RT-LAGC: Fragmentation-Tolerant Real-Time Memory Management Revisited

Isabella Stilkerich Michael Strotz Christoph Erhardt Michael Stilkerich
{istilkerich, strotz, erhardt, stilkerich}@cs.fau.de
Friedrich-Alexander University Erlangen-Nuremberg, Germany

ABSTRACT

The use of managed, type-safe languages such as Java in real-time and embedded systems is advantageous, as it offers productivity and especially safety and dependability benefits over dominating unsafe languages. A Java Virtual Machine (JVM) has to provide an implicit memory management system such as a garbage collector (GC), for example, as explicit memory management through allocation and release operations by the application developer is prone to programming errors and may result in a violation of the type system properties. Real-time systems have specific requirements regarding space and time bounds and a GC has to ensure that these defined upper limits will not be exceeded. A proper solution to address this issue is, for example, employing fragmentation-tolerant garbage collection as proposed by Pizlo et al. [16]. Their approach is called *SCHISM/CMR*. Based on their work, we developed an alternative fragmentation-tolerant GC variant called *RT-LAGC*, which is supported by our compiler *jino* and is part of the KESO JVM [18]. RT-LAGC is a *cooperative* GC, that is, the real-time system developer and the compiler assist the GC through system configuration (e.g. enough slack time for the GC to run) and program analyses, respectively. This is achieved by integrating the GCs in the design process of the whole system just as any other user application. In RT-LAGC, we designed a new bidirectional fragmented object layout. Furthermore, we implemented latency-aware management of fragmented memory as well as an alternative collection technique for array meta-information. Moreover, the execution properties of an exemplary application were improved by *jino*'s *extended escape analysis.* RT-LAGC is evaluated against KESO's purely incremental non-fragmentation-tolerant GC called *IRRGC* and a throughput-optimized stop-the-world collector named *CBGC*. A classification of typical memory patterns for Java objects supports the predictability of the examined embedded system.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
JTRES '14, October 13–14 2014, Niagara Falls, NY, USA
Copyright 2014 ACM 978-1-4503-2813-5/14/10 ...$15.00.
http://dx.doi.org/10.1145/2661020.2661031

Categories and Subject Descriptors

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

General Terms

Reliability, Design, Languages

Keywords

KESO, Java, garbage collection, memory management, embedded systems, real-time systems

1. INTRODUCTION

Java is a rather uncommon language in (deeply) embedded systems, though it provides a series of advantages such as memory safety [1]. As a type-safe programming language, Java also provides the foundation for comprehensive program analyses and runtime system support, which can be very useful in embedded and real-time systems. Memory management is an inherent part of such a runtime system. Dynamic memory management can be performed in various ways such as, for example, automated stack allocation and regional (heap) memory [9] by means of the compiler's escape analysis [5] as well as garbage collection techniques for heap management. Compiler-assisted regional and stack memories can be managed by low-cost operations, whose time-predictability can easily be determined. Also, those techniques are inherently fragmentation-tolerant, since memory areas are assigned to the respective control flows and those areas are cleared upon leaving certain parts of the program. However, some real-time applications exist, in which the lifetime of objects cannot statically be determined by the compiler. Moreover, the use of a garbage collector running during the system's slack time can provide a better throughput of the application execution, since the application code itself does not have to perform the memory release operations. The application developers have to make sure, for example, that there is enough slack time for the GC to run. The amount of slack time available can be determined during system design and by schedulability analysis [14]. Real-time garbage collection has been addressed in several projects. Schism [16] implements fragmentation-tolerant garbage collection on top of a concurrent mark-region collector. This technique harnesses fragmented allocation and concurrent replication. We revisited this ap-

proach by combining Schism and latency-aware garbage collection. This garbage collector is called RT-LAGC. The paper is organized as follows: Section 2 presents the KESO JVM in which RT-LAGC was implemented, as well as the Schism RTGC. Java objects used in applications are discussed and categorized in Section 3. Such a classification is useful to support the design of the embedded real-time application. Section 4 presents RT-LAGC, its modifications to Schism and Section 5 evaluates RT-LAGC by contrasting a throughput-optimized GC, a purely incremental GC and the fragmentation-tolerant RT-LAGC. The conclusion can be found in Section 6.

2. KESO AND SCHISM: AN OVERVIEW

In this section, we describe the characteristics of the KESO JVM that are relevant for the implementation of RT-LAGC and the basic approach of SCHISM/CMR, which has been adopted in RT-LAGC.

The KESO JVM.

KESO [18] is designed to be deployed in statically configured embedded systems. In such systems, all relevant entities of the application as well as the system software are known ahead-of-time. This type of application covers many, if not most, traditional embedded applications from the electronic control units found in appliances to safety-critical tasks such as the electronic stability program (ESP) and many other electronic functions found in modern cars. The important entities contain the entire type-safe source code of the application and operating-system objects such as threads (called tasks in AUTOSAR OS [2]), for example. Thus, it is not possible to dynamically load new code or create threads at runtime, which is why KESO is a *static JVM*. This scheme allows KESO's compiler *jino* to create a slim and efficient runtime system for Java applications in embedded systems. Applications can be isolated from each other by embedding them in so-called protection *domains*. Spatial isolation is constructively ensured by the type-safe programming language and strict logical separation of all global data (e.g. heap, static class fields). The RPC mechanism for communication ensures that object references are not propagated between domains. The runtime system provides control-flow abstractions such as threads and interrupt service routines (ISRs) and their respective activation and synchronisation mechanisms such as alarms and locks (called resources). KESO's ahead-of-time compiler *jino* generates ANSI C code from the application's Java bytecode. While most of the code directly translates to plain C code, the Java thread API is mapped onto the thread abstraction layer of an underlying OS. In the case of used JVM, that abstraction layer is normally provided by AUTOSAR OS, but the KESO approach can also be transferred to any other static OS. KESO's architecture of isolated domains implicates the strict separation of heaps and static fields, which leads to disjoint object graphs in the domains. This allows the memory management strategy to be chosen and performed individually for each domain.

SCHISM/CMR.

