Team Up: Cooperative Memory Management in Embedded Systems

Isabella Stilkerich Philip Taffner Christoph Erhardt Christian Dietrich Christian Wawersich
Michael Stilkerich

Friedrich-Alexander University, Erlangen-Nuremberg, Germany
{istilkerich, taffner, erhardt, dietrich, wawi, stilkerich}@cs.fau.de

Abstract

The use of a managed, type-safe languages such as Java in real-time and embedded systems can offer productivity and, in particular, safety and dependability benefits over the dominating unsafe languages at reasonable costs. A JVM that has dynamic memory-management needs to provide an implicit memory-management strategy, that is, for example, a garbage collector (GC) or stack allocation provided by the escape analysis of the JVM's compiler: Explicit management of dynamically allocated memory (i.e., by use of functions such as C's `malloc()` and `free()`) is vulnerable to programming errors such as neglected or false memory release operations causing memory leaks or dangling pointers. Such operations have the potential to break the soundness of the type system and are therefore usually not available for strongly typed languages. Type-safe languages in combination with static analyses – which respect hardware as well as system-specific information – can efficiently be employed to provide a runtime system including memory management (MM) that is specifically suited to an embedded application on a particular hardware device. In the context of this paper, we present novel memory-management strategy we implemented in our KESO JVM. It is a *latency-aware garbage-collection* algorithm called *LAGC*. Also, we introduce the static analyses that can assist LAGC. The application developers have to ensure that there is enough time for the GCs to run. Hardware characteristics such as soft-error proneness of the hardware or the memory layout can also be taken into consideration as demanded by the system configuration. This is achieved by integrating the GCs in the design process of the whole system just as any other user application, which is the reason why this approach is called *cooperative* memory management. The suggested strategies require reasonably low overhead.

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

General Terms Memory Management, Garbage Collection, Design, Languages

ESWEEK'14, October 12 - 17 2014, New Delhi, India.
Copyright 2014 ACM 978-1-4503-3050-3/14/10...$15.00.
http://dx.doi.org/10.1145/2656106.2656129

1. Introduction

Java is a rather uncommon language in (deeply) embedded systems, though it provides a series of advantages such as memory safety. 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 systems. Application software can be deployed on several kinds of microcontrollers that may vary in their hardware-specific features, such as availability of memories (RAM, ROM) and their respective layout or the occurrence of random soft errors. Combining the knowledge of the type-safe application, system software and the hardware features can help to create a runtime system that is tailored to a particular system setup. Memory management is an inherent part of such a runtime system and the focus of this work is to present how such a system and application-specific memory management for a particular hardware device can be created.

The paper is organized as follows: Section 2 will introduce the employed KESO JVM and the properties of the system, which are essential to understand our approach. Afterwards we will explain the overall idea of cooperative memory management in Section 3: The information sources given by the system and application developers are presented as well as our static analyses that process this information and assist the GC at runtime (Section 3.1). Afterwards, we propose a new low-overhead garbage collection mechanism in Section 3.2. The entire approach helps to create an automated tailored memory management. For space limitation reasons, we focus on one hardware-specific property: A case study of garbage collection and soft errors is given in Section 3.3. The evaluation (Section 4) of cooperative memory management in combination with soft errors is performed on a well-known real-time Java benchmark: We show to which extent the static analyses help the GC and also experimentally demonstrate how the garbage-collection algorithm proceeds in the presence of soft errors with and without our protection measures. The overhead in terms of memory footprint and execution time are also discussed. Related work and conclusions are presented in the last two sections.

2. Surrounding Conditions and Runtime System

The KESO JVM 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. These entities contain the entire type-safe source code of the application and operating-system objects such as threads (called tasks in AUTOSAR OS), for example. Thus, it is not possible to dynamically load new code or create threads at runtime. This scheme, however, allows to create a slim and efficient runtime environment for Java applications in embedded systems.

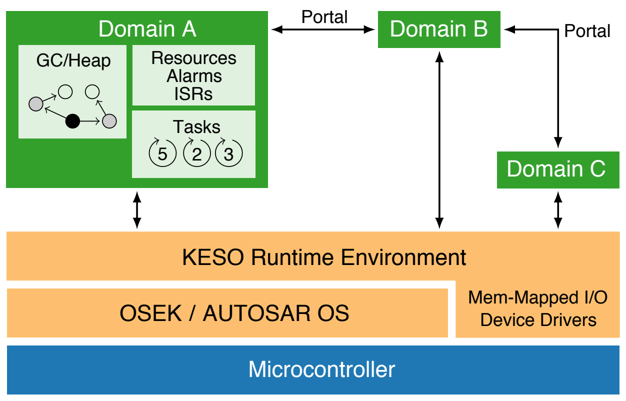

Figure 1: KESO's architecture

The architecture of KESO is shown in Figure 1. 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 (called *Portal*) for communication ensures, that object references[1] 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 applications benefit from Java features like type safety, dynamic memory management and optionally a garbage collector. The KESO's ahead-of-time compiler *jino* generates ANSI C code from the application's Java bytecode. During code generation *jino* also generates a runtime environment specific for that application. While most of the code directly translates to plain C code, the Java thread API is mapped onto the thread abstraction layer of an underlying OS. In the case of used JVM, that abstraction layer is normally provided by AUTOSAR OS, however, the KESO concept could also be transferred to any other static OS. Memory management applied under the given surrounding conditions can profit from the static and type-safe system setup, where the ahead-of-time compiler of the JVM can apply comprehensive analyses by incorporation of the system model, system configuration and the type-safe application code to assist the GC at runtime.

3. Cooperative Memory Management

To pursue the approach of cooperative memory management, LAGC is supported by KESO's compiler *jino*. This section gives an overview about the implications of the system model used in KESO and presents how *jino* processes the system information at hand (Section 3.1) and how the garbage-collection algorithm is supported by these analyses (Section 3.2). In order to demonstrate, to what extent hardware-specifics can be respected during the compilation and tailoring process as well as in the runtime memory management, we present a case study for taking random soft errors into account.

KESO's architecture of isolated domains implies the strict separation of the heaps and static fields. This trait leads to disjoint object graphs in the different domains, which allows (optional) garbage collection to be performed individually for each domain. It is also possible to deactivate garbage collection for applications, which do not need it.

