A JVM for Soft-Error-Prone Embedded Systems

Isabella Stilkerich Michael Strotz Christoph Erhardt Martin Hoffmann Daniel Lohmann
Fabian Scheler Wolfgang Schröder-Preikschat

Friedrich-Alexander University Erlangen-Nuremberg
{istilkerich, strotz, erhardt, hoffmann, lohmann, scheler, wosch}@cs.fau.de

Abstract

The reduction of structure sizes in microcontollers, environmental conditions or low supply voltages increase the susceptibility of embedded systems to soft errors. As a result, the employment of fault-detection and fault-tolerance measures is becoming a mandatory task even for moderately critical applications. Accordingly, software-based techniques have recently gained in popularity, and a multitude of approaches that differ in the number and frequency of tolerated errors as well as their associated overhead have been proposed. Using type-safe programming languages to isolate critical software components is very popular among those techniques. An automated application of fault-detection and fault-tolerance measures based on the type system of the programming language and static code analyses is possible. It facilitates an easy evaluation of the protection characteristics and costs as well as the migration of software to new hardware platforms with different failure rates. Transient faults, however, are not bound to the application code secured by the type system, but can also affect the correctness of the type system itself. Thereby, the type system might lose its ability to isolate critical components. As a consequence, it is essential to also protect the type system itself against soft errors. In this paper, we show how soft errors can affect the integrity of the type system. Furthermore, we provide means to secure it against these faults, thus preserving its isolating character. These measures can be applied selectively to achieve a suitable tradeoff between level of protection and resource consumption. [1]

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers; D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and Objects; D.4.5 [*Operating Systems*]: Reliability—Fault-tolerance; D.4.7 [*Operating Systems*]: Organization and Design—Real-time Systems and Embedded Systems

General Terms Reliability, Design, Languages

Keywords KESO; Java; RTSJ; Embedded Systems; Real-Time Systems; Reliability

[1] This work was partly supported by the German Research Foundation (DFG) under grants no. LO 1719/1-1 and SCHR 603/9-1

LCTES '13 16–21 June 2013, Seattle, Washington, USA.

1. Introduction

A lot of embedded systems have particular safety 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 categorize such errors into so-called *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. On the contrary, *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 manifesting themselves as bit flips are a result of hardware failures that are becoming more likely to happen as a consequence of shrinking structure sizes [7], extreme environmental conditions such as radiation [25], or voltage-supply problems.

Usually, functional-safety standards outline hardware-based redundancy and the employment of specialized error-correcting hardware components – such as ECC for memory devices or hardware watchdogs to recognize bogus behavior of components – as a possible solution. As these solutions entail additional or more expensive hardware components, this approach is often not feasible due to an immense cost pressure in many industrial domains. Products in such domains often are mass products, where cost differences of few cents on the single device can amount to huge values considering the whole of the produced devices. The tolerance towards added cost is particularly limited when caused by features that do not directly pose an added value visible to the customer, which is the case for robustness regarding soft errors. Besides the cost factor, hardware redundancy is often impractical due to physical size, weight and power constraints, which are an essential requirement in embedded systems.

Software-based fault-tolerance techniques such as spatially and temporally replicated execution of code or monitoring software components go without extra hardware and pose a cheaper alternative for increasing system dependability. A combination of both approaches may also be worthwhile in certain scenarios. In the absence of transient errors, a type-safe software system, for instance, could already provide constructively ensured spatial isolation of software components. However, soft errors can break the soundness of the type system and thus an integral part of software-based replication. For this reason, a memory-protection unit (MPU), for example, is necessary to maintain spatial isolation and to avoid error spreading [19, 26]. Hence, such software-based replication techniques usually cannot be applied at all to embedded systems where such an MPU does not exist. If an MPU is present, transient errors are recognized that cause an address to be out of preconfigured memory bounds. However, software-based fault detection at the granularity of objects – in contrast to region-based mechanisms, as for example provided by an MPU – has some advantages. The error-detection

time is lower and effects of errors can strongly be localized, thus allowing for fine-grained reliability measures that scale with the imposed costs and which are not confined by the limited number of address regions provided by an MPU. In this paper, we show how software-based isolation building on the type safety of a programming language can be preserved in the presence of transient errors, which allows for an early detection of bit flips and facilitates the application of software-based redundancy techniques without the need for an MPU. However, a combination of both software- and hardware-based memory protection is still possible to additionally harden the system.

As software-based fault tolerance does not come for free in terms of runtime overhead and different hardware also differs in the failure rates, it is necessary to tune the fault-detection or -toleration technique towards the safety requirements and also the hardware features. Therefore, a separation of the functional code and the non-functional property fault-tolerance is desired. For this, we use KESO [23] – a Multi-JVM for deeply embedded systems. KESO already allows for the automated application of fault-detection/tolerance measures using the results of comprehensive static analyses enabled by the type-safe programming language Java. In summary, the dimension of the fault-detection/tolerance measures could be adapted without touching the functional application code.

At first, a look at our fault hypothesis is taken in Section 2. Afterwards, the targeted domain and surrounding conditions are discussed in Section 3. Sections 4 and 5 describe the characteristics of a protected runtime system as well as the measures taken to ensure software-based memory protection in the context of KESO. In Section 6 we evaluate our approach using the Collision Detector (CD_x) benchmark [16]. Section 7 covers work that is related to ours before Section 8 concludes this paper.

2. Fault Hypothesis

As spatial isolation among the different replicas is an indispensable property of replication, we aim at improving the robustness of software-implemented spatial isolation by exploiting a type-safe programming language. Such programming 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 fault hypothesis our work bases upon.

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 bit flips in arbitrary memory locations and registers. It does not matter in which part of the processor these bit flips 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 covering such errors are possible. Thus, we cannot detect 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 strive for the protection of the type system but not the application itself. 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, pointers to virtual function tables or type information in object headers. We will explain how a corrupt type system could affect isolation and the measures to harden the type system later on in Section 4. Our intention is to improve the robustness of these elements in the presence of one soft error at a time so we can provide a reliable foundation to implement software-based spatial isolation. Thus, we do not guard application-specific data like computation results. This has to be accomplished on a higher level by means of e.g. replication as presented in a previous paper [26].

Thirdly, there are some elements that have the potential to compromise spatial isolation when affected by transient faults that are

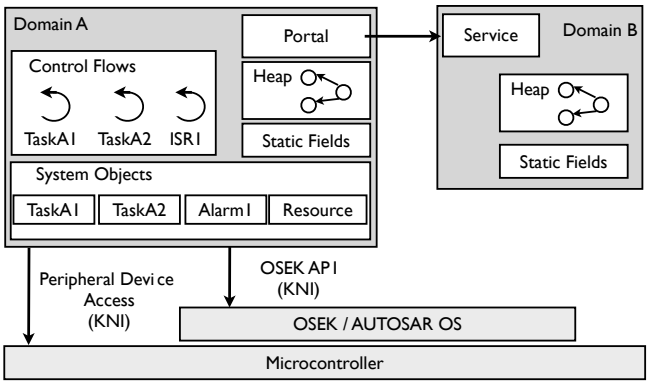

Figure 1: KESO's architecture

not safeguarded by our approach. In particular, these are the program-counter register (PC) and the operating system (OS). Additional measures are necessary to take care of these weak spots. A possible solution to detect the corruption of the PC is e.g. control-flow monitoring [15]. The OS, on the other hand, could be hardened by additional algorithmic measures [20] or – at best – the OS is implemented in the same type-safe programming language [14] that is used to achieve spatial isolation among different replicated components.