Schism [16] is set on top of a concurrent mark-region (CMR) GC. It applies fragmented allocation to objects and arrays if necessary: Objects are allocated in fixed-size non-moving fragments and larger objects span a set of possibly non-contiguous fragments. Array meta-data is handled by a replicated semi-space. Schism achieves heap operations to be of constant-time complexity by using *arraylets* that were previously employed in the Metronome GC [3]: Arrays consist of a contiguous meta-array called *spine*, which holds array meta-information (i.e. pointers to the array fragments carrying the actual array data). The Schism array layout is discussed in more detail in Section 4.1.2. Contiguous array allocation is possible, which is beneficial for throughput aspects. In a contiguous array, an element is addressed by

```
arrayBase + index * elementSize
```

Using arraylets, the addressing of an element is more complex:

```
offset = index * elementSize
fragmentIndex = offset / fragmentSize
fragmentOffset = offset % fragmentSize
address = spine[fragmentIndex] + fragmentOffset
```

Objects are represented as a linked list of fragments as proposed by the JamaicaVM [17]; object field access is easily predictable. In Schism, external fragmentation is no issue and fragmentation is bounded due to small fixed-sized non-moving fragments. Fragments are managed by a CMR GC. Spines may dynamically adjust their size, thus causing fragmentation, which is why they are handled by a replicating semi-space GC. Since spine pointers refer to non-moving fragments, their information does not need to be updated during the GC execution. Wait-free barriers for spine access can be used. Thus, spines can be copied concurrently without influencing the application's performance.

For RT-LAGC, the Schism approach is combined with a concurrent, incremental and *latency-aware* mark-and-sweep GC algorithm we implemented in KESO. Besides a replicating semi-space collector, a generational collector for spines has been added. A new fragmented object layout to efficiently discover references has been designed. The RT-LAGC can be assisted by *jino*'s extended escape analysis if specified.

3. MEMORY PATTERNS FOR OBJECTS

The memory usage of typical Java programs shows specific traits which should be respected during the design of a real-time system since information about those usage patterns can be beneficial. Such application knowledge comprises the size of an object (Section 3.1) as well as its *survivability* (Section 3.2). Objects can be categorized according to these characteristics. The categories support the prediction of the worst-case execution time (WCET) for memory management, the memory allocation and replenishment rates. The evaluation of a test application in Sections 5.2 and 5.3 analyses the program according to this classification.

3.1 Object Sizes

In Java, a memory request can clearly be assigned to either a regular object or an array. Arrays allow random access to their contents by index, while the position of an object field is known at compile-time. Usually, Java objects are relatively small. In contrast to the C++ language, for example, Java usually does not store instances by value but

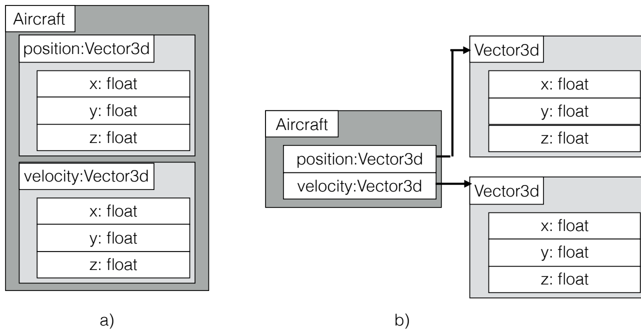


Figure 1: Value fields (a) and reference fields (b)

rather by reference. Due to this, Java objects are comparatively smaller than arrays. Figure 1 illustrates why C/C++ objects are often larger than Java objects: (a) shows objects (`Vector3d position, Vector3d velocity`) embedded in `Aircraft`. This object placement can be enforced manually by the C/C++ developer, whereas in Java no such means exists; (b) depicts the normal Java approach, where `position` and `velocity` are put into two separate objects that are referenced by `Aircraft`. However, it is possible for a Java compiler to automatically determine if an object can be *inlined* [13] into another object, that means the same constellation (a) is possible for Java objects as well. This can have a positive influence on the runtime and memory consumption, as indirections induced by reference fields are resolved and administrative data can be omitted. The existence of the object inlining facility should be respected during the design of memory management for real-time applications.

3.2 Survivability of Objects

The overhead imposed by garbage collection significantly depends on the number of surviving objects. An examination which objects of a real-time application will definitely not survive a GC execution helps to both improve upper space and time bounds and the estimation of those. The objects are put into categorization classes with respect to their *survivability*, that is, if they are able or unable to survive a GC run[1]. As *jino* performs whole-program analyses on type-safe code, those categories and the objects belonging to them can automatically be determined: On the one hand, objects may completely be extracted from garbage collection according to their categorization. On the other hand, due to the liveliness criterion and the knowledge of the points in time the GC is scheduled, the objects' survivability can be derived.

3.2.1 Method-Local Objects

Method-local objects are existent for the duration of the method they belong to and references to them do not leave the method scope. The information collected by alias analysis and the computation of the references' reachability is used by *jino* to automatically determine if an object *escapes* a method [6], i.e. if the object *has* to be allocated on the heap memory or if it can be stack-allocated. Applying escape analysis results in a series of benefits for the real-time system:

- Deallocation and allocation are performed by moving the stack pointer. These are low-cost and time-predictable operations in a CPU register.
- Method-local objects not residing in a potentially blocking method do not survive a GC run, which improves the upper space and time bounds.
- Only the stacks of potentially blocking tasks need to be scanned by the GC and this reference search is of complexity linear to the stack's depth. The stack depth is known at compile time by computing the call level of the *system call* `WaitEvent()`. However, blocking tasks must never be invoked in unbounded recursion.

Based on the escape state of an object, we developed the *extended escape analysis (EEA)* for extended stack scopes (ESS): Some method-escaping objects that are allocated in a method and returned afterwards can be allocated in the callers' stack frames (and passed to the original method as parameters) to further reduce heap objects. The expansion of the extended stack scope can be configured by the developer of the real-time application.

3.2.2 Region-Local Objects

Region-local objects escape the method they were created in, but they are not accessible by other threads. These objects can be allocated using region-based memory management. The safe C dialect Cyclone [9], for example, offers a region-based type system. Also, the RTSJ [4] specifies manual `ScopedMemory`. Regions are similar to method frames and co-exist with them. A region may span multiple method frames and in turn, a method frame may contain multiple regions. Region-local objects can have references into their own and into surrounding regions; however, none of their references must refer to an object of a subregion. Like method-local objects, region-local objects in a blocking method can survive a GC execution. Extended escape analysis can be used to fully automatically determine and handle memory regions: Besides potential stack allocation, objects eligible for ESS can still be heap-allocated[2], but can be managed on separate bump-pointer heaps which are cleared by resetting the bump pointer whenever the upper method scope(s) is/are left. We further categorize region-local results as follows:

- Region-local objects of fixed size
- Region-local objects of variable size

Depending on the configured stack scope expansion, fixed-size objects can be allocated on the stack or in a separate region, while variable-size objects are put into a separate region in the current state.

