Memory Protection at Option

Application-Tailored Memory Safety in Safety-Critical Embedded Systems

—

Speicherschutz nach Wahl

Auf die Anwendung zugeschnittene Speichersicherheit in sicherheitskritischen eingebetteten Systemen

Der Technischen Fakultät der Universität Erlangen-Nürnberg

zur Erlangung des Grades

Doktor-Ingenieur

vorgelegt von

Michael Stilkerich

Erlangen — 2012

Als Dissertation genehmigt von
der Technischen Fakultät
Universität Erlangen-Nürnberg

Tag der Einreichung:	09.07.2012
Tag der Promotion:	30.11.2012
Dekan:	Prof. Dr.-Ing. Marion Merklein
Berichterstatter:	Prof. Dr.-Ing. Wolfgang Schröder-Preikschat
	Prof. Dr. Michael Philippsen

Abstract

With the increasing capabilities and resources available on microcontrollers, there is a trend in the embedded industry to integrate multiple software functions on a single system to save cost, size, weight, and power. The integration raises new requirements, thereunder the need for spatial isolation, which is commonly established by using a memory protection unit (MPU) that can constrain access to the physical address space to a fixed set of address regions. MPU-based protection is limited in terms of available hardware, flexibility, granularity and ease of use. Software-based memory protection can provide an alternative or complement MPU-based protection, but has found little attention in the embedded domain.

In this thesis, I evaluate qualitative and quantitative advantages and limitations of MPU-based memory protection and software-based protection based on a multi-JVM. I developed a framework composed of the AUTOSAR OS-like operating system CiAO and KESO, a Java implementation for deeply embedded systems. The framework allows choosing from no memory protection, MPU-based protection, software-based protection, and a combination of the two. This decision can be made individually for each protection realm in the system. For both MPU- and software-based protection, the framework provides different trade-offs between the cost and the provided level of protection.

To achieve the configurability of MPU-based protection, I use aspect-oriented techniques to integrate the necessary changes to the operating system and the application. The configurability of software-based protection is based on static analyses in the Java compiler. The results of these analyses are also leveraged to improve the effectivity of MPU-based protection by aiding to determine private code and data items at a fine-grained level, showing significant improvements over the mostly manual existing approach in CiAO. The framework is completed by an extension that offers a soft-migration approach for existing applications.

At the example of the control software for a quadrotor helicopter, I evaluate the cost of using Java instead of the spread C or C++ languages and the qualitative and quantitative pros and cons of the different memory protection mechanisms. The evaluation shows that a Java implementation tailored towards the application domain can provide competitive performance to C and C++, but provides the added value of comprehensive software-based memory safety. The comparison of MPU-based and software-based protection showed that either can be more efficient depending on the type of application. The existence of both approaches is justified, and the two approaches complement each other in many aspects, which makes a combination of the two feasible as well.

Zusammenfassung

Zur räumlichen Isolation verschiedener Anwendungen auf einem Mikrokontroller wird im Umfeld zutiefst eingebetteter, statisch-konfigurierter Systeme üblicherweise eine Speicherschutzeinheit (MPU) eingesetzt. Diese beschränkt Speicherzugriffe auf eine begrenzte Zahl von Adressbereichen. MPU-basierter Speicherschutz zeigt jedoch Einschränkungen in den Bereichen der Hardwareauswahl, der Flexibilität und Granularität des gebotenen Speicherschutzes sowie des Aufwands der Benutzung. Softwarebasierte Verfahren bieten eine Alternative oder Ergänzung, haben bislang im Bereich der eingebetteten Systeme jedoch kaum Beachtung gefunden.

Diese Dissertation betrachtet vergleichend die qualitativen und quantitativen Vorteile und Einschränkungen von MPU-basiertem Speicherschutz und Speicherschutz basierend auf einer Multi-JVM. Aufbauend auf CiAO, einem Betriebssystem mit einer AUTOSAR OS Schnittstelle, und KESO, einer Java Laufzeitumgebung für zutiefst eingebettete Systeme, wurde ein Rahmenwerk entwickelt, welches die freie Wahl zwischen beiden Verfahren sowie deren Kombination erlaubt. Diese Entscheidung kann individuell für jeden Schutzraum im System getroffen werden. Das Rahmenwerk unterstützt für beide Verfahren verschiedene Abstimmungsgrade zur Beeinflussung des Verhältnisses von Kosten und gebotenem Schutzgrad.

Zur Realisierung des konfigurierbaren MPU-basierten Speicherschutzes wurden aspektorientierte Techniken verwendet, um die notwendigen Anpassungen in das Betriebssystem sowie die Anwendung einzubringen. Die Konfigurierbarkeit des softwarebasierten Schutzes basiert auf statischen Programmanalysen im Java-Übersetzer, deren Ergebnisse auch zur Identifikation privater Programmteile und Datenstrukturen verwendet wurden. Hierdurch konnte die Effektivität des MPU-basierten Speicherschutzes im Vergleich zum vorhandenen, größtenteils manuellen Ansatz in CiAO deutlich gesteigert werden. Eine Erweiterung zur Unterstützung eines weichen Migrationsansatzes für bestehende Anwendungen vervollständigt das Rahmenwerk.

Am Beispiel der Steueranwendung eines Quadrokopters wurden die Kosten für die Nutzung von Java als Ersatz für die verbreiteten Sprachen C und C++ evaluiert, sowie die qualitativen und quantitativen Vor- und Nachteile der verschiedenen Speicherschutzverfahren untersucht. Die Ergebnisse zeigen, dass die Verwendung einer auf die Anwendungsdomäne zugeschnittenen Java-Laufzeitumgebung keine Mehrkosten im Vergleich zu C und C++ mitbringen muss, jedoch den Mehrwert der umfassenden softwarebasierten Speichersicherheit bietet. Der Vergleich beider Speicherschutzverfahren ergab, dass jedes abhängig von den Anwendungseigenschaften das effizientere sein kann. Beide Verfahren haben ihre Existenzberechtigung und ergänzen sich in vielen Bereichen, so dass auch die Kombination sinnvoll sein kann.

Acknowledgments

My years at the FAU's system software group as a research assistant were some of the most fun yet. The product of those years – this thesis – would not have been possible in this form without the support and help of my great colleagues, to whom I want to express my gratitude.

My professor *Wolfgang Schröder-Preikschat* enables and supports his doctoral students in pursuing their own research interests. It is the freedom and self-responsibility he grants to his students that make his group such a great environment to work in. Together with *Jürgen Kleinöder*, I organized the exercises accompanying the systems programming lectures for some years; he enabled me to incorporate my own ideas into the course contents and I enjoyed those years of teaching a lot.

Some of my colleagues and former students contributed to parts of this thesis: Back when I was an undergraduate student, *Christian Wawersich* raised my interest in embedded systems and safety and brought me into the KESO project. *Daniel Lohmann* helped me throughout my research with invaluable discussions that often provided me with new perspectives on my research topics. *Peter Ulbrich* is the head of the I4Copter project; he not only provided me with the ideal evaluation scenario but also spent many hours helping me adapting and porting it; unforgotten, he showed very forgiving when I accidentally crashed the I4Copter into a building. *Jens Schedel* ported the I4Copter software to use the AUTOSAR OS interface and thereby made it usable for my setup. *Michael Strotz* implemented my ideas on gradual software-based memory protection in the KESO project. *Christoph Erhardt* did outstanding work on the compiler core; his work provided an important piece of infrastructure for my thesis.

My wife *Isabella* supported and motivated me in the challenging phases of thesis writing, proof-read and discussed with me the contents of this thesis.

Finally, I'd like to thank all my other colleagues that made working in the group such a great time.

Erlangen, December 2012

Contents

List of Figures

List of Tables

Listings

1 Introduction

1.1 Motivation

With eight billion units deployed in the year 2000, embedded microprocessors pose the lion's share of processors as opposed to 150 million general-purpose computers sold in the same year [117]. An embedded system is a computer system developed and deployed for a special purpose. Embedded systems are becoming increasingly ubiquitous in our everyday lives and are found, for example, in household appliances or cars.

Embedded systems are often part of a mass product and subject to an immense cost pressure. Cost differences of few cents for a single unit accumulate to a considerable amount for the whole of produced units. The microcontroller manufacturers commonly offer a product line of derivates around a specific microcontroller core. The derivates differ in features such as the amount of on-chip program and data memory, the number of I/O pins or the availability of optional functional units such as a memory protection unit (MPU) or communication interfaces such as CAN, SPI or UART. As a concrete example, STMicroelectronics' STM32F1 line of mainstream ARM Cortex-M3 processors as of 2011 offers derivates with program flash from 16 KiB to 1 MiB and RAM from 4 KiB to 96 KiB. The price range in this line for large quantities ranges from 1.76 € (STM32F100C4T6B, 16 KiB ROM, 4 KiB RAM) to 15.23 € (STM32F103ZGT6, 1MiB ROM, 96 KiB RAM) at a large retailer[1] – the largest derivate is more than eight times the price of the smallest one.

[1]Prices from http://de.farnell.com with the highest listed high-quantity rebate, retrieved on December 13, 2011

1.1.1 Electronic Control Units in the Automotive Industry

One industry where the electronic functionality has rapidly grown over the past decade is the automotive industry. A 2006 article [20] talks about ten million lines of code in a premium car, whereas a more recent Bosch article [78] already gives the number of up to 100 million lines of code in a luxury car. These numbers show the enormous growth of software in the automotive sector.

Embedded systems in a modern car take control of a wide-range of functionalities, covering comfort functions such as the fully automatic control of the air conditioning to safety-critical functions such as the anti-lock braking system (ABS) or the electronic stability program (ESP). With the trend towards more fuel-efficient cars that require sophisticated engine control and the introduction of electric cars, it is probable that the significance of electronic functions and the amount of software will continue to grow.

The automotive industry is vertically organized, that is, most of the development is carried out by a multitude of component suppliers. Traditionally, the electronic control units (ECUs) that implement electronic functions of a car have been shipped as black boxes comprising both the hardware and the software to the car manufacturer (OEM) [48]. The OEM then needed to integrate these black boxes by connecting them through communication busses such as the controller-area network (CAN) to form a cooperating network. In this *federated architecture* a dedicated microcontroller exists for each electronic function. A modern mid-class car is equipped with about 80 ECUs [78]. With the current amount of ECUs and the trend towards increasing electronic functionality, the federated architecture is running into a scalability problem as the integration is becoming increasingly difficult for the OEM for multiple reasons.

Heterogeneity: The ECUs are provided by a multitude of component suppliers and are heterogeneous in both hardware and software. Building a network of these black boxes becomes increasingly difficult with higher numbers of ECUs.

Reliability: The connectors that network the different ECUs are known to be fault-prone and a major cause of hardware failures.

Weight: The car wire harness by itself exceeds 100 kilograms in weight, comprising a length of more than two kilometers of wires. The electronic control units add additional weight. The increasing weight for cars of comparable class in recent years conflicts with the goal of fuel-efficiency.

Space: The space behind the car interior linings and in the engine compartment is limited.

Cost: Electronic components and software account for up to 40 percent of a car's production costs [20]. The price for copper has risen by more than 400 percent in the past decade.

1.1.2 Paradigm Shift Towards an Integrated Architecture

The automotive industry is undertaking various steps to address the above issues. One is the attempt to substitute copper as the wire material for the cheaper and more lightweight, but also less conductive, aluminum. The electrical differences between the two materials raise new issues that need to be addressed. Also, while the material transition lowers the weight and cost of the wire harness, the issues of requirement for limited space and the high number of fault-prone connectors remain.

A more fundamental approach is the departure from the federated architecture and a shift towards an *integrated architecture*, where multiple software functions are co-located on a single, but more powerful ECU. This architecture scales better, as more capable microcontrollers are developed in parallel to the increase of electronic functionality that provide more resources at the same price and physical size. The integrated architecture addresses many of the above issues: Two previously networked components that are integrated on a single ECU can communicate without the need for additional wires. The reduced amount of ECUs, wires and connectors reduce the material cost and weight and space requirements and at the same time increase the mechanic reliability of the electronic system.

1.1.3 New System-Software Requirements: Need for Isolation

With the shift to an integrated architecture, however, an important property of the federated architecture gets lost. With a dedicated ECU for every software function, each application executes in a physically isolated environment. A fault in one component is thereby contained and cannot directly affect the other components in the system. In addition, the faulty function can be identified relatively easy as the black box does not behave correctly and the supplier can be notified. This identification is important for both a quick resolution of the issue as well as for possible liability claims that may have been raised by the malfunctioning component.