Fourthly, 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 [8]. So, we do not make any effort to protect executable code and constant data stored there.

In summary, we certainly cannot tolerate arbitrary transient faults affecting the type system. But we try to reduce the probability that a corrupted type system breaks the isolating property of a type-safe programming language as far as reasonably possible.

3. The KESO JVM

We selected KESO [23] to evaluate the impact of soft errors to a type-safe runtime system, as it targets applications for statically configured embedded systems. In the following section, we introduce the key features of KESO as far as they are relevant for this paper.

3.1 Maxim and Concept

In statically configured embedded systems, all relevant entities of the application itself and the underlying system software are known at compile time. These entities comprise the complete code of the application and also operating-system-level objects (threads, interrupt service routines, synchronization locks, etc.) influencing the runtime behavior of the system. This type of application covers many, if not most, traditional embedded applications, from control units providing safety-critical functions such as the electronic stability program (ESP) and many other electronic functions found in nowaday's railway systems, airplanes or medical devices.

This scheme imposes some restrictions on applications building on top of KESO: It is not possible to dynamically load new code or create new threads at runtime. On the other hand, it allows to create a comprehensively tailored and efficient runtime environment for Java applications – even for small, deeply embedded systems.

3.2 KESO Architecture

The architecture of the KESO Java runtime environment is depicted in Figure 1. KESO provides the control-flow abstractions typical for this domain (i.e. threads called *Tasks* and interrupt service routines (ISRs)) and means to activate (e.g. alarms) and synchronize them

properly (e.g. via synchronization locks called *Resources*). Furthermore, KESO applications benefit from Java features like type safety, dynamic memory management and optionally a garbage collector. KESO even allows access to raw memory through Java objects. Thus, it is possible to implement complete embedded applications including device drivers (as long as these devices are interfaced via memory-mapped registers) in Java.

The ahead-of-time compiler *jino*, which is an integral part of the KESO toolchain, generates ANSI C code from the application's Java bytecode. During code generation *jino* also generates a runtime environment specific for that application. Additionally, *jino* can integrate e.g. reliability measures and software-based memory protection. While most of the code directly translates to plain C code, the Java thread API is mapped onto the thread abstraction layer of an underlying OS. In the case of KESO, that abstraction layer is normally provided by AUTOSAR OS.

Like KESO, AUTOSAR OS is configured completely statically, i.e. all relevant system objects (threads, ISRs, locks, etc.) and their properties (task and interrupt entries, runtime priority, interrupt source, etc.) have to be determined ahead of runtime and cannot be altered while the system is running. Thus, an application could not create threads dynamically or attach a different ISR to an interrupt source. Besides the application code itself, an AUTOSAR application needs to provide a system-description file that defines the instances of these OS objects and their attributes. The system-description file is used by the AUTOSAR OS implementation to create an OS variant containing statically allocated instances of the defined OS objects. In the context of KESO, this file is provided by *jino*. Furthermore, many AUTOSAR OS implementations ship with a code generator that outputs an OS implementation that is specifically tailored for the application in order to avoid unnecessary overhead.

3.3 Tailoring KESO

The KESO JVM adopts the idea of creating a tailored version of the infrastructure software that provides only the features required by the application. KESO's compiler *jino* uses a system-configuration file and the entire source code of the application as input to determine these features.

The system-configuration file contains all the information that is needed to generate an instance of AUTOSAR OS plus some KESO-specific extensions such as fault-detection and fault-tolerance options. The configuration file also explictly controls coarse-grained features such as the existence of a garbage collector (GC) or the replication of certain parts of the application.

The JVM features demanded by the application are implicitly extracted from the application code via static analyses. Features like floating-point support, 64-bit integers or virtual methods are detected by the post-reachability analysis. Some of those are not mere boolean features but are tuned on a more fine-grained level. For example, the dispatch tables needed for virtual method binding are only generated if *jino* failed to provide a full static binding.

3.4 Memory Protection in KESO

Being a Multi-JVM, KESO allows tasks to be spatially isolated in different protection domains, each of which appears as a JVM of its own from the application's point of view. If soft errors are not an issue, Java's type safety guarantees that an application can only access memory to which it has been given an explicit reference, and the type of the reference determines how an application can access the memory area pointed to by the reference. Type-safe programs are therefore also memory-safe [2]. In order to enforce type safety, the compiler inserts runtime checks into the code:

- For all invocations of non-static methods and accesses to object fields and arrays, the associated object reference must be valid, that is, non-`null`. A `null`-check is inserted before these operations.
- All array accesses must be within the array's bounds, so the index of the accessed element is checked against the array size.

Since the entire application is known ahead of time, *jino* can perform whole-program analyses and aggressively eliminate unnecessary checks by statically proving accesses to a reference or array to be correct.

Spatial isolation is established based on the logical separation of the object heaps and by maintaining a separate set of the static fields in each domain. Each control flow (i.e. task or ISR) and all other system objects are statically assigned to a domain. A system object can only be accessed from other domains if explicitly permitted by the KESO system configuration.

In order to allow control flows from different domains to exchange data with each other, domains may export a functional interface, a so-called *Service*, that can be invoked from other domains by using a proxy object, a so-called *Portal*, that represents the service in the foreign domain. Deep copying is used for parameters and return values in portal calls in order to retain the heap separation. As a copy-free alternative to the portal mechanism, KESO also provides shared-memory areas that can be accessed by a controllable set of domains using the same programmatic interfaces that are available for accessing raw-memory areas. To maintain isolation, all inter-domain communication mechanisms (IDC – i.e. portals and shared memory) must ensure that no reference values can be propagated to another domain.

In addition to the software-based spatial isolation, KESO can actively support an OS to provide hardware-based memory protection using an MPU. KESO supports the OS by physically grouping the domain data (i.e. the physically separated heaps and static fields) in separate memory regions to recognize addressing errors and so to additionally harden the system.

3.5 Fault Detection and Tolerance in KESO

KESO already supports the creation of dependable embedded systems by mechanisms for fault detection and tolerance such as software-based replication of critical application parts. This feature is smoothly integrated into the *jino* compiler and can be controlled through the system-configuration file. At this point, the developer can specify the number of bit flips that have to be tolerated by an application. KESO then instantiates the needed number of replicas, isolates them from each other spatially, either on the level of the programming language or by means of an MPU, and finally integrates them into the application. Moreover, *jino* is able to generate a majority voter to identify the faulty replica, and code to restore its state with the aid of the remaining intact replicas after a fault has been recognized. Either is only possible thanks to the type safety provided by Java: References can easily be distinguished from primitive data and both are strongly typed, enabling the automated generation of a majority voter and the recovery code.

As spatial isolation is the main prerequisite to build dependable systems based on replication, it is not sufficient in the presence of soft errors to rely only on software-based memory protection as it is implemented in KESO. Bit flips can corrupt references and thus break spatial isolation. This problem could be solved by using an MPU to separate the different replicas of a replicated dependable embedded system. However, many low-end microcontrollers do not offer an MPU and the protection offered by an MPU is rather coarsed-grained.