3.2.3 Thread-Local Objects

Thread-local objects cannot be assigned to regions or extended stack scopes, but they are never accessed by another

[1]A GC run or GC execution is one instance of slack time used for garbage collection.

[2]There are situations where extended stack allocation is not advantageous, e.g. for some virtual method call constructs: If the signature of a virtual method is changed due to ESS, the signature of all other methods that potentially share the same call site must be changed as well.

thread than the thread they were created by. They have to be handled by a heap management strategy. In case the respective control flow does not block, thread-local objects do not survive a GC run. The average runtime and WCET of the application in general can also be reduced if an object can definitely be determined to be thread-local by the compiler. Synchronization of the mutator threads of the application(s) is not needed anymore, which also improves latencies and blocking times of higher-priority threads.

3.2.4 Thread-Escaping Objects

These objects can be accessed by multiple control flows and further be categorized into

- Consumable objects
- Non-consumable objects
- Chronicle objects

Consumable objects can be released as soon as they are consumed. In case the thread blocks between creation and consumption time, the objects die in the next GC execution. Otherwise, the consumable objects survive. A use case is, for example, a higher-priority thread sending work assignments to a lower-priority thread. For the definition of proper upper space and time bounds, the minimum inter-arrival time [14] of consumable objects should be determined during system design, as the lower-priority thread has to have enough time to process incoming objects. The number of all possibly surviving objects can be estimated in the same way. **Non-consumable** objects are rarely or never released and fixed-sized objects of their kind can be allocated statically. One simple use case for this category is a shared-memory object expressing a state. Revisiting the above example, the work assignments could be put as reusable objects into an non-consumable queue. **Chronicle** objects allow the application to store information on the past. As these objects survive several GC runs, an upper bound of such objects has to be determined and they might also be allocated in a specific heap part to encourage contiguous allocation. As an example, automotive real-time applications employ AUTOSAR diagnostic system modules implementing the UDS (unified diagnostic services) protocol according to ISO 14229 by using chronicle memory. Also, the CD_x application presented later uses chronicle objects by storing byte arrays in a bounded buffer for diagnostic purposes.

3.2.5 Uncategorized Objects

By means of the aforementioned categories, it should be possible to construct type-safe real-time applications. An upper bound to the number of objects surviving GC runs is possible. The survivability of other uncategorized objects should be predictable. Moreover, undead objects – objects which are referenced but not used by the application – should be avoided. They are created when references to unneeded objects are not reset to *null*.

4. A LATENCY-AWARE REAL-TIME GC

This section describes the adoptions made from Schism, how the ideas are integrated into KESO, and the new features of RT-LAGC.

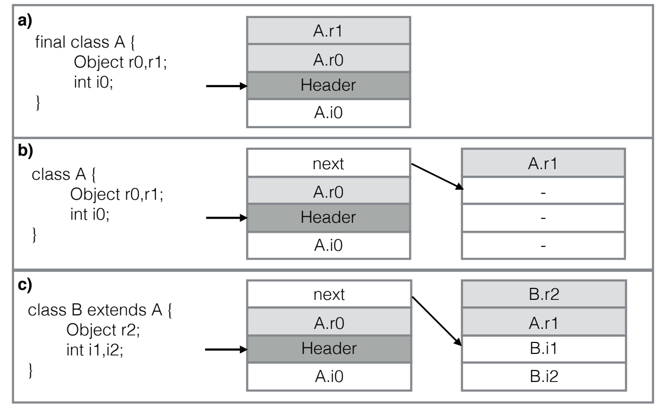


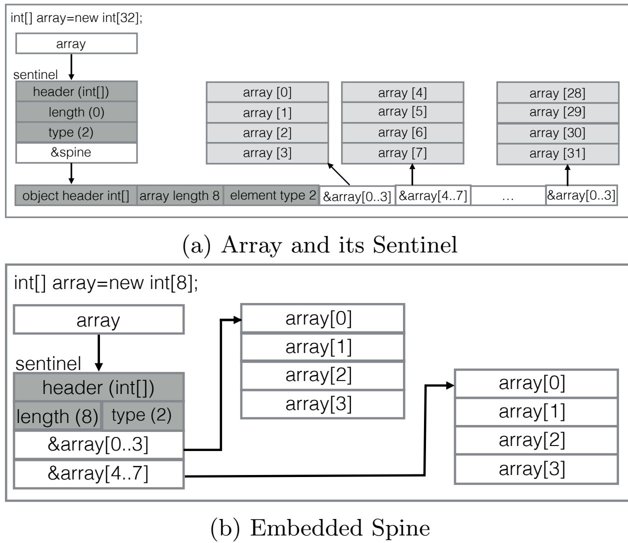
Figure 2: Fragmented bidirectional object layout

4.1 Object Layout

The object layout describes the arrangement of objects in memory, that is, the position and sequence in which their instance fields are stored. Instance fields can be subdivided into primitive-value fields and reference fields pointing to other objects. In KESO, each non-inlined object contains an object header that holds information such as the class ID or a color bit used by the GC for marking objects. Arrays additionally hold information about their length.