By co-locating multiple applications on a single microcontroller on top of an OSEK/VDX operating system [86] – the currently most-widely used operating system in the automotive domain – these properties of fault containment and easy identification of a malfunctioning component are lost, since OSEK/VDX does not provide any means for the isolated execution of multiple applications. The shift towards an integrated architecture therefore introduces new requirements for the system software, most notably in the areas of temporal and spatial isolation. The automotive industry reacted to the new requirements with the development of a new operating system standard AUTOSAR OS [10]. The designated successor to OSEK/VDX provides temporal isolation by enforcing execution time budgets, terminating tasks that exceed their supposed worst-case execution time (WCET) in one job. This prevents that the fault propagates to other applications by not leaving enough CPU time for the other applications' tasks to meet their deadlines. AUTOSAR OS also provides optional support for hardware-based memory protection on microcontrollers that are equipped with the needed memory protection unit. Memory protection provides the spatial isolation of the different applications, which ensures that a fault in one application cannot corrupt the state of another application. In my thesis, I put the focus on the issue of memory protection, but get back to the topic of temporal isolation where close correlations exist.

1.2 Problem Statement and Proposed Solution

The hardware-based memory protection mechanism based on the use of an MPU is not optimal for all applications. The first obvious reason is that the chosen microcontroller needs to be equipped with an MPU, which is not the case for many low-cost chips. From the STM32F1 line discussed above, only eleven of 94 derivates in the upper-quarter price segment come with an MPU, of which the cheapest costs 11.76 € (STM32F103VFT6, 768 KiB ROM, 96 KiB RAM). The need for an MPU therefore constrains the available choice to few higher-end derivates, which is problematic in cost-sensitive application domains. Besides this economic downside, there are further problematic aspects associated with spatial isolation established by an MPU with respect to flexibility, the granularity of protection provided and the required developer expertise.

An alternative to using a hardware unit to achieve spatial isolation is purely software-based memory protection. There are various software-based techniques to establish spatially isolated applications that address many of the drawbacks of using an MPU, but none of these have so far considerably been adopted by the embedded industry. Although software-based approaches provide benefits over MPU-based protection in many aspects, they also have disadvantages in other aspects, for example execution time overhead or less robustness with respect to transient hardware errors. A more detailed discussion of the properties of different hardware- and software-based memory protection techniques follows in Chapter 2.

With both hardware- and software-based mechanisms offering advantages over the other in some aspects, it is not possible to generally choose one technique as the superior one. The choice for a spatial isolation mechanism depends on many factors, some of which depend on the application, such as the overhead that a particular mechanism imposes for a particular application. In addition, even for a given application, the choice for a spatial isolation mechanism may depend on external factors, for example the same application might be deployed in different environments with varying safety requirements or different degrees of electromagnetic interference (EMI). There may also be constellations where a combination of hardware- and software-based memory protection may be sensible or necessary to meet the given safety requirements. To determine the best-suited option for a given scenario, not only the qualitative aspects need to be considered but also the quantitative cost of the different alternatives. The evaluation of the quantitative cost is only feasible if switching between the alternatives is easily possible and can be performed for the cost determination.

The aim of this thesis is to design, implement and evaluate a framework for the domain of safety-critical, deeply embedded systems that supports hardware-based memory protection using an MPU and software-based memory protection based on the type-safe language Java and a multi-JVM concept. The decision for these mechanisms is explained in Section 2.5. The framework achieves the following objectives:

Fine-Grained Configurability: The framework allows selecting from hardware- and software-based memory protection without needing to change the code of the application, thereby allowing to easily switch to the best suited mechanism as the external requirements or the characteristics of the application change. This thesis also explores further graduations of the two mechanisms to allow a more fine-grained trade-off between cost and the degree of protection provided.

Mixed-Mode Operation: To support the integration of applications with different characteristics while using the best-suited memory protection mechanism for each, the framework allows the coexistence and combination of different spatial isolation mechanisms in one system configuration.

Soft Migration: A huge base of legacy code exists in the industry. Only a minor part of the application code is newly developed for a new product, most is reused from the existing code [48]. To make a transition to Java feasible, this thesis explores the typical properties of deeply embedded applications. The developed framework supports the reuse of portions of legacy code and the transition at a manageable level, although not all the benefits of the framework are available for legacy code parts.

Support Quantitative Evaluation: To support the identification of the best-suited protection mechanism for a given application, the framework supports the evaluation of the cost imposed by each mechanism. This is implicitly enabled by the goal of configurability that easily allows switching between the mechanisms to measure the cost imposed by each.

1.3 Broader Scope of this Work

Although the electronic control units in the automotive sector are used as a motivating example in the previous section, the work presented in this thesis applies to many other fields of embedded systems as well, whereby the market conditions and technical requirements may vary moderately between the different fields. Mixed-criticality systems where software components of differing safety-criticality are integrated on one system in a manner that does not require the non-safety relevant parts to be verified using the same time-consuming and costly processes as the safety-critical parts are an active research topic in other fields such as aerospace as well. In fact, later in this thesis an avionic application is used as a running example, mainly due to the lack of availability of an equally comprehensive automotive application. The used hardware and system software are the same as used in the automotive context, however, and the requirements of the application concerning resources and safety-criticality are equally – if not more – stringent, than that of a safety-critical automotive component.

1.4 Structure of this Thesis

The remainder of this thesis is structured into the following chapters:

Chapter 2 presents and discusses properties of both hardware- and software-based techniques for establishing spatial isolation. The discussion is concluded by the choice of hardware-based protection using an MPU and software-based protection on the base of a multi-JVM, which show to be the most suited for the targeted domain of safety-critical embedded systems. In addition, Chapter 2 also reviews more broadly related approaches that aim at providing memory safety or spatial isolation with a focus on the domain of embedded devices.

Chapter 3 analyzes which properties of spatial isolation are required – and which are not – in the context of the safety-critical embedded field, and based on these properties develops a suitable application and isolation model that defines different levels of protection and lays the base for the developed framework. In addition, an example application is introduced that is consistently revisited throughout this thesis. Finally, I investigate how the two chosen isolation mechanisms map to that model.

Chapter 4 develops the design and architecture of the framework. The focus in this chapter is on the configurability that allows choosing between, mixing and combining the two chosen isolation mechanisms in a single system.

Chapter 5 studies the step-wise migration on the basis of components to the developed framework. To enable the interaction of legacy and migrated components, the most widely spread communication idioms found in legacy embedded code are identified and integrated into the framework.

Chapter 6 shows that the developed framework achieves the objective of allowing the easy and direct comparison of the cost imposed by the different isolation techniques by performing such a comparison at the example of the control software of a quadrotor helicopter. In addition, I evaluate quantitative aspects of the framework itself, including an evaluation of the overhead imposed by using Java as a language instead of C or C++ for the targeted type of embedded applications.

Chapter 7 wraps up the content of the thesis and concludes with the major findings and contributions of the presented work, and discusses possible directions for future work.

1.5 Own Publications Related to this Thesis

Parts of the ideas and results discussed in this thesis have previously been published. The following is a list of these publications:

[114] M. Stilkerich, C. Wawersich, W. Schröder-Preikschat, A. Gal, and M. Franz. „OSEK/VDX API for Java.“ In: *Proceedings of the Linguistic Support for Modern Operating Systems ASPLOS XII Workshop (PLOS '06).* (San Jose, CA, USA). New York, NY, USA: ACM Press, Oct. 2006, pp. 13–17. ISBN: 1-59593-577-0. DOI: 10.1145/1215995.1215999

[124] C. Wawersich, M. Stilkerich, and W. Schröder-Preikschat. „An OSEK/VDX-Based Multi-JVM for Automotive Appliances.“ In: *Embedded System Design: Topics, Techniques and Trends.* (Irvine, CA , USA). IFIP International Federation for Information Processing. Boston: Springer-Verlag, 2007, pp. 85–96. ISBN: 978-0-387-72257-3

[111] M. Stilkerich, D. Lohmann, and W. Schröder-Preikschat. „Memory Protection at Option.“ In: *Proceedings of the 1st Workshop on Critical Automotive Applications: Robustness & Safety.* (Valencia, Spain). New York, NY, USA: ACM Press, 2010, pp. 17–20. ISBN: 978-1-60558-915-2. DOI: 10.1145/1772643.1772649

[110] M. Stilkerich, D. Lohmann, and W. Schröder-Preikschat. „Gradual Software-Based Memory Protection.“ In: *Proceedings of the Workshop on Isolation and Integration for Dependable Systems (IIDS '10).* (Paris, France). New York, NY, USA: ACM Press, 2010. ISBN: 978-1-4503-0120-6

[119] I. Thomm, M. Stilkerich, C. Wawersich, and W. Schröder-Preikschat. „KESO: An Open-Source Multi-JVM for Deeply Embedded Systems.“ In: *JTRES '10: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems.* (Prague, Czech Republic). New York, NY, USA: ACM Press, 2010, pp. 109–119. ISBN: 978-1-4503-0122-0. DOI: 10.1145/1850771.1850788

[36] C. Erhardt, M. Stilkerich, D. Lohmann, and W. Schröder-Preikschat. „Exploiting Static Application Knowledge in a Java Compiler for Embedded Systems: A Case Study.“ In: *JTRES '11: Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems.* (York, UK). New York, NY, USA: ACM Press, 2011, pp. 96–105. ISBN: 978-1-4503-0731-4. DOI: 10.1145/2043910.2043927

[112] M. Stilkerich, J. Schedel, P. Ulbrich, W. Schröder-Preikschat, and D. Lohmann. „Escaping the Bonds of the Legacy: Step-Wise Migration to a Type-Safe Language in Safety-Critical Embedded Systems.“ In: *Proceedings of the 14th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '11).* (Newport Beach, CA, USA). Ed. by G. Karsai, A. Polze, D.-H. Kim, and W. Steiner. IEEE Computer Society Press, Mar. 2011, pp. 163–170. ISBN: 978-0-7695-4368-0. DOI: 10.1109/ISORC.2011.29

[118] 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: Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems.* (York, UK). New York, NY, USA: ACM Press, 2011, pp. 87–95. ISBN: 978-1-4503-0731-4. DOI: 10.1145/2043910.2043925

[113] M. Stilkerich, I. Thomm, C. Wawersich, and W. Schröder-Preikschat. „Tailor-Made JVMs for Statically Configured Embedded Systems.“ In: *Concurrency and Computation:*

Practice and Experience 24.8 (2012), pp. 789–812. ISSN: 1532-0634. DOI: 10.1002/cpe.1755

2 State of the Art

In this chapter, I present and discuss both established and academic techniques to achieve memory protection suitable for achieving fault-containment for memory access errors, and operating system architectures with protection facilities that build upon these mechanisms. Concluding the discussion, I select one hardware- and one software-based mechanism that I pursue further in this thesis.

2.1 Levels of Memory Protection

Memory protection can be applied at different degrees. Two basic levels that can be distinguished are the sandboxed execution of untrusted code as opposed to the – potentially trusted – execution of memory-safe code.

2.1.1 Sandboxing

The sandboxed execution of code focuses on faults in potentially untrusted code. Sandboxing does not detect semantic errors in the executed code, such as out-of-bounds array accesses (or, more generally, buffer overflows), use of uninitialized values, or accesses to dead stack frames, except where these accesses would impact other applications' data. What faults are detected by a sandboxing approach varies for different techniques. Some would, for example, detect memory access errors that address unused portions of the address space, whereas others would silently ignore the error. Another common trade-off is the detection of errors versus the simple masking. In the former case, a memory access outside the memory area of the sandboxed application raises an error, whereas in the latter case the sandbox merely ensures that such accesses cannot happen, for example by applying a bitmask to all memory addresses prior to dereferencing that ensures that the access is within the application memory. The basic guarantee common to all sandboxing approaches is that memory access errors of one software component do not affect the data of other software components, which is sufficient to provide spatial isolation.

2.1.2 Memory Safety

As opposed to sandboxing, where executing the code in a controlled environment contains the defects of potentially untrusted code, memory safety is a property of code that guarantees that the code itself does not use memory in an unsafe way. The term *memory safety* has often been used in the literature without a uniform definition of the guarantees given by such code. One definition [3] given by Aiken, Fähndrich, Hawblitzel, Hunt, and Larus (Microsoft Research) in the context of the Singularity project is the following:

> Memory safety ensures the validity of memory references by preventing `null` pointer references, references outside an array's bounds, or references to deallocated memory.

Other projects that define approaches to produce memory-safe code may vary in the safety properties provided. A common exclusion is the absence of references to deallocated memory (that is, dangling pointers/references). In the discussion of related approaches later in this chapter I point out what memory-safety properties a particular approach has.