[1] A reference is a compound of address and object information such as the object's dynamic type or memory management meta data

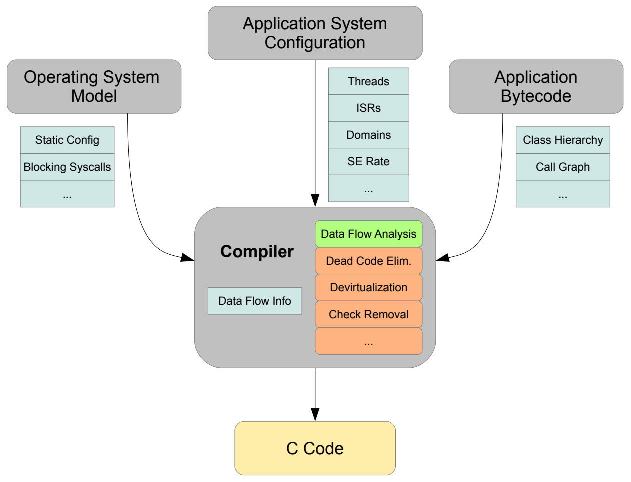

Figure 2: Cooperative memory management: KESO's compiler *jino* processes the application system configuration to assist the GC at runtime

3.1 Compiler-Assisted Memory Management

jino applies a series of high-level analyses and transformations on type-safe code. Due to the strict separation of references and primitive values, these compiler passes can be executed far more efficiently on an application written in Java in contrast to an application written in the (unsafe) C language. We developed new compiler passes for *jino*, which are particularly interesting for JVM's cooperative memory management. The three sources of information needed by the compiler are depicted in Figure 2:

Operating System Model The underlying OS provides several information that is needed for the compilation and tailoring process such as the knowledge about blocking and non-blocking system calls or the priority-based task scheduling mechanism. Taking the first-stated example into consideration, the compiler can, for instance, decide, whether particular structures for stack scanning (so-called linked stack frames, see Section 3.1.3) in the GC phase have to be emitted.

Application System Configuration The system configuration – provided by the application developer – informs *jino* about the concrete setup of the application world such as the thread architecture or the protection domains, into which the system is subdivided. For each domain, the entry points to the respective control flows are specified. Moreover, the application developer configures if these tasks have run-to-completion semantics (basic tasks) or can potentially block (extended tasks). It can be defined too, which events may occur and in which way they should be handled as well as the heap management strategy. Regarding hardware-specifics, also *soft errors* may be addressed: Application protection by spatially isolated replication can be configured as well as level of protection needed for hardening the runtime system against soft errors (i.e. the soft error rate (**SE**) needs to be derived from the hardware specification). Other hardware-specifics are, for example, the existence of a *memory protection unit (MPU) [22]*, existing *read-only memory (ROM)* (Section 3.1.2) or the *address layout* of the microcontroller (e.g. Section 3.1.4). In case of an MPU, *jino* may perform its reachability analysis to assist the MPU to physically group application data or decide, which memory safety checks (i.e. null checks) can be performed by the MPU or the runtime system. Application code annotations by the developer are **not** necessary for that. *All* this information is important for *jino*'s control-flow-sensitive analyses.

Application Bytecode The application bytecode itself states the information that is processed and transformed by the compiler. Effectively, the application system configuration determines a specialized variant of the code base. Due to the system's static nature, the class hierarchy, the call graph, etc. remain constant at runtime – it is not possible to load any additional code at runtime. This allows to perform a whole-program analysis and respective optimizations.

Starting from these three classes of static knowledge, it is possible to collect extensive information about the program – bundled with a specific operating system and deployed on a specific hardware device – that can be exploited in order to enhance the emitted C code that includes the application and the runtime environment. The technical details of the static analyses are not part of this paper, however, we give a short insight how they support system-specific memory management exemplary in the context of soft errors. Evaluation results can be found in Section 4.2:

3.1.1 Extended Escape Analysis

The information collected by alias analysis and the computation of the references' reachability can be used to automatically determine if an object *escapes* a method [7] i.e. if the object has to be allocated on the heap memory or if it can be stack-allocated. 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 caller's stack frame to further reduce heap objects. EEA should be combined with stack protection to support control flow isolation, i.e. the stack must not overflow. It is also used to compute the thread-locality of an object. Besides potential stack allocation, the EEA is useful for other issues as well: Such objects can still be heap-allocated[2], but are managed by a separate heap region to unload the GC. Molnar et al. too support stack allocation in the CACAO JVM [17] using a Steensgaard-based escape analysis. Unlike KESO, they employ just-in-time compilation and escape analysis is performed at runtime. Stack allocation of arrays is not supported.

We use EEA too for determining the *survivability* of an object, i.e. if the object survives a GC phase. This is helpful to reduce the upper space and time estimations for real-time memory management. As a consequence of EEA, the number of objects that need to be garbage collected is reduced and this implicates – particularly in the context of soft errors – a series of advantages:

- Deallocation and allocation is performed by moving the stack pointer. These are low-cost and time-predictable operations in a CPU register.
- Due to the reduction of heap objects, the GC phases are shortened. Furthermore, it reduces the GC's heap fragmentation as short-living objects are located in a separate memory area.
- The overhead of the overall application in a multi-threaded environment is reduced, since synchronization of the mutators on method-local and thread-local objects is not needed.
- Soft error-protection of references to stack objects is implicitly done be reference checking (see Section 3.1.4) and the overhead of protecting the GC phase against soft errors (Section 3.3.3) is significantly lower. Moreover, the size of the isolation domain can be reduced, which is particularly interesting, if application replication (Section 3.3) for data protection against soft errors is activated.

[2] There are situations, where stack allocation is not advantageous, such as e.g. some virtual method call constructs: EEA takes place after devirtualization, however, if lots of method candidates remain, the ESS object had to be allocated in every possible method, which may result in a waste of memory

```
void foo( obj_t **llref , obj_t *thisp ) {
  // 3 reference variables +frame link pointer
  obj_t *references [4] = { NULL, NULL, NULL };

  // link to previous frame
  *llref = references ;

  bar(&references [3], thisp );
}

void bar( obj_t **llref , obj_t *thisp ) {
  // terminate the list ...
  *llref = KESO_EOLLREF;
  // ... before calling a blocking system service
  WaitEvent(EventID);
}
```

Listing 1: Linked Stack Frames Implementation

3.1.2 Immortal Object and Runtime Final Analyses

In Java, it is not possible to mark objects such as fields as constant (`final`). Also, it is up to the application programmer to label constant data and this manual approach can lead to missed optimizations, if the marking is neglected. Regarding *final* references or arrays, the content of the referenced object can still be manipulated, only the object or array to which it is referring must not change. The knowledge if such objects are definitely constant is very precious, as this data can be placed in ROM, such as – for example – flash memories. A ROM unit is significantly cheaper than the scarce RAM. More importantly, ROM is often much less susceptible to soft errors [9]. Data located in ROM does usually not need to be protected by means against soft errors. Other advantages are similar to the ones mentioned with escape analysis (items 2.-6. in Section 3.1.1) such as reduced overhead in the GC phase. Korsholm proposed a manual annotation-based solution [15], whereas we extended *jino* to derive constant and write-once (*runtime final*) data automatically. These compiler passes are assisted by escape analysis to use aliasing information. If specified, constant objects become *immortal* by storing them in ROM. Runtime final objects always reside in RAM. Immortal objects have to be treated differently by the GC as their memory can naturally not be reclaimed. Headers of immortal objects contain a special marker value. Memory safety checks can be eliminated on such objects, as they are known to be non-`null`, which can contribute to a significant performance improvement.