4.1.1 Fragmented Bidirectional Object Layout

The fragmented allocation of objects results in a redesign of KESO's current object layout which we call *fragmented bidirectional object layout* (FBOL). FBOL is based on the approach proposed by SableVM [8]. Figure 2 shows exemplary Java classes and the resulting fragmented objects assuming a 16-byte fragment size on a 32-bit host. The first lineup (a) illustrates an object of class A with no fragmentation since it fits entirely into a single fragment. All reference fields are located above the object header, while primitive fields are put below. As class A is marked `final`, no subclasses are possible. Since KESO is a static JVM, the `final` property of the class can also be implicitly determined by the compiler. In the second illustration (b), subclasses can be derived from class A, which necessitates another fragment and a `next` pointer referring to it. It should be noted that `next` does not point to the beginning of the fragment, but rather to the last reference field in that fragment. This is the location of the object header in case of an non-fragmented object. The last lineup (c) shows a subclass B, whose reference fields are inserted into the fragment from above and the primitive ones from below, with `next` pointing to the location behind the last reference. Since B has no further subclasses, no third fragment is needed. The access to an instance field is of constant complexity: The access to the A.r0 field resembles the non-fragmented bidirectional layout, while field B.r2 is located in the second fragment. To get B.r2, one indirection through the `next` pointer has to be taken and afterwards two address slots have to be subtracted. Since regular Java objects usually have a small size, they do not occupy more than a couple of fragments, so the overhead imposed by indirections is acceptable. By means of compile-time heuristics or runtime profiling, hot fields can be placed into fast front fragments for longer chains to optimize the throughput. However, some compiler optimizations



(a) Array and its Sentinel

(b) Embedded Spine

Figure 3: Fragmented Spine Arrays

may have a negative influence on RT-LAGC and the effects have to be kept in mind: Object inlining inflates object sizes but reduces reference accesses during the application execution. For such large objects, one possible solution is the use of spines instead of a linked list for fragments to avoid long chains. Another possibility is to host a separate heap with an enlarged fragment size that holds the pointers to the remaining fragments *in-place* in the first memory chunk. The tradeoff between the usual linked-list version using sparse chain pointers, in-place fragment pointers and spines has yet to be determined.

4.1.2 Array Layout

The array layout using spines was adopted from Schism. As mentioned before, arrays are managed by spines. To avoid updating all references referring to a spine in case of moving a spine, an unmovable sentinel element which points to the spine is allocated in the fragment memory. Moving a spine entails an update to the sentinel fragment only. Figure 3a shows a fragmented vector. Both the sentinel fragment and the spine contain a length attribute as it is still possible to allocate contiguous arrays. A zero-length attribute in the sentinel indicates a fragmented vector and a non-zero value a contiguous array, respectively. Statically allocated fixed-size arrays can be stored contiguously and the spine can be omitted. This procedure saves spine space and speeds up array accesses. Execution time and saved space degrade with an increasing number of fragmented arrays. Alternatively, contiguous arrays may not be supported at all by using *embedded spines* as illustrated in Figure 3b. In case of using this array layout, all arrays have a spine. The choice of embedding a spine depends on the array's size and saves spine space.

4.2 Latency-Aware Fragment Memory

Fragments are handled by a latency-aware collection algorithm called LAGC. LAGC is a precise, non-moving mark-and-sweep GC. It proceeds incrementally, i.e. it can be interrupted at any time except for short and predictable critical sections as well as stack scanning. In order to ensure the consistency of memory management data structures, critical sections are protected by write barriers. Critical sections are short and of constant complexity to keep the worst-case reaction time to external events as low as possible, which is why we name this approach *latency-aware*. RT-LAGC is handled by a dedicated control flow called `GCTask`, which is assigned the lowest priority in the system so that the task is scheduled at slack time, that is, other tasks are either inactive or blocked. This is a good moment to perform the `GCTask`, as most tasks will be suspended[3] and only the stacks of blocked tasks need to be scanned. A GC run is performed in the two typical phases of mark-and-sweep GCs. In the scan-and-mark phase, the live set of objects is determined by scanning all the reference values present in the application, and the parts of the heap occupied by these live objects are marked using the traditional tricolor scheme [7]. In the beginning of the scan phase, all objects are white. When the GC discovers an object reference reachable from the root set of the application, the memory occupied by the object is marked as being used and the object becomes gray. After having scanned all references within the object (and having colored all referenced objects gray), the object becomes black. Upon completion of the scan-and-mark phase, all objects on the heap are either white or black. In the subsequent sweep phase, the memory of all still-white objects is reclaimed.

Reference Scanning and Fragment Size.

The LAGC is able to easily discover references: The root set is defined by static reference fields and local references on the stacks of blocked tasks. Static reference fields of a protection domain are managed by an array and local references in stack frames are handled by linked stack frames [10]. FBOL is used to physically group references of fragmented objects. To compute the beginning of the object from a reference pointing to that object, the number of fragments and references has to be known. The fragment count is stored in the runtime type information table. It would be possible to also put the reference count of each fragment there, but this would result in a higher memory consumption. Instead, the start address of fragment memory is aligned to the fragment size to be divisible by the fragment size. Provided the fragment size is a power of two, the start address of a fragment referred to by the `next` pointer can be easily computed: The least-significant bits of the pointer are filled with zeros according to the fragment size. The fragment size should be chosen carefully: Too large fragments will result in large internal cutoff, whereas too small fragments will inflate external fragmentation and cause increased overhead for accesses. Thus, the average object size for the individual real-time application should be determined e.g. by profiling.

Write Barriers and Atomic Scanning of Task Stacks.

The latency-aware GC uses Yuasa write barriers [19], which cause an object to be colored gray whenever a reference to it is overwritten. This approach keeps up the last discoverable path to a living white object. Write barriers are generated by our compiler for static references, reference fields in objects and field writes in an array of object references only, since they do add noticeable overhead to write operations.

[3]Consequently, only few objects will survive the GC execution.

As write barriers are not active on local variables, the task stacks have to be scanned atomically. To allow for a precise detection of references, linked stack frames are used for all methods which do not have run-to-completion semantics – that is, methods during whose lifetime the GC task may run because they perform a blocking operation. This property can be determined at compile time. The stacks of blocked tasks (being in the *waiting* state) need to be scanned only, as other tasks in the *suspended* state have empty stacks, since garbage collection is performed at slack time. Scanning the stack of a task has a complexity linear in its stack size, which is predictable at the time the stack is scanned by the GC: The GC will scan the task's stack when it is in the waiting state, which is only the case in few well-known locations, where the `WaitEvent()` function of the operation system is invoked. Recursive calls complicate the prediction of the stack size. However, they can be identified at compile time and the programmer is advised to check the real-time capability of the application code.

The simple solution of interrupt disabling for the entire stack scanning phase results in a high interrupt latency, which depends on the stack size and consequently on the user application. It would also negate the low interrupt latency accomplished by optimized critical sections in other parts of the GC. However, it is not necessary to scan all task stacks atomically: Only the task whose stack is currently being scanned has to be delayed until stack scan completion. During stack scan, all discovered objects are merely colored gray, that is, the references are pushed onto a separate working stack. The scanning of the objects referred to by the references on the working stack is performed after stack scanning.