Instead of relying on an MPU, we examined how spatial isolation via software-based memory protection can be preserved in the presence of soft errors by systematically protecting references and type information. Utilizing the static nature of the system and respective compiler-based techniques makes an efficient implemen-

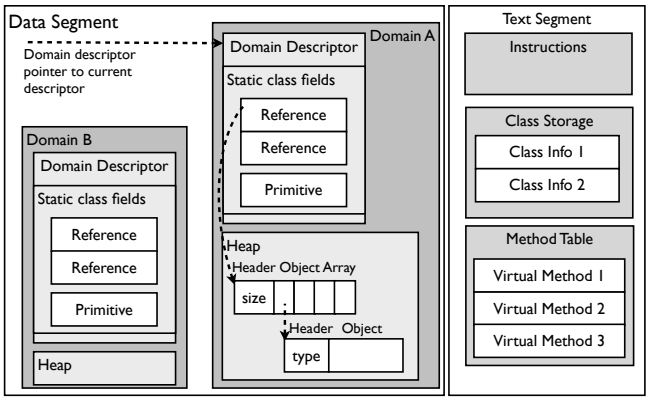

Figure 2: Relations between domain descriptor and object references: The domain descriptor is used by the application through the domain-descriptor pointer. A static reference (accessed through a constant offset) is employed to access an array object that contains references to plain objects. A virtual method is invoked on the first element in the array after type-checking that object. The method table located in the text segment is accessed at a constant offset.

tation of reference checking (RC) possible. This advance can further be assisted by accounting for microcontroller specifics – such as failure rates, special instructions and address layout.

As software-based memory-protection techniques apply on a very fine-grained level, they also call for a fine-grained application of reliability measures and thus a low error-detection time in contrast to MPU-based systems. Furthermore, the application developer can continue to rely on the benefits of a type-safe programming language even in a fault-prone hardware environment.

4. Runtime-System Integrity

The scope of this paper is a reliable type-safe middleware for embedded applications that retains software-based isolation also in the presence of soft errors. Therefore, we focus on the protection of the runtime system itself and not of the application data.

With regard to the fault hypothesis stated in Section 2, we identified the following critical spots which influence the integrity of the runtime system in the context of KESO:

- The type system ensuring memory safety. Here, e.g. **references**, **class identifiers** and **virtual method tables** could be corrupted.
- Per-**protection-domain data** mainly comprising static fields and heaps for each domain. Regarding the memory-management system, data elements used by **memory allocators** and the **garbage collection** for book-keeping purposes could be corrupted, compromising the software-based spatial isolation.

In this paper we only consider the protection of the type system as we want to evaluate its cost and effectiveness in isolation. Thus, we spare complex memory-management strategies using garbage collection and leave this topic as future work. This is not a real drawback as garbage collection is not necessary for many embedded applications [23, 27]. The Java language does, however, not allow a static allocation of objects. For such applications, KESO provides a simple heap strategy that does not provide garbage collection at all. The advantage of this heap implementation (bump-pointer or pseudo-static allocation) is the short, constant and thus easily predictable time required for the allocation of an object. Since there is no way of releasing the memory of objects that are not required any more, the application should only allocate memory objects during the initialization phase.

4.1 Effects of Soft Errors on the Runtime System

In Java, type safety ensures that programs can only access memory regions to which they were given an explicit reference; the type of the reference also determines in which way a program can access the memory region pointed to by the reference. This is utilized by KESO to establish spatial isolation by preventing any shared data between protection domains. For this purpose, heap objects and static class fields of the different protection domains are logically separated as mentioned earlier.

Preserving software-based memory protection in the presence of transient errors is two-fold: Firstly, global book-keeping information of the type system must not be corrupted. Secondly, protection-domain data has to be handled in a correct way, so that each domain only gets access to its private data. Thirdly, it has to be ensured that wild references and faulty type-system information caused by bit flips are recognized in order to maintain type safety and thus, preventing faulty accesses to other memory locations.

In the following, an overview is given how bit flips in book-keeping information of the runtime system and in references can influence the runtime environment. The relationships between the data structures and references are illustrated in Figure 2.

4.1.1 Global Information of the Type System

KESO incorporates a set of runtime-system-internal data structures relevant to enforce the type system. The *method table* is used in conjunction with virtual method invocations, whereas the *class storage* holds essential data about the individual classes such as the size of an object instantiated from a class. Bit flips in the class storage would invalidate static assumptions on the object's type and dynamic type checks.

The virtual method table as well as the class storage are computed statically and are constant, allowing for locating these data structures in robust ROM. Accesses to the structures are performed with constant indices, which is the reason why those accesses can be regarded to be safe (see Section 2).

The remaining part of type-system information is incorporated in each object (in plain objects and the derived array objects).

4.1.2 Global Information of each Protection Domain

For every domain, there exists a descriptor which holds the static class fields (containing both primitive data and references) and a pointer to the domain's heap. Both the descriptor and the heap are non-constant and therefore reside in RAM, which is subject to transient errors. While primitive data is considered as part of the application, a static reference must be secured just like any other reference.

In addition, there is one global pointer which references the descriptor of the protection domain in whose context the current thread of control is running. Context changes are performed by setting the pointer to another domain descriptor. This global pointer is of particular interest and must be explicitly secured. As the static fields and the heap of a protection domain are accessed by dereferencing this global pointer, its corruption could have severe consequences. A domain could get access to the static fields and the heap of another domain and thereby break spatial isolation.

Heaps and management strategies are available on a per-domain basis. In the context of KESO, we consider *pseudo-static* allocation, where the heap can instantly be corrupted due to a bogus bump pointer. This pointer is part of the domain descriptor. In case of garbage collection, bit flips in references in the scan-and-mark phase as well as in the sweep phase cause inconsistencies in the object graph, leading to wild references.

4.1.3 Local Information inside Objects and Arrays

The remaining part of the type system is directly embedded into the objects created and manipulated by the application. Although this information is local to these objects, it has the potential to compromise the complete type system if it is corrupt. In any of the cases explained below, spatial isolation provided through the type system could not be guaranteed any more.

Object and Array References: Bit flips in object references may produce wild object references with devastating impact. Primitive values could be read and interpreted as a reference values, for example. Also, data could be stored to illegal locations so that runtime information of other objects is corrupted. Existing `null`-checks inserted by the runtime system are invalidated – as they are checking a corrupted reference – and thus faulty dereferencing is not detected any more.

Object-Header Information: In KESO, each object has a header block holding meta-information such as the type of the object, which is represented by a class identifier (ID), and thus determines in which way the memory the reference points to can be used. The class ID is used as an index for lookups in the method table and the class storage. Using a wrong type – because the class ID is affected by a transient fault – breaks the soundness of the type system. If a method not suiting the type of the object is invoked, this may entail memory accesses outside the scope of the particular object and also outside of the protection domain.

Array Header Information: In case of the specialized object variant *array*, there is an additional size parameter that defines the length of an array, and it has to be ensured that any array access is within bounds. The size parameter can be affected by soft errors: In the best case, it has changed to a smaller value. This may cause additional array-bounds exceptions in some cases. If the flipped array size is greater than the actual one, an array bounds check is not able to detect an illegal array access any more. At array locations where the index and array size are both known to be constant, bit flips in the array size information do not affect type safety.

4.2 Protection of Type Safety

Having identified these critical location, we now take a closer look into how isolation and type safety can be preserved in the presence of soft errors. First of all, the protection-domain descriptor and references to it have to be correct. Secondly, an object reference has to refer to the correct memory location and an object which is placed at that location must only be accessed by operations that suit the type of that object to preserve memory and type safety. Both characteristics are needed to establish software isolation.