Memory safety by itself does not provide strict spatial isolation and ensure the containment of logical faults, however, two memory-safe components can be spatially isolated by eliminating shared data, given that the memory safety is complete with respect to the above definition.

2.1.3 Type Safety

A property at the level of the programming language that is closely related to memory safety is *type safety*. The same authors [3] as above define it as:

> Type safety ensures that the only operations applied to a value are those defined for instances of its type.

Concerning the memory accesses of a program, type safety provides many of the properties of memory-safe code, for example, dangling references must not exist (or be used) in a type-safe program. While `null` pointer dereferences or out-of-bounds array accesses are commonly not considered to be type errors, these errors are practically avoided (or detected) by any type-safe programming language (or its runtime system). Thus in practice, the code generated from a program written in a type-safe language is also memory safe, even though most type-safe languages need to check for some error conditions at runtime.

2.2 Comparison Criteria

In the following, I present both hardware- and software-based techniques to establish memory protection and review their suitability for use in the domain of deeply embedded, low-cost and real-time systems. My review takes the following criteria into account, which are discussed in detail in the text and summarized in Table 2.1 on page 32. For space reasons, the criteria are abbreviated in Table 2.1. The criteria are:

HW Req Hardware requirements of the approach, by availability in typical microcontrollers for the target domain.

⊕ No special hardware needed

○ Commonly available hardware needed

⊖ Hardly existent hardware needed

Flex Flexibility and granularity of the protection, incorporating the size and provided maximum number of memory regions protected.

⊕ Granularity of language-level objects in sufficiently large numbers

○ Coarse-sized regions in sufficiently large numbers

⊖ Few, coarse-sized regions

Leg Code Effort needed to adapt legacy code, which is assumed to be present in the form of C source code.

⊕ No modifications necessary

○ Moderate (up to 10 %) changes to existing code needed

⊖ Considerable code refactoring or rewriting required (exceeding 10 %)

Cost Average cost in execution time or memory of using the protection approach in relation to a plain C application that executes fully privileged. The higher of the average CPU versus memory cost is considered.

⊕ Negligible (up to 10 %)

○ Moderate (up to 50 %)

⊖ Intolerable (exceeding 50 %)

This classification is based on the spectrum of the actually published overheads to provide a gross diversification. In cost-sensitive domains, added costs of up to 50 percent are considered intolerable as well.

Fast IDC (Inter-Domain Communication) Supports fast communication among spatially isolated protection domains. Shared memory areas are excluded from this consideration, given that they present an exception to the isolation.

✔ Supports fast IDC

- ✖ Does not support fast IDC
- N/A Not applicable, or no numbers published

EMC Robustness of the spatial isolation with respect to transient hardware faults, for example bit flips in memory caused by electromagnetic interference. Rating based on the protection-relevant context stored in memory.

- ✔ Negligible risk, only few words of protection context in memory
- ✖ Considerable amount of protection-related context in memory

Prot Level Level of protection provided, as discussed in Section 2.1

Pred Predictability of the mechanism, assessing its suitability for the use in real-time systems. Based on whether the worst-case execution time of a program can differ heavily from the average caused by external factors such as the presence of other programs, particularly on the state of caches. Caches not related to memory protection are not considered, as these need to be incorporated independent of the protection technique.

- ✔ The execution time is mostly independent of that of other programs.
- ✖ Predicting the worst-case execution time of a program under the respective protection approach either requires sophisticated analyses that are not commonly available or yields impractically high worst-case estimates.

CPU Prot Whether the approach provides timing protection (that is, prevents monopolization of the CPU by one application)

TCB Size Size of the code base that needs to be relied on for the protection to work correctly.

- $\oplus$ Simple mechanism, typical implementations less than 100 lines of code
- $\bigcirc$ Relies on a verifier that performs simple program analyses
- $\ominus$ Requires the correct function of a full compiler or involves complex whole program analyses

It should be noted that Table 2.1 is only meant to provide a coarse overview. The ratings are not directly comparable for various reasons. Published results are based on different test or benchmarks applications, often rooted in diverse target domains and with differing characteristics. The coarse rating may also conceal large deviations, for example relative costs of 60 percent and 1000 percent are both equally rated as unacceptable. In few cases, no data is published for a particular criterion and the table instead contains a personal estimate based on data of similar approaches. Nevertheless, the overview should be sufficiently accurate to rate the various approaches.

2.3 Hardware-Based Memory Protection Approaches

Hardware-based memory protection mechanisms build on special hardware mechanisms that check every memory access instruction for compliance with the memory protection policy enforced by the particular unit. Commonly, read, write and execution permissions can be granted for portions of the memory. While the way the memory is organized differs among the different approaches, there are some similarities common to all implementations.

Although the memory accesses are fully checked by the hardware, software is always needed that configures the protection hardware in a way that sets up the appropriate memory access permissions for the currently executing code. This software is part of the operating system. For the protection to be effective, the application that is subject to memory protection should not be able to change the configuration of the protection hardware[1]. The code responsible for managing the protection hardware thus needs to be granted additional access privileges compared to the regular application code. A processor that is equipped with memory protection hardware therefore needs to provide at least two privilege levels, often referred to as the *supervisor mode* and the *user mode* of the execution. Only code executing in supervisor mode is allowed to modify the configuration of the protection hardware. In addition, the processor needs an exception mechanism that can be used by the protection hardware to signal violations of the configured protection concept, such as the attempt to execute a privileged instruction from user mode or a memory access that violates the setup protection, and enables handling of such error conditions by the privileged memory management software. Finally, a mechanism for the controlled transition between the privilege levels is needed. For the latter, most processors provide a distinct *supervisor call* instruction that raises a special exception and transfers control to the operating system.

2.3.1 Coarse-Grained Approaches Without In-Memory Data Structures

2.3.1.1 Region-Based Memory Protection Unit

A memory protection unit (MPU) establishes range-based memory protection. The basic principle is that a number of address ranges (regions) of the physical address space can be assigned particular access rights. Memory accesses are only allowed to the defined regions with permissions appropriate for the respective access. Despite sharing this simple principle, concrete implementations differ. For example, the heterogeneous MPU of the Infineon Tricore TC1796 microcontroller provides two sets of region registers, with two code and four data regions each. Execute permissions can be granted only on code regions, whereas read and write permissions are only available for data regions. If multiple overlapping regions contain an address, the combined

[1]Some approaches that target trusted – but potentially buggy applications – do deliberately not prevent the applications from modifying the protection hardware, trading improved efficiency for a slightly increased risk of memory corruption.

rights of all involved regions are effective for the address. The optional MPU of ARM Cortex-M3 processors is a homogenous MPU that supports eight regions usable for both code and data. However, whereas the Tricore's MPU allows defining regions for arbitrary ranges, the MPU of the Cortex-M3 requires regions to be of a size that is a power of two with a minimum size of 32, and the start address of the region to be aligned to its size. For overlapping regions, rights do not accumulate but the regions are ordered and the first matching region according to that order is effective.

2.3.1.2 Segmented Memory

Segmented memory is similar to the region-based protection using an MPU in that each segment is defined by a base address, a length and access rights, but segmented memory additionally involves address translation. The hardware unit is called a memory management unit (MMU) and may provide additional memory management functionality beyond the scope of protection. The addresses used by the program are offsets into a particular segment. On address translation, the offset is checked against the segment's length and the permissions of the segment are checked with the type of memory access. If both tests succeed, the offset is added to the segment's base address to determine the physical memory address. In addition to the direct translation, segmented memory can also be combined with paged memory.

2.3.2 Caching Approaches with In-Memory Data Structures

2.3.2.1 Page-Based Memory Management Unit

In a page-based memory organization, the address space is organized in pages of a fixed size. Typical page sizes are in the range of 1–64 KiB. As with segmented memory, the hardware unit is called an MMU and provides additional, non-protection related functionality. The most notable one is address translation on a per-page basis. The address mapping is defined in the page table, an in-memory data structure that contains the physical address and permissions for each page, and possibly additional attributes not related to memory protection. Commonly, a multi-level page table is used to reduce the size of the page table. A page table is defined for each protection realm. To avoid expensive memory lookups in the page table on each memory access, the MMU is equipped with a fast, fully-associative cache that holds recently used page table entries, the *translation look-aside buffer (TLB)*. Typical TLB sizes are in the range of up to 1024 entries. The optional MMU in the Tricore architecture specifies a TLB with a capacity of 4–128 entries. Some MMUs handle a TLB miss and the following page-table lookup in hardware (for example, x86), whereas others (for example, Tricore) leave the management of the TLB to the operating system.

2.3.2.2 Mondrian Memory Protection

Mondrian [126] memory protection (MMP) is a hardware implementation that allows defining memory access permissions at the granularity of single memory words. MMP supports an arbitrary number of segments of variable length for that permissions can be specified. The central data structure used by MMP is the *permission table*, comparable to the page table used in page-based memory management, but restricted to only contain protection-related data. Mondrian uses a two-stage cache concept: At the first level, a so-called side-car register accompanies each of the processor's address registers. The side-car register caches the access permissions for the segment used in the last memory access addressed using the respective address register. The second level is the *permissions lookaside buffer* that caches entries of the permission table similar to the TLB in MMU implementations, with a typical capacity of 64 or 128 entries. Based on Mondrian is Mondrix [127], a variant of the Linux kernel that uses Mondrian to isolate kernel modules. Mondrian has so far only been implemented and evaluated in simulators. For Mondrix, a modified version of the Bochs x86 simulator was used.

2.3.2.3 UMPU: Software-Fault Isolation in Hardware

Kumar et al. presented a hardware extension [66] called UMPU to an AVR ATmega103 8-bit microcontroller that implements the runtime checks required for a software-based fault isolation approach (Section 2.4.1.1), specifically Harbor [65], in hardware. The mechanism supports one trusted and up to seven untrusted protection domains. A part of the memory is reserved for the use by the untrusted domains and organized in blocks of a fixed, configurable size. The authors used a block size of eight bytes in their experiments. For each of these blocks, the domain ownership is tracked in the *memory map*, an in-memory data structure containing four bits for each block. To provide fault containment of memory access errors, the `store` instruction of the processor is extended to check in the memory map whether the accessed block belongs to the currently active domain, which is tracked in a separate processor register. To enable inter-domain communication, each domain can statically export a jump table of code entry points in a designated flash page. When another domain calls one of the exported functions through the jump table, the active protection domain is changed to that of the exporting domain and execution continues at the exported entry point. Upon return, the previous domain is restored. For this, the `call` and `return` instructions of the architecture have been modified. A single runtime stack is used for all domains. The stack resides in a memory area that is not in the range managed by the memory map. To avoid stack corruptions among domains, a separate stack bound register is used, that contains the current lower limit of the upward-growing stack. This bound is updated in the `call` and `return` instructions, and inspected by the `store` instruction to avoid stack corruption. Return addresses are managed on a separate *safe stack* to provide control flow integrity [1].

It is not entirely clear what happens if an illegal store operation is encountered. This is an interesting issue as the AVR architecture does not provide a mechanism for the processor to signal synchronous exceptions. The published text [66] suggests that such illegal writes are simply not performed without signaling the exception condition.

2.3.2.4 HardBound

HardBound [33] is a hardware extension to directly support fat pointers in the processor. A fat pointer is a pointer enriched by the base address and the bounds of the referenced object. HardBound is an academic project and has been evaluated by extending the Simics [73] full system simulation of an x86 processor. The processor is extended by a new unprivileged instruction `setbound`, which can be used to define the base address and bounds information for an address stored in a register. The processor subsequently checks all memory accesses to be within the bounds of the used pointer. HardBound is a mechanism that works at the user-level and is not sufficient for providing spatial isolation, as it does not provide full memory safety as defined above (in particular, it is only safe in the absence of dangling pointers).

To store the bounds information for an address, HardBound uses two in-memory data structures. The *shadow space* stores the base and bounds information for an address in a separate memory region to retain the internal pointer representation for compatibility reasons. The location where the base and bounds information for an address are stored can be computed from the address value. HardBound assumes a page-based address management that allocates pages for the shadow space on demand. To avoid the allocation of a shadow space twice the size of the program's data space, HardBound manages a second in-memory data structure that contains a bit for each memory word to mark whether the word contains a pointer, or not, the *tag metadata space*. Fat pointer metadata in the shadow space is only allocated for pointers.

The fat pointer's meta data not only needs to be accessed for every load or store instruction, but also for many other CPU instructions to correctly propagate the bounds information, for example in addition or subtraction instructions for supporting pointer arithmetic. To reduce the number of memory accesses, HardBound implements an additional first level cache, the *tag metadata cache*, to cache blocks of the tag space. To further reduce the size of the shadow space, the authors present various alternative approaches to compressed pointers, encoding bounds information for common object in additional bits in the tag metadata space or unused bits in the pointer value itself.

