Escaping the Bonds of the Legacy: Step-Wise Migration to a Type-Safe Language in Safety-Critical Embedded Systems

Michael Stilkerich, Jens Schedel, Peter Ulbrich, Wolfgang Schröder-Preikschat, Daniel Lohmann
Department of Computer Science 4 - Distributed Systems and Operating Systems
Friedrich-Alexander University Erlangen-Nuremberg
{stilkerich,schedel,ulbrich,wosch,lohmann}@cs.fau.de

***Abstract*—Type-safe high-level languages such as Java have not yet found their way into the domain of deeply embedded systems, even though numerous attempts have been made to make these languages cost attractive. One major challenge that remains is the huge existing code base in many industries. Completely reengineering this code base is not viable for cost and time reasons. We present an approach that allows to isolatedly combine legacy software components and safe software components in an embedded system using the two most common communication idioms found in this domain. Our approach allows the developer to freely choose between hardware- and software-based isolation mechanisms. We demonstrate the feasibility of our approach by porting a non-trivial part of a real-world, hard real-time embedded avionics application. Our results show that the cost of this mixed-mode operation is on the same scale as the pure operation.**

***Keywords*-Real-time and embedded systems; Protection mechanisms; Reliability; Java;**

I. Introduction

Modern, type-safe high-level languages such as Java have become quite widespread in many computing domains from enterprise computing to less resource-constrained embedded systems. Java provides a number of benefits compared to low-level languages such as C: Case studies [1], [2] found software development in Java to be more productive and that Java avoids or detects many common programming errors such as out-of-bounds array accesses or dangling references. Software-based memory safety can easily be built on top of Java, an important benefit particularly for vertically organized industries where software components of different suppliers are integrated and software defects need to be contained in the faulty component. Looking at the domain of real-time systems, code written in a type-safe language tends to be a good subject to static analyses.

A. The Issue

Although Java has appeared back in 1995 and many research efforts have been done to overcome its limitations with respect to low-level programming, resource consumption and predictability [3], software targeting statically configured, deeply embedded systems is still almost exclusively developed in unsafe low-level languages such as C, Assembler and some C++. Java is basically nonexistent in this domain.

Searching for the reasons we looked at a particular example for such an industry, the electronic control units (ECU) found in modern cars. The amount of software in cars has been growing exponentially over the last 30 years, and reached a volume of more than ten million lines of code in a premium car [4]. Software functions and the electronics necessary therefore make up to 40% of a car's production costs [5].

The automotive industry is vertically organized. Electronic functions are normally not developed at the car manufacturer (OEM) but provided by a multitude of component suppliers, which traditionally ship the electronic component bundling both hardware and software as a black box. The OEM needs to integrate these ECUs to form a cooperating network. With currently about 80 ECUs in a premium car this integration has become increasingly difficult and costly for the OEM. This triggered a paradigm shift from the federated to an integrated architecture where multiple software functions are cohabited on one microcontroller platform. The cohabitation introduces new requirements to the underlying system software, which led to the development of AUTOSAR OS [6] that extends its widely used predecessor OSEK/VDX [7] particularly in the areas of temporal and spatial isolation.

B. The Causes

Reasoning with automotive developers about the causes for the reluctance to adopt a modern language, we identified the following concerns:

1) Apprehension of increased hardware costs: High-level language features such as dynamically-bound method calls and safety checks (e.g., null checks, array bound checks, type checks) are commonly perceived as major sources of runtime overhead, which leads to the association of Java and slow/expensive in the minds of many embedded developers. Much of this runtime overhead can, particularly in the domain of statically-configured software, however, be eliminated by means of static analyses. Concerning the in importance gaining spatial isolation of applications, there is a common misbelief among many software developers that hardware-based protection comes for free while software-based protection incurs the overhead of a type-safe language.

2) Huge existing code base: The second, maybe more significant issue, is the huge existing code base that has developed over the years and, for both cost and time reasons, cannot completely be re-engineered in a new language.

C. Proposal: Mixing C and Java Applications on an MCU

In this paper, we propose the concurrent operation of both C and Java applications targeting statically-configured embedded systems. Statically configured here means that the entire code base as well as all operating system (OS) level objects (threads, locks, events) are known at compile time. Besides enabling the development of new components in Java while preserving the existing investment, our approach also enables the incremental migration. This is a prerequisite for any new technology for being accepted by many industries.

