is cleared. The bitmap is then searched for contiguous portions of the heap marked as free. By design, the use of a bitmap allows to easily merge multiple neighboring blocks of memory into one coherent free block without any further effort. When an area of free memory has been found, a new list item is being created. The list item is then enqueued into the free list. Once the bitmap has been searched completely, the GC run is finished.

2.2.2.2 Native Interface to the System

Access to the heap array of the domain descriptor by the CoffeeBreak GC is comparable to that of the RDS allocator described in 2.2.1.2. However, the CoffeeBreak GC makes use of a set of other system resources that need access to internal implementation details.

Firstly, marking objects relies on storing color information in object structures. For this purpose, a part of the object header is reserved for usage by the garbage collector (see Section 1.2.2). When a live object is scanned by the GC, it is marked with a certain color. This coloring prevents re-scanning object references that have already been handled in this run of the garbage collector, what could eventually lead to an endless loop. The current color bit is also contained in the domain's heap descriptor. Access to both of these data structures requires access to the object header – an easy task in C, but it requires a weavelet if it should be performed by Java code.

Secondly, scanning an object relies on finding all references stored in it. This requires access to its fields, be they public or private. As a reflection interface is not available within KESO, this information can only be accessed by directly manipulating the object header. All references contained within an object are stored subsequently in its header. Information about the number of references accessible in an object header must be acquired by querying KESO's class store. This is easily possible, as the object's class identifier can be used as an index into the class store array, as explained in Section 1.2.2

Lastly, scanning local object references in blocked tasks can only be performed by accessing task descriptors and traversing the linked stack frames of blocking tasks.

2.3 Comparison of Language Features: Java vs. C

The focus of this thesis lies on porting system components of KESO from their original implementation in C to Java. However, the two languages have several differences which make a direct translation from one language to the other difficult.

This section is intended to give an overview over the features of the two languages. Where a feature is not directly available in Java, but is needed in KESO's implementation of system services, possible workarounds are shown.

2.3.1 General Overview

C has its origins in the development of the UNIX operating system by Kernighan and Ritchie [9]. Although it is not only useful in that context, C has many features that make it fit for the purpose of system programming. C is an imperative language that allows to declare functions and structures, but it offers little of the abstractions such as object-oriented design present in many modern languages such as Java.

C programs are compiled into machine code. The compilation processors typically includes a preprocessor run before the actual C compiler run, a mechanism that is examined closer in the next section.

Java is an imperative, object-oriented language. The goal behind its invention was to provide a common language for implementing platform-independent applications. Also, Java should facilitate creating more robust programs by ensuring memory safety. This means restricting memory access to the bounds of actually created objects. As a trade-off, low-level access to hardware is normally not desired in Java. In contrast to C, The language also guarantees type-safety. With this feature, it is always possible to determine the type of a variable – for example, if a variable is a reference to an object or a primitive value. This feature allows the creation of automatic garbage collection that frees memory areas not needed by objects.

A Java program is normally shipped in the form of a standardized byte code file. The byte code is then being executed by a *Java Virtual Machine* (JVM), which is the only part of the system that has to be ported to new machines. In KESO, Java is translated to C code using JINO, which can in turn be compiled to a binary suitable for execution on a machine.

2.3.2 Preprocessor Usage

The C compilation process puts a preprocessor run before the actual compilation in which the C code is parsed and transformed. The preprocessor itself does not parse the C source code. Instead, it performs basic text manipulation: Files can

be included, symbolic value replacement is possible and source code lines may be omitted or included based on conditions.

```
1  #include "header.h"
2
3  #define NUM_ENTRIES 42
4
5  int main(int argc, char **argv) {
6  #ifdef SYSTEM_HAS_PRINTF
7      printf("Test value: %d\n", NUM_ENTRIES);
8  #endif
9      return 0;
10 }
```

Listing 2.1 – Example of preprocessor usage in C.

Some of the features of the preprocessor are shown in listing 2.1. These features are the following ones:

File Import In line 1, a header file is included into the current file. This step places the content of the header file at the place of the `#include` statement. It can for example be used to import variable of function declarations.

Text Replacement Line 3 shows the use of symbolic constants: All occurrences of `NUM_ENTRIES` will be replaced by the value 42. This step may also include macros. Using macros, the preprocessor will then handle inserting parameters that can be passed to the macro call into the replacement string.

Conditional Compilation This feature of the preprocessor is shown in the lines 6 to 8 of the listing. Line 7 will only be included in the preprocessor output if the value `SYSTEM_HAS_PRINTF` has previously been defined.

Especially the latter feature is often used in KESO's source code. Depending on the configuration options which were enabled in the system configuration, or compile-time optimization by JINO, features may be available or disabled in the final system that consists of a set of system components and is running on the actual hardware. For example, systems that do not make use of arrays of objects do not have to include the definitions needed for such data structures. This means, that a garbage collector such as the CoffeeBreak GC does not have to include the ability to handle arrays of objects – the necessary part of the code can thus be omitted. This is done via preprocessor conditions.

Similarly, preprocessor constants are used to communicate the values of constants determined by JINO during compilation to the final system. During the coding phase, JINO emits preprocessor statements, e.g. for the number of tasks in the system, into

the appropriate header files. Those files can then be included where the information is needed.

Contrary to C, Java does not include a preprocessor per default. Instead of including symbolic constants and portions of code before the actual compilation step, object-oriented techniques are employed. This is due to the fact that the preprocessing is performed outside of the normal language scope, a frequent reason for criticism. In the context of object-oriented programming, basic preprocessor features like including constants by text replacement hurt the principle of data encapsulation – all data should be found in the object hierarchy and not come from outside of it. Configuration of the system should also rather happen by methods such as dependency injection instead of conditionally including the needed parts of code. With dependency injection, objects are created dynamically at runtime and passed to their places of use via references. Having to rely on object instantiations does impair the applicability for system services that may require a static allocation scheme.

However, some features in system programming may only be implementable by using a preprocessor. The lack of a preprocessor in Java for exactly that purpose has therefore been noted by the Jalapeño project in [3]. There are Java projects where a custom preprocessor was implemented. One example would be the Squawk VM presented in section 2.1.3. For KESO, no such preprocessor was deemed necessary, as its system interface is mostly written in C. The C code can be emitted conditionally by JINO or the original C preprocessor can be used.

Relying on the above methods means that KESO does not adhere to the principle of data encapsulation on its system level. However, on the application level, where the programmer expects to use *pure* Java, data encapsulation is provided, with the only exception of side-effects introduced by weavelets. As weavelets should be used sparingly, and are only necessary in use-cases not intended by Java (such as the one in this thesis), this is a minor impact.

2.3.3 Memory Management and Usage

As C has its origin in system programming, it allows very fine-grained control over its memory usage behavior. The programmer can decide whether to place a date on the stack, the heap or – in the case of global variables – in the data segment of the program.

Allocating heap memory in C is traditionally implemented using library functions such as `malloc` and `free`. Thus, the programmer has full control over the memory usage if their program. They can also explicitly create a pointer to arbitrary memory addresses. As a consequence, C code can possibly be written without having a dedicated heap area where new objects can be created. The way current GCs are

implemented in KESO is exemplary for this way of programming – all objects in the garbage collectors are either statically allocated or exist temporarily on the stack frame of their method hierarchy.

As noted in Section 2.1 during the discussion of Java-based memory management in other projects, Java heavily relies on an existing memory management technique for its implementation, as it is one requirement for a full-featured JVM. New instances of objects are typically created on the heap, but modern JVM implementations can allocate objects on the stack. KESO's mechanism for stack allocations will be examined in Section 3.1. There are no keywords for deliberately deleting an object that goes out of scope and will not be needed any more. This is done to prevent *dangling pointers*, object references that point to memory areas where an object has previously existed, but was deleted. The reference would then point to memory space that does not belong to an object, hurting Java's memory model.

A full-featured Java implementation therefore always requires a working garbage collection mechanism. Where no such mechanism is available, only a subset of Java can be used. This subset needs to be restricted to either abstain completely from object instantiations by forbidding the use of the `new` keyword or ensuring that only allocations that can be performed on the stack are allowed. The latter solution will be enforced by the implementation presented in this thesis in Section 3.1.

This subset can either completely forbid heap allocations – this is the way memory management is implemented in existing systems. Within KESO, objects may also be allocated on the heap.