3.1.3 Linked Stack Frame Elimination

We use linked stack frames (referred to as *llrefs*) to allow the GC to scan the task stacks for references. Listing 1 shows how the linked stack frames are implemented in KESO in the generated C code. The local reference variables of a function are stored in an array `references` rather than individual variables to ensure their physical collocation in memory. Besides the reference variables, the last element of the array contains a link pointer that is used to link to the `references` of the next frame, or contains a marker value `KESO_EOLLREF` to let the GC detect the end of the linked list. To maintain the list, the function interface is extended by a parameter `llref` that points to the previous link pointer and is updated in the prologue of a called function. Initially, the head pointer location which is known to the GC is passed. Before calling a blocking function, the list is terminated with the marker value. Linked frames add obvious overhead to the function prologues and calls for maintenance work on the linked list of references. In addition, the reference values need to be initialized with `null` in case the GC scans the list before the program has assigned a value to the variable. More severely, however, is the hidden overhead: the effect on the C compiler's optimizations, since alias analysis is more complicated when multiple variables are stored as a compound rather than individually. To reduce the overhead caused by linked frames, we only generate them for functions that are potentially active while the GC is running. In the AUTOSAR OS programming model, we can limit these as follows:

- Garbage collection is performed at slack time in KESO. This means that all application tasks are either suspended (empty stack) or blocked at that time.
- Heap space exhaustion will never cause a task to block during an allocation. Instead, an exception will be generated.
- Functions reachable only from basic tasks are never active while the GC is running due to the run-to-completion semantics of basic tasks.
- Functions reachable from extended tasks can be active during garbage collection only if they (transitively) invoke a blocking system service.
- All blocking system services are known to our compiler. In AUTOSAR OS, there is only a single blocking system service, `WaitEvent()`.

Based on these observations, we only use linked stack frames in blocking functions, that is, functions from which a path in the call graph to a call of the `WaitEvent()` service exists. If a non-blocking function is invoked from a blocking function, the linked frames list is not maintained in this sub-graph of the call graph. An interesting special case are dynamically bound method calls for which both blocking and non-blocking candidates exist. The function interface needs to be the same for all candidates to remain call-compatible. This issue can easily be solved by adding the `llref` as an unused parameter to the non-blocking candidates. Rafkind et al. also restricted the use of linked stack frames to allocating functions in the Magpie C source-to-source compiler [19]. We can apply this technique more aggressively due to KESO's side-stepped garbage collection that will never cause an allocation to be interleaved by a garbage collection. In our target domain, tasks are often periodically executed and only block at a shallow stack depth to wait for the next period, while the actual periodic activity is performed in a called method. For such applications, the periodically executed code will not require the use of linked frames at all and therefore not suffer from the performance penalties in our system model.

3.1.4 Runtime Checks

The compiler emits *memory safety* checks (i.e. null and array bounds checks) at those code locations, where its data flow analyses could not statically prove that the access will always succeed or that an MPU (if present and assisted by *jino* to be used in that way) will generate a hardware exception due to the microcontroller's memory layout [23]. When considering soft errors, the *integrity* of the regarded *reference* has to be checked prior to the regular memory safety checks to preserve software-based memory protection as proposed by Stilkerich et al. [21]. Two of these integrity runtime checks have been integrated in *jino* and are distinguished **by the time** the check is executed:

Dereference Check (DRC): In this variant, object references stay encoded all the time. They are tested and decoded every time they are dereferenced, thus the probability of detecting an illegal reference is very high. Dereferencing takes place **whenever** one of the following bytecode instructions is executed: `putfield`, `getfield`, `*aload`, `*astore`, `arraylength`, `instanceof`, `checkcast`, `invokevirtual`, `invokeinterface`, and `invokespecial`.

Load Reference Check (LRC): In this variant, object references are checked **as soon as** they are loaded from memory – that is, from a static field (bytecode instruction `getstatic`), an object field (`getfield`) or an object array (`aaload`). They are encoded when being stored and decoded upon being loaded.

An object reference integrity check is two-fold: First of all, the address the object is located at has to be proven valid and secondly its respective header information (e.g. dynamic type information) is checked. We describe later in Section 3.3 how these mechanisms can be used during garbage collection. One possibility is to protect references with parity information, however, also any other protection mechanism is conceivable and easy to plug into the system with respect to the application developer's choice. The protection information's layout may respect the hardware device: As an example, selected target platform (TriCore TC1796), uses 32-bit addresses, but the microcontroller has only 1 MB addressable RAM. This hardware characteristic can be exploited to embed up to 12 bits of parity information in the references without increasing the attack surface of the program.

3.2 Runtime Memory Management

We have implemented a novel incremental latency-aware heap strategy called LAGC for KESO: Embedded systems usually receive external events (e.g. measurements from sensors connected to bus systems) and compute – in reaction to these events – results to drive actors (e.g. an engine). Thus, an important aspect in such systems is a low latency on external events (that is, interrupts). KESO's incremental garbage collector was designed to restrict all critical sections to constant complexity. The worst case reaction time to interrupts is therefore low and predictable. LAGC is a precise, tracing, non-moving mark-and-sweep GC and its working principle is as follows: Garbage collection is executed by a dedicated control flow (called `GCTask`). This single task is responsible for the garbage collection in all domains that use the particular heap management strategy, but only processes the heap of one domain at a time. The `GCTask` is assigned the lowest priority in the system, thus the slack time of the system is used to perform garbage collection runs. This is a good moment to perform a garbage collection, since most tasks will be suspended 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 [8]. 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. The LAGC can be interrupted during a GC run and needs to synchronize with the application. All critical sections within the incremental GC where interrupt handling needs to be suspended are of constant complexity.

Scanning References The root set comprises the static reference fields and local reference variables on the stacks of blocked tasks. The GC needs to be able to find these references in memory with as little overhead as possible. This issue is solved by

- Grouping all static reference fields of a domain into an array that can simply be traversed.