To evaluate the overhead and test the feasibility of our approach, we have ported the core component of a complex real-world quadrocopter control application to Java and flew the aircraft with our ported component combined with the remaining original components. Our contributions are:

- The ability to directly compare the costs of hardware- vs. software-based memory protection by making both application-independent configurable properties of the system software.
- The quantitative evaluation and qualitative experiences of software- vs. hardware-based memory protection and low-level vs. a safe high-level language at the example of a complex real-world embedded application on hardware and system software that is typical for the domain of deeply embedded systems.
- A mixed operation of newly developed or re-engineered safe components with existing legacy software at a price that is comparable to the pure operation of the unsafe code components.

II. Spatial Isolation Mechanisms

Spatial isolation can be realized either software-based or by employing a hardware protection unit. Software-based isolation normally involves checking some or all memory accesses of the application at runtime. While there are software-based approaches that can be applied to binary code [8], approaches based on a type-safe language can constructively provide spatial isolation to a large degree and only require a small amount of runtime checks. Regarding hardware-based memory protection, the memory protection units (MPU) found on some low-cost chips for deeply embedded systems normally work by restricting the memory access to regions of the physical address space. The number of these regions is limited by the number of ranges that the specific MPU implementation supports. Memory management units (MMU) that support virtualization of the physical memory by providing logical address spaces are commonly not found in the domain of deeply embedded systems for cost reasons and predictability issues.

Comparing MPU- versus software-based spatial isolation, each approach has advantages in some aspects: Isolation based on the use of an MPU

- incurs no overhead while the execution stays within the context of a particular application, whereas software-based protection adds overhead by runtime checks.
- is less vulnerable to transient hardware errors, since the protection concept relies on fewer memory locations that normally reside in radiation-hardened areas.

Isolation based on the use of a type-safe language

- does not require a special hardware unit, which enables the use of cheaper microcontrollers.
- is more flexible with regards to the granularity of protection provided. MPU protection is limited to a fixed, small number of memory regions that need to include all the memory that is accessed by a particular application.
- allows for efficient communication among applications and system calls, since no protection-mode change is necessary.
- not only provides spatial isolation, but can also detect memory errors within a particular application such as out-of-bounds array accesses.

The choice for a particular protection scheme consequently highly depends on the characteristics of the application. It may be reasonable to use different schemes for different components of an application. Even for a given application the choice may be dependent on external factors such as the deployment scenario (e.g., testing vs. release development phase, use in environments with differing safety requirements), thus the isolation scheme should not be hard-coded in the application, but be a configurable property.

III. Application Model

Our application model is similar to that specified in AUTOSAR OS. The application software is structured in so-called *OS applications*. An OS application can be a distinct logical application, but it is also possible to separate the software components of an application into multiple OS applications. As to our application model, this distinction makes no difference. In the following, we will simply speak of applications.

Applications span the realms of spatial isolation. Each application control flow (task/thread, interrupt service routine (ISR)) is assigned to exactly one application. Every portion of memory that is accessed by applications also belongs to exactly one application. The interaction and data exchange among applications is exclusively performed by using OS services.

We distinguish trusted and non-trusted applications. Trusted applications together with the kernel compose the

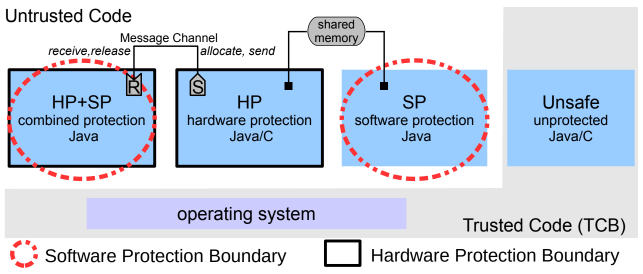

Figure 1. Application Model

trusted computing base. Control flows that belong to a trusted application are not subject to the enforcement of spatial isolation. On the other hand, control flows within non-trusted applications may only access memory regions that belong to the application. Restricting read accesses is not required from a safety perspective, but could provide an earlier detection of certain software defects.

A. Isolation Schemes

Spatial isolation is either enforced by an MPU or constructively by the use of a type-safe language. Figure 1 gives an overview of the isolation types that are available in our model. All four types can be derived from a Java application, while only two types apply to C applications.

Hardware-Based Protection (HP): The application's memory accesses are checked to affect only the data segments of the current application by employing an MPU. A Java application falls in this category if the runtime checks are disabled but MPU protection is turned on.