However, this restriction renders large parts of the standard Java class library unusable. Data structures like the working stack implementation of the CoffeeBreak garbage collector need to store immortal data on the heap. It is also not possible to use methods that internally create an object and return it to their caller. This is, for example, done in the `Integer.toInt()` method of Java's standard class library.

2.3.4 Pointers and Direct Memory Access

As noted above, C's intended use case was the programming of operating systems. Hence, it makes direct memory access available to the programmer: With the use of pointers, memory addresses may be referenced, read from and written to. Pointer arithmetics is also available. This allows to write efficient low-level programs as only a minimum of abstraction is available between the higher-level language and the bare machine.

Java does not have the concept of a pointer to arbitrary memory addresses that can be manipulated manually by the programmer. At the time of its creation, Java was primarily intended for application programming and thus did not have the need to directly interfere with hardware devices. Additionally, pointers are a

notorious source of errors: Incorrect use may accidentally alter memory locations, thereby leading to data corruption or damage to the runtime environment. Java's programming model makes use of references to objects. A reference may either be invalid (`null`) or point to an object. Consequently, Java ensures *memory safety*. Memory safety is the guarantee that references may not point to invalid addresses such as an out-of-bounds access of an array at an invalid index or addresses of objects that are now deallocated [10].

One goal of KESO was promoting the use of a type-safe language for as many system components as possible to exploit features such as static analysis of . This created the need of directly accessing hardware features of the processors the system would run on: Hardware drivers should be written in Java. For this purpose, there are two mechanisms which break the boundaries of Java: Weavelets and memory-mapped objects.

Weavelets, as shown in Section 1.2.3, allow to call snippets of C code from a Java program. Therefore, they may be used to access and manipulate memory contents hidden on Java's language level. This topic will be discussed further in chapter 3.

Memory mapped objects allow to map certain memory addresses to Java objects. The memory address to map is statically configured in the Java source file. This mechanism is useful to access ports available in micro controllers. With memory mapped objects, device drivers may be entirely implemented in Java. An example of this feature, and a discussion whether it may be used for KESO's system services, can be found in section 3.2.

2.4 Summary

This chapter provided an overview over existing projects using type-safe garbage collector implementations in high-level programming languages. Among the problems commonly encountered by these projects are the lack of low-level memory manipulation and the need to abstain from using memory management itself in the Java code of the aforementioned implementations.

Two of the memory management tools provided by KESO – the RDS Allocator and the CoffeeBreak GC – were examined closely. Attention was paid not only to their functionality, but also to their interface to KESO's internal data structures. These data structures must be handled with care, as there normally is no representation for them on Java's language level.

In closing, a comparison between Java and C was performed. The two languages differ not only in their language paradigm – an imperative, type-unsafe and an object-oriented, type-safe language – but also in their targeted use cases. Where C was originally intended as a system programming language running on the bare

metal, Java aims at developing applications running in a managed environment. This is problematic because it consequently – while bringing additional benefits – lacks some features found in C. Among those features are some like memory pointers that are critical for the implementation of a heap management strategy.

Solutions to the above problems that were developed during the implementation of KESO's garbage collectors will be suggested in the next chapter.

Chapter 3

Implementation

This chapter presents the implementation of type-safe garbage collection services that was created in the course of this thesis. The first section shows how stack-allocation may be enforced by JINO upon garbage collectors that must not perform allocations on a heap. After a discussion of the possibility of using KESO's memory-mapped objects in the garbage collectors, the last two sections present the static implementations of both the RDS allocator and the CoffeeBreak GC and in closing the dynamic CoffeeBreak GC.

3.1 Restriction to Stack Allocation

As seen in the previous chapter, allocation of objects on the heap is not feasible for garbage collection services that do not have a heap. This section shows how stack allocation can be enforced in KESO at compile-time and shows additional benefits for systems using the RDS allocator.

3.1.1 Motivation

As seen in the language comparison in section 2.3, many features of Java rely on allocating objects on the heap. This makes reimplementing the operation of system services that were originally written in C difficult.

All variables used by garbage collection algorithms implemented in C are either method-local or allocated ahead of time in the data section of the program. The only exception is made by the free memory list of the CoffeeBreak GC: It is placed within the free memory areas of the domain heap it manages.

3.1.2 Java Annotations in JINO

The Java Virtual Machine Specifications [11] define so-called annotations. Annotations allow the programmer to enrich the program code with non-functional information. Annotations can be used either by the JVM or accessed by the program via reflection. An example of an annotation that is bound to a method can be found in listing 3.1. Here, the method `someMethod` has been equipped with the annotation `SomeAnnotation`. New annotations – like the annotation `SomeAnnotation` – are defined as Java classes.

```
1 @SomeAnnotation
2 public void someMethod(int arg) {
3     // ...
4 }
```

Listing 3.1 – Example usage of method annotations. The annotation `SomeAnnotation` has been added to the method.

According to the specification, annotations are stored in the byte code of a Java program. They are contained in the `attributes` section of classes, fields or methods. There are two categories of annotations: runtime invisible and runtime visible ones. Runtime invisible annotations are intended to be used by the JVM. Runtime visible annotations can be accessed using Java's reflection interface. Each annotation can obtain additional parameters. These are stored as a list of key value pairs in the byte code of the annotation.

For checking the allocations performed in statically executed garbage collector code, JINO was expanded to support a limited subset of annotations. Parsing method annotations has been implemented into the `MethodData` parser. As JINO is an ahead-of-time compiler – and a reflection interface is not available in the KESO system – only `RuntimeInvisibleAnnotations` can be parsed. If any other annotation type is encountered while parsing of Java class files, a message is printed and the corresponding section of the Java byte code will be skipped.

For the limited support needed in JINO, annotations do not need to support parameters. Any encountered parameters to annotations will hence be discarded in the parsing process. This further facilitates storing the annotation information, as omitting parameters to annotations obliterates the need to actually instantiate the class defining the annotation. Therefore, all annotations bound to a method definition will be stored as a list of class names in the compiler's internal method representation.

3.1.3 Compiler Checks

JINO should enforce two constraints on methods that do not allow heap allocation:

1. Methods that were marked as *stack-allocation-only* should be able to exclusively call methods to which the same restriction applies.

2. During the coding phase, JINO decides whether an object creation should create an object on the stack or perform a heap allocation. When a method is marked as *stack-allocation-only*, any attempt to create a heap allocation for an object instantiation must be prevented here.

To make the programmer's intent known to JINO, methods that require to be run in a static context must be marked. This is achieved by creating a custom annotation named `@NoHeapAllocation` that can be attached to methods. Usage of this technique is demonstrated in Listing 3.2, where the method `restrictedMethod` should not perform any heap allocations.

The first goal is then implemented as an additional compiler pass called `ForcedStackAllocationAnalysis`. For every method marked with the `@NoHeapAllocation` annotation, all callees are found and checked for the same annotation. If a callee is not flagged correctly, an error is thrown to inform the programmer and translation is aborted.

Obviously, this pass requires knowledge about all callees of a given method and benefits from KESO's closed-world assumption – no dynamic class loading at run time is possible. The information can be obtained by querying the call graph of the application, where all possible method calls are analyzed. Its construction has already been implemented in JINO and its creation is required in the pass configuration of the `ForcedStackAllocation` pass.

The second goal is implemented by inserting additional checks in JINO's coding phase. Here, the decision whether an object will be allocated on the stack or on the heap is made. This decision relies on several parameters like the size of the object or its *escape state*.

The escape state describes whether an object leaves the scope of its creation. This information is created during the `EscapeAnalysis` run of JINO that determines where an object must be created. In JINO, there are three different escape states [1, p. 19]:

```
1 @NoHeapAllocation
2 public int restrictedMethod(B input) {
3     A aInstance = new A(input);
4     return aInstance.doStuff();
5 }
```

Listing 3.2 – Method `restrictedMethod` must allocate the local object `aInstance` on its stack.

Local The object is only used locally and does not escape the scope boundaries of its creating method.

Method The object escapes its creating method but not its thread.

Global The object also escapes its thread.

Concerning stack allocation, only objects that have an escape state of *local* may be created on the stack. An example can be seen in Listing 3.2. Here, the object `aInstance` is not stored in a class variable or used as a return value. It thus has an escape state of *local*, allowing the object `aInstance` to be created on the method's stack.