4.2.1 Protection Techniques

Before we examine these issues in detail, we discuss possible concepts to harden the type system against soft errors.

Attack-Surface Reduction Many transient errors occur directly in RAM or when data is transferred on the bus, that is, on load and store accesses to memory. A reduction of such accesses leads to fewer memory-protection errors [11]. Validating that a reference read into a register is correct reduces illegal addressing, thus error spreading, and can maintain type safety. Bit flips that might occur in registers are currently not in the focus of our approach, but may at least be recognized if a register value is checked whenever it is stored to memory. Such errors are also likely to be detected on the next load of a reference in case of a corrupted address or by an additional protection of application data.

Soft errors which occur on the bus when the data is stored are also outside the scope of our experiments in this paper. If the microcontroller has an MPU that is situated on memory, it can complement reference checking to detect bus store errors, however if the MPU is located on the processor, such errors cannot be detected at the time of the store instruction.

Checking References An RC can be performed in many different ways such as, for example, replication and voting or via a checksum. There is no focus on a specific technique, as it should be adjustable as demanded by the system configuration with regards to runtime overhead, memory consumption and safety requirements. Due to the static nature of the system, all reference accesses can be determined ahead of time and reference checks can be automatically inserted by *jino*. At runtime, these checks ensure the integrity of the respective memory accesses. As an initial technique, we instrumented *jino* to enrich references with parity information. This technique has some advantages over reference replication [6], as it does not inflate the reference size and so the attack surface of the program, which is the reason why we have selected this approach.

We now briefly describe three possible variants of reference checking and in which way this leads to an improvement of protection in the context of the effects of illegal references, which were mentioned in Section 4.1.

Dereference Check (DRC): In this variant, 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 when one of the following bytecode instructions is executed: `putfield`, `getfield`, `*aload`, `*astore`, `arraylength`, `instanceof`, `checkcast`, `invokevirtual`, `invokeinterface`, and `invokespecial`.

A drawback is the possibility that the check will be executed more often than actually necessary – for instance, if the reference in question is used several times in succession but is kept within a processor register between the uses, the first check would suffice[2]. Also, corruption of a reference will not be detected until the reference is actually used. Reference comparisons in the application code may result in a wrong branch being taken.

Load Reference Check (LRC): In this variant, 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.

This approach offers an early error detection and needs fewer checks than the DRC variant. The disadvantage is that it has a higher false-negative probability, since local variables may be spilled to the stack in case of high register pressure or across the boundaries of method invocations.

Header-Only Check (HOC): A third conceivable possibility is to not check the reference ifself, but instead only test upon dereferencing if the reference points to a valid object header containing a certain bit pattern. For efficiency, it can be combined with the class-ID check described in Section 4.2.4. In case of a corrupted reference, it is likely that the target is not an object header but a random piece of data, given that object headers are relatively small (usually four bytes).

This approach is less safe than the LRC and DRC variants. Firstly, the check can result in false negatives if the data pointed to by the faulty reference happens to look like a legal object header. Secondly, the reference may be corrupted in a way that it points to another valid object, which may have a different type than the intended object. In both cases, type and memory safety

[2] Of course, the register may be temporarily spilled to the stack at any point in time by an interrupt service routine. Even so, it will stay in memory only for a very short time.

can no longer be maintained. It is even possible that the reference points into the memory belonging to a different protection domain. To prevent breaching the isolation, the runtime system would have to catch such cases by checking the reference against the domain's memory bounds or by using an MPU as a safety net. In theory, the chance of false negatives could be reduced by arranging the heap layout in a way that all valid references have a Hamming distance of at least two from each other. This approach would cause a higher memory clipping and raises the need to adapt garbage collection techniques, where fragmented allocation can be a possible solution. This may be a subject of further research, but will not further be examined here. In the remainder of this paper, we will focus on DRC and LRC rather than the HOC variant.

Choosing a variant for a concrete application scenario involves a tradeoff between protection level and costs. Various combinations of DRC and LRC are thinkable – among others:

- both DRC and LRC (slowest, but highest level of protection and early error detection)
- DRC only (faster, but no early error detection)
- LRC within leaf methods, DRC within other methods (even faster and only slightly less protection)
- LRC only (fastest, but lowest protection)

The reference checks are performed before any existing `null`- and array bounds checks so that the latter are executed on valid references.

4.2.2 Protecting the Current-Domain Pointer

As described above, the protection domain in which the current control flow is running is determined by a global pointer that points to the descriptor of the domain. To maintain type safety, that pointer must be checked whenever it is dereferenced. Such events include accesses to static fields and memory-management operations. To implement the verification of the pointer, one of the techniques for references to user objects described above is used analogously.

4.2.3 Memory Management

Memory managment by means of pseudo-static allocation is safeguarded by protecting the bump-pointer on access. Dependable GCs are part of our future work.

4.2.4 Protecting Object Headers

By encoding and checking object references, we are able to test the correctness of the memory pointers and preserve the validity of `null`-pointer checks. In case of bump-pointer allocation, references to deallocated memory are also prevented. However, memory safety is still not attained: Some operations which rely on meta-information stored within the object header definition affect type safety if that information is incorrect. For instance, they may cause writes beyond the object's boundaries. With the employment of more eloborate GCs, bogus object information may cause dangling references. Consequently, the integrity of the object headers must also be encoded and checked.

Each object header contains a class ID. In addition, array objects hold a field containing the array length. Optionally, management data used by the garbage collection is also present. Only if the meta-information is verified can type and memory safety be preserved.

As we do not make use of garbage collection in the scope of this paper, but rely on simple bump-pointer allocation, we do not need to secure any GC information. Thus, a header check is only necessary in the following scenarios:

Access to Type Information The class ID in the object header is used by `instanceof` and `checkcast` to determine if a given object is of a specific type. Illegally downcasting an object to a bigger type and then writing to one of the instance fields added by the subclass is guaranteed to write beyond the bounds of the object. Under normal conditions, such behavior is the result of a programming error, but it can also occur if a soft error affects the control flow. Consequently, to preserve type safety, such illegal downcasts have to be caught reliably. This requires checking the class ID before performing the actual type check.

Virtual Method Invocations The target for virtual method invocations is determined by performing a lookup in the global method table. While the table itself is constant and resides in ROM, the index for the lookup is computed using the class ID of the `this`-object. In case the class ID in the object header is affected by a transient error, the CPU will jump to an arbitrary wrong address. To prevent this, the runtime system must again make sure that the class ID is untouched.

Devirtualization, as described in Section 5.1, can help mitigating the costs of such checks.

Array Accesses An access to an array element always has to be within the array's bounds to guarantee memory safety. If soft errors are not an issue, this property can partially be verified by the compiler, or array bounds checks are necessary at runtime. Taking bit flips into consideration is two-fold:

On the one side, a soft error can affect the index and cause it to become too large or negative. This can be caught by a regular bounds check.

On the other side, the length information of the array can be corrupted, which invalidates any conventional array bounds check. In order to handle this issue, the existing array bounds check is extended to verify the validity of the array size before determining if a given index is within the array's bounds. Thus, this check incorporates an array-header check. It is referred to as *extended array bounds check* in the remainder of this paper. As a consequence, the compiler must be especially careful when deciding whether an array access can be left unchecked or not. The criteria for this optimization are presented in Section 5.1.