Software-Based Protection (SP): Software-based protection is based on an application written in Java and a Java Virtual Machine that supports a mechanism for the isolated execution of multiple applications, for example by implementing the Java Isolation API [9].

Combined Protection (HP+SP): Hardware- and software-based memory protection may also be combined by additionally configuring the MPU regions for a Java application. This type of protection combines the strengths but also the costs of both isolation mechanisms.

Trusted Applications (Unsafe): Spatial isolation is not enforced for trusted applications. This type of application can be a C application running without memory protection or a Java application that runs with runtime checks disabled.

B. Data exchange among applications

We support two OS abstractions for the exchange of data among different applications. For both mechanisms, the data exchange paths among different applications are defined statically.

1) Shared Memory: Shared memory is a typed area of memory that is made accessible to multiple applications. The applications directly access this area without using explicit OS primitives, thus completely bypassing the OS. This mechanism has a number of problems, for example it requires the applications to ensure the proper synchronization of accesses to the area. Nevertheless, shared memory is a data exchange mechanism that is found in many legacy applications. For this reason, we decided to support it in our model. Access to the shared memory area is permanent for all accessing applications, thus it does not require protection mode changes or MPU reconfigurations when accessing the area.

2) Messaging: The second data exchange mechanism is message-oriented and similar to comparable mechanisms found in other operating systems for our target domain (e.g., OSEK COM [10]). The mechanism allows the definition of message channels between different applications. The sender side of the channel uses the primitives `allocate` to request a message buffer from the operating system and `send` to send the message to the receiving end. Sending a message triggers an ownership transfer of the message from the sending application to the receiving one. The receiver side uses the primitives `receive` to receive a message and `release` to release the message buffer to the operating system after having processed the message data. Both ends must only access a particular message between the two respective OS primitives. Consequently, when using MPU protection, the MPU has to be reconfigured upon invocation of any of these primitives to grant or revoke the access to a message. The message protocol provides implicit synchronization, since it ensures that only one application can access a particular message at a time, even if the OS chooses to use the same memory for the send and receive buffers.

IV. Implementation

Our prototype uses CiAO [11], a family of embedded operating systems that supports an API similar to that of AUTOSAR OS. CiAO aims at providing a high level of static configurability of even fundamental properties of the system by employing aspect-oriented programming (AOP) [12], thereunder MPU-based memory protection. The MPU protection in CiAO has been developed specifically with focus on safety rather than security and is highly optimized for this domain. Details on the implementation of MPU protection in CiAO are available in a separate paper [13].

To support Java applications, we use KESO [14], a JVM implementation for deeply embedded systems. KESO requires static applications and does not support all features of the Java 2 standard platform, most notably the dynamic loading of classes. Rather than defining a fixed profile, that defines a subset of JVM features that are supported by the runtime, KESO's approach is to use the available ahead-of-time knowledge to create a Java virtual machine with a feature set that is specifically tailored to the particular application, thereby reducing the overhead to the necessary minimum without statically restricting the available feature set too much. KESO supports the isolated execution of different applications. This is achieved by a strict logical

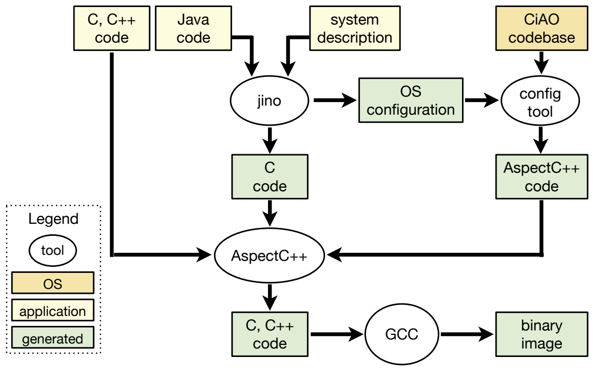

Figure 2. Toolchain

separation of the object heaps of the different applications, and the maintenance of a separate set of global variables (i.e., the static class fields in Java) for each domain. These measures are sufficient to eliminate any shared data. From the viewpoint of the application, each application seems to be executing in a JVM of its own.

KESO compiles Java bytecode to C code that includes the Java runtime and runtime checks. This enables us to leave CiAO completely oblivious of the Java code running on top. To CiAO, the Java applications compiled to C code appear as native C applications.