Correct enforcement of stack allocation therefore also relies on escape information for the application being created. The additional compiler pass introduced above therefore depends on the `EscapeAnalysis` pass. As noted earlier, up to now JINO decides the mode of the allocation (heap or stack) in its coding phase. The decision does not only depend on the escape state but also on other factors such as aspects of the configuration or the object's size. The object may either be too large to fit on the stack or its size may be dynamic.

This behavior could not be changed fundamentally. Instead, an error message is now given by JINO at compile-time if an object can only be allocated on the heap but this operation is forbidden in its scope. The programmer must then manually circumvent the source of the problem.

3.1.4 Additional Benefit for Systems Using the RDS Allocator

Besides its use in implementing Java code to be executed in an environment without a heap, there is yet another benefit of automatically checking for heap allocations using method annotations. As noted in section 2.2.1.1, using an RDS allocator forces the programmer to initialize all used Java objects at the start of the application. After that, no heap allocations should take place to prevent memory exhaustion. The program is therefore divided into two parts: initialization and execution.

Initialization methods can be left unchanged, and all heap allocations can be performed within them as usual. However, it is possible to mark the execution methods of the program with the `@NoHeapAllocations` annotation. Now, JINO will automatically find unwanted heap allocations and alert the programmer if such are found. That way, the program can be statically checked and programming faults that would lead to failure during runtime can be prevented.

3.2 Implementation Using KESO's Memory Mapped Objects

3.2.1 General Overview

As KESO's target architecture are embedded systems, it must provide a way of interacting with its environment. On small-scale systems, full-featured device drivers are often not available or needed. To control peripheral devices, the out- and input pins of the micro controller are mapped into the address space.

```
1  package atmega128;
2  import keso.core.*;
3
4  public class PortA implements MemoryMappedObject {
5      // PortA consists of three 8-bit registers (unsigned)
6      public MT_U8 PINA; // address 0x39
7      public MT_U8 DDRA; // address 0x3a
8      public MT_U8 PORTA; // address 0x3b
9
10     // create a mapping of this class at base address 0x39
11     static PortA regs = (PortA) ↘
           MemoryService.mapStaticDeviceMemory(0x39, ↘
           "atmega128/PortA");
12
13     // configures one port PIN as output PIN
14     public void setOutput(byte pinNumber) {
15         DDRA.or(1 << pinNumber);
16     }
17     // the class can contain more mthods and also regular (not ↘
           mapped) fields
18 }
```

Listing 3.3 – Example usage of memory-mapped objects as a device driver [12].

Listing 3.3 shows the use of memory-mapped objects in KESO. Three 8-bit registers are made accessible by Java code in the lines 6 to 8. Special classes, identified by the prefix `MT_`, are used to denote memory types. To set up the memory mapping, a system service must be called (line 11). The objects are then accessed using ordinary Java methods (line 15). In the example it can be seen that memory-mapped objects can make use of Java's visibility modifiers. An elegant object-oriented design of driver classes that makes use of information hiding is therefore possible.

3.2.2 Evaluation of Memory-Mapped Objects for Low-level Access in Garbage Collectors

At first glance, memory-mapped objects seem well-suited for the purpose of implementing KESO's garbage collection in Java. Internal structures such as the domain descriptors could be mapped to Java objects. This way, for example the root set of a domain could be conveniently accessed by the garbage collector.

By implementing additional memory types for pointers, a Java interface for this low-level construct could be provided. It would also be possible to execute sanity checks when accessing pointers and thereby enforce some kind of access restrictions.

However, there are two major drawbacks that render memory-mapped objects unusable for garbage collectors:

1. Memory-mapped objects are mapped to static addresses (see line 11 of Listing 3.3).

2. The layout of memory mapped objects is fixed.

The first item especially concerns the usage of memory-mapped objects for data structures that can be placed at arbitrary positions in the systems main memory. It would be possible to adopt the `MemoryService`'s functionality so that the mapping to structures such as the domain descriptors would work, as the position of these descriptors is known to the linker. However, the access to objects that are placed at a variable position in the memory is not possible that way. The only way of accessing such structures, for example the references in an object, is using the weavelet mechanism.

Concerning the second problem, the difficulty arises from the way internal data structures of the runtime environment are constructed in KESO. For example, the layout of the domain descriptor C `struct` is created based on the configuration options chosen by the user for this system build at compile-time by JINO. This means that the number of fields in the descriptor varies. If a memory mapped object with a given data layout would be mapped onto a domain descriptor that had been given another data layout by JINO, the system would not work correctly or not at all.

To make matters even worse, error checking during runtime would come with high costs and is therefore not reasonable. Therefore, crashes or other unpredictable behavior caused by the mismatched data could occur.

Especially the second problem makes KESO's memory-mapped object mechanism unusable in garbage collectors. The impulse to use memory-mapped objects was therefore discarded and KESO's weavelet technology was adopted to provide access to low-level language constructs. The weavelet mechanism allows to access data at arbitrary memory locations and is independent of the structure of the domain

descriptors, as the elements of the structure can be accessed by their name and not their position.

3.3 Static Implementation

As noted before, implementations of memory management in Java must not rely on the memory management itself. Previous projects like the ones in presented in section 2.1 therefore abstained from using Java's object allocation.

In this section, implementations of the RDS allocator as well as the CoffeeBreak garbage collector that were created in the context of this thesis will be presented. Both do not make use of Java's dynamic heap management and are hence called static implementations.

With this task come certain problems concerning the storage of program data that must be available over the whole program runtime and must survive the scope of its creation and the usage of the Java's standard class library. These problems will be discussed and possible solutions will be presented.

3.3.1 Restricted Domain Scope Allocator

This section explains the implementation of the static RDS allocator that was performed for this thesis. Both the compiler-side changes and the system-side implementation are shown here.

3.3.1.1 Compiler-Side Implementation

During the compilation process of KESO, parameters for its system components are read from the configuration file. To store this information as well as additional data during the compilation, an intermediate representation for parts of the system is needed. In JINO, each heap management mechanism is represented by its own *intermediate representation* class. At compile time, this class will be instantiated within JINO, equipped with values from the configuration and computed information and emit the code needed in KESO.

```
1 Heap = UserDefinedDomainScope {
2     HeapSize = 4096;
3     ClassFile = "javards/RDSHeap";
4     AllocMethod = "keso_rds_alloc(III)Ljava/lang/Object;";
5 }
```

Listing 3.4 – Configuration of the Java RDS allocator within a domain configuration section.

For the Java implementation of the RDS allocator, a custom heap class called IMHeapUserDefinedDomainScope with a custom system configuration entry was created. A sample entry may be seen in listing 3.4. It is apparent that three configuration values are needed:

HeapSize The size of the heap in bytes. It is needed to create a heap array of the correct size in the KESO system.

ClassFile The class file of the implementation of the RDS heap.

AllocMethod The method within the class that handles allocations. Note that the signature of the method must be given explicitly.

It may seem needless to write the exact method signature of the allocation method into the configuration file. However, this is needed to ensure correct compilation of the system.

During the compilation process, various optimizations are applied to JINO's internal program representation. Some of these optimizations include the removal of dead code. In normal use – when JINO is just translating a Java application – all classes needed by the application are deduced from the applications main class. All classes and methods required by the application are determined by examining call and reachability relations from the main class.

When translating the Java Restricted Domain Scope allocator, the class of the allocator is not reachable by direct calls from the application's source code. Instead, it is indirectly needed via new calls. This relation is not known to JINO. Hence, it must be ensured that the Java RDS code is not optimized out by JINO. To do so, the necessary classes are specified in the configuration and made known to the reachability analysis.

There are three places within JINO where special care must be taken to keep the classes of a Java heap in the class store. The first is the JavaDomain class, which must manually require the files needed for its heap implementation. The second is the CallGraphAnalysis class, which must be made aware of the classes belonging to the heap implementation it is currently optimizing. The last is the ReachabilityAnalysis component of JINO. Heap methods of a Java heap must be added to its root set, as they are not reachable for the program by direct calls.

3.3.1.2 System Implementation

The implementation follows the general functionality presented in Section 2.2.1.1. To perform memory manipulation, and access domain descriptor fields, the weavelet Domain_RDS was created. Besides accessing the domain descriptor, two larger methods are managed by the weavelet: Allocating a new object and setting the class identifier for the object.