The software must properly initialize the bounds information. In their experiments, the authors of HardBound extended the C library's dynamic allocation function `malloc()` to properly set the bounds information. For statically allocated objects and stack objects, a C compiler was modified to initialize the bounds whenever a pointer to such an object is created. Propagation of the bounds information when passing the pointer or doing pointer arithmetic is completely handled by the processor. The authors performed an evaluation by counting the micro operations performed by the CPU with and without the HardBound extension. The runtime overhead for a benchmark suite is five to nine percent on average for different variants of compressed pointers, with peaks of up to 23 percent for some applications. These numbers assume that bound checking can be implemented in hardware without increasing the time required for the affected processor instructions. An additional evaluation that counts an additional micro operation for each bound check of an uncompressed pointer increases the average runtime overhead by three to ten percent. The memory overhead was compared by counting the number of pages allocated for the application. The average overhead of additionally allocated pages is from ten to 55 percent on average for different compression schemes, with up to 200 percent for some applications.

2.3.3 Discussion

None of the academic approaches is built into a commercially available processor. MMUs are rarely found in embedded processors. Of the Tricore family, only the TC1130 is equipped with an MMU [77]. The reasons are manifold: The MMU in the ARM7TDMI processor increases its area tenfold and its power consumption twofold [66]. The comprehensive in-memory data structures that hold the memory access permissions result in vulnerability with respect to transient hardware errors that can break address mappings and permissions. The cache used to achieve acceptable runtime cost impairs the predictability and worst-case execution time analysis cannot predict the cache states in the presence of multiple threads. Assuming a cache miss on every memory access is impractically pessimistic. A case study [52] that investigates several MMU designs finds that the overhead to the virtual memory system is in the range of ten to 30 percent. The published runtime overheads for the scientific approaches are in the same range. The in-memory data structures also add to the memory footprint of the program, the more the finer the granularity of protection. For HardBound, the reported memory overheads are up to 200 percent. It should be noted that of the scientific approaches only UMPU was designed for the embedded domain, albeit not for real-time applications.

From all hardware protection units, MPUs are the most common in processors for the deeply embedded and real-time systems for their simple design and predictable runtime behavior. Nevertheless, as outlined in Section 1.2, MPUs are not available in the low-cost segments of microcontroller product lines. The limited number of regions constrains the flexibility and may not be sufficient for all applications, especially if the operating system grants temporary access to some memory areas in the course of inter-domain communication such as messaging. The application needs to organize its data physically grouped in memory to achieve a small number of memory regions, which may require changes to legacy applications. The small number of regions may also impair the ability to use special memory types (for example, fast but small on-chip memory versus slower but larger external memory) for parts of its data if that required an additional region that may not be representable in a static MPU configuration. Execution time overhead occurs when the protection realm needs to be changed, that is, when communicating with another software component or when invoking a system call. The mechanism may therefore be expensive for communication-intensive applications. Concerning the robustness with respect to transient hardware faults (bit-flips), MPU protection is robust as the protection registers are normally hardened against radiation. While the region settings for inactive tasks are stored as part of the task context in RAM and thus subject to transient faults, the statistical risk of a bit flip affecting a saved protection context is low and acceptable. MPU protection does not introduce any source of indeterministic behavior and is thus suitable for the

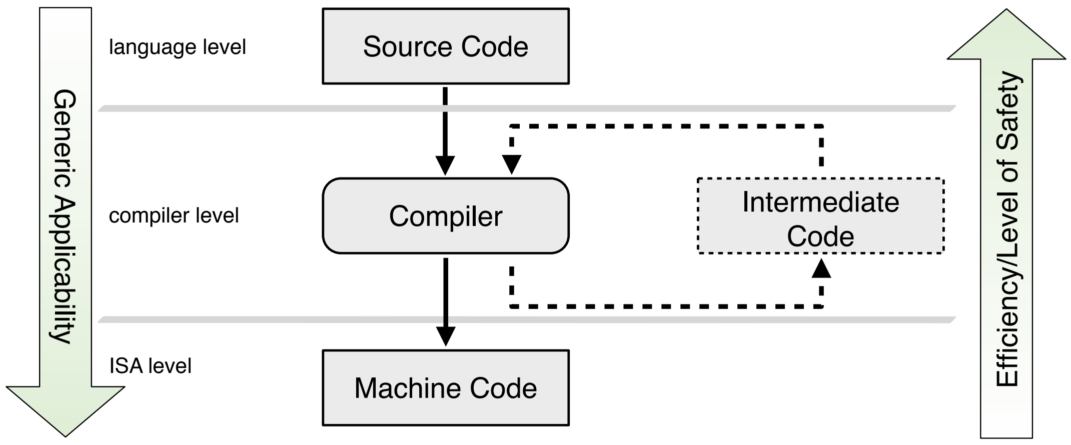


Figure 2.1: Levels at the Software-Based Memory Protection Mechanisms Take Effect

use in real-time settings. An operating system can virtualize the MPU, simulating an MPU with an (almost) arbitrary number of regions, which is for example done by Hightec's PXROS-HR operating system, allowing a more flexible MPU use by the application. This virtualization comes at the price of reduced predictability and potentially high overhead for certain usage patterns that collide with the operating system's region replacement strategy.

In a nutshell, MPUs are the only hardware memory protection units that are reasonably available in low-cost microcontrollers. This is for good reasons, as MPUs provide low-cost and low-overhead memory protection that is suited for cost-sensitive markets. The simplistic design and the robustness with respect to transient hardware faults make MPU-based protection easily verifiable and therefore a good choice for safety-critical applications. Finally, the absence of caches and in-memory data structures results in deterministic behavior that does not increase the complexity of worst-case execution time analysis, thus MPUs meet the requirements of real-time applications.

2.4 Software-Based Memory Protection Approaches

Software-based approaches provide memory-safe code to varying degrees without requiring a hardware protection unit. The available software-based approaches are manifold. Many approaches build on the same underlying concept, however. The approaches can coarsely be classified by the level at that they provide memory safety. Figure 2.1 shows these levels: Language-level approaches build on a safe programming language that constructively provides memory safety at the source code level. Following in the code generation process is one or more compilation processes. Compiler-level approaches work on the intermediate representation of the compiler and analyze and transform it in such a way that the output of the transformation is memory safe. Though working on the compiler's intermediate representation is theoretically independent of the source-level programming language, the transformations commonly assume certain properties that are implied by properties of the programming language, or place other restrictions on the input code. The transformation may fail with the rejection of the program if these expected preconditions are not met, or the analysis fails to prove that the input meets the expected constraints. At the final stage, a binary image containing executable machine code of the target instruction-set architecture (ISA) is produced. ISA-level approaches transform a binary image in a way that ensures memory safety.

The lower the level an approach hooks in the more general it can be applied. While language-based approaches mandate the use of a particular programming language, compiler-level approaches only require the use of a special compiler and binary-level approaches are theoretically entirely independent of the used language and toolchain. On the other hand, as source code is transformed to machine code, with each step down the toolchain information is lost and fewer assumptions can be made on properties of the program code without limiting generality. In the remainder of this section, I discuss software-based approaches categorized by these three levels, starting bottom up with binary level approaches.

2.4.1 Instruction-Set-Architecture-Level Approaches

2.4.1.1 Software-Based Fault Isolation

Software-based fault isolation (SFI) [122] provides sandboxing of application modules by patching the binary code so that all critical machine instructions are replaced by a more elaborate routine that checks whether the operation complies with the guaranteed safety model before executing it. The original approach aimed at isolating closely coupled software modules for which hardware-based memory protection incurred too high cost, caused by the context switching overhead in frequent communication between modules, for the example the sandboxed execution of third-party program plugins. The common concept is that each isolated entity has own code and data segments, and each critical operation (for example, store, jump) is patched to a checked version.

The original implementation by Wahbe et al. [122] is actually a compiler approach, as their prototype has been implemented by modifying the GCC compiler and therefore benefits from extended code transformation capabilities available at that stage. The motivation of the original approach is not to avoid the need for a hardware unit but to avoid the communication cost implied by using it. Two mechanisms are discussed: Segment matching detects faulty memory accesses by preceding each critical operation with a check for the address. The implementation requires four reserved registers that are not used by the normal application code and therefore a modified compiler. Sandboxing only ensures that the target memory location is within the segment of the running module by inserting code that patches the upper address bits to point to the segment but without checking it, requiring only two dedicated registers.

In the embedded systems context, particularly the wireless sensor networks (WSN) community has picked up SFI as a research topic. WSN nodes are typically equipped with very small (for example, AVR 8-bit) microcontrollers that provide no hardware support for memory protection or unprivileged execution modes.

The t-kernel [43, 44] provides CPU protection by preemption, horizontal memory protection that protects the kernel's data from being modified by a faulty application module, and virtual memory with swapping of heap pages to external flash. In a process called *naturalization*, the t-kernel patches the binary code of the application at load time. All branch instructions are modified so that the naturalized code yields the CPU to the t-kernel frequently. Three types of virtual memory are distinguished: stack memory, which is contiguous and not swappable, physical address sensitive memory (for example, memory mapped device registers) that is not relocatable, and heap memory that is both swappable and relocatable. The application is rewritten so that every memory access for that the address is not known at the time of naturalization is rewritten to a branch. The target code first determines the type of memory region. Stack accesses are directly executed. Heap accesses may require the page to be swapped in. Finding the page entry takes linear time. The t-kernel supports a single application with a single thread only. The reported overhead is 50 to 200 percent for execution time, and six to nine times code size expansion.

XFI [37] targets the isolated execution of pluggable program extensions in the form of relocatable library code. Example applications include kernel extensions and browser plugins such as a JPEG decoder. XFI rewrites the binary and inserts inline runtime checks that ensure control-flow integrity [1] and that the module only accesses the memory segments as defined by the object file format with constraints also defined in the object file. XFI uses a separate stack for data that is accessed via pointers and a scoped stack with integrity properties that holds directly addressed values and return addresses. On the host system, a simple verifier, which is the only trusted component, checks with the help of debug information that the module fulfills the properties mandated by XFI and contains the proper runtime guards. The published execution time overhead is five to 200 percent, with code expansion of up to four times for some benchmark applications.

Harbor [65] is an SFI approach targeted at WSN that supports the isolation of multiple application modules on top of the SOS [47] sensor node operating system. Harbor rewrites binaries offline but verifies them at load time. Harbor is a pure-software implementation of the UMPU [66] protection model presented in Section 2.3.2.3. The rewriter adds a runtime check to each memory store and control flow transfer function that performs the same checks as the extended hardware instructions in the UMPU. Harbor only ensures spatial isolation; it does not prevent monopolization of the CPU by a faulty application like t-kernel. The overhead reported in [65] is an 8 times slowdown for an FFT application and a 13.3 times slowdown for a memory intensive buffer writer application. Code size expands by 30 to 60 percent in the rewritten application modules.

2.4.2 Compiler-Level Approaches

At the next higher level in the toolchain are memory-safety approaches that work in the compiler. As opposed to the binary-level techniques, the source code of the application is required but no particular programming language is presumed, although all approaches place some constraints on the input code. An advantage over the lower-level approaches is that at the compiler level the code can be transformed in a more general and efficient way. A simple example is that introduced runtime checks can be executed inline, which is only possible with limitations when transforming binary code, since branch targets may move as code is inserted. Updating the branches to work for the new targets is not always possible, especially for indirect (computed) branches.

2.4.2.1 Software-Based Segment Protection

Software-based segment protection [104, 18] is a compiler approach that aims to provide the same type of memory protection that a region-based MPU (Section 2.3.1.1) provides without the need for a hardware protection unit. The published implementation transforms the static single assignment (SSA) [31] form of the Low Level Virtual Machine (LLVM) [68] compiler framework. The transformation adds runtime checks to memory accesses that ensure that the access is within the bounds of the segment that it is *intended* to access. The intended segment is determined by a whole program analysis that collects for each pointer access all terminating definitions of that pointer value. For each terminating definition, the segment is known. For each possible intended segment, a runtime check is inserted that checks whether the actual value of the pointer is that of the corresponding definition and then checks whether the pointer is within the bounds of the segments. The implementation contains some optimizations to reduce the runtime check overhead, most notably an optimization that moves checks out of loops where the pointer value within the loop depends on the loop invariant only and the minimum and maximum possible values are checked before entering the loop. For a range of embedded benchmarks, the approach shows attractive cost of less than one percent runtime and four percent code size on average.