A. Toolchain

Figure 2 shows the Toolchain to illustrate how CiAO and KESO are combined. The applications are provided as C/C++ and Java source code. In addition, a system description is provided by the system integrator that contains the static configuration information on the complete system (e.g., task and application definitions with their attributes, assignments of tasks to applications). KESO's Java-to-C compiler *jino* compiles the Java applications to C code and generates the static configuration for CiAO from the system description. The generated CiAO configuration is used by CiAO's configuration tool to generate a CiAO variant tailored to the feature requirements of the applications at hand. The implementation language of CiAO is AspectC++ [15]. The AspectC++ weaver performs a source-to-source translation of both the OS and the application code to apply the advice given by the aspects in the generated CiAO variant. Finally, the woven C++ code is compiled by a C++ compiler and linked to the binary image.

B. Extensions to the KESO Multi-JVM

For our prototype, we extended KESO to support CiAO's MPU-based memory protection and low-level communication mechanisms. For the MPU-based memory protection, we needed to physically group all data that belong to an application in memory. This is basically achieved by physically separating the heaps of the different applications (i.e., statically partitioning the available heap space) and to group it with the static field instances of each application. This process is transparent to the software developer and completely handled by the compiler.

To access CiAO's shared memory and messaging communication mechanisms from a Java application, we created wrapper APIs using KESO's native interface. These APIs enable access to the shared memory or message area through the base abstraction of `RawMemory` as defined by the RTSJ [3]. `RawMemory` provides a low-level interface for accessing *primitive* data that resides outside Java's managed memory areas and is often used for writing device drivers. The area is of a fixed size and every memory access to this area requires an explicit offset into the area, which needs to be bound checked for safety reasons.

V. Case Study: The I4Copter Framework

To evaluate the practicability and costs of our approach, we ported a central component of the I4Copter [16] project to Java. The I4Copter is a quadrotor helicopter with an overall span of 91 cm. Quadrotor helicopters are simple in mechanical design and rely on four fixed pitch propellers, pair-wise spinning in the opposite direction, and a simple gearless drive. The flight attitude[1] is solely controlled by varying the rotation speed of the engines, which requires a challenging control software to reliably control its inherently unstable flight characteristics. The electronic control unit on the I4Copter is a board based on the Infineon Tricore TC1796 32-bit microcontroller, clocked at 150 MHz with a 1 MiB of external MRAM used for both code and data. The TC1796 is equipped with an MPU that supports four data regions for an application.

The control software of the I4Copter currently consists of approx. 26,000 physical lines of C++ source code and comprises a total of 13 mostly periodical tasks and four ISRs. The software is structured in five major components, which are modeled as separate OS applications that exchange data only by means of shared memory and messaging. Figure 3 shows these components and the data exchange paths among them. SignalProcessing samples the various sensors and performs simple preprocessing (e.g., noise filtering). CopterControl receives steering commands via radio control and transmits monitoring data to the base station by wireless LAN. The Ethernet component contains the network stack and interface drivers and does the actual WLAN communication. The core of the control application is the FlightControl component, which takes as input the steering and sensor data provided by the CopterControl and SignalProcessing components via shared memory and computes as output the thrust levels of the four engines. These are actuated to the engines by sending a message to the SerialCom component, which performs the communication with bus-bound devices (e.g., via SPI), that is the engine controllers and some sensors.

[1]Angle of the aircraft in regard to a reference point

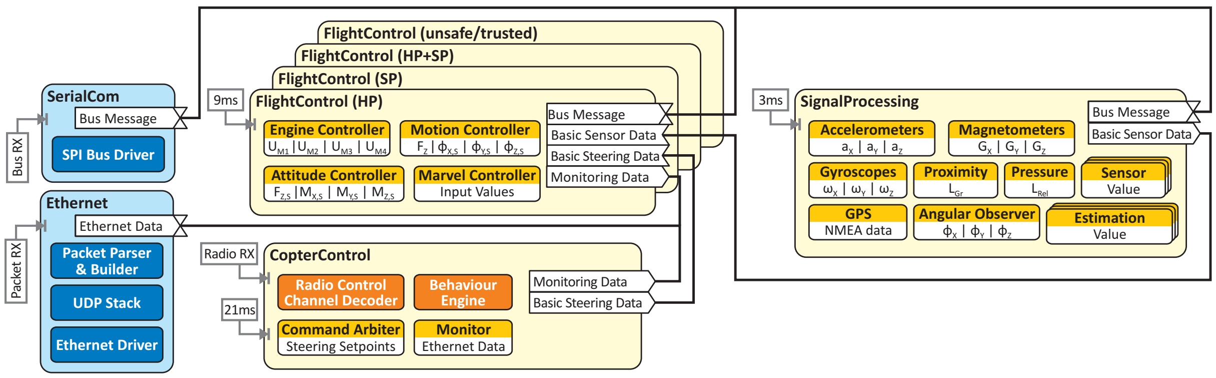

Figure 3. I4Copter Software Architecture: Components and Data Connections

	null checks	bound checks
Input	12 (+48 AOT)	26
Controller	0 (+125 AOT)	68
Output	23 (+83 AOT)	4
FLIGHTCONTROL	218 (+873 AOT)	164