Two methods were necessary that do not belong to the Domain_RDS weavelet: Overwriting the new object's memory with zero and interfacing with KESO's error message output, for printing out a message in case the heap memory is exhausted. Those two methods were created in the Marshal weavelet.

Normally, applications within KESO may use standard Java IO mechanisms. Given the restriction that no heap allocations may occur, this is not possible any more. Java's implementation of IO streams internally manages a buffer that is located on the heap, as it may be increased in size during run time. Consequently, the Marshal weavelet implements adapter methods for KESO's normal output primitive as well as the error handling function.

3.3.2 CoffeeBreak Garbage Collector

This section is intended to show the implementation of the CoffeeBreak garbage collector that was created with the restriction of not allocating objects on the heap. After an overview over the structure of the component, the usage of weavelets for storing data structures of the garbage collector will be explained in greater detail.

3.3.2.1 Overview

The CoffeeBreak garbage collector is more complex than the RDS allocator presented in the previous section. This is due to the fact that the CoffeeBreak garbage collector must store an internal state that goes beyond of the few fields of the RDS allocator that can be embedded into domain descriptors: The free list of the garbage collector, the working stack for the marking phase and the bitmap for marking free heap areas must all be stored within the CoffeeBreak GC.

The structure of the static CoffeeBreak garbage collector as it was implemented for this thesis can be seen in Figure 3.1. The class JavaCoffeeBreak is the main class of the garbage collector. It contains two methods, one for allocating a new object and one for starting the actual garbage collection. These methods redirect to the actual implementation and serve as an interface to KESO.

Both tasks of the garbage collector – allocation and collection – have been implemented in their own classes: CoffeeBreakCollector or CoffeeBreakAllocator. All three of the mentioned garbage collector classes are static. As no new object allocations are allowed in the garbage collector, its classes also may not be instantiated.

At a quick glance, it may seem possible to store, for example, the bitmap of the garbage collector in class fields. However, class fields are stored within the domain descriptor in KESO. This way, class field values do not hurt the principle, that references may not be shared between domains: Each domain has its own set of class variables. Concerning the garbage collector that is running within the TCB of the Domain Zero, where no domain descriptor is available that could contain class

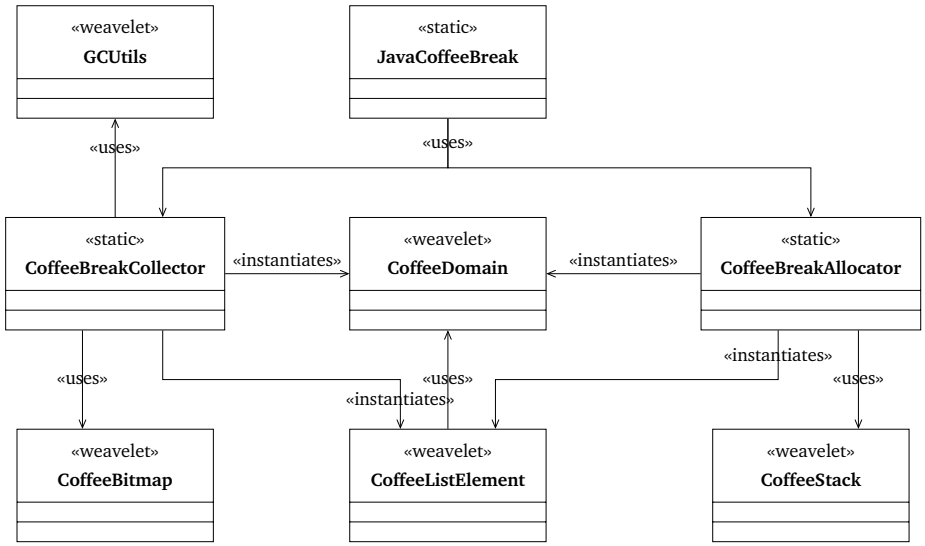


Figure 3.1 – Class structure of the static CoffeeBreak GC. Classes implemented using weavelets are marked with `«weavelet»`.

fields, it is impossible to use class variables. The Domain Zero has no actual domain descriptor that could contain class fields, as there was no need to provide a domain descriptor for the native C implementation of the domain's tasks.

This means that the garbage collector classes cannot manage state information needed during their execution. All this information must therefore be stored in memory accessed through weavelets. The usage of weavelets for storing the GC's state will be discussed in the next section.

3.3.2.2 Usage of Weavelets/KESO Native Interface (KNI)

As already suggested in Section 1.2.3, weavelets are a mechanism that enables JINO to substitute method bodies or even method calls in Java code with C code at compile time. This section shows that weavelets were used in the static CoffeeBreak implementation in two ways: Providing a way to store CoffeeBreak's internal data structures and accessing system properties such as domain descriptors and object headers.

The existing native implementation of the garbage collector stores its data primarily in a bitmap and in the free list that is embedded in the free memory chunks of the heap (see 2.2.2). The bitmap is a fixed size array declared in the CoffeeBreak source file.

The Java implementation needs to provide a similar bitmap, but it cannot declare such a data structure due to its memory usage restrictions – an array must not be created on the heap and it is too large in size to create it on a method stack. For this

reason, the actual data of the CoffeeBitmap class visible in figure 3.1 is stored in a native C array that is accessed via weavelets.

The CoffeeBitmap provides methods to set the status of contiguous blocks of slots to either free or used. The logic of mapping slots to entries in the bitmap was implemented purely in Java. Only a minimal set of methods that contain no algorithmic code and are restricted to performing access to the bitmap are implemented in C. For this purpose, KESO's weavelet mechanism was extended so that it can emit static C arrays into the header files generated for a weavelet class, a functionality that was not possible previously for weavelets.

The CoffeeDomain weavelet provides access to data structures stored in the domain descriptor and methods that allow for example manipulating an object's class identifier. Its functionality is similiar to the weavelet used in the RDS allocator in the previous section of this thesis.

Iterating the free memory list of the garbage collector was also implemented using weavelets. The already existing list element structure written in C was reused for the Java implementation. This allows to keep the benefit of managing the list elements within the free heap blocks. However, implementing the iteration over the list proved to be a challenge, as pointer manipulation is required.

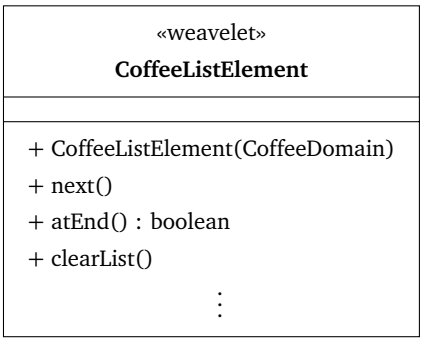


Figure 3.2 – CoffeeListElement as an iterator. Only methods necessary for iterating over the list are shown.

For this reason, the CoffeeListElement class was implemented following the pattern normally used for an iterator in Java. The methods of the class can be seen in Figure 3.2. Traversal of the list can be controlled via the methods `next()` and `atEnd()`. Per default, constructing a instance of a CoffeeListElement element sets its internal list element pointer to that of the current domain – that is why a CoffeeDomain object must be passed to the constructor. The CoffeeListElement class hides the internal "magic" occurring when traversing the free memory list. The pointer to the current list element is stored in a field in the object header. This field is injected in the weavelet translation process by JINO and is therefore not visible in the Java class – only the weavelet's C code can access it. Contrary to the

C implementation, the Java CoffeeBreak never has to perform any of the ongoing pointer manipulations and interacts with the CoffeeBreak list using a well-defined interface.

The `CoffeeDomain` and `CoffeeListElement` both store internal state. For that reason, they must actually be instantiated. The Java CoffeeBreak therefore makes use of KESO's stack allocation facility: Instances of the objects are created at the top of the call hierarchy and then passed down to the leaf nodes.

3.4 Dynamic Implementation

The previous section introduced both an RDS allocator and a CoffeeBreak garbage collector written in Java. Where the RDS allocator was fairly easy to port, as its complexity is limited, the CoffeeBreak Java implementation is very restricted: Much of the functionality is still implemented as weavelet and therefore actually written in C. This problem is mainly caused by the lack of the `new` operator in Java within the GC's code.

It is therefore desirable to implement the CoffeeBreak GC as a Java application that behaves more idiomatic for the programmer. Weavelets cannot be replaced for accessing KESO's system interface, but they should not be necessary for the internal data structures of the garbage collector.