5. Efficient Implementation

Since our approach targets embedded systems, it is crucial to consider factors such as memory consumption, footprint and performance of the application while providing suitable protection for the application and runtime system. KESO's compiler optimizations facilitate creating a tailored runtime environment that suits the safety requirements to balance protection and cost. Less code and less memory usage also leads to a lower susceptability to soft errors, since the probability for a transient error to affect the application decreases with the memory usage. Static type-safe programs show a very good analyzability that allows for optimizations which are not possible if that type information is missing. KESO uses this information for its analyses and optimizations. Further optimizations through the C compiler also have a positive effect on the soft-error susceptability due to [11].

The reference runtime checks are inserted by KESO's backend, when optimization passes have already been run. We used and extended several analyses to gather the information necessary to eliminate, to simplify or to emit RCs. Characteristics of the hardware platform, e.g. if ROM is available, are also included in our analyses. In the following, an explanation on which checks have to be inserted and which can be erased is given.

5.1 High-Level Compiler Optimizations

KESO's static programming and system model enables the compiler to perform aggressive whole-program optimizations that would

be far less effective without a closed-world assumption. The optimizations are not only able to increase the application's runtime performance and reduce its memory footprint in general, but also have two additional effects that suit our purposes:

1. They reduce the attack surface for soft errors by decreasing the number of potentially error-prone operations such as memory accesses. The more information can be computed statically ahead of time by the compiler, the fewer operations have to be executed at runtime on the target system.
2. They mitigate the overhead introduced by the integrity checks discussed above.

In the following, we present selected compiler optimizations we implemented in *jino* that serve these purposes. Although modern C/C++ compilers offer comparable optimizations and we rely on such a C compiler as the final stage of our tool chain, the optimizations in *jino* make specific use of the high-level application knowledge available at the early compilation stages – for example, information about the system's designated entry points, about the assignment of tasks to protection domains, or about the target platform. Moreover, most of these whole-program transformations influence each other. Hence, we implement our own high-level transformations in *jino* while at the same time benefiting from the C compiler's low-level optimizations.

Constant Propagation: The constant-propagation algorithm in the KESO compiler is based on Wegman and Zadeck's interprocedural *Sparse Conditional Constant Propagation* [28]. Uses of variables which would normally reside in registers or on the stack are replaced with immediate values that are embedded directly in the code, reducing the number of potentially error-prone variable accesses. In addition, the register pressure is lowered, thereby reducing the need to spill registers to the (vulnerable) stack.

Copy Propagation and Variable Coalescing: Converting the intermediate representation back from SSA form is done using the algorithm proposed by Sreedhar [22], which coalesces variables and eliminates redundant copy instructions in the process. This, too, reduces register pressure and stack usage.

Dead-Code Elimination: Among other things, the *Sparse Conditional Constant Propagation* algorithm can fold conditional branches into unconditional jumps, and it can be used as a reachability analysis: Any basic block or entire method that was never visited is dead and can be safely eliminated. While this optimization itself does not directly contribute to better code performance, the other analyses and optimizations – for instance, method inlining or the rapid type analysis described below – benefit from its results.

Method Devirtualization: The principle of polymorphism in Java requires dynamic dispatch – that is, a method invocation whose target is not known at compile time is dispatched at runtime. As described earlier, this involves a lookup in the global dispatch table, plus an object-header check to enforce type safety in the face of soft errors. When translating the source code into bytecode, the Java compiler by default generates such `invokevirtual` instructions for all calls to an instance method of an object. The programmer could avoid this performance bottleneck by abstaining from the features of polymorphism, writing only static class methods. However, this would not suit the Java programming model very well.

To overcome this, the KESO compiler performs devirtualizations [1, 24] where possible. Since no additional classes can be loaded at runtime, the complete set of callee candidates for each call site is known ahead of time. Invocations whose candidate set contains a single element are converted into `invokespecial` instructions, which are bound statically and require neither a dispatch-table lookup nor an associated integrity check of the object header.

Whole-program analyses allow to further shrink the candidate sets, possibly yielding more single-element sets and thus increasing the effectiveness of the devirtualization optimization. In KESO, we apply a combination of class-hierarchy analysis (CHA) [12] and rapid type analysis (RTA) [3]. CHA analyzes the data flow of reference variables and tries to determine the dynamic types of the referenced objects at call sites as specifically as possible, whereas RTA purges all candidate sets of methods whose class is never instantiated – profiting from the elimination of dead object-allocation sites.

Method Inlining: For method invocations that can be bound statically, the compiler can choose to embed the body of the callee at the call site, eliminating the overhead of the function call. After inlining, it is worthwhile to re-run the constant-folding and -propagation pass because it is now possible to specialize the embedded method body according to the concrete arguments passed at the original call. At the same time, the arguments passed by that call no longer have to be considered when re-analyzing the callee method, which may in turn be further optimized.

Runtime-Check Elision: As the consistency checks are often coupled with regular runtime checks, the efficiency of the compiled code profits from the regular check-elision optimizations performed by the compiler. `checkcast` instructions are eliminated if the data-flow and type analysis proves the respective reference to always be of the correct type – for example, if it follows an `instanceof` case differentiation. In this case, the object-header check preceding the `checkcast` is unnecessary as well.

For array accesses, the following cases have to be differentiated:

1. Both the index and the array size are constant (possibly thanks to constant propagation) and the index is within bounds. Consequently, no bounds check is needed.
2. The array size is constant, but the index is variable. A simple bounds check is sufficient since the size is not read from the array header.
3. The data-flow analysis proves that the access will always be within bounds, but one of the two values is not constant. This can be the case when iterating over an array in a canonical `for`-loop, for instance. In a scenario without fault-detection/tolerance requirements, the bounds check could be elided. In our case, however, the access index may have been corrupted or the array may have been created with a wrong, possibly too small size. Hence, an extended array bounds check must be emitted.
4. None of the above conditions is met. An extended array bounds check has to be inserted.

If it is sufficient to maintain spatial isolation, the restrictions can be somewhat loosened: It can be argued that extended array bounds checking is not necessarily required if reading from an array that contains primitive data. Reading a wrong value would lead to wrong application data, but such data errors could be caught using additional application-specific safety measures. Other protection domains would not be affected. However, if the array contains object references, reading from it must be protected with an extended bounds check, since reading from an invalid position might return a reference pointing into the heap of another protection domain.

ROM Allocation: One of the major drawbacks of Java as a programming language for embedded systems is its insufficient handling of constant data. While there is the keyword `final` to mark variables of primitive type or references themselves as immutable, there is no equivalent concept for the contents of an object. This is especially cumbersome for arrays containing primitive constant values: Such arrays are allocated on the heap and initialized at the time the class is loaded – one element at a time. This scheme induces a number of disadvantages:

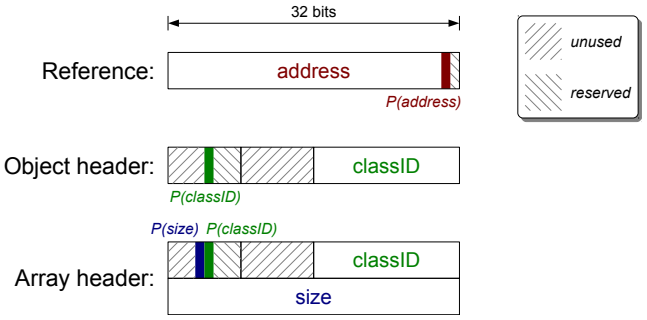

Figure 3: Encoding of parity information on a 32-bit platform

- The explicit initialization code – especially for arrays – needlessly inflates the text segment.
- On small embedded devices, RAM is scarce. Placing constant array data in the ROM instead could save precious memory.
- According to our fault hypothesis, integrity checks (object/array header) for ROM-allocated objects could be elided.
- Protection redundancy techniques for application data can profit from a reduced replication set.