For synchronization, AUTOSAR OS resources are used. Instead of disabling the scheduling similar to the non-preemptive critical section protocol [15] – which may cause a high-priority task to be delayed by garbage collection, even though its stack is currently not being scanned – AUTOSAR OS resources are employed in a fine-grained manner: A garbage collector resource (GCR) is assigned to each task that uses the system call `WaitEvent()`. This is statically determined by *jino*. During stack scanning of a blocked task, the GCR of the task is occupied and the AUTOSAR OS priority ceiling raises the priority of the `GCTask` to the ceiling priority of the GCR, which is the priority of the waiting task. This still allows any other higher-priority task, ISRs and alarm callback routines to interrupt the GC during stack scanning. Tasks assigned a lower priority than the ceiling priority of the GCR are delayed, which is necessary to prevent unbounded priority inversion. Employing a GCR for each task that invokes `WaitEvent()` is the finest synchronization granularity for stack scanning and results in a minimum number of tasks being deferred. If this is not needed due to the user application, it is possible to define synchronization groups for a coarser level of synchronization. It is important that the stacks of the blocked task are scanned before the remaining parts of the root set. Otherwise, the only reference to a white object on a task stack could, for instance, be written to a static reference field that was already scanned and removed from the stack. Since write barriers only color the object to which a reference is overwritten, they do not color the object in this case: the reference is overwritten on the stack where write barriers are not active.

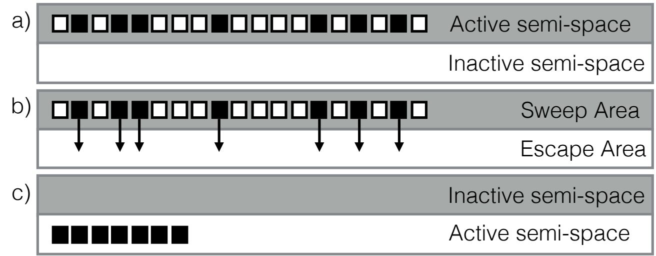


Figure 4: Replicating Spine Memory

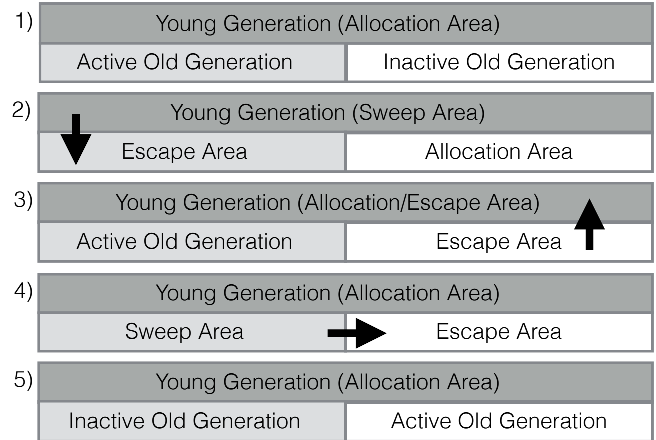


Figure 5: Generational Spine Memory

4.3 Management of Spines

Spines can be managed by replicating memory as proposed by Schism. In addition to that, RT-LAGC offers *generational spine memory*: A huge share of the GC effort is caused by long-living objects as they have to be marked and searched for references. At once, it can be noticed that most objects exist for a short period of time only. Generational collectors address this observation by reducing the work on long-living objects and focusing on young ones. The heap is divided into multiple generations and these generations can be managed by several memory strategies such as, for example, a replicating technique. The collection phase for the young generation is triggered more often than that for the older generation. An object surviving a determined number of GC runs is moved to the old generation. Allocation and collection of the replicating and generational spine memory are described in the following.

4.3.1 Allocation

Spines are placed contiguously by bump pointer allocation, which is guarded by a lock (i.e. disabling interrupts for uni-processors or a spinlock for multi-processors in our current implementation) due to the possibility of overlapping allocations. The guarded allocation could also be implemented using an atomic addition operation; however, this is not supported by many target platforms. Bump pointer allocation leads to a constant, short and predictable allocation time and reaction time to external events with no fragmentation.

Replicating Spine Memory.

Replicating collectors divide the heap into an *active* and an *inactive* semi-space as depicted in Figure 4. Allocations are handled by the active semi-space. A single index is shared between the semi-spaces, but only accessed by the currently active area. The request for the currently active semi-space does not face concurrency issues, as the lowest-priority GC task will never interrupt an allocation. All reachable (black) objects in Figure 4 are copied to the inactive area and the semi-spaces switch parts. As both semi-spaces cannot be used at the same time, the mandatory heap space is doubled.

Generational Spine Memory.

The RT-LAGC spine memory is divided into three areas – *young generation*, *active* and *inactive old generation* as depicted in Figure 5 – which correspond to three managing indices. If the allocation does not interrupt the GC run, the spines are placed in the young generation, which is similar to replicating memory. In case the GC's marking phase is interrupted, spines are allocated in the inactive old generation and are moved to the young generation at time of the completion of the GC run.

4.3.2 Collection

Spine memory is cleaned up by copying reachable spines from the *sweep area* to the *escape area*, while unreachable ones are left behind. The pointers stored in the spine remain constant after initialization, since the referenced fragments do not move. Thus, the consistency of spines is automatically ensured during the copy process. Allocations of higher-priority tasks interrupting the GC execution are performed in the *allocation area*. The location of the respective areas is dependent on the implementation of the spine memory. If an arraylet is marked reachable, the GC task allocates a new spine in the escape area, in case the reachable arraylet was located in the sweep area. The contents of the old spine are copied to the new one and the spine pointer is redirected to the new spine.

Replicating Collection of Spines.

To prepare the mark phase, the active semi-space becomes the inactive semi-space and vice versa. The inactive area is the sweep area and the currently active area is both escape and allocation area. Regardless of whether the marking phase is currently in progress or not, spines are allocated in the active semi-space, which is also true for the spines being copied from the sweep to the escape space.

Generational Collection of Spines.