- Using a bidirectional object layout as proposed by SableVM [11] that physically groups reference fields of objects in memory, even when inheritance is being used.
- Grouping all local reference variables in a stack frame, and building a linked list that links the groups of the different stack frames. This is a variation of Henderson's linked frames [13]. The list is maintained in the prologues and epilogues of potentially blocking methods.

Incremental Scanning of the Object Graph The LAGC uses write barriers as proposed by Yuasa [24] during the mark-and-scan phase to allow incremental scanning of the object graph, except for local reference variables on the stack. Whenever a reference to a white object is overwritten, the object is colored gray by the barrier which prevents that a white object can become invisible to the GC after the start of the mark-and-scan phase. Stack scanning of a task is not incremental and of complexity linear to the size of the stack: The stack is only scanned when a task is waiting for an event, which is only the case in few well-known places where `WaitEvent()` is called. (Unbounded) recursive invocation of blocking methods may, however, complicate the prediction of the stack size. The stack size can be computed by our compiler ahead of time and *jino* is able to detect recursive calls. LAGC uses AUTOSAR OS resources (i.e. priority ceiling) when scanning the stack of a task. This delays the task whose stack is being scanned until the entire stack has been scanned, but allows higher priority tasks and ISRs to interrupt the stack scan operations. In addition, the immediate priority ceiling prevents tasks with a priority lower to that of the task whose stack is being scanned to interrupt the scan, which prevents unbounded priority inversion. The scan-and- mark phase consists of the following steps:

- Enable write barriers, change the allocation color to black
- Scan the stacks of blocked tasks and mark reachable objects gray
- Scan the root set (static fields) and mark reachable objects gray
- Proceed scanning gray objects until there are no more gray objects
- Disable write barriers, change the allocation color to white

It is important that the task stacks are scanned first since (expensive) write barriers are not used for stack variables. Otherwise, the only reference to a white object W on a task stack could, for instance, be written to an already scanned static field and removed from the stack. This would remove the last discoverable path to W without marking W, causing the object to remain white and being reclaimed during the sweep phase though still being visible to the application, which may compromise the type system and must not happen. Garbage collection can be preempted between scanning two task stacks, since a reference value cannot move from one (white) stack to another (black) without first being stored in a (white/gray) field on that write barriers are active.

3.3 Type-Safe Runtime Systems and Soft Errors

Safety-critical embedded systems have specific requirements regarding hardware and software components to avoid or mitigate malign errors. Functional safety standards such as the IEC 61508 and ISO 26262 address this issue and distinguish between *systematic* and *random* errors. Systematic errors can occur in hardware and software components and are the result of design and implementation defects. Engineering processes and methods exist to avoid and mitigate systematic defects. In order to fight typical implementation errors in software (bugs) that often occur when using the programming C – which is wide-spread in embedded systems engineering – programming standards such as MISRA-C have been released. Such rules and standards restrict unsafe programming languages to get a safer language subset. Dynamic memory management or the use of function pointers, for example, are prohibited in systems compliant to MISRA-C. Random errors do not reside in the system in the first place and only occur in hardware. They are referred to as permanent (hard) and transient (soft) errors, where soft errors have – in contrast to hard errors – only a temporary effect on the logical circuits or memory. Soft errors become noticeable as bit flips and are a result of hardware failures that are becoming more likely to happen as a consequence of shrinking structure sizes [4], extreme environmental conditions such as radiation or voltage-supply problems. We state that type-safe languages are not only beneficial for the reduction of systematic errors, but also for the mitigation of random soft errors as well as the consideration of other system-specific traits: Comprehensive analyses at compile time help to create a lean memory-safe runtime system suited to the deployed application. Complex and safeguarded runtime memory management has not been used in a soft-error-prone setup so far and we would like to address this issue with our case study. Precise garbage collection is usually performed in combination with type-safe languages. These languages, however, can lose their isolating character in soft-error-prone environments as presented later in Section 3.3.1. As the memory management in general deals with object references we adopt the reference protection mechanisms from [21] as earlier described in Section 3.1.4. The authors did not use a garbage collector and to address this, we combine reference checking with additional safety runtime checks to build a dependable complex memory management (MM) variant. In general, several ways to safeguard MM in the presence of soft errors and – as a consequence – software-based memory protection, are conceivable:

No usage of GC at all: Lots of embedded application do not require garbage collection. Instead, region-based [12] or bump pointer heap management can be employed. Bump pointer access has to be protected as otherwise type system information in the objects located on the heap or elsewhere could be corrupted. It has to be combined with reference checking (either LRC or DRC) in the application to assure the isolation properties of the type-safe programming language [21].

Type-safe GC: The proposed LAGC could be implemented in a type-safe language and treated the same way as a KESO application. Reference checking (LRC or DRC) will protect references. The integrity of the GC data structures used for book-keeping (see Section 3.3.3) purposes has to be protected in addition.

Implement GC in unsafe language: The GC is part of the trusted computing base, i.e. it does not compromise the type system, if soft errors do not occur. When accounting for soft errors, the references used during its run phase as well as the GC data structures have to be protected. As the GC is not implemented in (type-safe) Java, naturally the information of load and store instructions from the Java bytecode level is not present. This is the reason why DRC should be used[3]

The first approach was presented in [21], a GC was not employed as no dependable GC was available. The second proposal has currently been skipped due the unsuitable implementation of *memory-mapped objects* in KESO: Such objects describe the layout of a specific region in memory and are comparable to C `structs` with a more fine-grained access control. Without any further runtime system support, it is usually not possible to manipulate object

[3] This checking mechanism can be applied to the generated (memory-safe) C code, as the reference information is compiled into the runtime environment. It would be possible to apply LRC to the machine code level, but this approach currently not in our scope.

information in a type-safe language, which is on the other hand a mandatory property for any garbage collector, as it has to read and write object information at runtime. Special abstractions such as the memory-mapped objects provided by KESO can be used to access the type information on the heap. The GC has to access the application objects through these special abstractions, which causes an additional overhead in space and time. These abstractions are planned to be optimized to reduce this overhead, as this one is too high in the current state. As the GC is usually part of the (small) trusted code and computing base, it is usually not required to have a type-safe GC to establish memory protection. However, with the consideration of soft errors, this assumption is not true anymore, as bit flips may cause wild references: Thus, we present a possible solution for the third idea in the context of this work: We implemented the GC algorithm in the unsafe C/C++ languages and further support it with the static analyses mentioned in Section 3.1. Moreover, protection of the garbage collection information in addition to reference checking is applied to preserve the characteristics and implications of the (memory-safe) runtime system.