A static analysis could find objects and arrays with immutable contents and mark them as ROM-allocatable, provided this is supported by the target platform. This currently being implemented in KESO and promises to be a worthwhile optimization.

5.2 Incorporation of Platform-Specific Features

Tailoring the KESO runtime environment to a concrete application scenario also involves awareness of the underlying hardware platform. Thus it is possible to adapt the runtime system to the conditions of the target platform and to make use of specific hardware features where available.

Alignment and Address Layout: For 1-bit error detection using parity information, one additional bit is needed for every word that is to be protected. Since every object on the heap is aligned at a minimum of four bytes, each valid pointer always has its two least-significant bits set to zero. In KESO, if a garbage collector is present, it reserves the lowest address bit for its purposes, leaving the second-lowest bit free for storing the parity information. This allows parity-encoded references to be represented as regular pointer variables that can be loaded and stored with a single memory access.

Given certain platform characteristics, the implementation of other, more complex fault-tolerance mechanisms such as ECC (which uses more than one redundancy bit) would not necessarily require inflating the reference size either. For example, the TriCore TC1796 platform has a 32-bit address space but only 1 MiB of physical RAM. Depending on the concrete memory mapping, this would allow up to 12 additional bits in each 32-bit pointer variable to be used by the runtime system.

For the headers of regular objects and of arrays, the parity information can be stored in unused bits of the header as depicted in Figure 3. Consequently, enabling parity-based error detection in KESO does not increase the memory footprint of the application.

Processor Instruction Set: Many processor architectures have special instructions for efficiently computing the parity of a data word. If such instructions are available on the target platform, KESO makes use of them by generating calls to the GCC built-in function `__builtin_parity()`. For instance, on the TriCore platform the parity computation itself amounts to a mere four CPU instructions as can be seen in Listing 1.

```
keso_check_and_decode_reference:
    mov.d   %d4,  %a4
    mov.d   %d15, %a4
    parity  %d2,  %d4       ; Compute parity
    bsplit  %e2,  %d2
    bsplit  %e2,  %d2
    parity  %d2,  %d2
    jz      %d2,  .Lsuccess ; Catch parity error
    nop
    call    keso_throw_error
.Lsuccess:
    andn    %d15, %d15, 2   ; Decode address
    mov.a   %a2,  %d15      ; Return it
    ret
```

Listing 1: Reference checking and decoding on the TriCore MCU

Memory-Protection Unit: If an MPU is available, KESO and the underlying AUTOSAR OS can be configured to use it as an additional safety net. If a severe error were to cause the type system to become corrupted and memory safety could no longer be guaranteed, the MPU could maintain the isolation of the protection domains. To achieve this, KESO physically groups the memory portions belonging to a domain (heap, stacks, etc.) in memory and provides the OS with the start and end addresses of these regions.

6. Evaluation

In this section, we evaluate the costs imposed by our reference-checking approach (Section 6.2) and the protection provided by it using the Fail* fault-injection framework, which is explained in Section 6.3. As an exemplary application, the Collision Detector (CD_x) benchmark is employed and a brief introduction to it is presented in Section 6.1.

6.1 The CD_x Benchmark

For our fault-injection and performance evaluation, we use the *Collision Detector (CD_x)* [16] – an open-source benchmark that is available in a C (CD_c) and a Java (CD_j) version with almost equivalent algorithmic behavior – as a representative Java application for embedded systems. KESO bundled with CD_j has already been evaluated against CD_c and results can be found in [23].

The core of the CD_x benchmark 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 [16]. Since CD_j allocates temporary objects and uses collection classes of the Java library, it normally requires the use of dynamic memory management. Since protected garbage collection in KESO is currently a work in progress, we use pseudo-static allocation instead.

6.2 Overhead to Unprotected KESO

To determine the overhead imposed by a secured type system, we use CD_x in the *onthegoFrame* variant configured processing 24 frames (it would run out of memory after that), which is suitable to be deployed on the Infineon TriCore TC1796 device (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. Our experiments cover the LRC reference-checking variant. Checks were statically inlined.

	NoChk	SafeChk	AddrChk	HdrChk	LRC
text	45246	49044	59634	61810	64310
data	4005	4005	4109	4109	4109
bss	836138	836138	836138	836138	836138
text ov	-7.74%	0.0%	17.76%	4.60%	24.33%
data/bss ov	-2.53 %	0.0%	0.0%	0.0%	0.0%

(a) Memory footprint

	SafeChk	LRC
NullChecksEmitted	153	153
BoundsCheck	91	0
BoundsCheckElided	15	0
BoundsCheckKnownIndex	12	0
BoundsCheckFullExtended	0	91
BoundsCheckKnownSizeExtended	0	27
ReferenceCheckEmitted	0	301
HeaderCheckEmitted	0	97

(b) Runtime-check emission

	NoChk	SafeChk	AddrChk	HdrChk	LRC
Overhead	-11.84%	0%	18.42%	8.33%	30.71%
# Checks	0	0	225715	69936	417069

(c) Runtime overhead and number of LRC executions

Figure 4: Overhead induced by LRC

Footprint. Figure 4a shows the footprint of various KESO checking variants. `NoChk` denotes that runtime safety checks, i.e. `null` and array bounds checks are disabled. However, the memory allocator still checks the bump pointer against the heap bounds to prevent a heap overflow. The generated code is not memory-safe at those locations where *jino* was not able to statically prove the validity of those accesses. This CD_x variant is comparable to a version directly implemented in unsafe C.

The `SafeChk` variant follows the Java specification and is memory-safe in the absence of soft errors, wherefore it is selected as the baseline variant. `AddrChk` (address checks), `HdrChk` (header checks) and `LRC` are built on top of `SafeChk` (combined address, `null`, header and extended array bounds checking). Extended array bounds checks are applied to all array types (not just reference arrays) at necessary locations.

Class-storage information and the method table are located in the text segment in all variants. Although the parity information itself does not increase the memory footprint of the application, the data segment grows by 104 bytes when type-system protection is enabled. This is caused by a missed optimization: Two constant strings in the application are actually dead, but their headers and/or references are encoded at startup, respectively. Hence, the linker regards them as used and is unable to discard them. As expected, the bss segments, containing the heaps and stacks, are of equal size in case of LRC and *unprotected* configurations (safety checks enabled only, no protection against soft errors).

The text segment is inflated by 7.74% due to safety checks. Supplementarily, LRC causes an increase by a total of 24.33%, where 17.76% are caused by address checks and an additional 2.90% by combining them with object-header checks. Extended array bounds checks enlarge the the text segment by 3.67%.

The header checks reuse code from address checks, hence the aggregated code-size increase (20.65%) is smaller than that of the individual parts (17.76% for address and 4.60% for header checks).

KESO performed devirtualizations on 343 method invocations, omitting the header and address checks while retaining type safety, whereas 22 non-static method invocations remained. Seven `instanceof` occurrences were statically dissolved, which allowed to omit header-check insertion at those locations.