2.4.2.2 Fail-Safe C

Fail-Safe C [83] is a source-to-source compiler for ANSI C programs. It aims at providing memory safety for C programs without requiring modification of the original code. The transformed code contains runtime type information for all memory blocks and uses fat pointers and fat integers to support conversion between pointers and integers. Accesses to memory blocks are redirected to access methods specific to the runtime type of the memory block. Fail-Safe C does not target embedded applications and does not support low-level code that accesses memory areas not allocated by the runtime environment. The execution time overhead exceeds 1000 percent for some benchmarks. Numbers of the memory overhead have not been published, but the use of fat representations for all pointer and integer values and the fallback to allocating twice the requested memory if the runtime type of a dynamically allocated block of memory cannot be statically determined suggest that the memory overhead may be significant.

2.4.2.3 SoftBound

SoftBound [80] is a software implementation of the fat pointer approach realized in hardware by HardBound discussed in Section 2.3.2.4. It is implemented as a pass in the Low Level Virtual Machine (LLVM) [68] compiler framework and transforms LLVM's intermediate static single assignment (SSA) [31] representation. The code is instrumented so that base and bounds information for every pointer are carried with the pointer. These properties are carried in separate variables that are allocated to registers by the LLVM compiler in the same manner as regular program variables. The properties are initialized at `malloc()` call sites from the `malloc()` return value and the passed size parameter, or at locations where a pointer is created by using the address operator, where the size of the referenced object is used. The intermediate code is instrumented so that the bounds information is propagated and adjusted during pointer assignments and arithmetic, and checked whenever a pointer is dereferenced, or passed as a function parameter (for the latter, procedure cloning [29] is used to create a clone function with an interface extended to include the pointer metadata as additional parameters is created). SoftBound also supports optional *pointer narrowing* where bounds of a pointer are narrowed if a reference to a subobject is created, which detects subobject overflows but breaks compatibility with some code patterns where a structure field has an actual size larger than its declared size. When pointers are loaded from or stored to memory, the base and bounds are stored in a separate data structure. Load and store operations are instrumented so that the pointer metadata is loaded and stored as well. The location of the metadata of in-memory pointers is computable from the pointer value. SoftBound has published average runtime overheads of 93 percent for checking both loads and stores or 54 percent for store-only checking, and memory overhead of up to 300 percent (average 87 %) in the used benchmarks.

2.4.2.4 CCured

CCured [81, 28] is an extension to the type system of C that categorizes pointers into different kinds to achieve memory safety using a combination of static whole-program analyses and runtime checks. `Safe` pointers point to a single memory object of a static type and are not subject to pointer arithmetic. A `Safe` pointer always contains a valid address to an object of its static type or `null`, and dereferences of `Safe` pointers imply a `null` check. `Seq` (sequence) pointers are 3-word fat pointers that point to a memory area containing an array of objects of a static type. Pointer arithmetic is allowed on a `Seq` pointer and dereferencing the pointer incurs a bound check. `Wild` pointers are the only type that can be used in arbitrary type conversions. `Wild` pointers are two-word fat pointers that contain a type tag in addition to the pointer value itself. In addition, the memory area that a `Wild` pointer points to needs to be tagged with the size of the area and information that allows determining for each word in the area whether it is a pointer or not. Dereferencing a pointer incurs a type check to determine whether the `Wild` pointer actually contains a pointer value, and a bound check. For writes, the tag data of the affected word needs to be updated additionally. CCured performs a source-to-source transformation on C code and implements a static whole-program analysis to infer the best-suited pointer types from the unmodified C program. To reduce the number of expensive `Wild` pointers in the program, CCured supports a form of physical subtyping [21, 102] to support common subtyping design patterns found in C code of larger projects. Briefly formulated, a type `T` is considered a physical subtype of another type `S` if the physical memory representation of `S` is a prefix of that of `T`, distinguishing different CCured pointer types and non-pointer types. CCured supports upcasts and checked downcasts in a physical subtype hierarchy through a fourth `Rtti` fat pointer type that tracks the actual type of the pointee object. The type is initially set when the `Rtti` is created from a `Safe` pointer (upcast) and checked when cast back to a `Safe` pointer (downcast). CCured has been evaluated at the example of server applications and kernel modules and shows overheads of up to 90 percent. A conservative garbage collector is suggested for applications that require dynamic memory management.

2.4.2.5 Memory Safe C

Dhurjati et al. presented a series of compiler techniques to achieve a strong level of memory safety for essentially a type-safe subset of the C language. The approach [34] has been given no particular name but is for brevity referred to as MSC-GC (Memory Safe C without Garbage Collection) on the basis of the paper title. The approach provides a level of memory safety close to type safety for the proposed C subset except that it fails to prevent the use of dangling pointers. The restrictions posed on C are briefly:

- Casts to pointer types are only allowed between pointers to primitive types where the pointee type of the target needs to be of smaller or equal size as the pointee type of the source.
- Unions may only contain types that are cast-compatible.
- Array indices must be a restricted linear expression that involves the array's size, constants and, with limitations, symbolic values.
- Pointers to stack locations must not be stored in global variables, heap objects, or be returned from a function.
- Pointers must be initialized before being first used. A use includes taking the address of the pointer.
- Pointers to arrays must not be loaded from global variables or heap locations.

The main goals of the approach are to avoid runtime checks entirely and to allow the manual deallocation of objects. To achieve this, a hardware protection unit and a reserved address range that is at least the size of the largest individual object are needed. The base of this address range is used as the `null` address and thus dereferences of a `null` reference are expected to raise a hardware exception. The approach combines several static analyses to check the safety of the program. A conservative escape analysis is used to ensure that no dangling pointers to stack locations are created. Global pointers, heap memory and local pointer arrays are initialized with the `null` value to prevent the use of uninitialized pointers. Dangling pointers to deallocated objects are not avoided but the compiler ensures that the use of a dangling pointer is no memory safety concern. For this, the compiler transforms the code so that all heap objects are allocated from type-homogeneous memory pools. This is similar to region-based memory management as in Cyclone [53] or scoped memory in real-time Java [55] in that the memory of a region is reclaimed at one time. Pools differ from regions in that they contain only objects of a single type and that reuse is possible within the lifetime of the pool. MSC-GC implements an automatic pool allocation transformation that automatically determines the memory pools and transforms `malloc()` and `free()` operations in the program to the corresponding pool operations. Therefore, a dangling pointer to a de- and reallocated object may exist, however, the reallocated object has the same type and therefore a memory safety violation is not possible by using the dangling reference (although it represents a semantic error). MSC-GC has been evaluated using 20 small- to mid-size programs. Since MSC-GC does not use any runtime checks, the only runtime overhead is caused by the initialization of pointer values and the use of pool allocation instead of the regular heap allocation. The overhead is therefore negligible with maximums of seven percent, however, the pool allocation limits memory reuse and causes increased RAM usage of up to 44 percent in some cases. The required changes to the source code of existing applications are also minor in the sub-percent range. The downsides of the approach are that the analyses only work on single-threaded applications and that programs that the compiler fails to prove safe are rejected. Of the 20 benchmark applications, one failed the conservative escape analysis, and nine were rejected because the array uses could not be proven within bounds. The latter shows that the given restrictions on array indexing are too restrictive for many programs. Runtime `null` checks could be used to eliminate the need for a hardware protection unit. The ability to inject runtime bound checks is limited due to the absence of runtime array bounds information and only possible where the compiler is able to infer a symbolic expression to compute the array bounds from runtime state available in the local scope of the array access.

2.4.3 Language-Level Approaches

Approaches on the language level fundamentally build on properties of the programming language to achieve memory-safe code, although the compiler and possibly a runtime system are involved in establishing and maintaining this property as well.

An obvious requirement for using language-level approaches is that the source code must be available in the respective programming language, a considerable hindrance if legacy code needs to be reused. As a result, besides the lower-level approaches discussed before, a fair amount of research aims at retrofitting the programming language most widely used in legacy code (that is, C) to achieve memory safety while requiring only modest changes to existing code.

2.4.3.1 Retrofitted Unsafe Languages

Cyclone [53] is a safe dialect of C that uses a combination of intra-procedural analyses, interface annotations and runtime checks to provide memory safety for C programs with an acceptable adaption effort for legacy code. In addition to the standard C pointer type, whose semantics are restricted, Cyclone introduces two additional pointer types designated by extra type symbols in the source code. Regular C pointers ($*$-pointer) cannot be used in pointer arithmetic and are `null`-checked on access. Never-`null` pointers (`@`-pointers) must never hold a `null` value. This invariant is checked when the pointer is initialized rather when it is dereferenced. This allows propagating `null` checks up the call graph, ideally to the source definition to statically verify the never-`null` property without a runtime check. The third pointer type is three-word fat pointers (`?`-pointer) with bounds information, the only type that may be used in pointer arithmetic and on that dereferencing incurs a bound check. To avoid dangling pointers to deallocated memory regions, Cyclone disables the C `free()` function completely and suggests the use of a conservative garbage collector or alternatively provides memory regions as a concept to memory management. A region is a portion of memory that is deallocated as a whole. A region may contain objects of different types and with different allocation times, but all are deallocated at once. Cyclone implements a static region analysis [42] that determines a region for each pointer used in the program and raises a compile time error if a pointer outlives its region (for example, if a pointer to a local variable is returned from a function). Cyclone allows programmer-defined growable regions that open a new scope and special variants of some C library functions (for example, `malloc()`) that allocated from such regions instead of the heap. Adopting a program to use regions to substitute for heap-based allocation can pose considerable reengineering overhead. The effort of porting legacy applications to Cyclone requires about ten to 20 percent of the source lines to be changed. The runtime overhead imposed by Cyclone is up to 185 percent, no numbers have been published concerning the effect on the memory footprint.

Deputy [27] is a sound system of dependent types for C. The type system enables the developer to define dependencies between program variables in the form of program annotations. As an example, the standard C function `memcpy()` could be annotated as shown in Listing 2.1. With the `count(n)` expression the programmer annotates that the bounds of the destination pointer `d` is `[d; d+n)`, and analogously for the source pointer and the returned pointer. Deputy's compiler checks that these constraints hold and inserts runtime checks where the static analysis fails to prove the constraints.

```
void *count(n) memcpy(void *count(n) d,void *count(n) s,size_t n)
{
    while(n --> 0) d[n] = s[n];
    return d;
}
```

Listing 2.1: Deputy Annotations for `memcpy()`

To reduce the amount of annotations that need to be provided by the programmer, Deputy infers missing annotations for local variables. In case the inference algorithm fails Deputy raises a compile-time error and the programmer needs to manually add the annotation. Similar to the never-`null` pointers in Cyclone, the expression of type invariants at interfaces allows propagating the check of the invariant back to the source without requiring an inter-procedural analysis. The compiler also checks (or inserts appropriate runtime checks) that type invariants are not invalidated when a variable that other types depend on is modified. Deputy does not check memory deallocations and dangling pointers can impact the soundness of the dependent type system. The required changes to the code of existing applications are reported to be in the single percent range and therefore fewer than those required for Cyclone. The runtime overhead for a variety of benchmarks is one to 98 percent. Deputy has been used to implement memory-safe Linux device drivers in the SafeDrive [128] project and is also the basis for a safe variant [30] of the sensor network operating system TinyOS. To address the issue of dangling pointers, Deputy has been combined with HeapSafe [40], a reference counting scheme to check the sound use of free at runtime, but requires additional source code modifications and adds additional overhead (11 % execution time, 13 % RAM on average).

2.4.3.2 Type-Safe Languages

Type-safe languages provide strong type safety by proper language design and a supporting runtime environment. The key concept in a type-safe language to provide memory safety is that of the unforgeable, strongly typed reference, which represents a capability at the language level to access an area of memory in limited ways as defined by the type associated with the reference. There may be additional memory regions such as stack variables accessible to the program, but the management of such areas is under full control of the runtime system. A combination of language design and runtime checks ensures that the type safety is maintained. The types of runtime checks needed depend on the actual language. Common checks include checks of potentially unsafe type conversions (for example, downcasts), `null` reference checks before dereference operations and array bound checks to avoid buffer overflows. While `null` dereferences and out-of-bounds array accesses are not commonly considered as type errors, failure to avoid these errors would normally affect the soundness of the type system and need therefore be avoided.

Ada, C#, Java, Lisp, and Standard ML are some examples from the multitude of programming languages providing strong type safety, and type safety is a property of almost any new programming language that is being developed. In the embedded context, type-safe languages have not broadly been adopted yet, compared to the spread of C. Besides the existence of legacy code, type-safe languages face a conflict between the language restrictions that make the language type safe and the ability to write low-level programs such as device drivers, which is a common need in embedded programs. Currently, Ada is probably the type-safe language that is most widely used in embedded code, however, due to the higher availability of trained programmers there is currently a shift towards Java in these communities [82].