This section introduces an implementation that removes the restriction to the object of allocations within the CoffeeBreak garbage collector at least for the mark-and-sweep phase. To do so, KESO's Domain Zero concept is first expanded: By introducing an actual Java Domain for trusted code base instead of treating the Domain Zero as a concept that only exists at compile time in JINO, static fields become usable. This *Java Domain Zero* will then be equipped with a heap using an RDS allocator.

New object instances of the CoffeeBreak GC can be created on this heap. Resetting this heap cyclically will allow the CoffeeBreak collector to allocate permanent as well as temporary Java objects. Because this implementation does not require the GC's data structures like the working stack or the bitmap to be defined statically in the binary of the system, it shall be called *dynamic* implementation.

3.4.1 Introducing a Java Domain Zero

Up to now, the Domain Zero in KESO's system architecture, as shown in Section 1.2.1, was a purely theoretical concept: No actual domain descriptor was emitted into the domain configuration file by JINO, as the garbage collector implementations written in C declared their own data structures in their source code and needed no domain descriptor to store them.

To implement a full-featured garbage collector in Java, the Domain Zero must support all features of a normal Java Domain: Class variables and a heap need to be available. However, compatibility with existing code that runs statically in a Native Domain must be kept.

For this reason, JINO's representation of the Domain Zero was remodeled to support a hybrid approach. The actual Domain Zero representation serves as a wrapper and contains both a Native Domain and a Java Domain. The implementation of the Domain Zero can be seen in figure 3.3.

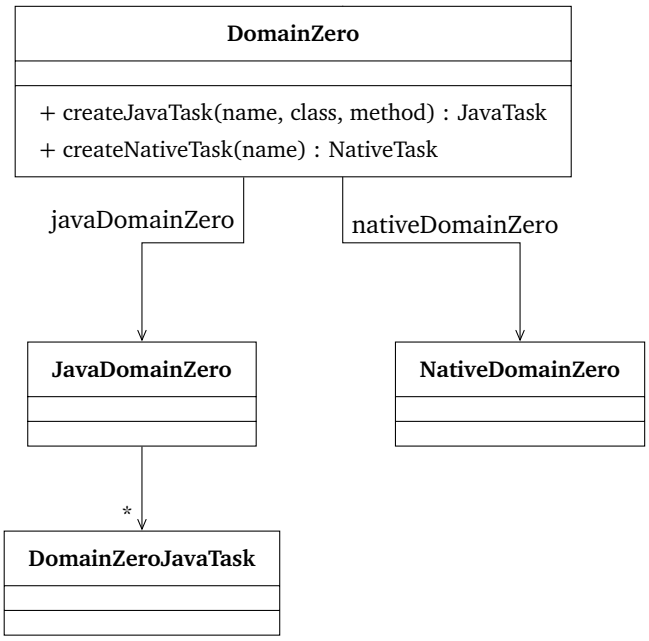


Figure 3.3 – The new Domain Zero implementation hides two actual domain structures – one for native tasks, one for Java tasks.

To add a task that should be running in the privileged TCB of the Domain Zero, one must use the `create...()` methods provided by the class `DomainZero`. This ensures that the domain descriptors will only be added to the system if the domain is actually needed. As a domain descriptor for Java domains is quite large (it contains a heap array), this saves space for system configurations that do not make use of system services in Java. Also, within JINO, the domain object for the Java Domain Zero must be treated differently from ordinary Java domains: Where normal Java domains are being added to the sets of application domains, system domains and Java domains, the Java Domain Zero must only be contained in the sets of system domains and Java domains. The latter step prevents JINO from performing reachability analysis on the GC tasks. Reachability analysis determines which system objects and code belong to which domain. As the garbage collector runs "outside" of the normal system, and

especially the allocation method is never explicitly called, it must be made sure that its code is not removed by JINO.

Having the GC task created by a factory method also enforces correct priority settings. The priority of the task must always be the lowest priority, so that it can only run when all other tasks are blocked (see Section 2.2.2).

One problem that arose during the creation of the `JavaDomainZero` was a faulty initialization of the domain's data structures. This is due to the fact that JINO emits a sequence of initialization operations for all domain descriptors in the order the domains were added to the system configuration. As the intermediate representation of the Java Domain Zero is added late in the system, it would be initialized after all other domains – these domains however require the CoffeeBreak GC to be set up to handle their allocation requests.

The problem was solved by treating the initialization of the Java Domain Zero as privileged and adding it before any other domain's initialization in the system.

3.4.2 Separation of Allocation from Collection

This section describes how the allocation method of the CoffeeBreak garbage collector does not need to create new object instances for the CoffeeBreak collector and therefore enables usage of the memory management model described in the next section. For increased clarity of writing, the term *allocation* shall refer to a memory request by the application that must be handled by the CoffeeBreak's allocator method and *new object instantiation* shall refer to an object that needs to be created within the code of the CoffeeBreak itself.

The implementation of the dynamic CoffeeBreak garbage collector is based upon the following observation: The method for allocation of new objects is executed in the context of the calling domain. Therefore, whenever a new object instantiation is performed within the allocation method of the CoffeeBreak GC, its new object will be created on the heap of the domain that is managed by the CoffeeBreak instance. This heap is managed by the CoffeeBreak, which means its allocation method would be called again – an infinite loop would occur. That happens, because KESO redirects allocations to the allocator method for the current domain in the macro `KESO_ALLOC`. Figure 3.4 illustrates this dilemma.

However, the allocation method does not need to perform its own allocations. The only data structure of the CoffeeBreak GC used in the method is the free memory list. If the free memory list is implemented purely in Java, its list elements shall reside in the heap assigned to the Domain Zero. The free memory list can then be used in the following two ways in the allocation method:

1. A memory block referenced by one element in the free memory list exactly fits the size needed for a memory request. The block is then removed from the

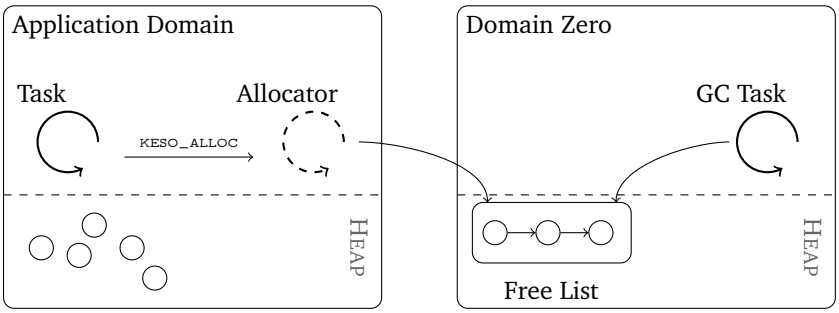


Figure 3.4 – Separation of call sites of the CoffeeBreak code: Allocation is executed in the context of the application domain, collection within the Domain Zero.

pool of available free memory blocks. The respective list element is discarded. No new object instantiation within the allocation method is needed.

2. A free memory block is large, and only a portion of it is used to fulfill a memory request by the application. The rest of the free memory block must be kept in the free memory list by enchaining a free list element referencing it. In this case, the free memory element originally referencing the large object may be reused by equipping it with the size and start address of the new, smaller memory block. No new object instantiation is needed.

Consequently, the allocation method can be implemented completely in a way so that no object instantiations are needed. The mark-and-sweep phase of the dynamic CoffeeBreak implementation however does need object instantiation. Its design, and how memory management within a Java Domain Zero can be managed, will be shown in the next section.

3.4.3 Architecture

The architecture of the dynamic CoffeeBreak garbage collector is similar of the one presented in Section 3.3.2 that does not have its own heap for object instantiations. However, there are several differences concerning weavelet usage and adaption to KESO's task model. The overall architecture can be seen in Figure 3.5.

In comparison with the previous implementation of the CoffeeBreak shown in Figure 3.1, the reduced number of weavelets is instantly visible. In my new CoffeeBreak implementation, all data structures that are restricted to the garbage collector were implemented in Java. The weavelets `GCUtils` and `MinimalCoffeeDomain` are therefore only needed to access domain descriptor properties and object headers.