Figure 4b lists the emitted runtime checks. The number of `null`-checks is unchanged. In the protected variant, 118 *extended* array bounds checks were emitted: 91 of those array bounds checks include the size-integrity check, whereas the remaining 27 array checks are performed on arrays with constantly known size, but varying indices.

Unprotected KESO includes 91 *normal* array checks whereas 15 checks could statically be computed to always succeed. Assuming soft errors, these 15 array bounds checks have to be inserted as extended array bounds check (discussed in Section 5.1). In addition, 301 reference checks and 97 object-header checks (76 on non-static method invocations, 21 on `instanceof` and `checkcast`) were emitted in protected KESO.

Runtime. The runtime overhead of LRC is listed in Figure 4c: 18.42% of the increased demand were contributed by address checks and 8.33% by object-header checks. The additional runtime of 3.96% was caused by extended array bounds checks (including 121838 size-integrity checks), which are needed to accomplish full LRC.

Most of the bounds checks that contribute to the increased runtime reside within loops iterating over arrays. In a scenario without soft errors, these checks could be easily elided, but in our fault-detection scenario they have to be kept because the loop index could be corrupted. *jino* is able to statically determine the array size in some cases, which results in a light-weight array bounds check with constant and known size. This check detects corrupted indices. If this optimization is not possible, a full extended array bounds check is emitted.

As a possible future optimization, sufficiently small loops that iterate over an array of known size could be unrolled by the compiler. Since both index and size would be constant, the extended array bounds checks could then be omitted.

6.3 The Fail* Fault-Injection Framework

In order to get an insight into the effects on the protection level provided by the reference-checking extension, the fault-injection (FI) framework Fail* [21] is used. It is currently available for the Bochs simulator [17] 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 on the application and software-based isolation.

For the FI campaign, garbage collection is disabled and pseudo-static allocation is used to exclude effects of bit flips that occur during the unprotected GC phase. Faults that occur during the selected pseudo-static allocation strategy are, however, considered in our experiment. Hardware-based memory protection is also 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 256 KiB, which is enough to compute a set of 5 frames in CD_x before it runs out of memory.

We injected single bit flips into each bit position of each word of the allocated heap space. 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. 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 was applied to three variants of the system: No runtime checks at all (`NoChk`, program is not type-safe), `SafeChk` (type-safe, unprotected against soft errors), and a full LRC-hardened variant, resulting in a total fault space of approximately 4,341,038 experiments.

Table 1 shows the overall results of the FI campaigns grouped by the reachable destination points. In the `NoChk` variant, approximately 64% of the injections resulted in *No Effect* and 56% for

Result	NoChk	SafeChk	LRC
No Effect	809,918	820,835	867,205
Error Exception	39,655	109,312	37,747
Null-Pointer	0	116,245	43,287
Out-of-Bounds	0	62,926	29,992
Illegal Memory Access	289,159	246,981	17,286
Trap	72,816	56,824	6,084
Timeout	44,762	43,453	37,916
Parity Exception	0	0	588,635
Total	1,256,310	1,456,576	1,628,152

Table 1: Fault-injection results

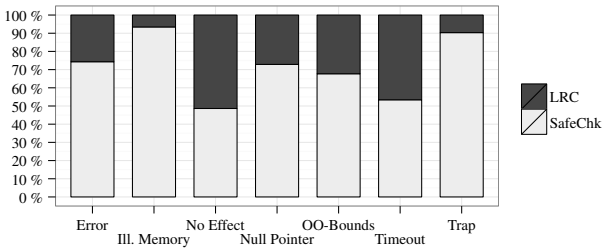

Figure 5: Proportional distribution of the fault-injection results

`SafeChk` and 53% for LRC, respectively. Here the faults were either masked or silently corrupted the application's data. The LRC variant caused CD_x to pass through without triggering exceptions or traps in some cases: As a result, effects of bit flips could be mitigated on the application layer, where additional application-protection mechanisms can recognize faulty data or control-flow errors (e.g. through replication or control-flow monitoring). Assuming a redundancy and recovery approach as presented in [26], the recovery mechanism is able to restore both application data and respective valid references, as spatial isolation between replicas is maintained.

Comparing the type-safe system (`SafeChk`) against the plain memory-unsafe version (`NoChk`), it can be concluded that safety checks in form of `null`-pointer and array bounds checks already detect some corrupted data and references and reduce illegal memory accesses and traps. However, the triggering of these exceptions is more an effect of the executions of safety checks on invalid application data and references. Regarding these inherent fault-detection mechanisms of the KESO runtime system, the LRC variant reveals an overall decrease of these exceptions, since LRC found corrupted references that illegally triggered those exceptions in many cases. Others were still raised due to, for example, corrupted application data causing to select another control-flow path.

The portion of *Null Pointer*, *Out of Bounds* and *Error* exceptions (caused by failed `checkcast` or heap-memory bounds errors) and hardware *Traps* is considerably higher in the unprotected system, as depicted in Figure 5. Here, the parity check can detect the injected fault before the error can propagate and result in a hardware trap or error exception.

The extended array bounds check detected 29,992 errors, whereas 588,635 errors were found by additional parity checks. A certain amount of bit flips resulted in a *Timeout* behaviour. These errors influenced a loop-controlling variable residing in the unprotected application data. The *Traps* occurring in the LRC variant were division-by-zero exceptions, which are also caused by the application's computation with faulty operands. Such errors can be handled by fault-tolerance measures at the application level. *Traps* in the unsafe `NoChk` variant were mainly induced by dereferencing `null`-pointers (address 0), which leads to a trap on the x86 architecture.

The FI experiments also caught any *Illegal Memory Access*, that is, any access beyond the defined sections and writing accesses into the text section. With LRC, these illegal memory accesses can be traced back to failed executions of `instanceof`. Some bit flips were injected after the reference check and cause the reference to point to a valid object. The probability of that happening increases with the heap size and the number of objects located there. The object-header check is therefore unable to detect an error, thus producing a wrong outcome of `instanceof`. A wrong object type is assumed and type safety is corrupted, which leads to the illegal memory usages. This drawback of LRC in contrast to DRC was discussed in Section 4.1.3. Nevertheless, LRC decreased memory isolation violation by 94% (-271,873) compared to `NoChk` and 93% (-229,695) compared to `SafeChk`. Based on LRC, application replication as provided by KESO to protect the application data can be applied on top of our approach. As LRC retains software-based memory protection to a certain degree, hardware-based isolation is not necessary, but can complement LRC if required.

7. Related Work

Reliability in JVMs has been addressed in a few existing projects. Friedman and Napper [13, 18] have their focus on the replication of the entire JVM to tolerate fail-stop errors in distributed systems. In [9] the susceptability of application data in JVMs and their protection was analyzed, targeting JVMs for workstations.

Chen [10] 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 subheaps which contain the latest changes and those which have not changed. Unchanged heap parts and moving new heap data is protected by a memory management unit (MMU), however, type safety is not retained. 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 any research of type-system protection. Besides reduced illegal memory accesses, our approach was able to detect and signal some bogus behavior in the application. Additional application-specific fault-tolerance measures based on static analyses on static type-safe programs can be employed on top of our approach. In [5], it was quantitatively analyzed how an exemplary piece of software reacts to corrupted virtual function calls in the context of C++ and the authors propose to protect the virtual function pointer by a dependability aspect that is applied by the AspectC++ weaver. In their experiments, 75% of all bit flips in virtual function pointers led to an application crash. In contrast to this approach, safeguarding of virtual function calls is included by protecting the type system itself. Thanks to static application knowlegde and type information, an efficient protection of method calls is possible.