Java has been widely adopted in many computing domains from server and desktop applications down to less resource-constrained embedded systems such as mobile phones and is nowadays part of most training or academic education programs that teach software development. Java is backed by an active research community and many efforts have been taken to overcome Java's limitations with respect to resource-constrained embedded systems. The real-time specification for Java (RTSJ) [55] and more recently safety critical Java (SCJ) [60] define scheduling models, memory allocation policies and execution models suited for using Java for real-time programs. For the event-driven processing model of some real-time applications, a number of alternate execution models [106, 107, 108, 9, 8] have been developed that better reflect this execution model than the standard Java thread model.

Spatial Isolation Based on a Type-Safe Language Type-safe (or fully memory-safe) code can easily be spatially isolated by eliminating access to shared data between the isolated entities. In Java, such shared data is initially the static class fields, and during runtime all objects exchanged through this channel. By eliminating or restricting access to these shared fields, different application parts can easily be spatially isolated from each other.

There have been several research projects that use a safe language as the base for software-based spatial isolation. SPIN is a microkernel that supports the safe execution of extensions written in Modula-3 in the kernel space. J-Kernel [49] and KaffeOS [12, 11] provide the isolation of subsystems in a single JVM process. Singularity [3, 51, 38] similarly provides software-isolated processes but is based on C#.

JX [41] provides first-class processes that are solely isolated by the type safety of Java and the logical separation of the data structures. This architecture is also referred to as a multi-JVM architecture, because as a result of the strict data separation each isolated entity appears to be executing in a separate JVM. KESO [113] is a follow-up project to JX that provides the multi-JVM architecture of JX for resource-constrained embedded systems.

The Java Isolate API [56] defines a standard interface for isolating subsystems in a JVM and is implemented by SquawkVM [103], a research VM by Sun that targets resource-constrained devices such as Sun's SPOT wireless sensor network motes.

Exotasks [9] allocate memory from a private heap and exchange data through strongly typed data connections, which are defined in the form of an Exotask graph.

Data passed through such a connection is deep copied to the receiving Exotask. Exotasks support a strong isolation model, which limits communication with other Exotasks to the connections of the Exotask graph by disallowing writes on any static fields while limiting read access to (recursively) final fields. The restrictions are enforced by static analysis at initialization time. The usage restrictions with respect to static fields are the main point where Exotasks contrast from a multi-JVM architecture, in which the global state is maintained separately for each isolated entity. The multi-JVM architecture is preferable in this aspect, as it does not require non-standard Java applications and allows creating multiple isolated instances of one application.

The Crux of Implicit Memory Management C uses manual deallocation for dynamic memory management, which poses the risk of dangling pointers. Dereferencing of dangling pointers is a semantic error, and proving the absence of dangling pointer use in the presence of explicit memory deallocation is extremely difficult [34]. Listing 2.2 shows a C++ code snippet that illustrates how a dangling pointer can void the soundness of the type system. The classes `MyInteger` and `MyString` both contain a single field, a primitive integer value in the former and a pointer field in the latter case. Transferred to a type-safe language such as Java, instances of two equivalent Java classes would have the same physical memory representation in many JVMs. The example first allocates a `MyInteger` object and then explicitly deallocates it. The code retains its now dangling pointer to the former `MyInteger` object, and then allocates a new object of type `MyString`. The memory allocator might reuse the memory of the previously freed `MyInteger` object. Given that the two classes have the same physical representation, the code can now use the dangling pointer to create forged references. The program is not type safe. While code analysis could easily detect the use of a dangling pointer in the above example, such analyses rapidly gain complexity in the presence of pointer aliases and concurrent threads.

For this reason, dangling references need to be avoided, which is usually achieved by replacing explicit deallocation with implicit memory management (that is, garbage collection). The previously discussed approaches to achieve memory safety for C programs address this issue in various ways:

- Disabling `free()` entirely (Cyclone)
- Using a conservative garbage collector, that uses heuristics for the lack of precise runtime type information to identify reference values in the program state (CCured, Deputy)
- Providing alternate memory management techniques such as regions or memory pools (Cyclone, MSC-GC)

Disabling deallocation entirely is a viable option for many embedded programs that do not actually require dynamic memory management and only allocate memory in the initialization phase, however, some programs may require dynamic memory

```
struct MyInteger { int value; };
// same physical representation as MyInteger
struct MyString  { char *value; };

void danglingPtr() {
    MyInteger *i  = new MyInteger();
    // explicit deallocation of i
    delete i;

    // memory of deallocated object may be reused
    MyString  *s  = new MyString();

    // use dangling pointer to access s->value as int
    i->value    = 42;

    // s->value contains a forged reference of type char*
    s->value[0] = 'a';
}
```

Listing 2.2: Unsound Type System Caused by Dangling References

management. Conservative garbage collection [19] cannot reliably identify references and memory and therefore give no guarantees that dead objects are deallocated, which is an issue in safety-critical applications. The region-based techniques are subject to usage restrictions that are not suitable for all memory usage patterns. The region-based approaches furthermore postpone deallocation to the point where the region is destroyed, which can lead to increased and possibly unacceptable memory consumption for certain usage patterns.

The preferable memory management strategy therefore depends on the memory usage behavior of the individual application. Java provides standardized mechanisms for all of the above variants. The RTSJ defines *immortal memory* that exists throughout the lifetime of the program, and *scoped memory* regions that are subject to similar usage restrictions as Cyclone's regions. As a type-safe language, Java also supports *precise* garbage collection, and research on real-time garbage collection [4, 100, 90, 87, 88, 13, 15, 22, 54, 96] has brought up garbage collection techniques that bound fragmentation and pause times and are suited for the use in real-time applications.

2.4.4 Discussion

As raised in the opening of this section, there is a trade-off between reusability of existing source or binary code and the cost and protection level provided. SFI approaches are the most general ones but the provided safety level is limited to the sandboxed execution of the application. At the same time, the overhead introduced by patching all store instructions in binary code to call a runtime check routine is simply unacceptable in cost-sensitive markets.

The compiler approaches focus on providing source code reusability and most achieve memory safety at the level of application-level objects. Hooking into the compiler as a transformation pass on the intermediate representation, the analysis framework of the compiler can be utilized and runtime checks or pointer metadata are subject to the same standard compiler optimizations as the application code, resulting in a lower, but still considerable, overhead compared to the SFI techniques. On the downside, some assumptions are made on the input code that may cause the rejection of typical low-level embedded code. As an example, casting an integer value to a pointer value is commonly forbidden, but this is a common operation in driver code that accesses memory-mapped registers. Another example is the fat integer representation in fail-safe C, which renders the approach unusable for low-level programming.

The safe C dialects Cyclone and Deputy add programmer annotations to aid the compiler in proving the memory safety of a program. By annotating parameters and return types at interface definitions, many runtime checks can be omitted without the need for whole-program analyses by transitively propagating safety checks to the caller side, eventually reaching the source definition in many cases. None of the retrofitted approaches provide a solution to the dangling pointer issue discussed in Section 2.4.3.2 that preserves the original way of allocating and deallocating memory using the `malloc()` and `free()` functions. As solutions, conservative garbage collection or specialized forms of memory management that are not generally applicable are provided.

Languages that are strongly type safe by design have concepts to reliably handle this problem, for example by applying precise garbage collection. In addition, such languages provide the highest level of memory safety, type safety at the semantic level of the programming language. Concerning an existing C code base, however, moving to a type-safe language means that the entire code needs to be rewritten. The effort of such a rewrite depends on how different the targeted type-safe language is from C, but it can be expected to be higher than adapting the code base to one of the safe C dialects.

All software-based approaches are prone to electromagnetic interference by relying on considerable amounts of protection-relevant data situated in RAM. In type-safe languages, the protection relies on the integrity of reference values, array bounds and runtime type information. Fat pointer approaches similarly rely on the integrity of pointer properties. Even the more low-level SFI approaches base on an in-memory permission table that is comparable in size to a page table.

The TCB size includes a compiler or sophisticated static analyses for all software-based approaches except the ISA-level approaches, which only rely on a comparably simple verifier. Such a verifier still exceeds the software portion of simple hardware-based mechanisms.

CPU Protection in Software-Based Approaches

In the absence of an unprivileged processor mode, CPU protection can either be enabled by ensuring that the application does not make uncontrolled use of privileged instructions that disable the preemption mechanism of the operating system (for example, by disabling the interrupt handling), or the application can be instrumented to regularly yield to the operating system. Type-safe languages provide no mechanism to use privileged instructions[a] and therefore the absence of privileged instructions in the application code is guaranteed. C provides an inline assembly mechanism to make use of such instructions, and binary code may contain such instructions anyway. Depending on the target architecture, however, the absence of privileged instructions in the application code may be easily checkable by a simple verifier if the instruction set provides dedicated CPU instructions. Such a verifier can then be used for any software-based approach. If this is not possible, or the application is deliberately allowed to use privileged instructions, an instrumentation approach as done in t-kernel can be used to ensure that the operating system is invoked in statically bounded intervals. Such approaches can be used independent of the respective protection mechanism. It should be noted that the unprivileged processor mode could no longer guarantee CPU protection either, if the application is given access to privileged instructions. Given that the application can contain low-level driver code and interrupt service routines, this scenario is not unrealistic and OSEK/VDX provides explicit operating system services to control the interrupt handling.

[a]Assuming that the runtime system does not provide explicit interfaces for that purpose.

	HW Req	Flex	Leg Code	Cost	Fast IDC	EMC	Prot Level	Pred	CPU Prot	TCB Size
MPU	○	⊖	○	⊕	✖	✔	Sandbox	✔	✔	⊕
Segments	⊖	⊖	⊕	⊕	✖	✔	Sandbox	✔	✔	⊕
Pages	⊖	○	⊕	○	✖	✖	Sandbox	✖	✔	○
Mondrian	⊖	⊕	⊕	○	✖	✖	Sandbox	✖	✔	○
UMPU	⊖	⊕	○	⊖	✔	✖	Sandbox	✖	✖	○
HardBound	⊖	⊕	⊕	⊖	N/A	✖	MemSafe	✖	✖	⊖
t-kernel	⊕	⊖	⊕	⊖	N/A	✖	Sandbox	✖	✔	○
XFI	⊕	⊖	⊕	⊖	N/A	✖	Sandbox	✔	✖	○
Harbor	⊕	⊖	⊕	⊖	✔	✖	Sandbox	✔	✖	○
(Wahbe's) SFI	⊕	⊖	⊕	○	✔	✖	Sandbox	✔	✖	⊖
CCured	⊕	⊕	⊕	⊖	N/A	✖	MemSafe	✔	✖	⊖
MSC-GC	○	⊕	○	○	N/A	✖	MemSafe	✔	✖	⊖
SW Segment Protection	⊕	⊖	⊕	⊕	N/A	✖	Sandbox	✔	✖	⊖
SoftBound	⊕	⊕	⊕	⊖	N/A	✖	MemSafe	✔	✖	⊖
Fail-Safe C	⊕	⊕	○	⊖	N/A	✖	MemSafe	✔	✖	⊖
Cyclone	⊕	⊕	○	⊖	N/A	✖	MemSafe	✔	✖	⊖
Deputy	⊕	⊕	○	○	N/A	✖	MemSafe	✔	✖	⊖
Type-Safe Language + Isolation	⊕	⊕	⊖	○	✔	✖	TypeSafe	✔	✔	⊖
MPU + Multi-JVM	○	⊕	⊖	○	⊖	✔	TypeSafe	✔	✔	⊕

Table 2.1: Comparison of Memory Protection Mechanisms

2.5 Decision for MPU-Based Protection and a Multi-JVM

Table 2.1 contains an overview of the rating criteria that I introduced in Section 2.2 for all presented memory protection approaches. From the discussed techniques, I selected MPU-based protection and software-based memory protection on the basis of a multi-JVM, which when combined provide fine-grained protection at the semantic level of programming-language-level objects, robustness against transient hardware-faults and a small TCB. What follows is a discussion of the reasons that led to this decision.

For safety-critical applications deployed in environments prone to EMI, the robustness of the approach with respect to EMI is crucial. Given the shrinking structure sizes in hardware manufacturing, hardware becomes more prone to transient faults caused by EMI and tolerating such faults will gain relevance in the future. As shown in Table 2.1, only the two coarse-grained hardware protection approaches of the ranged-based MPU and the segmented protection are robust in this aspect. Safety-critical applications may also require a formal verification or other processes to ensure the correct functioning of the protection system to sufficient certainty. Such processes are more feasible for approaches that build on a small TCB than for those that rely on the correct functioning of a complex software package. This is a second aspect where MPU and segmented protection stick out. Of the two, however, only MPUs are present in the low-cost embedded segment.