Table I
RUNTIME CHECKS IN THE JAVA PORT

We have ported the FLIGHTCONTROL component to Java and combine it with the remaining original C++ components. For reasons of comparability, our port is as close in structure to the original code as the language permits.

A. *FlightControl Processing Stages*

A run of the FLIGHTCONTROL can be divided in three stages, which differ in the expected cost regarding the different protection schemes.

Input: In the input phase, the sensor and steering input data are copied from shared memory areas to the internal buffers of the control algorithm. The Java port uses `RawMemory` to access the shared memory area, which implies a bound check on each access. No MPU reconfigurations are needed to access the area, since it is accessed read-only and global read permissions are granted.

Controller: The controller is code generated using Matlab's real-time workshop from a Simulink model. It computes the four engine thrust levels from sensor data, steering commands and internal state retained from previous runs. This stage is intensive in single-precision floating-point computation and does not use any system services.

Output: The computed thrust levels are actuated to the engine controllers by sending a message to the SERIALCOM component and activating the receiving task. Sending the message requires two MPU reconfigurations if MPU protection is used (`allocate` and `send`). An additional protection mode change is required for the task activation. For accessing the message buffers, the Java port uses a combination of `RawMemory` and KESO's memory-mapped objects [14] to access the message memory.

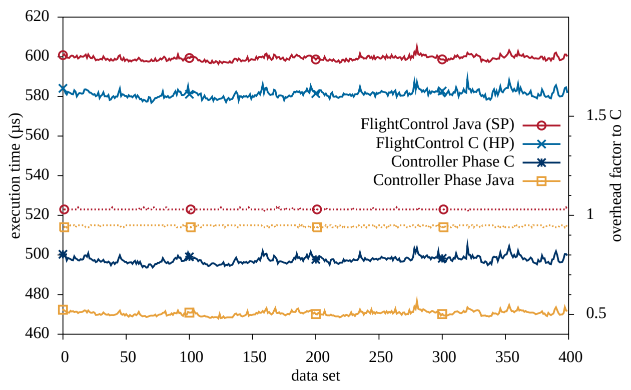

Figure 4. In-flight Execution Times C++ vs. Java

B. *Performance Java vs. C++*

We first examine how our Java port of FLIGHTCONTROL compares to the original version. To create a realistic execution environment for the runtime measurement, we extended the I4Copter application such that it runs both variants in sequence. Both variants are periodically activated with a period of 9 ms and use the same input data for computation. The code executed in each variant is exactly the same as in a standalone configuration. The C++ version is isolated by MPU-based protection, whereas the Java version is isolated by software-based means only. We measure the execution time of the entire FLIGHTCONTROL task and separately the execution times of the three earlier mentioned processing stages by reading the value of the free running system timer (clocked with 75 MHz) before and after the respective code section. For our measurement, we disable the interrupts during the execution of the FLIGHTCONTROL task. The measured values along with the inputs used and the outputs generated are transmitted by WLAN to the base station.

Figure 4 shows the results of the measurement. The dotted lines are drawn against the right scale and show the overhead factor of the Java version represented by the corresponding solid line over the original variant. We only draw a slice of 400 of the more than 12,000 data sets that our measurement

	C++	Java	Java without runtime checks
code	72902	80654	72470
data	2412	2844	2844
bss	97264	104632	104632

Table II
FOOTPRINT OF THE COMPLETE SYSTEM

produced to keep the curve progression distinguishable. The remaining data looks similar and does not provide additional findings. For space reasons, we have not drawn separate curves for the input and output phases; the execution times of both phases are constant for all data sets.

In the input and output phases, the Java version needs 2.5x (input) and 1.2x (output) the time compared to the C++ version. This is due to the bound check that is performed for each copied value, which is more expensive than the actual operation. KESO is currently not very strong at optimizing bound checks by static analyses. By using KESO's memory-mapped objects [14] instead of `RawMemory` to access the shared memory areas, we expect that most of this overhead could be eliminated. The few system calls in the output phase could not compensate these checks.