3.3.1 Garbage Collection and Soft Errors: A Case Study

Soft errors occurring in the GC phase can have a devastating effect on the integrity of the runtime system. On the one hand, there are the application references, which are accessed during the GC run and on the other hand, there are internal memory-management structures for book-keeping purposes. Corruptions in these spots can produce wild references and thus violate the type-safety property of the programming language and the assumptions used during the compilation process. An obvious solution to this problem is to disable the GC, which is possible in lots of embedded applications and to only allow safeguarded checked bump-pointer management of objects on the global heap in combination with application replication [21] However, some reasons for a soft-error hardened GC exist. In many embedded systems, memory is a scarce resource. A protected GC can help to keep the heap size reasonably low, which reduces the overhead of application replication techniques. Also, the probability of detecting corruptions in object references is very high, since the object graph needs to be scanned. For this reason, we would like to give an insight into dependable garbage collection. First of all, we introduce our fault model in Section 3.3.2 followed by an inspection of critical GC-related runtime structures in Section 3.3.3. In the evaluation (Section 4), an experiment with an exemplary application is done, which reveals the fault detection rates as well as the overhead imposed by a protected LAGC. Moreover, a study on the size of the application replication sphere with and without LAGC is given.

3.3.2 Fault Model

Protection of memory management against soft errors is essential for the runtime environment to preserve the isolating character of the type-safe programming language. Strongly typed languages can safely isolate different software objects as long as the integrity of the type system can be maintained. This naturally also has an impact on the used fault model. Firstly, we only consider soft errors that become visible at the programming interface of the processor, as we propose a software-based solution. This comprises soft errors in arbitrary memory locations and registers. It does not matter in which part of the processor these errors actually occur – in the memory or the register itself or while data is transferred from memory to a register on the bus – but it is important that software-based checks are able to cover such errors. Thus, we cannot detect soft errors when data is corrupted after we have checked for its integrity while it is copied from e.g. a register to memory or an output location. Secondly, we safeguard memory management but not the application itself in the experiment. That is, we only protect those items which are necessary to preserve spatial isolation provided by a type-safe programming language. Mainly, such items comprise object references, the associated type information and the memory-management data structures. Application protection has to be accomplished on a higher level by means of e.g. application replication. Thirdly, we assume that program code and data that is located in non-volatile read-only memory like flash does not suffer from transient faults, as these memory areas normally are more robust than e.g. SRAM or registers [9] and their ECC hardware protection mechanisms are sufficient to protect the code section against transient faults. So, we do not make any effort to protect executable code and constant data stored there.[4] In summary, we certainly cannot tolerate arbitrary transient faults affecting the memory management. However, we try to reduce the probability of soft errors to compromise the isolating property of a type-safe programming language as far as reasonably possible. In Section 4, it is *experimentally* exhibited how compromised heap management could affect isolation and we will present measures to harden it.

3.3.3 Critical GC-Related Structures and Soft Errors

The GC data structures in KESO are designed to cause low overhead. This is both beneficial for restraint-constraint systems and to minimize the overhead of protecting the JVM against soft errors:

Free Memory List The GC uses a linked list to maintain the free memory of the domain's heap. This list is referred to the *free memory list (FML)*. Its elements contain book-keeping information on a block of continuous free memory such as their 16-bit *size* field (expressed as *slots*), the 8-bit *colorbit* and the 8-bit *locking mode*. Soft errors in the FML may cause addressing errors.

Slot Division and Bitmap The GC employs a bitmap (BM) to mark the used memory by setting respective bits in the BM. An object has a minimum size due to the object header, which is currently four bytes. Thus, the heap is divided in slots of a fixed, statically-configured length (*slot size*) and an object uses one or more consecutive slots on the heap. Thus it is sufficient to map one bit in the BM to a heap slot. Since slots are not shared between objects, there is a certain cutoff, if objects only partially occupy a slot. The use of statically known slot size information is an immediate (ROM) access and does not need to be protected against soft errors under the given fault model. However, the bitmap might be compromised:

- A bit flip in a marked slot, may cause the GC to falsely reclaim used memory, which may cause the application code to overwrite type information and break the type system.
- Flipping the bit information in the unmarked slot may cause the application to run out of memory, as the memory chunk cannot be reclaimed. The occurrence of this error still preserves the integrity of the type system.

Working Stack The working stack (referred to as *WS*) is an array of references and is used by the `GCTask` during the scanning phase to keep track of references to objects that still have to be scanned. The stack pointer is an index into the WS and contains the index of the next unused array element. Therefore, pushing to the stack is a post-increment operation on the stack pointer while popping references from the stack is a pre-decrement operation. This immediate access is also regarded as soft error-safe. The stack is empty when the stack pointer is 0. The maximum stack size has to be predictable, since the stack must not overflow during the scanning phase. The GC's scan algorithm assures that an object reference is never present on the stack more than once. Thus, the worst-case size of the stack is

[4] If less robust ROMs are used, additional means have to be applied to protect them. Such measures could still be combined with our approach.

determined by the sum of maximum number of objects that can be allocated from the heap and number of immortal objects in the entire system. The maximum number of allocatable objects matches to the number of heap slots, whereas the immortal objects are allocated by *jino* and is therefore known at compile time. During scanning, object references can potentially be corrupted.

Managed Domains The managed domains array has all information of the domain identifiers of the domains managed by LAGC. It is used to select a certain domain for garbage collection by means of specific criteria, such as the filling state of the heap. Having configured more than one domain, a soft error in the domain selection phase may cause the wrong domain being collected. This error does not lead to corruptions of the type system, but may result in a *out-of-memory* situation that is signaled as exception and can be handled by KESO.

GC Domain Since the LAGC of this heap can be interrupted during a garbage collection cycle, the mutator has to know, if the GC thread currently runs in this domain. Transient errors in this identifier can lead to synchronization issues when using the LAGC or to slow down program execution as unnecessary synchronization code may be processed. The interruptibility is discussed in detail in Section 3.2. The access to the GC Domain is an immediate (ROM) access and thus does not need to be protected due to the fault model.

Linked Stack Frames To allow the garbage collector to scan the references on the tasks' runtime stacks, KESO uses linked stack frames as mentioned previously in Section 3.1.3. The entry to the linked stack frames list as the object references contained in it may be corrupted due to a bit flip.

3.3.4 Protection of Garbage Collection

In the following, we shortly describe how the two traditional GC phases proceed in the face of transient errors. Additional information can be found in Sections 3.2 and 3.3.3. The runtime checks were carefully inserted in neuralgic code spots of the GC only to keep the overhead and the increase of the attack surface of the program as low as possible.