Figure 5 illustrates the generational collection of spine memory: During the collection of generational spine memory, surviving objects from the young generation are moved to the old one, which turns the young generation into the sweep area, while the escape area resides in the active old generation semi-space. In contrast to the replicating collector, the generational GC has to treat spines differently in the mark phase: The allocation in the young generation may lead to

- Lost spines as references may escape marking
- Instant spine aging, which can in turn result in a premature exhaustion of old generation memory.

Thus, instead of allocating spines in the young generation, they are put into the inactive semi-space of the old generation, which becomes the allocation area as can be seen in line 2 in Figure 5. Since the inactive old generation is used for allocation, all spines allocated during the mark phase have to fit into this area. Therefore, the slack time available must be chosen according to the allocation rate[4] to complete the mark phase. Otherwise, the space for the old generation has to be increased. This approach is compliant to KESO's cooperative memory management approach. For the preparation of the mark phase, the number of sweep phases is recorded. According to a configurable number of sweep phases, the collection of the old generation is triggered: the meaning of the color bit of the old generation is inverted and a reachable spine in the old generation is colored accordingly. After the marking phase, the allocation area is assigned to the young generation again. Spines allocated in the old inactive generation during the mark phase are moved to the young semi-space and the spine reference of the respective arrays is adjusted (Figure 5, line 3). If necessary, the old generation is collected by traversing spines of the old active generation. A reachable spine is moved to the other semi-space (illustrated in line 4), the spine reference is adjusted and the semi-spaces of the old generation switch parts (line 5/6).

5. EVALUATION

In this section, we evaluate the costs and benefits imposed by real-time garbage collection. For this, we employ the real-time *Collision Detector (CD$_x$)* benchmark, which is available in a C (CD$_c$) and Java (CD$_j$) variant. We only evaluated the CD$_j$ version for this work, a detailed comparison of CD$_j$ on KESO and CD$_c$ can be found in a separate paper [18]. A brief introduction to CD$_x$ is presented in Section 5.1. Section 5.2 shows the analysis results for the typical Java object profiles presented earlier in Section 3 in the context of CD$_j$. An incremental evaluation of the garbage collection mechanism is given in Section 5.4.

5.1 The CD$_j$ Benchmark and KESO's Setup

The core of the CD$_j$ application is a periodic task that detects potential aircraft collisions from simulated radar frames. A collision is assumed whenever the distance between two aircraft is below a configured proximity radius. The detection is performed in two stages: In the first stage (reducer phase), suspected collisions are identified in the 2D space ignoring the z-coordinate (altitude) to reduce the complexity for the second stage (detector phase), in which a full 3D collision detection is performed (detected collisions). A detailed description of the benchmark is available in a separate paper [12]. Since CD$_j$ allocates temporary objects and uses collection classes of the Java library, it requires the use of dynamic memory management. In this evaluation, three garbage collection algorithms are compared against each other to determine the costs caused by RT-LAGC. A throughput-optimized stop-the-world GC (CBGC) is turned into an incremental latency-aware GC (IRRGC) that uses write barriers to synchronize with the mutator. On top of

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

the latter, the fragmentation-tolerant variation RT-LAGC is set up. We employ CD_j in the *onthegoFrame* variant. The configuration uses six airplanes, the collision detector processes 10000 frames and the detector task is periodically released with a period of 30 ms. The GC task is assigned the lowest priority in the system and uses the slack time for garbage collection that the detector tasks leaves in each period. A heap size of 600 KiB is used for CD_j. The object inlining transformation has been disabled. The generated code is deployed on an Infineon TriCore TC1796 board (150 MHz CPU clock, 75 MHz system clock, 1 MiB SRAM). The application is compiled with GCC (version 4.5.2) and bundled with KESO and an AUTOSAR OS implementation. The code is executed from internal flash memory. CD_j runs in a single protection domain. We also ported a multi-domain version of CD_j to KESO; however, this variant does not fit onto the TC1796 device in the current state.

5.2 Object Categorization for CD_j

The object categories presented in Section 3.2 are revisited in the context of CD_j. The results were computed by *jino* ahead of time.

Method-Local Objects.

On per-frame computation, 57 of 146 (39.04%) of all allocations in CD_j are marked as *local*. 47 objects (32.19%) escape their methods and 42 allocations (28.77%) are marked as *global-escaping*. Overall, 44 of 146 (30.14%) allocations in CD_j are eligible for stack allocation.

Region-Local Objects.

For this experiment, we simulated smaller regions by using extended stack scopes. Method-escaping objects are candidates for such scopes. In CD_j, 44.00% of all allocations can either be performed in the own or the caller's stack frame. Other region definitions for both stack or separate heap memories are possible. Such heaps can be managed by simple bump-pointer allocation, but this approach was not selected for this evaluation.

Thread-Local and Thread-Escaping Objects.

36.30% of all allocations are non-thread-local, that is, the created objects are accessed by more than one task. In contrast to this, *jino* computes 42.55% of all allocations to be non-thread-local for the multi-domain CD_j. The increase can be traced back to data transferred between the protection domains. In summary, 56–70% of all objects – depending on the use of extended stack scopes – are managed by RT-LAGC for the single-domain CD_j.

5.3 CD_j Profiling

To profile CD_j, we implemented object lifetime tracing for KESO. Figure 6 shows the frequency of object sizes occurring in CD_j. Objects sized 12 bytes appear most frequently, where 4 bytes are occupied by the object header and 8 bytes by the instance fields, respectively. As object inlining is turned off, the average object size is 12–16 bytes. Thus, the fragment size should be at least 16 bytes to avoid unnecessary runtime overhead caused by the linked-list structure of fragmented objects. By setting the fragment size to 32 bytes, all objects fit into a single fragment at the cost of a certain internal cutoff. Figure 7 depicts the objects' lifetimes: Most of the objects (81057) are released after the first GC cycle, so CD_j confirms the weak generational hypothesis [11].