In the controller phase, the Java port unexpectedly slightly outperforms the original variant despite the runtime checks within the controller by about 5%. We identified two major reasons for this result: Firstly, KESO was able to perform most of the runtime checks at compile time. As shown in Table I, all null checks for non-array accesses could be performed ahead of time (AOT) and only bound checks remain in the controller. Secondly, the C++ version calls down to the C library for many simple floating-point operations (e.g. `fabsf`, `fmodf`), which are inlined in the Java port.

In the total execution time, the Java version has an overhead of about 4% compared to the C++ version. To our surprise, the Java version performed worse in the I/O-intensive phases but outperformed the C++ variant in the computationally intensive part. The reason is that the small number of three system calls does not pose a significant overhead compared to the total execution time of the remaining computations. The overhead factor in all phases is constant within the accuracy of the measurement.

Footprint: Table II shows the footprints for the complete system. The Java port is about 11% larger in code size than the C++ version. The causes for this are mostly the runtime checks in the code, as a version without these checks is about the same size as the C++ version. Since our Java port does not use garbage collection, it only contains a very slim KESO runtime. For initialized data, the Java port requires about 17% of additional RAM. This is mainly caused by KESO's runtime type information. The 8% overhead in uninitialized data is caused by the heap that all Java objects are allocated of. We chose a heap size that is very close to the memory requirements of the application. The increased size stems from object headers that are needed for Java objects and are not present in the respective C++ counterparts.

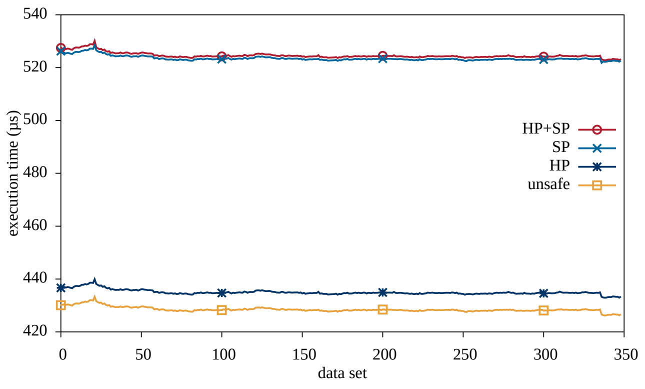

Figure 5. Offline Comparison of Different Isolation Variants

C. Cost of Protection Variants

After comparing the costs of C++ and Java, we want to determine the cost imposed by the use of MPU- vs. software-based memory protection without noise generated by the difference of our two ports. For this, we derive all four protection schemes from our Java port by either disabling MPU-based protection in the OS or disabling the runtime checks. To achieve comparable results, we need to run the component with identical input data, which is not the case when performing consecutive flights. We therefore isolated the FLIGHTCONTROL component from the framework and created an offline test that feeds the component with compiled-in input data that we take from a flight data trace.

The results of this measurement are depicted in Figure 5. For our ported component, enabling MPU protection causes an overhead of 1–2% in the overall execution time of the component, while the version that includes runtime checks introduces an overhead of 20–22%. The reason for this is the high share of internal computation as opposed to only three system calls that require an MPU reconfiguration in the FLIGHTCONTROL component. Thus, for this particular component the MPU can be used without much impact on the performance. The software-based isolation, on the other hand, comes at a higher price but has the potential of detecting additional memory errors within the component itself. We expect that many of the bound checks could be eliminated by improving KESO's static analyses. In the current schedule of the I4Copter application, there is enough slack time to leave the checks active in the application. We expect that more communication intensive components such as the SERIALCOM component with a simple internal logic will show a different picture and are currently working on a port of this component to check if these expectations apply.

D. Discussion

In this section, we want to share some experiences we gathered in the process of porting the FLIGHTCONTROL component.

1) Bugs found: During our port, we found that simple programming errors such as off-by-one happen to even experienced programmers. The runtime checks happened to be a quick aid in finding these bugs, especially since the debugging tools available for our platform failed to work reliably in the presence of MPU-based memory protection. Besides discovering bugs in our port, we could also find three bugs in the original code, one of which was already discovered at compile time by Java's stricter type system. One of these bugs resulted in the delayed mode change of the FLIGHTCONTROL component to the flight mode on takeoff.