Scanning of the Object Graph The traversal of the object graph starts at the root set (consisting of static reference fields, local references of the blocked tasks' stacks). The integrity of the references – that is object addresses as well as the associated `type` and as `color` data – has to be tested on dereference operations (DRC) to prohibit the propagation of wild references. In addition, the entry pointer to the linked stack frames entry point used for blocking control flows must be checked. It should be noted that DRC has been applied to the **type-safe application** and **object references** in the past by Stilkerich et al. [21] as described earlier in Section 3.1.4. To transfer the DRC *concept* to the **C/C++ language level** and to **memory managements elements** (which can be pointers or data items), the technical implementation of checking mechanism depends on the intended use:

1. To protect **object references** of the **application** during the GC phase, the object pointer representing a Java reference has been instrumented, so that it is automatically checked (address plus object header check) on every dereference operation.
2. To protect **pointers** of the **garbage collector**, their type is instrumented to perform an address check only on dereference operations.
3. To protect **data** of the **garbage collector**, its integrity is checked as soon as its information is propagated to the application (i.e. **not** on every dereference operation).

The maximum root set size (including the number of static references) is statically known and is used in immediate read operations. The valid objects are pushed to the working stack.

Mark-and-Sweep To check the integrity of the bitmap – that implicitly contains type system information – on every read and write access is extremely expensive (approx. 300% allover GC run overhead). As an example, assume a heap size of 768kB and slot size of 16. This setup results in 49152 slots and a bitmap size of 6144 bytes. In contrast, an integrity check checks 32-bit data (4 byte) in our experiment, for which special processor instructions may exist. Still, for every object marking, checking the bitmap is a huger overhead. This approach is also not necessary to protect the system's type safety: Indeed, the bitmap holds runtime system information, however, the propagation of its erroneous state is only critical, if it causes the propagation of invalid references. Due to the interruptibility of the `GCTask` by higher-priority threads, two bitmap error scenarios can occur that might lead to the corruption of software-based memory protection:

- Interleaved allocation: The application may seize a falsely reclaimed memory chunk and overwrite type information of another application object. The FML integrity of other FML elements may still be valid.
- The FML becomes corrupted, as information in the falsely reclaimed block may overwrite FML book-keeping information.

As a solution, the ISR has to check the bitmap integrity in its prologue, if the sweep phase marker is set. The overhead of an ISR increases by executing the checking code, however, the additional cost is constant and preditable.

4. Evaluation

In this section, the costs imposed by KESO's LAGC algorithm is evaluated. For this, we employ the Java version of the real-time *Collision Detector (CD$_x$)* benchmark. A brief introduction to it is presented in Section 4.1. The effects of static analyses provided by the *jino* compiler to assist vanilla as well as soft-error hardened collection are discussed in Section 4.2. We contrast LAGC and the soft-error hardened `S-LAGC` in relation to the protection level provided by our GC-protection measures by using the Fail* fault injection framework, which is explained in Section 4.3. Finally, the evaluation is concluded by comparing these two variants with respect to runtime and footprint overhead in Section 4.4.

4.1 The CD$_x$ Benchmark

The core of the CD$_x$ 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 [14]. Since CD$_j$ allocates temporary objects and uses collection classes of the Java library, it requires the use of dynamic memory management.

4.2 Static Analyses: Compiler-Assisted MM

To determine the benefit of compiler support for garbage collection, selected analyses of those we implemented have been evaluated in the context of KESO. We will show static results as well as their influence on runtime behavior.

Extended Escape Analysis On per frame computation, 57 of 146 (39,04%) of all allocations in CD_j are marked as *local*, from which 13 allocations have overlapping liveness regions and are thus not considered for stack allocation. 47 objects (32,19%) escape their methods and 42 allocations (28,77%) are marked as *global-escaping*. In summary, 44 of 146 (30,14%) allocations in CD_j are eligible for stack allocation. Method-escaping objects are candidates for extended stack scopes. In this benchmark, another 14% of allocations can be performed in the caller's stack frame: Around 44% of all objects can be stack-allocated or managed by a thread-local heap. Thus, 56-70% – depending on the usage of extended stack scopes – of all objects are managed by LAGC. EEA speeds up CD_j by up to 9.5% in contrast to pure heap allocation. Also the heap usage at runtime is more than halved.

Immortal Object and Runtime Final Analyses The analyses reduce the data segment size by 41% and the text segment by 14%. The number of null pointer checks emitted has been reduced by 30%, which contributed to the code size reduction. The runtime final analysis reduced the execution time of the overall CD_x application by 5-12% due to dead code removal and runtime check elimination. Afterwards, placing constant data in ROM instead of RAM increased the CD_x's runtime by 4-8% in turn.

Linked Stack Frame Elimination With respect to the system model, the compiler determines that only 18 out of 195 methods need linked stack frames. For LAGC, the runtime overhead of the CD_x benchmark is reduced by 23.4%. As well, the text segment size is decreased by 14.1%. Linked stack frames do not add noticeable overhead to the GC run, as garbage collection is performed at slack time. The elimination of linked stack frames is sensible in soft-error-prone environments as it supports an efficient application of reference checking: The overhead of runtime checks for reference integrity is decreased by 11.1%.

As the yet suggested analyses contribute to a better memory footprint, execution time and support runtime memory management in general but also support integrity reference checking, they are enabled for our case study with soft errors. Experiments have shown, that the soft error susceptibility drastically rises, when deactivating these optimizations. For space limitation reasons, the numbers are not shown.