8. Conclusion and Future Work

We presented a possible solution to protect the type system of a strongly typed programming languange in the presence of soft errors. The approach targets embedded systems and allows to continue relying on the benefits of Java to the extent discussed in the evaluation section. The effects of soft errors on the type system and software-based memory protection built on top of the type system have been analyzed exemplarily in the context of the KESO JVM. By means of LRC, it is more likely to retain software-based spatial isolation needed for many fault-tolerance techniques such as replication without the need for an MPU. Moreover, LRC can retain type safety and allows for an early error detection at the granularity of objects. The overhead imposed by our solution is dependent on the application itself (e.g. frequent array usage). Also, it is bound to the effectiveness of compiler optimizations, that is, the extent to which they are able to elide dynamic program information that can

be corrupted during execution. To the best of our knowledge, we presented and evaluated the first implementation of type safety and software-based memory protection in the presence of soft errors.

There are several aspects which we would like to cover in our future work. At first, there is the evaluation of the more expensive DRC variant, which is able to detect more illegal memory accesses. Our RC variants are easily adaptable to detecting and tolerating more bit flips at a time, so an evaluation of these scenarios is also in progress. Secondly, KESO's optimizations will be extended: The handling of `checkcast`, for example, will be improved. Also, we plan to implement the idea of possible loop unrolling allowing for the elision of expensive extended array bounds checks. Error-detection support for the program counter by means of automated use of control-flow information available in KESO will be examined.

Up to now, we have had to use the protected pseudo-static memory allocation technique. The available GCs are currently extended to detect and tolerate soft errors. We are interested in the overhead imposed by dependable garbage collection, which also facilitates to significantly reduce the heap size and thus the attack surface of the heap. A ROM allocation analysis is currently being developed to be able to place more constant data in ROM to decrease RAM usage. Also, the effects of an *extended escape analysis* – which allows for stack allocation and moreover an automated application of RTSJ's [4] `ScopedMemory` – will be taken into consideration.

References

[1] G. Aigner and U. Hölzle. Eliminating virtual function calls in C++ programs. In *10th Eur. Conf. on OOP (ECOOP '96)*, pages 142–166, London, UK, 1996. Springer. ISBN 3-540-61439-7.

[2] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus. Deconstructing process isolation. In *MSPC '06: Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, pages 1–10, New York, NY, USA, 2006. ACM. ISBN 1-59593-578-9. doi: 10.1145/1178597.1178599.

[3] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. *SIGPLAN Not.*, 31(10):324–341, 1996. ISSN 0362-1340. doi: 10.1145/236338.236371.

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

[5] C. Borchert, H. Schirmeier, and O. Spinczyk. Protecting the dynamic dispatch in C++ by dependability aspects. In *Proceedings of the 1st GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '12)*, Lecture Notes in Informatics, pages 521–535. German Society of Informatics, Sept. 2012.

[6] C. Borchert, H. Schirmeier, and O. Spinczyk. Generative software-based memory error detection and correction for operating system data structures. In *Proceedings of the 43nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*. IEEE Computer Society Press, June 2013.

[7] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6): 10–16, November 2005. ISSN 0272-1732. doi: 10.1109/MM.2005.110.

[8] G. Cellere, S. Gerardin, M. Bagatin, A. Paccagnella, A. Visconti, M. Bonanomi, S. Beltrami, P. Roche, G. Gasiot, R. H. Sorensen, A. Virtanen, C. Frost, P. Fuochi, C. Andreani, G. Gorini, A. Pietropaolo, and S. Platt1. Neutron-induced soft errors in advanced flash memories. In *IEDM 2008*. IEEE, Feb. 2009. ISBN 978-1-4244-2378-1.

[9] 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 *Java Virtual Machine Research and Technology Symposium*, pages 67–78, Berkeley, CA, USA, Apr. 2001. USENIX. ISBN 1-880446-11-1.

[10] 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. doi: 10.1145/1065579.1065761.

[11] J. J. Cook and C. B. Zilles. A characterization of instruction-level error derating and its implications for error detection. In *DSN*, pages 482–491. IEEE, 2008. doi: h10.1109/DSN.2008.4630119.

[12] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *LNCS*, 952:77–101, 1995.

[13] R. Friedman and A. Kama. Transparent fault-tolerant java virtual machine, 2003.

[14] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder. The JX operating system. In *2002 USENIX ATC*, pages 45–58, Berkeley, CA, USA, June 2002. USENIX. ISBN 1-880446-00-6.

[15] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante. *Software-Implemented Hardware Fault Tolerance*. Springer, Heidelberg, Germany, 2006. ISBN 0-387-26060-9.

[16] 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. doi: 10.1145/1620405.1620412.

[17] K. P. Lawton. Bochs: A portable pc emulator for unix/x. *Linux Journal*, 1996(29es):7, 1996.

[18] J. Napper, L. Alvisi, and H. Vin. A fault-tolerant java virtual machine. In *In Proceedings of the International Conference on Dependable Systems and Networks (DSN 2003), DCC Symposium*, pages 425–434, 2002.

[19] S. Poledna, A. Burns, A. Wellings, and P. Barrett. Replica determinism and flexible scheduling in hard real-time dependable systems. *IEEE TC*, 49(2):100–111, 2000. ISSN 0018-9340. doi: 10.1109/12.833107.

[20] H. Schirmeier, R. Kapitza, D. Lohmann, and O. Spinczyk. DanceOS: Towards dependability aspects in configurable embedded operating systems. In A. Orailoglu, editor, *3rd HiPEAC W'shop on Des. f. Reliability (DFR '11)*, pages 21–26, Heraklion, Greece, Jan. 2011.

[21] H. Schirmeier, M. Hoffmann, R. Kapitza, D. Lohmann, and O. Spinczyk. 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.

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

[23] M. Stilkerich, I. Thomm, C. Wawersich, and W. Schröder-Preikschat. Tailor-made JVMs for statically configured embedded systems. *Concurrency and Computation: Practice and Experience*, 24(8):789–812, 2012. ISSN 1532-0634. doi: 10.1002/cpe.1755.

[24] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. *SIGPLAN Not.*, 35(10):264–280, 2000.

[25] A. Taber and E. Normand. Single event upset in avionics. *IEEE Transactions on Nuclear Science*, 40(2):120–126, Apr. 1993. ISSN 0018-9499. doi: 10.1109/23.212327.

[26] I. Thomm, M. Stilkerich, R. Kapitza, D. Lohmann, and W. Schröder-Preikschat. Automated application of fault tolerance mechanisms in a component-based system. In *JTRES '11: 9th Int. W'shop on Java Technologies for real-time & embedded Systems*, pages 87–95, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0731-4. doi: 10.1145/2043910.2043925.

[27] P. Ulbrich, R. Kapitza, C. Harkort, R. Schmid, and W. Schröder-Preikschat. I4Copter: An adaptable and modular quadrotor platform. In *26th ACM Symp. on Applied Computing (SAC '11)*, pages 380–396, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0113-8.

[28] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13:181–210, Apr. 1991. ISSN 0164-0925. doi: 10.1145/103135.103136.