2) Developer burdens when using MPU protection: We found the code structure of the I4Copter application sometimes being more oriented on the memory protection implementation of the OS rather than the logical structure of the solved problem. Whenever dealing with data items of non-local scope, the programmer had to take care of placing each data item in the correct memory region. Sometimes, this can become troublesome, for example if static members of a shared class need to be used from different components. We found many custom workarounds for such issues throughout the code base. Getting back to the example of ECU software in the automotive industry, even though the importance of spatial isolation has been recognized and became part of the AUTOSAR standard, the situation is that MPU protection is still almost unused. We believe that one major advantage of using a managed language is that it allows to fully hand the job of placing data items in the appropriate memory areas to the compiler tools and thus greatly facilitates and encourages the use of MPU-based memory protection.

3) Safety checks in embedded applications: As we observed with the bound checks in our case study, runtime checks can pose a considerable overhead for an application. Static analyses can greatly reduce this overhead if they can proof that a particular safety check will never fail at runtime and thus allow for a safe omission of the runtime check from the generated code. We believe that embedded applications and particular those that need to fulfill real-time requirements are a good target for such static analyses, as these applications are often developed with analyzability as an explicit design goal in mind. Table I supports this claim: 80% of the null checks could be eliminated by static analyses, mitigating the commonly assumed unacceptable costs of the increased safety level.

VI. RELATED WORK

In the area of general purpose computing, RPC-based mechanisms allow the isolated co-existence of safe and unsafe applications [17] by employing traditional operating system processes. This isolation mechanism is based on the use of multiple address spaces and requires the availability of a memory management unit, which is rarely found in deeply embedded systems.

Safe dialects [18], [19], [20] of the C language extend the type-system of C and also enable the incremental migration of a legacy code base. Most of these approaches work similar to that of a type-safe language such as Java and combine static analyses with runtime type information and safety checks, however, since the extensions made to C are solely with scope on the safe type system, these approaches do not provide the high-level language features and leave the task of memory management in the hands of the developer. Safe code is not isolated from unsafe code, thus portions of the code base that have not been migrated to the safe dialect become part of the trusted code base.

Foreign function interfaces (FFI) allow code written in different languages to directly interact. The Java Native Interface [21] is directly integrated with the Java 2 standard platform, however, the mechanism has been developed to be portable across many JVM implementations and is rather expensive. The Java 2 micro edition does not contain a native interface, but many embedded Java implementation come with own, more lightweight implementations of a native interface to enable tasks such as device driver development. The common problem is that the native code is not executed in isolation from the safe code and hence needs to be trusted. There are also approaches that combine a native interface with a type-safe C dialect [22], [23]. These require the foreign code parts to be re-engineered in the chosen safe dialect. In addition, the isolation is purely software-based.

VII. CONCLUSION

In this paper, we presented an approach that offers a safe incremental migration path for existing statically-configured embedded C applications to a type-safe language. We have developed a prototype that uses the memory protection mechanism most commonly found in the domain of deeply embedded systems and a Java ahead-of-time compiler that was specifically targetted towards this domain. We extended KESO with the low-level communication mechanisms shared memory and message passing, which are widely spread in existing applications, to enable the safe communication between re-engineered Java and legacy components. To test the feasibility of our approach, we presented a case study in which we migrated a significant portion of a complex real-world, hard real-time application to Java.

While the overhead introduced by the runtime checks required by most type-safe languages can indeed pose a significant overhead to the application, compiler optimizations and static analyses can often compensate this overhead. In our case study, where our ported component turned out to be subject to many safety checks and thus a piece of software that can be isolated by using the MPU in a much cheaper way, the 22% overhead of the safety checks could be reduced to a total overhead of only 4% compared to the C version by other optimizations in other places. MPU protection can additionally be used with only little added expense and without the need for the software developer to manually cope with the low-level requirements of region-based memory protection.

To conclude, type-safe languages can be introduced step-wise without imposing unacceptable costs. Hardware- and software-based memory protection are not mutually exclusive but complementing ways of achieving spatial isolation of applications. Having these ways as application-independent configuration options allows to easily determine the cost of the different variants and supports the decision based on the costs and the safety requirements for the given deployment scenario. The use of Java not only opens up the option of software-based memory protection but also makes the use of MPU-based memory protection more attractive as it relieves the developer of the burden to manually arrange for the physical grouping of application data in memory. Given that MPU protection is barely used today particularly for this reason, the use of Java can pave the way to a broader application of MPU protection.

References

[1] G. Phipps, "Comparing observed bug and productivity rates for Java and C++," *Softw. Pract. Exper.*, vol. 29, no. 4, pp. 345–358, 1999.