The data structures that could be implemented in Java due to the newly created Java Domain Zero are visible at the bottom of the figure: A class `Stack` for the

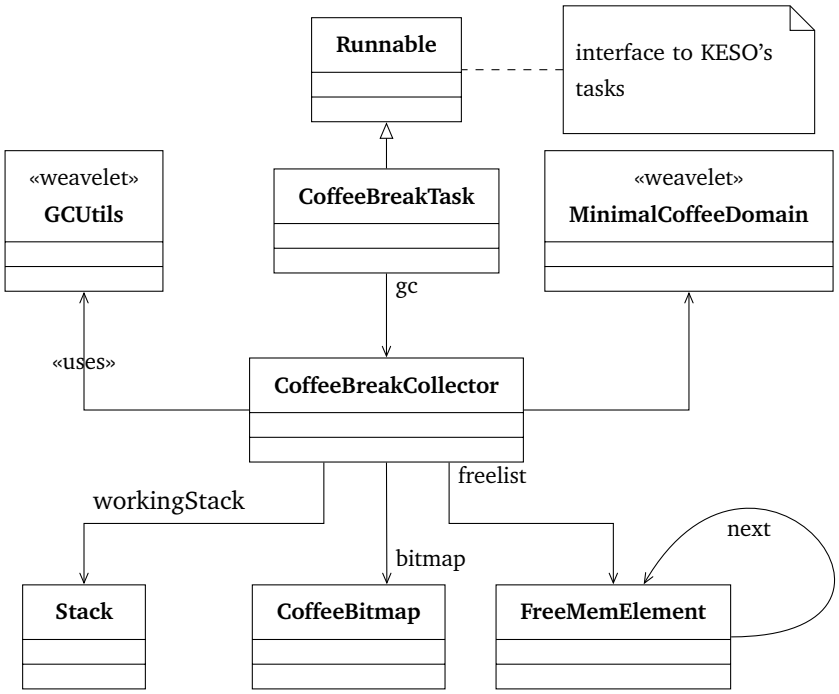


Figure 3.5 – Class structure of the dynamic CoffeeBreak GC. Classes implemented using weavelets are marked with «`weavelet`«

working stack used in the mark-and-sweep phase, a class `CoffeeBitmap` for the bitmap used to mark the status of memory slots and the `FreeMemElement` class that implements a list element of the free memory list. How exactly these objects can be kept in the CoffeeBreak's RDS allocator managed heap without exhausting memory after some time will be explained in the next section.

3.4.4 Cyclically Resetting the RDS Heap

The objects created by the CoffeeBreak GC in its run time can be divided into two groups. One group will be referred to as *base set* of the CoffeeBreak GC. It contains objects that must exist during the lifetime of the garbage collector, which is also the lifetime of the whole system. The other group refers to *short-lived objects* that do not need to survive the mark-and-sweep phase.

The base set of the CoffeeBreak GC in its current implementation contains an object of the class `MinimalCoffeeDomain`. The instance of the `CoffeeBitmap` class used in the mark-and-sweep phase is also stored here. Because the size of the latter is constant for the runtime of the garbage collector, it can be created once, but must then be reset at the start of the marking phase. This is done to avoid accidentally marking slots as still used that have been freed by the program in the meantime.

The set of short-lived objects consists mainly of the free memory list. The free memory list is not kept in the heap memory it describes any more (like in the C implementation and the static Java CoffeeBreak), but in the RDS-managed heap of the Java Domain Zero. This comes with a penalty on memory usage, but enables the free memory list to be implemented purely in Java.

The working stack that stores objects in the mark-and-sweep phase can also be placed in the RDS heap of the Java Domain Zero. Contrary to the C implementation, which used a fixed-size array and needed the CoffeeBreak GC to manually keep track of the index into the array, the class `Stack` offers a well-defined interface to the garbage collector. Internally, it manages an array that is dynamically increased when necessary.

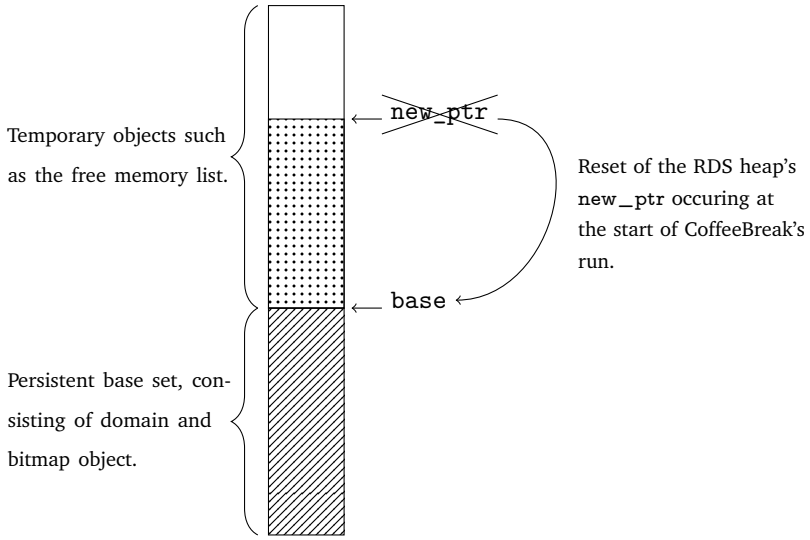


Figure 3.6 – Cyclically resetting the RDS heap of the Java Domain Zero.

Figure 3.6 shows the composition of the RDS heap of a Java Domain Zero used by a CoffeeBreak garbage collector. At the start of the heap are the persistent objects: The domain object and the bitmap will survive as long as the system is active. The middle part of the heap is taken up by temporary objects like the working stack or the free memory list. At the top of the heap, a free area of memory is still available for the creation of new objects within the garbage collector.

At the right site of the heap array, two pointers into the address space of the heap array are visible. The pointer `base` marks the end of the persistent memory area. The memory area above the `base` pointer is available for dynamic object creation. The `new_ptr` pointer is equivalent to the *new pointer* of the RDS heap – it marks the first empty memory address and new objects will be created here. Its target moves toward the top of the heap with each performed allocation on the heap.

Resetting the heap does not work automatically, but requires the programmer to identify and mark two critical points in the code:

1. The creation of the base set is finished in the CoffeeBreak's constructor. At this point, the current position of the RDS allocator must be stored. Later on, the new pointer of the RDS allocator will be reset to this memory address.

2. The reset of the new pointer must occur at the start of each CoffeeBreak run. As the CoffeeBreak collector runs in an endless loop, this is done start of the loop's body.

It must be made sure that no object references into the memory area that will be reused after the reset survive. For this purpose, it is advised not to declare temporary objects in the scope of the mark-and-sweep method before the RDS heap was reset, and to set references stored in class fields to `null`. The latter operation allows KESO to throw an error when an invalid reference, that was not assigned to a newly allocated object again, is accessed.

Resetting the RDS heap is currently implemented using the weavelet `RDSHeapResetter`. Its two methods support querying the base address of the RDS allocator as well as resetting the new pointer.

3.4.5 Configurability

KESO strives to be as configurable as possible. This allows to create a system that is tailored exactly to the limited hardware available in embedded systems. Within the CoffeeBreak's C implementation, conditional compilation is often used to en-

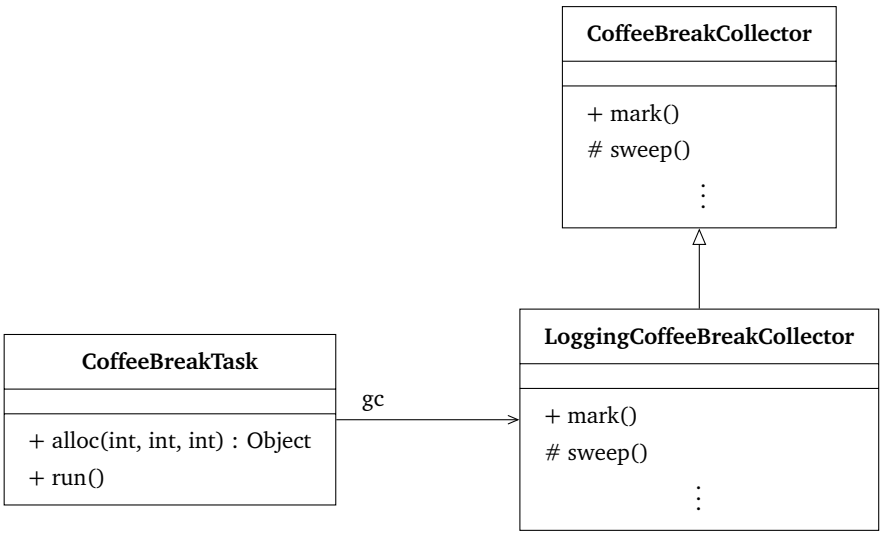


Figure 3.7 – Structure of the logging CoffeeBreak GC.

or disable features based on configuration options that were passed by JINO as compile-time constants. As stated in Section 2.3.2, Java does not allow conditional compilation.