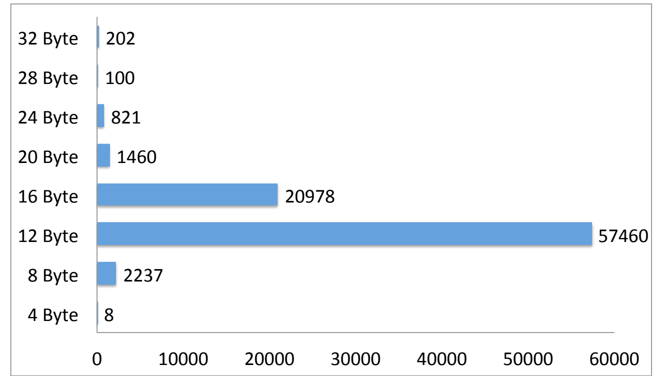


Figure 6: Frequency of regular objects' sizes

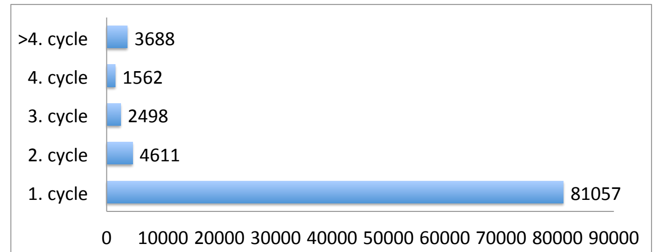


Figure 7: Frequency of regular objects' lifetimes

5.4 RT-LAGC and CD_j

In the context of CD_j, we evaluated the runtime, heap usage and footprint of RT-LAGC. We contrasted it by opposing KESO's other available heap strategies.

Runtime.

Table 9 and Figure 8 (100 frames are visualized) show an excerpt from our experiments The five RT-LAGC configurations are distinguished by either using:

- Generational (**Gen**) or replicating (**Rep**) spines
- Force fragmented arrays (**FFA**)
- Extended escape analysis (**EEA**)

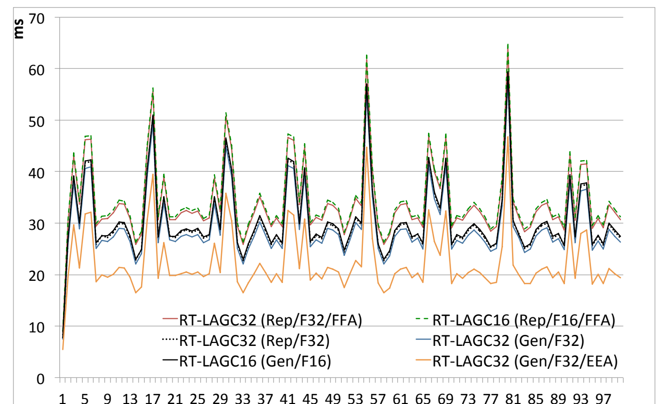


Figure 8: CD_j runtimes of RT-LAGC collectors

RT-LAGC	Gen	Rep	FFA	EEA	F16	F32	Ov
RT-LAGC32	✓	–	–	✓	–	✓	0%
RT-LAGC32	✓	–	–	–	–	✓	25.0%
RT-LAGC32	–	✓	–	–	–	✓	27.4%
RT-LAGC16	✓	–	–	–	✓	–	27.8%
RT-LAGC16	–	✓	–	–	✓	–	27.9%
RT-LAGC32	–	✓	✓	–	–	✓	35.3%
RT-LAGC16	–	✓	✓	–	✓	–	36.3%

Figure 9: CD_x runtime overhead of RT-LAGC collectors

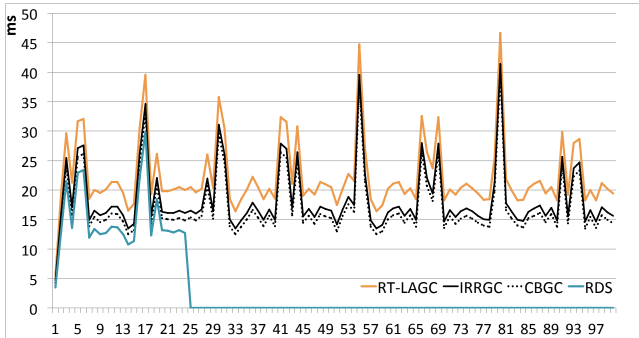


Figure 10: Runtime of the fastest RT-LAGC in contrast to KESO heap strategies RDS, CBGC, IRRGC

- Fragment size of 16 or 32 bytes

The baseline is formed by an RT-LAGC32 variant with enabled extended escape analysis to source out short-living objects to stack memories. Furthermore, it uses 32-byte-fragments, generational collection of spine memory and supports contiguous array allocation. Using a 16-byte fragment size instead of 32 bytes increases CD_j's runtime by 1–4%, which is attributed to the fragmented object layout. The number of linked fragments an object consists of was bounded to three. Configuring a 32-byte fragment size causes all regular objects to fit into a single fragment. In KESO's system configuration, the maximum number of tries to contiguously allocate arrays can be defined to constitute an upper time bound. Our experiments have shown that contiguous array allocation often instantly succeeds, which is attributed to CD_j's missing fragmentation behaviour. Therefore, we introduced the FFA option to always force fragmented array allocation to examine the costs: This GC variant shows a 35.3% runtime overhead to the baseline RT-LAGC. Allowing contiguous array allocation speeds up the execution time of CD_j by approximately 10% due to better data locality and fewer indirections. Generational spine memory is slightly faster (1–3%) than the replicating semi-space. The huger benefit of generational memory is that half of the spine memory can be saved in contrast to replicating variant. The biggest benefit is gained by using EEA (approximately 25–27%) to swap out short-living objects to extended stack scopes.

Table 11 and Figure 10 contrast RT-LAGC configured

	RDS	CBGC	IRRGC	RT-LAGC
Overhead	-15.4%	0%	6.8%	23.5%

Figure 11: Runtime overhead of the fastest RT-LAGC in contrast to KESO heap strategies RDS, CBGC, IRRGC

Heap Type	text	data	bss
RDS	40132	1925	655822
CBGC	45088	1993	661262
IRRGC	49816	1994	661294
RT-LAGC32 (Rep/F32/FFA)	57775	4049	814894
RT-LAGC16 (Rep/F16/FFA)	58251	2721	817294
RT-LAGC32 (Rep/F32)	58695	3345	814894
RT-LAGC16 (Rep/F16)	59167	2449	817294
RT-LAGC32 (Gen/F32)	59559	3361	814894
RT-LAGC32 (Gen/F16)	59071	2737	817294
RT-LAGC32 (Gen/F32/EEA)	60157	3361	814894