4.3 Dynamic Analyses: Fault Injection

In order to get an impression on the effects of transient errors onto GC and the protection level provided by safeguarded memory management, the fault injection (FI) framework Fail* [10] is used. It is currently available for the Bochs simulator [16] on the x86 platform and ARM simulators. The selected AUTOSAR OS is available for x86 and TriCore platforms. Therefore, we built a variant of the CD_x benchmark that runs on top of the x86 port of the AUTOSAR OS. Even though the x86 platform is not a deeply embedded platform, we argue that FI experiments on that platform can be used to evaluate the functional effects of bit flips in MM structures and on reference accesses. The x86 (multicore) platform allows us to run lots of fault injections and cover a huge faults space in a reasonable amount of time. For the FI campaign, LAGC is enabled. Hardware-based memory protection is disabled, but we use *jino*'s reachability analysis to physically group application data as input for Fail* to determine illegal memory accesses. The heap size is set to 48 KiB and a set of 50 frames was processed. We injected single bit flips into each bit position of each word of GC runtime structures and the references accessed during the collection phase. To reduce the resulting huge fault space, we made use of the fault-space pruning methods of the Fail* framework to concentrate on memory locations that are actually read according to a golden run. As a result, idempotent faults are merged: Ineffective FI experiments (instructions between write and read of a memory location) are not considered as they lead to the same FI result. This allows to filter out all injections that are known to be ineffective, e.g. bit flips that are overwritten before they are actually read. The campaign applied to LAGC resulted in a total of seven FI campaigns. An excerpt from the fault injection results for the LAGC and the selected reachable destination points can be found in Table 1. The `Vanilla` variant does not include any protection against soft errors at all. Gradual protection is shown adding object header checks in combination with `DRC` and the safeguarding of the GC data structures `FML, BM, llref, WS` and the respective combinations. The `S-LAGC` includes protection of *all* GC data structures combined with DRC during the GC run and application. We picked 1-bit error detection in KESO's system configuration file. `Total` denotes the entire count of performed experiments for a particular variant. In Table 1, for example, 154724 injections were performed in the Vanilla variant, where 13697 injections (8.6%) led to a breach of type system and isolation properties and thus illegal memory accesses. In summary, 2.435.990 experiments have been performed. It should be noted that the experiment count varies, since the protection mechanisms naturally change the program and thus also the count of read operations change. As the program variants are not directly comparable, we show absolute values instead of relative ones. We are especially interested in the preservation of type system properties, that is, a reduction of illegal memory accesses and falsely triggered memory safety checks (null and array bound checks) as well as cast and heap overflow (both denoted as error) exceptions. For the unprotected LAGC, a huger share of the injections resulted in *NoEffect*, that is, bit flips where either masked or silently corrupted the application's data. Regarding software-based memory protection, a portion of 324 of the injected faults (0.08%) led to a breach of the type system (*Illegal Memory Access*, i.e. any access beyond the defined sections and writing accesses into the text section). Due to the integrity checks and fault space pruning, the number of injected faults (e.g. 154724 in Vanilla and 399912 in S-LAGC) varies.

Using reference checking only, is not sufficient at all to keep isolation properties, as errors in crucial memory-management structures other than object references, lead to corruptions of the type system as discussed in Section 3.3.3. The application of FML protection in combination with DRC reduces this share significantly. The gradual protection of the bitmap (BM), working stack (WS) and the entry in to the linked stack frames list (llref) in combination with DRC for reference accesses, led to a smaller amount of illegal memory accesses and false triggering of memory safety checks. An interesting observation is that it does not really matter – assuming an equal distribution of soft errors - which of these GC structures is protected. A gradual protection with these structures is not a huge benefit, but rather the combination of FML/DRC. Full type-safety protection (F-TS: S-LAGC plus application reference checking) resulted no illegal *Traps* (such as division-by-zero) and *Timeouts* at all. Timeouts may occur, for example, due to flawed loop conditions. Software-based memory safety checks are rarely illegally triggered anymore, whereas *GC* and *Parity* exceptions (both denote detected errors by KESO) caught lots of the injected faults (261095).

4.4 Overhead to Unguarded Garbage Collection

To illustrate the costs imposed by safeguarded garbage collection and application, we have examined the footprint and runtime of selected protection configurations. We use CD_x in the *onthegoFrame* variant, which is deployed on the Infineon TriCore TC1796 device (150 MHz CPU clock, 75MHz system clock, 1 MiB SRAM). The application is compiled with GCC (version 4.5.2) and bundled with KESO and an AUTOSAR OS implementation. All integrity checks have been inlined, which increases the text segment (ROM), but drastically reduces the runtime by up to 20%. It should be noted

LAGC	Vanilla	WS/DRC	FML/DRC	BM/DRC	llref/DRC	Color	S-LAGC
No Effect	131034	130971	132642	72181	76800	76675	138406
Timeout	883	880	877	798	794	920	0
Trap	1560	1569	0	1660	1657	1657	0
Illegal Memory Access	13697	13330	567	11864	11840	11840	324
Error Exception	5060	4951	183	4604	4613	4613	34
Null-Pointer	2385	2251	28	2252	2246	2246	11
Out-of-Bounds	105	183	91	320	313	313	42
Parity Exception	0	699	772	2742	2865	2605	545
GC Exception	0	0	251396	8035	67	915	260550
Total	154724	154834	396312	104456	100968	101784	399912

Table 1: Fault injection results for LAGC

in advance that the Vanilla variant already profits from escape, immortal and runtime final analyses as well as the elimination of linked stack frames (see Section 4.2 for details). We just measured the additional overhead of runtime integrity checks as these memory-management optimizations are not only beneficial when taking soft errors into account, but also in every other system setup.

Memory Footprint. Table 2 shows the footprint for the text segment for a set of protection variants. As protection information has been embedded in the existing GC structures, there is no change in the data and bss segment. The text section is inflated by 0.93% up to 1.97% for the respective MM structures and GC algorithms. Combining DRC (full protection of all object references in the application and GC phase) leads to an overhead of 35.17%. Protection of all runtime system information available can achieved at a 35.92% overhead for LAGC. With the employment of GC protection, the heap size of an application can be shrunk significantly. Considering CD$_x$ as an example, 256 KiB heap size is needed to compute a set of five frames, whereas with safe garbage collection, a 48 KiB is sufficient to process an unlimited number of frames. The use of application replication on top of our approach is thus much less memory consuming thanks to the shrunk heaps.

Runtime. Reference checking (DRC), which is needed to protect object references during the scan-and-mark-phase causes approx. 44-48% additional runtime overhead for the LAGC. The working stack is implicitly protected by DRC. Safeguarding of color information, linked stack frames and the free memory come at a reasonable cost as can be derived from Table 3. Bitmap protection is expensive, as the parity over the entire structure – 384 bytes as a result of the 48 KiB heap with a slot size of 16 – has to be computed. We are currently working on a compacted version of the bitmap, which is also particularly advantageous if having lots of small-size objects. This approach has also the potential to improve the runtime of the integrity checking. Disabling the synchronization code turns LAGC into a stop-the-world GC and speeds up CD$_j$'s runtime by 6.8%. Disabling garbage collection and using bump-pointer allocation reduces the runtime by another 15.4%. We also compared the C version of CD$_x$ called CD$_c$ against CD$_j$ bundled with KESO: CD$_c$ is 6% slower than CD$_j$ using bump-pointer allocation due to whole-system static analyses. Thus, the use of a managed language with automated memory management – which can adequately be protected against soft errors – gets along with moderate performance penalty by using our cooperative memory management approach.