The protection provided by an MPU is coarse-grained and merely provides spatial isolation. For a more fine-grained type of protection, and to provide the option of memory protection for low-cost microcontrollers without an MPU, a software-based approach can be used. The SFI approaches thereby provide little additional protection properties over the MPU, and implementations for embedded systems such as Harbor showed unacceptably high overheads. Language- and most compiler-based techniques provide protection at the granularity of programming-language-level objects. With the exception of type-safe languages, however, none of these approaches addresses the issue of dangling pointers without requiring the use of specialized forms of memory management, which requires experienced programmers, is not applicable to all programs and may dramatically increase the memory requirements for certain usage patterns. Type safety prevents semantic programming errors in addition to the memory access errors prevented by memory-safe approaches. Finally, type safety is a feature of most newly developed programming languages, so it can be expected that such languages will also find their way to the deeply embedded domain in the long run.

From the multitude of type-safe programming languages in circulation, Java is attractive for a number of reasons. Firstly, the syntax of Java is very similar to that of C++. Although the effort of porting C applications to Java is presumably higher than the adaption to one of the safe C dialects, the syntactic similarity facilitates this process. Due to the wide spread of Java in other computing domains and in education, there is a high number of trained programmers available that has already introduced a shift from Ada towards Java in the real-time systems domain [82]. A great amount of research has led to standard programming interfaces for low-level programming, and memory management and execution models suitable for using Java in real-time systems. Safety-critical Java (SCJ) is in the progress of finalization and includes a new verification-friendly execution model for Java. With KESO, a multi-JVM for deeply-embedded system exists that supports the targeted devices and provides a spatial isolation concept.

The bottom row of Table 2.1 shows the expected protection properties of combining MPU protection with a multi-JVM such as KESO. Some factors, such as the cost, accumulate negatively, whereas both approaches complement each other in other aspects such as the protection flexibility and level of Java, combined with the small TCB and the robustness with respect to EMI of the MPU. In the combination, the MPU protection can be regarded as a simple and robust safety net that ensures fault containment and compensates the TCB size and EMI vulnerability of the multi-JVM. In Chapter 3, I will explore in more detail the synergies that one can benefit from by combining MPU protection and a multi-JVM.

3 Analysis Application Model and Protection Levels

In this chapter, I develop the application model of the framework. I begin with a presentation and analysis of the protection model defined in AUTOSAR OS [10], a model developed by a consortium formed of automotive suppliers and OEMs that suits the requirements of this domain. Following, I introduce the I4Copter project, which serves as a running example throughout the remainder of this thesis, and how the I4Copter control software maps to the AUTOSAR OS model. I use this model as a base for the framework and extend it to include software-based memory protection and combinations of hardware- and software-based memory protection. The AUTOSAR OS model includes optional graduations to vary the degree of memory protection provided. I examine possible graduations for the software-based protection based on the multi-JVM concept, and the cost and synergies of using a combination of both mechanisms to isolate an application.

3.1 The AUTOSAR OS Application Model

AUTOSAR OS [10] defines an application model that includes MPU-based memory protection with a mandatory base set of isolation guarantees and a set of optional extensions that provide additional error detection. Figure 3.1 shows the AUTOSAR OS application model, the possible isolation boundaries, and which parts are optional (blue requirement boxes) and which are mandatory (red requirement boxes). Table 3.1 shows the requirements cited from the AUTOSAR OS specification [10].

The basic realms of spatial isolation are called *OS-Applications* in AUTOSAR OS. The control flow abstractions provided by AUTOSAR OS are tasks, which are comparable to the real-time threads in the real-time specification for Java (RTSJ), and

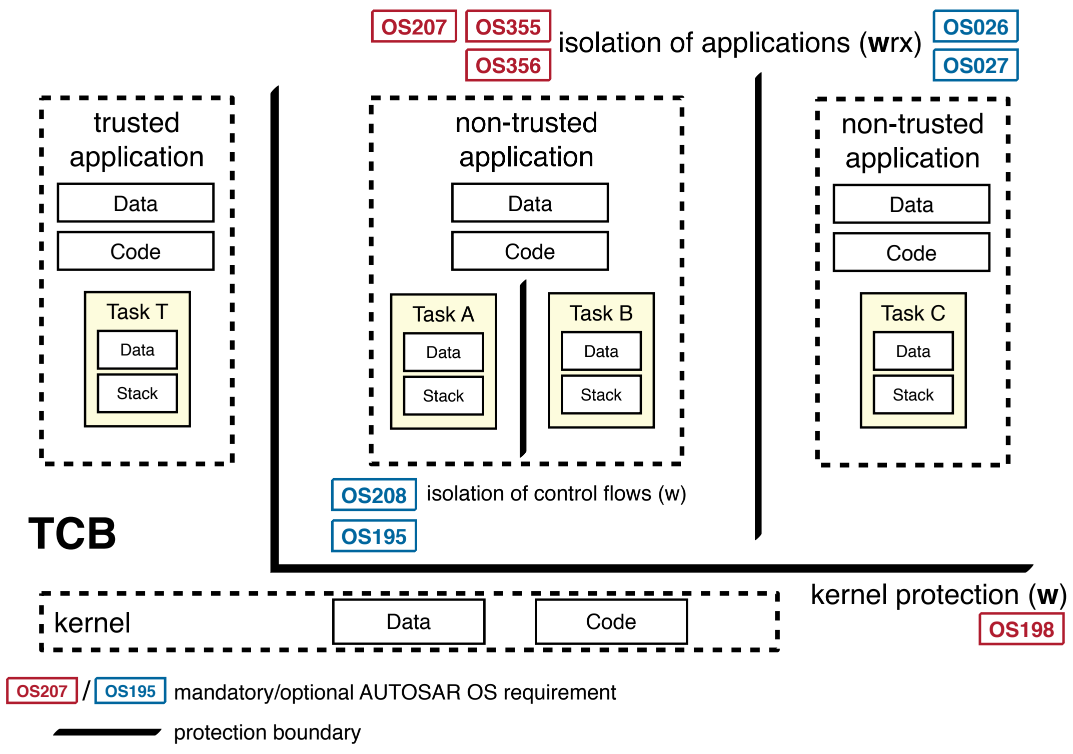


Figure 3.1: The AUTOSAR OS Application Model

interrupt service routines, which are comparable to the asynchronous event handlers defined by the RTSJ. Each control flow is statically assigned to an OS-Application that defines its memory access permissions. Memory access control is based on the different memory segments of an OS-Application. Each OS-Application can have a private data segment, and each control flow has a stack and an optional private data segment. For code, there exist both globally shared segments as well as private code segments available to only one OS-Application.

3.1.1 Layers of Protection

AUTOSAR OS distinguishes *trusted* and *non-trusted OS-Applications*. Trusted OS-Applications are not subject to memory access control and, together with the kernel, form the trusted computing base (TCB) of the system. The main purpose of trusted OS-Applications is to extend the functionality of the kernel with additional services. Kernel protection is a mandatory requirement that defines that the data segments of the trusted code base need to be protected against modifications from non-trusted OS-Applications.

The next layer provides the actual spatial isolation of non-trusted OS-Applications. Support for write protection, which prevents a non-trusted OS-Application from

	Kernel Protection
OS198	The Operating System module shall prevent write access to its own data sections and its own stack from non-trusted OS-Applications.
	Isolation of Applications
OS207	The Operating System module shall prevent write access to the OS-Application's private data sections from other non-trusted OS-Applications.
OS355	The Operating System module shall prevent write access to all private stacks of Tasks/Category 2 ISRs of an OS-Application from other non-trusted OS-Applications.
OS356	The Operating System module shall prevent write access to all private data sections of a Task/Category 2 ISR of an OS-Application from other non-trusted OS-Applications.
OS026	The Operating System module may prevent read access to an OS-Application's data section attempted by other non-trusted OS-Applications.
OS027	The Operating System module may provide an OS-Application the ability to protect its code sections against executing by non-trusted OS-Applications.
	Isolation of Control Flows
OS195	The Operating System module may prevent write access to the private data sections of a Task/Category 2 ISR of a non-trusted application from all other Tasks/ISRs in the same OS-Application.
OS208	The Operating System module may prevent write access to the private stack of Tasks/Category 2 ISRs of a non-trusted application from all other Tasks/ISRs in the same OS-Application.

Table 3.1: Memory Protection Requirements from the AUTOSAR OS Specification

writing to data segments of another OS-Application, is mandatory. In addition, the operating system *may* prevent non-trusted OS-Applications from executing code from other OS-Applications' private code segments or from reading data from other OS-Applications' private data segments.

The third layer of isolation protects the control-flow-local data (that is, its stack and optional local data segment) from being modified by other control flows of the same OS-Application. This layer is fully optional and only comprises write protection.

3.1.2 Required Isolation Properties

The mandatory and optional requirements of the AUTOSAR OS model show, which isolation properties the automotive industry considers as necessary, and which as optional. To recapture the motivation: The main reason for spatial isolation in mixed-criticality systems is the containment of faults. The establishment of write protection that protects the data of the kernel and other applications from direct modification by a non-trusted OS-Application provides containment of faults that occur within the isolated non-trusted OS-Application.

Restricting code execution and read accesses to the private segments of an OS-Application provides additional error detection capabilities: It detects certain cases where an OS-Application does not behave as expected, but does not otherwise contribute to fault containment. Notably, the execution of arbitrary code does not pose a problem as long as the data of the kernel and other OS-Applications is protected from modification. Restricting read accesses has the additional property of hiding possibly confidential data of an OS-Application, for example cryptographic keys. This is a security feature, but normally not relevant for safety-motivated memory protection, where no malicious applications are assumed in the system.

Permitting global read and execution permissions has several benefits. Global read permissions enable the efficient implementation of unidirectional data flow among OS-Applications and operating system services that only query the state of the operating system. In addition, no additional MPU regions are required for code segments and read-only data segments, leaving more regions available for memory regions that write access should be granted for. As discussed in Chapter 2, MPU protection is bounded by a fixed number of regions, and can provide memory protection in a predictable and efficient manner only if the number of regions is sufficient for all data areas that an OS-Application needs to access. Lastly, the enforced protection of control-flow-local data can detect certain classes of errors, most notably stack overflows, but is not needed for fault containment.

3.1.3 Graduations of Hardware-Based Memory Protection

The protection varies in the memory access types that are constrained (that is, read, write and instruction fetch memory accesses) and the isolated entities. If MPU-based protection is used, isolation of the trusted code (kernel protection) and write protection are mandatory features, as the two provide the minimum base needed for the reliable functioning of MPU protection. Application isolation would also be mandatory according to the AUTOSAR OS requirements, but I opted to leave it optional in my model because application scenarios exist that do not necessarily require application isolation.

Kernel protection is sufficient for some scenarios where only the availability of the infrastructure software needs to be ensured. An example is the protection of a boot loader that allows replacing the installed application modules and therefore ensures that a node does not become unreachable. One domain for such settings is wireless sensor networks, where the nodes are often not physically accessible anymore after having been deployed in the field and a failure of the infrastructure software would mean that the node is lost. This problem can be solved by kernel protection, and the t-kernel (Section 2.4.1.1) provided just this level of isolation for that target domain.

Restriction of read and instruction fetch accesses is an optional error detection facility, as is the isolation of control-flow-local data. The latter is only feasible if application isolation is also used.

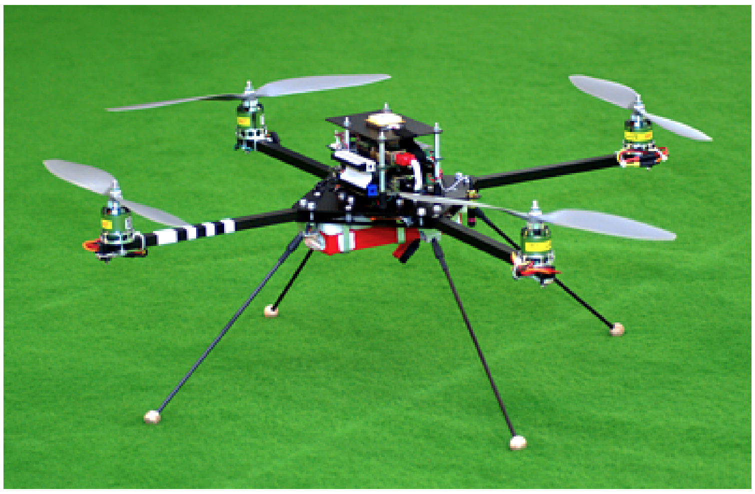


Figure 3.2: Photo of the I4Copter, Version 2.4 (“Apollo”)

3.2 The I4Copter Application

The I4Copter [120] is a quadrotor helicopter, an aircraft driven by two pairs of oppositely-spinning rotors. Figure 3.2 shows a picture of the I4Copter. The I4Copter is a suitable demonstrator application for a safety-critical, hard real-time system. Quadrotor helicopters are simple in mechanics compared to regular helicopters, but the four rotors are fixed and the aircraft can only be stabilized by varying the rotation speeds of the rotors, requiring a fairly sophisticated controller to achieve a stable flight behavior.