One notable example for a feature which can be enabled or disabled is the support for tracing, that prints debug information to the standard output during execution of the garbage collector. To display the possibility of using object-oriented design instead of preprocessor features, a class `LoggingCoffeeBreakCollector` was implemented that also prints debug output.

The logging garbage collector was implemented following a proxy pattern [13]. The class `LoggingCoffeeBreakCollector` inherits from the class `CoffeeBreakCollector`. It wraps every method call of the super class in debug statements. If functional changes were to be made in the standard CoffeeBreak class, no additional changes would be needed in the sub-class of the logging GC.

By instantiating the correct instance of the collector object in the CoffeeBreak's task, the system can execute either the standard or the logging CoffeeBreak collector.

A benefit of this method, in comparison to the C implementation, is especially that there are no debug statements and preprocessor conditions cluttering the code that is needed for the GC's functionality.

3.4.6 Summary

In this section, the implementation of a CoffeeBreak garbage collector that is not restricted to storing its own data structures in weavelets was presented. To circumvent the restriction to stack allocation, a resettable RDS heap was added in a custom Java domain. An example also showed how this implementation of the garbage collector can be extended using object-oriented design instead of conditional compilation previously used in KESO.

Chapter 4

Evaluation

The implementation of the CoffeeBreak GC that was presented in the previous chapter is the final implementation of one of KESO's memory mechanisms in Java. This chapter is intended to evaluate the work that was done: In the first section, a critical discussion of the solutions that were found for the main challenges in implementing garbage collection in Java will be given. The following section will show the changes in the size of the final binary in comparison to the native C implementation of the memory management. In closing, the runtime of the system will be examined.

For the latter two steps, the benchmark CD_j was used. It implements a real-time collision detection for aircraft traffic. The program consists of two components: A generator that generates radar frames and the actual collision detector. In this thesis, the "on-the-go" variant where both components are located in one KESO domain will be used.

4.1 Evaluation of the Implementation and its Features

This section is intended to give an overview over the solved challenges in the implementation of the CoffeeBreak GC and the limitations that were encountered in this thesis.

4.1.1 Overview over Solved Challenges

The implementation of the CoffeeBreak garbage collector that can allocate objects on its own, *Restricted Domain Scope* (RDS)-managed, heap shows a solution to the two basic problems of implementing garbage collection in Java: Complications in handling new object instantiation and access to low-level system data structures. In addition, an attempt at making the resulting implementation configurable and extendable was made.

Two solutions to the *restrictions to object allocation* within the garbage collector's code have been found:

Stack allocation allows to allocate objects on the stack frame of the current method. The mechanism for allocating objects on the stack had already been implemented for KESO. In this thesis, I introduced a mechanism to ensure that only stack allocation is used in places where no objects may be allocated on the heap. Nonetheless, stack allocation is not suitable for all object creations, as objects created on the stack perish when their creating method is left.

Cyclic RDS heap resetting was created in this thesis to allow the instantiation oeps, the benchmark CD_j was used. It implements a real-time collision detection for aircraft traffic. The program consists of two components: A generator that generates radar frames and the actual collision detector. In this thesis, the "on-the-go" variant where both components are located in one KESO dof long- as well as short-lived objects from within the garbage collector. However, this solution requires additional work by the programmer to identify suitable points for resetting the heap in the source code.

Low-level system access has successfully been provided by weavelets. These serve as an adapter between KESO's C structures and the garbage collector's code. However, the implementation of the adapters requires each weavelet to be implemented manually. In addition, some of the weavelets need to store internal state information such as the progress in an iteration. For this purpose, the weavelet mechanism was expanded in the course of this thesis.

Configurability is a basic feature of the CoffeeBreak garbage collector. Contrary to the original C implementation, the Java implementation cannot rely on a preprocessor to conditionally include code when needed. Instead, object-oriented design patterns can be used, as was shown in this thesis with the implementation of the CoffeeBreak garbage collector with additional debug output.

4.1.2 Limitations

Weavelets for low-level system access and object-oriented patterns for configuration of the garbage collector show the limitations of implementing type-safe garbage collection in KESO: The related projects shown in Section 2.1 are runtime environments for Java (or C#) with garbage collectors written in Java (or C#). The runtime environments themselves are written in Java (or C#).

However, KESO is a Java runtime environment for Java with garbage collection written in Java that is itself written in C. This means that access to data structures such as object headers that may be available as a Java/C# object in related projects must always be implemented in weavelet-based adapters. This is for example the

case for advanced features such as the scanning of object references found on the linked stack frames of methods: Scanning of those object references is handled completely by the corresponding weavelets and cannot be done in Java.

In addition, configuring available features is difficult – in case of the CoffeeBreak GC with additional debug output, its implementation class must be selected manually in the configuration by the user. Previously, only required code for system features found in the analysis performed by JINO would be emitted and features in the templates for the C implementation of the CoffeeBreak GC could be included conditionally by the preprocessor. Due to the lack of a Java preprocessor and the handling of Java classes in JINO, this is not possible for the Java implementation.

Additionally, the resettable RDS heap is by far not an ideal solution. It does solve the problem of creating Java objects within the garbage collector's code. However, this comes at a high cost: The programmer must manually identify object sets that are long- and short-lived and must take special care to prevent access to objects that are not valid any more. Therefore, the resettable RDS heap regrettably hurts Java's principle of memory-safety.

4.1.3 Summary

In conclusion, it has been shown that implementing garbage collection in Java for KESO is feasible. Solutions for accessing low-level data structures and circumventing the restrictions on new object allocations have been found.

However, the lack of a preprocessor makes system programming difficult. In addition, the nature of KESO – the system is written in C – requires the frequent use of weavelets. This makes using more advanced features of the runtime environment, such as the linked stack frames, cumbersome for the programmer, as they constantly have to switch between Java and C to implement the actual algorithm of the garbage collector and its accompanying weavelets.

4.2 Size of the Application

The KESO system containing the CD_x was configured to use a heap size of 800 kB. The complete configuration for the benchmarks is shown in Table 4.1. Table 4.2 shows the changes in size for the various configurations of the garbage collectors.

The total overhead for the *weavelet-based implementation* of the CoffeeBreak garbage collector adds up to 0.36 % in comparison to the system using the existing C implementation of the garbage collector.

The majority of the increase is caused by the larger text section and the bigger data section. The text section size difference is can be explained by additional functions in the implementation due to the usage of weavelets.

	Configuration
CPU	Infinion TriCore TC1796
CiAO	git revision `14defd2b`
KESO	SVN revision 4303
TriGCC compiler	version 4.6.4

Table 4.1 – Configuration used for running the CD_x benchmark.

The data section of the binary is 2.63 % bigger than its native counterpart. Its increase insize adds up to as little as 52 Byte. Its increase in size can be explained by a minimally longer string constant for the "out of memory" error message given by the GC (the GC name was prepended) and the increased size of the class store of the system.

The small overhead of the bss section can mainly be traced back to the fact that the fixed-size array of the GC's working stack is bigger in the weavelet-based CoffeeBreak implementation as it is not configurable by the user in the configuration file.

The overhead of the size of the binary configured to use the *RDS-heap-based implementation* of the CoffeeBreak is 3.84 % compared to the CoffeeBreak's C implementation.

The text section is 9.05 % bigger as it contains more Java code: Here, the Java code must also implement features not previously found in the CoffeeBreak GC, such as dynamically increasing the size of the working stack of the CoffeeBreak. This added complexity is also represented by the increase of the size of the data section.

The absolute increase of the bss section adds up to more than 30 kByte. This is not surprising, as the additional heap of the Java Domain Zero has a size of 40 kByte. The discrepancy between those two numbers can be explained by the fact that the working stack and the bitmap are both allocated on said heap at runtime.

All in all, both Java implementations come with a measurable increase of the size of the binary.