Figure 12: `Footprint` in bytes of CD_j using KESO heaps

with generational spine memory to KESO's other heap strategies. EEA has been enabled for all variants and the baseline is defined by the stop-the-world collector CBGC. To get an impression of the overhead imposed by garbage collection in general, *pseudo-static allocation* (RDS) is also contrasted. Pseudo-static (bump-pointer) allocation is comparable to `ImmortalMemory` specified in the RTSJ. RDS is 15.4% faster than CBGC, but it runs out of memory after 23 iterations. The incremental collector IRRGC has a higher overhead for the mutator (6.8% on average, 30% in the observed worst case), which is caused by the added overhead of write barriers and due to the complex linked-list implementation used to manage the free memory that allows a list traversal to be interrupted by higher-priority mutators. Disabling the synchronization code in the incremental collector shows indeed a very similar allocation performance to that of the stop-the-world GC. Adding the fragmented object and array layout to IRRGC causes an additional overhead of 16.7% (23.4% compared to CBGC).

Heap Usage.

The spine size can be reduced from 96 KiB to 48 KiB when using generational instead of replicating spine memory. The young space is set to 42 KiB, while 6 KiB are used for mature spines. EEA reduces the heap usage of CD_j by 42.6% on average. Comparing RT-LAGC16 in generational mode (fragment size of 16 bytes, contiguous arrays are allowed) to the IRRGC16 (allocation unit is 16 bytes), the heap usage increases by 0.7%, whereas forcing fragmented arrays will exalt the heap usage by 45%. Resizing RT-LAGC's fragments from 16 to 32 bytes engrosses the heap with a 44.7% rise.

Footprint.

Table 12 shows a comparison of the footprint for CD_j using different heap management strategies, distinguishing between the size of the `text` and data (`data, bss`) segments. The incremental IRRGC and RT-LAGC have a more complex implementation than CBGC and introduce write barriers in the application code. Comparing IRRGC and CBGC, the text segment is thus inflated by 9.5%. Due to the fragmented object layout, the size of the text segment for the RT-LAGC variants is increased by 13.8–17.8% in contrast to IRRGC. The access to instance fields of a fragmented object is more expensive, as indirections have to be followed to get the desired field. The RT-LAGC16 (Rep/F16) and RT-LAGC32 (Rep/F32) confirm this, since all objects of CD_j fit into a fragment sized 32 bytes. By using generational instead

of replicating spine memory, the code size of the application does not change. However, the GC code is slightly more complex due to special handling of the marking phase. This procedure leads to a moderate growth (1.5%). With FFA, contiguous arrays are not supported at all, which leads to a smaller text segment (1.6%). Support for contiguous arrays claims an additional allocator function. Moreover, in the mark phase and the application code, fragmented and contiguous arrays have to be distinguished. The data segment inflates due to a certain cutoff for alignment demanded by RT-LAGC depending on the fragment size. On the contrary, the IRRGC requires a 4-byte alignment. As an example, an 8-byte-sized global object and a 32-byte fragment leads to a 24-byte cutoff. The transition from a 16-byte to a 32-byte fragment increases the data segment by 0.1%. Taking IRRGC as base, the data segment size grows approximately 19% for the shown RT-LAGC configurations. In summary, for CD_j a RT-LAGC variant using generational spine memory, EEA and a fragment size of 16 bytes is a good configuration as it combines a moderate runtime overhead in contrast to a fragment size of 32 bytes with an acceptable heap usage behaviour.

6. CONCLUSION

For RT-LAGC, we applied the Schism technique on top of the incremental and cooperative LAGC that offers a short and predictable reaction time to external events. Our approach features an alternative object layout called FBOL for the easy discovery of references. In addition to replicating spine memory, generational spine memory is available in RT-LAGC, which results in a relevant reduction of spine memory. KESO's compiler *jino* is able to assist runtime memory management with its extended escape analysis. Furthermore, *jino* classified objects of the exemplary CD_j benchmark according to their survivability and size. This approach supports the machine-independent upper time and space bounds analyses for memory management of real-time applications. By combining the analyses of *jino* and RT-LAGC, a space-efficient and latency-aware solution for real-time runtime management has been developed.

7. ACKNOWLEDGMENTS

The authors would like to thank Christian Wawersich and Clemens Lang for their valuable hints and ideas for this project. This work was partly supported by the German Research Foundation (DFG) under grants no. SCHR 603/9-1 and SFB/TR 89.

8. REFERENCES

[1] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus. Deconstructing process isolation. In *MSPC '06: 2006 Memory System Performance and Correctness*, pages 1–10, 2006.

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

[3] D. F. Bacon, P. Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 81–92, June 2003.

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

[5] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *14th (OOPSLA '99)*, pages 1–19, 1999.

[6] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, Nov. 2003.

[7] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. In *Language Hierarchies and Interfaces, International Summer School*, pages 43–56, London, UK, 1976.

[8] E. M. Gagnon and L. J. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium*, pages 27–40, Apr. 2001.

[9] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *(PLDI '02)*, pages 282–293, 2002.

[10] F. Henderson. Accurate garbage collection in an uncooperative environment. In *ISMM '02: 3rd Memory Management*, pages 150–156, 2002.

[11] R. Jones and R. Lins. *Garbage collection: algorithms for automatic dynamic memory management*. New York, NY, USA, 1996.

[12] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek. CD_x: A family of real-time Java benchmarks. In *JTRES '09: 7th*, pages 41–50, 2009.

[13] O. Lhoták and L. Hendren. Run-time evaluation of opportunities for object inlining in Java. In *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande*, JGI '02, pages 175–184, New York, NY, USA, 2002. ACM.

[14] J. W. S. Liu. *Real-Time Systems*. 2000.

[15] A. K.-L. Mok. *Fundamental Design Problems of Distributed Systems for Hard Real-Time Environments*. PhD thesis, Massachusetts Institute of Technology, MIT, Cambridge, MA, USA, May 1983.

[16] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: Fragmentation-tolerant real-time garbage collection. In *(PLDI '10)*, pages 146–159, 2010.

[17] F. Siebert. Realtime garbage collection in the JamaicaVM 3.0. In *JTRES '07: 5th*, pages 94–103, 2007.

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

[19] T. Yuasa. Real-time garbage collection on general-purpose machines. 11(3):181–198, 1990.