5. Related Work

Garbage collection in embedded systems has been addressed in prior research projects such as, for example, ([18],[1],[2], [3],[20]). They are also suited to be deployed under real-time conditions, but do not address microcontroller and operating system-specific features. In [5] the susceptibility of application data in JVMs and their protection was analyzed, targeting JVMs for workstations. The authors also point out that lots of errors may be uncovered in the garbage-collection phase. Chen [6] proposed to detect and recover from transient errors by adding a dual-execution and check-pointing extension to KVM. The heaps of both instances are compared against each other. To keep the overhead low, the heaps are divided into sub-heaps which contain the latest changes and those which have not changed. Unchanged heap parts and moving new heap data is protected by hardware-based memory protection in the shape of a memory-management unit (MMU), however, garbage collection is not protected and also type safety is not retained. Therefore, this approach is not suitable to provide software-based memory protection in the presence of soft errors. Errors that trigger a trap can be corrected by copying the state from the sane instance to the corrupted one. The focus is on 1-bit error detection and state recovery of heap objects. We are not aware of any research of *explicit* garbage-collection protection against transient errors nor any other incorporation of hardware and operating system specific features for memory management. Our approach of combining information from all system layers provides an efficient and safer automated memory management for embedded systems.

6. Conclusion and Future Work

In this work, a novel latency-aware garbage-collection mechanism with short and predictable response times to external events has been introduced. The employed GC data structures have been designed to cause reasonably low overhead. Moreover, we have presented a cooperative memory-management approach and have shown how static analyses considering the hardware specification, operating system model and the type-safe code can be used to support runtime memory management with respect to the application developer's configuration. New compiler passes namely the extended escape analysis, immortal object and runtime final analyses have been developed. We also integrated the linked stack frame optimization into our system model and applied reference integrity check insertion by means of DRC to the references used by the GCs. To further assure memory safety, we have demonstrated that gradual protection of garbage collection data structures against transient errors is possible with acceptable runtime and footprint overhead if the checks are inserted at particular code locations. The system developer can decide how much effort should be spend for protection techniques, while having the effects of such errors in mind and still provide an efficient solution. The dynamic analyses with the integrated Fail* tool to support this approach. With the employment of a safeguarded GC, the replication base can be shrunk effectively in contrast to using pseudo-static allocation only. The compiler's optimizations are very useful to support the GC and keep the overhead of dynamic safeguarded memory management reasonably low.

For our future work, we would like to improve our static analyses such as the extended escape analysis and stack protection to further reduce the efforts of runtime memory management. The garbage-collection mechanism will be extended to support supplementary

Text	Vanilla	DRC	FML	FML/DRC	BM	Ilref	WS	Color	F-TS
CD_j + LAGC	0%	33.20%	1.97%%	35.17% %	0.89%	0.81%	0.68	0.50%	35.92%

Table 2: Footprint (Text segment)

Runtime	Vanilla	DRC	FML	BM	Ilref	Color
LAGC	0%	47.72%	7.64%	23.07%	8.94%	6.44%

Table 3: Runtime Overhead in GC phase KESO MM

control-flow monitoring (single- and multi-core machines, i.e. using additional cores if available) to further harden the GC phase.

7. Acknowledgments

The authors would like to thank Michael Strotz, Clemens Lang and Simon Kuhnle 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.

References

[1] J. Auerbach, D. F. Bacon, B. Blainey, P. Cheng, M. Dawson, M. Fulton, D. Grove, D. Hart, and M. Stoodley. Design and implementation of a comprehensive real-time Java virtual machine. In *7th ACM Conf. on Embedded Software (EMSOFT '07)*, pages 249–258, New York, NY, USA, Oct. 2007. ACM. ISBN 978-1-59593-825-1. .

[2] J. Auerbach, D. Grove, B. Mccloskey, D. F. Bacon, B. Biron, A. Micic, P. Cheng, C. Gracie, and R. Sciampacone. Tax-and-spend: Democratic scheduling for real-time garbage collection. In *8th ACM Conf. on Embedded Software (EMSOFT '08)*, Oct. 2008. ISBN 978-1-60558-468-3. .

[3] D. F. Bacon, P. Cheng, and V. T. Rajan. The Metronome: A simpler approach to garbage collection in real-time systems. In R. Meersman and Z. Tari, editors, *Proceedings of the OTM Workshops: Workshop on Java Technologies for Real-Time and Embedded Systems*, volume 2889 of *LNCS*, pages 466–478. Springer, Nov. 2003.

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

[5] D. Chen, A. Messer, P. Bernadat, G. Fu, Z. Dimitrijevic, D. J. F. Lie, D. Mannaru, A. Riska, and D. Milojicic. JVM susceptibility to memory errors. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium*, pages 67–78, Berkeley, CA, USA, Apr. 2001. USENIX.

[6] G. Chen and M. Kandemir. Improving Java virtual machine reliability for memory-constrained embedded systems. In *Proceedings of the 42nd annual Design Automation Conference*, DAC '05, pages 690–695, New York, NY, USA, 2005. ACM. ISBN 1-59593-058-2. .

[7] 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. ISSN 0164-0925. . URL `http://doi.acm.org/10.1145/945885.945892`.

[8] 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. Springer. ISBN 3-540-07994-7.

[9] G. C. et al. Neutron-induced soft errors in advanced flash memories. In *IEDM 2008*. IEEE, Feb. 2009. ISBN 978-1-4244-2378-1.

[10] H. S. et al. FAIL*: Towards a versatile fault-injection experiment framework. In G. Mühl, J. Richling, and A. Herkersdorf, editors, *25th Int. Conf. on Architecture of Computing Systems (ARCS '12), Workshop Proceedings*, volume 200 of *Lecture Notes in Informatics*, pages 201–210. Gesellschaft für Informatik, Mar. 2012. ISBN 978-3-88579-294-9.

[11] 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, Berkeley, CA, USA, Apr. 2001. USENIX.

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

[13] F. Henderson. Accurate garbage collection in an uncooperative environment. In *ISMM '02: 3rd Int. Symp. on Memory Management*, pages 150–156, New York, NY, USA, 2002. ACM. ISBN 1-58113-539-4. .

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

[15] S. Korsholm. Flash memory in embedded Java programs. In *JTRES '11: 9th Int. W'shop on Java Technologies for real-time & embedded Systems*, pages 116–124, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0731-4. .

[16] K. P. Lawton. Bochs: A portable PC emulator for Unix/X. *Linux Journal*, 1996(29es):7, 1996.

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

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

[19] J. Rafkind, A. Wick, J. Regehr, and M. Flatt. Precise garbage collection for C. In *ISMM '09: 2009 Int. Symp. on Memory Management*, pages 39–48, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-347-1. .

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

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

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

[23] M. Stilkerich, D. Lohmann, and W. Schröder-Preikschat. Gradual software-based memory protection. In *Proceedings of the Workshop on Isolation and Integration for Dependable Systems (IIDS '10)*, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0120-6.

[24] T. Yuasa. Real-time garbage collection on general-purpose machines. *J. Syst. Softw.*, 11(3):181–198, 1990. ISSN 0164-1212. .