4.3 Runtime of the Application

To measure the runtime of the application, the configuration of the CD_x collision detector was set to calculate collisions in 50 radar frames. The time used for each frame is measured by the program. The percentual overhead for each frame was then calculated for both the weavelet-based CoffeeBreak as well as the CoffeeBreak that is implemented purely in Java. The results can be seen in Figure 4.1.

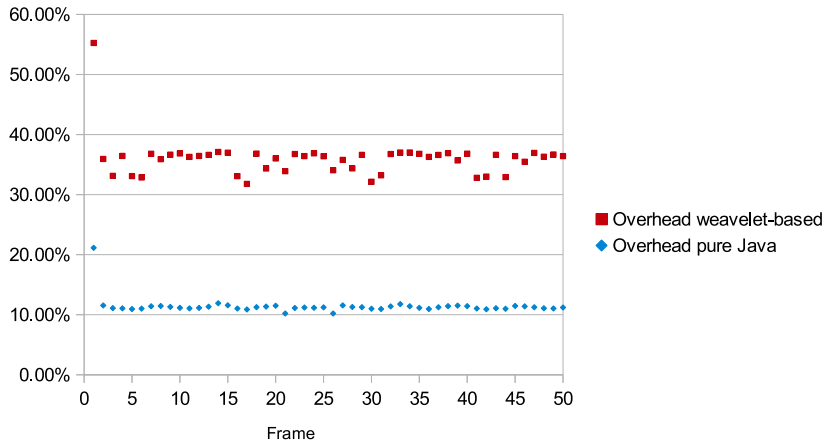


Figure 4.1 – Percentual runtime overhead for each frame of the CD_x application.

The results show that the weavelet-based implementation of the CoffeeBreak garbage collector has a runtime for each frame that is on average 36 % higher than that of the C implementation. The implementation using a Java-based approach for the GC's own data structures is significantly faster, with an average overhead of only 11 % compared to the native C implementation of the CoffeeBreak.

The spike in the overhead for the first frame is caused by the fact that the time taken for its calculation is much lower than that of the other frames. Deviations in execution time for this first frame therefore show much more prominently than those of other frames.

Consequently, the wide-spread use of weavelets for CoffeeBreak can only be discouraged: It comes with a huge penalty on performance. The additional complexity incurred by adding a Java Domain Zero however is well justified: It more than halves the time overhead incurred by implementing the GC in Java.

	text		data		bss		total	
	size/Byte	increase/%	size/Bytes	increase/%	size/Bytes	increase/%	size/Bytes	increase/%
CD_x (standard CoffeeBreak)	42 476	–	1977	–	992 882	–	1 037 335	–
CD_x (weavelet CoffeeBreak)	44 040	3.68	2029	2.63	994 996	0.21	1 041 035	0.36
CD_x (RDS-heap CoffeeBreak)	46 320	9.05	2349	18.82	1 028 502	3.59	1 077 171	3.84

Table 4.2 – Size of systems with different garbage collectors, grouped by the segment of the binary. The increase in size is given in percent compared to the binary using the standard implementation of the CoffeeBreak GC.

Chapter 5

Conclusion

This thesis evaluated the possibility of implementing KESO's garbage collection services in the type-safe language Java instead of C. To do so, changes were not only made to the implementation of the actual garbage collectors but they were also needed in all three stages of KESO's compiler, JINO.

In a first step, the task was analyzed: Related work of garbage collection in type-safe languages was reviewed and common problems found in the Jalapeño/Jikes RVM, Singularity OS and Squawk VM projects were identified. Among them are especially the need to provide *low-level access* to memory in Java and to restrict Java's *object allocation*. Additional problems stem from language differences between Java and C.

JINO's previously existing *stack allocation* was reused. An additional mechanism in the compiler was implemented to restrict the garbage collector implementations in Java that were then implemented to stack allocation. In this process, KESO's *memory-mapped objects* were discarded as a method to provide low-level system access, as they showed to be too inflexible. KESO's *weavelets* were used instead.

This implementation heavily relied on weavelets, not only for accessing system properties but also for storing garbage collection data structures. As this was deemed unsatisfactory, a new approach was developed that allows the garbage collector to use an allocation-only heap that is *reset* at pre-defined points in the algorithm. Additionally, a way of expanding the garbage collector's code using object-oriented patterns instead of conditional compilation was shown.

In conclusion, the initial problems identified in the analysis could be circumvented, and it was shown that writing garbage collection in Java for KESO is certainly possible. However, the development proved to be cumbersome due to KESO's nature: As KESO's runtime system is written in C (contrary to related systems), weavelets have to be written for virtually any system interaction. In addtion, KESO's fine-grained configurability that relies both on procedural generation of program code by

JINO, as well as conditional compilation, complicates propagation of configuration options to the Java code of the garbage collector.

Future Work

Resetting the RDS heap in the final, dynamic implementation of the CoffeeBreak garbage collector has two drawbacks: The programmer must make sure that no object references into the re-usable area of the heap survive. They must also manually determine the points in the algorithm where the base set has been allocated and where resetting the heap is appropriate.

To improve the safety of the resettable RDS heap, it would be interesting to evaluate whether additional compiler passes could automatically identify the point in code where the reset should occur. The base set could be identified by data flow analysis. JINO could then insert the necessary instructions automatically.

Another area that shows potential for improvement is the propagation of KESO's configuration to the Java implementation of the garbage collectors. As shown in Section 3.4.5, it is possible to re-create configuration options present in the original CoffeeBreak collector. However, the class that implements the desired option has to be selected manually by the user.

To facilitate option passing and be able to access constants read from KESO's configuration file or determined during the compilation process, a preprocessor could be included. Other projects, such as the Squawk VM presented in Section 2.1.3, already have a working preprocessor for Java. It would be interesting to evaluate if this preprocessor could be used for KESO.

List of Acronyms

GC garbage collector

JVM Java Virtual Machine

KESO “Konstruktiver Speicherschutz für eingebettete Systeme“ – German for ”Constructive memory protection for embedded systems“

RDS Restricted Domain Scope

RE runtime environment

TCB Trusted Code Base

List of Figures

List of Tables

Bibliography

[1] C. Lang, "Improved stack allocation using escape analysis in the KESO multi-JVM," *bachelor's thesis, Friedrich-Alexander Universität Erlangen-Nürnberg*, 2012.

[2] C. W. A. Wawersich, "Keso: Konstruktiver Speicherschutz für eingebettete Systeme." Ph.D. dissertation, University of Erlangen-Nuremberg, 2009.

[3] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen, "Implementing Jalapeño in Java," *ACM SIGPLAN Notices*, vol. 34, no. 10, pp. 314–324, 1999.

[4] D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev, "Demystifying magic: high-level low-level programming," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2009, pp. 81–90.

[5] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Oil and water? High performance garbage collection in Java with MMTk," in *ICSE 2004, 26th International Conference on Software Engineering, Edinburgh, Scotland, May 23-28, 2004*. IEEE, May 2004.

[6] D. Simon and C. Cifuentes, "The squawk virtual machine: Java™ on the bare metal," in *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 2005, pp. 150–151.

[7] Sun Microsystems Inc., "Squawk VM source code," https://svn.java.net/svn/squawk~svn, 2004 – 2013, revision 973.

[8] F. Henderson, "Accurate garbage collection in an uncooperative environment," in *ACM SIGPLAN Notices*, vol. 38, no. 2 supplement. ACM, 2002, pp. 150–156.

[9] B. W. Kernighan, D. M. Ritchie, and P. Ejeklint, *The C programming language*. Prentice-Hall Englewood Cliffs, 1988, vol. 2.

[10] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus, "Deconstructing process isolation," in *Proceedings of the 2006 workshop on Memory system performance and correctness*. ACM, 2006, pp. 1–10.

[11] A. Tim/Yellin Lindholm (Frank/Bracha, Gilad/Buckley, *The Java Virtual Machine Specification: Java SE 8 Edition*. Prentice Hall, 2014.

[12] M. Stilkerich, I. Thomm, C. Wawersich, and W. Schröder-Preikschat, "Tailor-made JVMs for statically configured embedded systems," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 8, pp. 789–812, 2012.

[13] E. Gamma, R. Helm, C. Larman, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, ser. Addison-Wesley Professional Computing Series. Pearson Education, Limited, 2005.

[14] C. Erhardt and M. Stilkerich, "A Control-Flow-Sensitive Analysis and Optimization Framework for the KESO Multi-JVM," 2011.