[2] E. Quinn and C. Christiansen, "Java Pays – Positively," IDC Bulletin W16212, Framingham, MA 01701 USA, May 1998.

[3] "JSR 1: Real-time Specification for Java," Sun Microsystems JCP, Palo Alto, CA, USA, May 2006. [Online]. Available: http://jcp.org/en/jsr/detail?id=1

[4] B. Hardung, T. Kölzow, and A. Krüger, "Reuse of software in distributed embedded automotive systems," in *4th ACM Conf. on Embedded Software (EMSOFT '04)*, Pisa, Italy, Sep. 2004, pp. 203–210.

[5] M. Broy, "Challenges in automotive software engineering," in *28th Int. Conf. on Software Engineering (ICSE '06)*. New York, NY, USA: ACM, 2006, pp. 33–42.

[6] AUTOSAR, "Specification of operating system (version 4.0.0)," Automotive Open System Architecture GbR, Tech. Rep., Dec. 2009.

[7] OSEK/VDX Group, "Operating system specification 2.2.3," OSEK/VDX Group, Tech. Rep., Feb. 2005, http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, visited 2009-09-09.

[8] R. Kumar, E. Kohler, and M. Srivastava, "Harbor: Software-based memory protection for sensor nodes," in *IPSN '07: 6st Int. Conf. on Information Processing in Sensor Networks*. New York, NY, USA: ACM, 2007, pp. 340–349.

[9] "JSR 121: Application Isolation API Specification," Sun Microsystems JCP, Palo Alto, CA, USA, Jun. 2006. [Online]. Available: http://jcp.org/aboutJava/communityprocess/final/jsr121/

[10] OSEK/VDX Group, "OSEK/VDX communication 3.0.3," OSEK/VDX Group, Tech. Rep., Jul. 2004, http://portal.osek-vdx.org/files/pdf/specs/osekcom303.pdf.

[11] D. Lohmann, W. Hofer, W. Schröder-Preikschat, J. Streicher, and O. Spinczyk, "CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems," in *2009 USENIX ATC*. Berkeley, CA, USA: USENIX, Jun. 2009, pp. 215–228.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *11th Eur. Conf. on OOP (ECOOP '97)*, ser. LNCS, M. Aksit and S. Matsuoka, Eds., vol. 1241. Springer, Jun. 1997, pp. 220–242.

[13] D. Lohmann, J. Streicher, W. Hofer, O. Spinczyk, and W. Schröder-Preikschat, "Configurable memory protection by aspects," in *4th W'shop on Progr. Lang. and OSes (PLOS '07)*. New York, NY, USA: ACM, Oct. 2007, pp. 1–5.

[14] I. Thomm, M. Stilkerich, C. Wawersich, and W. Schröder-Preikschat, "KESO: An open-source multi-jvm for deeply embedded systems," in *JTRES '10: 8th Int. W'shop on Java Technologies for real-time & embedded Systems*. New York, NY, USA: ACM, 2010, pp. 109–119.

[15] O. Spinczyk and D. Lohmann, "The design and implementation of AspectC++," *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software*, vol. 20, no. 7, pp. 636–651, 2007.

[16] P. Ulbrich, R. Kapitza, C. Harkort, R. Schmid, and W. Schröder-Preikschat, "I4Copter: An adaptable and modular quadrotor platform," in *Proceedings of the 26th ACM Symposium on Applied Computing (SAC '11)*. New York, NY, USA: ACM, 2011, to appear.

[17] 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*. New York, NY, USA: ACM, 2006, pp. 1–10.

[18] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *2002 USENIX ATC*. Berkeley, CA, USA: USENIX, 2002, pp. 275–288.

[19] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer, "CCured in the real world," in *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '03)*. New York, NY, USA: ACM, 2003, pp. 232–244.

[20] J. Condit, M. Harren, Z. R. Anderson, D. Gay, and G. C. Necula, "Dependent types for low-level programming," in *ESOP*, ser. LNCS, R. D. Nicola, Ed., vol. 4421. Springer, 2007, pp. 520–535.

[21] "Java Native Interface Specification 1.1," Sun Microsystems, Palo Alto, CA, USA, Aug. 2005.

[22] G. Tan, A. W. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, and D. Wang, "Safe Java native interface," in *Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering*, 2006, pp. 97–106.

[23] M. Furr and J. S. Foster, "Checking type safety of foreign function calls," in *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '05)*. New York, NY, USA: ACM, 2005, pp. 62–72.