Figure 3.2 shows a photo of the I4Copter. It is based on an Infineon Tricore TC1796 microcontroller (32-bit, 150 MHz, 2 MiB program flash, 64 KiB SRAM) with a heterogeneous MPU providing two sets of memory protection regions, containing two code and four data regions each. The used Hightec Easyrun board is equipped with one MiB of persistent external MRAM. The I4Copter is controlled by a radio remote control and additionally transmits monitoring and debugging data by wireless LAN to a base station.

3.2.1 Core Subsystems of the I4Copter Framework

Figure 3.3 shows the five core subsystems of the I4Copter control software. The dashed boxes depict one of the subsystems each. The shaded boxes within each subsystem represent the control flows it contains, which are either tasks or interrupt service routines (ISRs). The lines connecting the different subsystems show the data exchange paths between subsystems. The five subsystems have the following function:

The Signalprocessing subsystem periodically collects the data of the sensors. The I4Copter is equipped with a multitude of sensors: three gyroscopes to determine

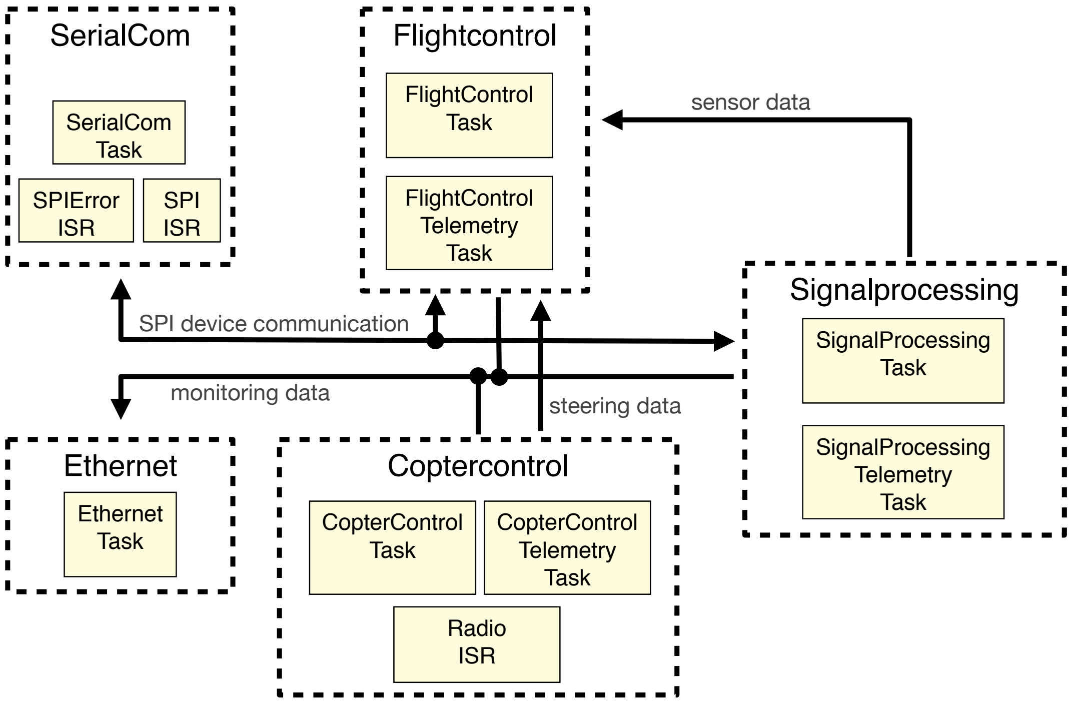


Figure 3.3: Subsystems and Data Exchange Paths in the I4Copter

the angular position, an accelerometer that measures the proper acceleration, a compass, two pressure sensors, and two proximity sensors to determine the altitude above ground level utilizing either infrared or ultrasonic. The Signalprocessing subsystem mainly comprises driver and filtering code for all these sensors. Signalprocessing is periodically triggered every three milliseconds.

The Coptercontrol subsystem contains the main behavioral logic and mode management. Coptercontrol manages a global flight mode that is provided to and affects the behavior of all other subsystems. The mode management is implemented as a finite state machine. State transitions are triggered when certain input data exceeds a configured threshold, or when a mode change is suggested by another subsystem. In addition, Coptercontrol also manages incoming steering commands, which can currently be given either by a radio remote control or sent by wireless LAN. The driver for the radio remote is also part of the Coptercontrol subsystem. The chosen steering commands are provided to other subsystems. Finally, when the Coptercontrol subsystem detects a connection loss to both remote control sources, it provides an emergency mode management to bring the aircraft down. The Coptercontrol subsystem is periodically triggered every 21 milliseconds.

The Flightcontrol subsystem contains the controllers that manage the flight behavior of the aircraft to different aspects. The controllers take as input the sensor data provided by the Signalprocessing subsystem and the steering commands provided by the Coptercontrol subsystem, and calculate as output the individual thrust levels of the four rotors to reach the flight attitude aimed for. The core controller is the flight attitude controller that stabilizes the aircraft. The second controller, which is currently in experimental state, is the altitude controller that aims to control the height of the aircraft, an otherwise manual process (for example, it can keep the aircraft at its current height if no steering commands to change the height are received). Both controllers are modeled using Simulink MATLAB. The code is generated from these models using Simulink Real-Time Workshop. Flightcontrol is periodically triggered every nine milliseconds.

The SerialCom subsystem serializes accesses to the SPI bus by other subsystems. Currently, the Signalprocessing subsystem accesses the SPI bus to control SPI attached sensors, and the Flightcontrol subsystem requires access to the SPI bus to propagate the computed thrust levels to the engine controllers, which are also attached to the SPI bus. SerialCom is triggered by events sent from the client subsystems.

The Ethernet subsystem manages the reception and transmission of UDP packets. Received packets contain remote control commands and are provided to the Coptercontrol subsystem. Outgoing packets contain telemetry data of the various subsystems that allow the base station to remotely monitor the internal state of the different subsystems, for example as a debugging aid. Telemetry data is collected periodically each 27 milliseconds by dedicated low-priority tasks in the Coptercontrol, Flightcontrol and Signalprocessing subsystems. These tasks collect the data for the respective subsystem and provide it to the Ethernet subsystem, where the pieces are assembled to a network packet and transmitted. The Ethernet subsystem does not contain a network protocol stack or interface drivers. These facilities need to be provided by the operating system. The Ethernet subsystem is triggered aperiodically[1] by the operating system to handle incoming packets, or by the other subsystems when telemetry data needs to be transmitted.

Besides these core subsystems, there currently exist two additional optional subsystems. One allows controlling an optional camera that is movably mounted to the aircraft. The second is in experimental state and aims at enabling the aircraft to follow a preprogrammed waypoint plan in the form of GPS coordinates. These subsystems are disabled in my evaluation configuration and not further considered in this thesis.

Table 3.2 summarizes all control flows in the I4Copter application and also shows the priorities and, for periodic tasks, the activation period. The priority space is

[1]The terms *periodic* and *aperiodic* are used here with respect to the activation type, not with respect to the type of a possibly associated deadline as in *aperiodic* and *sporadic* tasks.

Component	Control Flow	Priority	Period
Coptercontrol	CopterControlTask	10	21 ms
	CopterControlTelemetryTask	5	27 ms
	Radio RX ISR	12 (IRQ)	-
Ethernet	EthernetTask	7	-
Flightcontrol	FlightControlTask	12	9 ms
	FlightControlTelemetryTask	4	27 ms
SerialCom	SerialComTask	15	-
	SPI RX ISR	10 (IRQ)	-
	SPI Error ISR	11 (IRQ)	-
Signalprocessing	SignalProcessingTask	13	3 ms
	SignalProcessingTelemetryTask	3	27 ms
Operating System	Ethernet RX ISR	7 (IRQ)	-

Table 3.2: Control Flows' Properties in the I4Copter

divided into task level priorities and ISR level priorities. A higher priority number represents a higher priority level, with ISR level priorities being above all priorities of the task level. The Ethernet RX ISR technically belongs to the operating system, but still needs to be considered as it affects the application schedule.

The I4Copter is not only interesting as an evaluation scenario for being a real-world application (as opposed to a synthetic benchmark), but also because its subsystems cover a quite broad range of code characteristics. While the Signalprocessing subsystem consists mostly of low-level driver code, Flightcontrol contains higher-level code that is computationally intensive in single-precision floating-point arithmetic. Coptercontrol has a mixed code-base containing a driver and a state machine. Finally, the SerialCom and Ethernet subsystems contain little internal logic and are relatively communication-intensive with other subsystems and the operating system.

3.2.2 Mapping to the AUTOSAR OS Application Model

The I4Copter has an operating system abstraction layer that allows it to run on Hightec's PXROS-HR [94] and operating systems implementing the AUTOSAR OS interface[2]. In both cases, each of the I4Copter's subsystems is placed in a memory protection realm. In the case of AUTOSAR OS, each of the subsystems is mapped to a non-trusted OS-Application.

[2]In its current state, the I4Copter still requires the operating system to provide a messaging mechanism that is not part of the AUTOSAR OS standard.

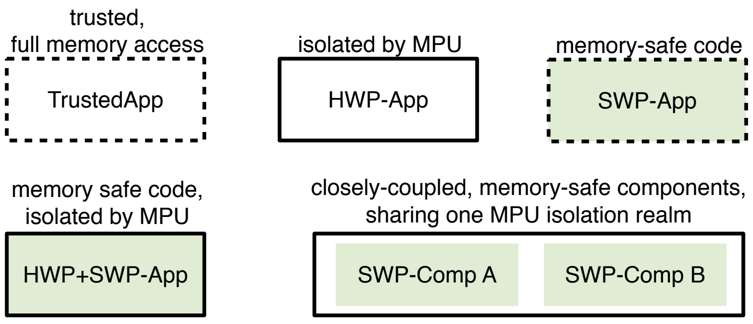


Figure 3.4: Isolation Variants

3.3 Model Refinement: Software-Isolated Components

I have so far introduced the protection model of AUTOSAR OS that defines trusted OS-Applications, which are not subject to any memory access restrictions, and non-trusted OS-Applications, whose memory accesses are constrained to their local data segments by employing an MPU. The type of memory accesses that are constrained always comprise store accesses to warrant fault containment, and optionally loads and instruction fetches to provide additional error detection or protection of confidential data for security-relevant applications.

A straightforward approach to extend this model by software-based memory protection would be to use the entity that defines hardware-based memory protection realms, that is non-trusted OS-Applications, for software-based protection realms as well. As discussed in Section 2.5, however, MPU-based protection is rather coarse-grained and data exchange among isolated entities may incur the overhead of context switches. Consequently, it can be expected that closely coupled software components are placed in the same OS-Application. On the other hand, language-based memory protection is very flexible at the granularity of objects, and data can principally be exchanged without requiring an OS-level context switch. Following the conclusion in Section 2.5 that the MPU can provide a robust safety net for software-isolated components, I add the entity of *software-isolated components* to the model, which are assigned N:1 to OS-Applications. This model is similar to the protection model in Singularity [3], where also multiple software-isolated processes can be contained in one hardware-isolated process. In the example of the I4Copter application, different drivers in the Signalprocessing subsystem could be placed in different software-isolated components, or the telemetry tasks could be placed in a different component than the task performing the main function of a subsystem to protect the core functionality of a subsystem from faults in the telemetry task.

Figure 3.4 shows the possible isolation variants in this model, and also introduces the notation to indicate a particular type of isolation used in other figures. The first two types represent the application types present in AUTOSAR OS, unconstrained trusted applications and MPU-protected non-trusted applications, indicated by the solid box border. The third type is a software-isolated component mapped 1:1 to an OS-Application, that is memory-safe code, indicated by the shaded background, not constrained by an MPU. The fourth type combines a memory-safe component with an MPU-isolated OS-Application. The fifth variant finally shows an N:1 mapping of multiple memory-safe components in the same MPU-isolated OS-Application.

3.4 Graduations of Software-Based Memory Protection

Section 3.1 introduced AUTOSAR OS' application model and the variability that it defines with respect to its MPU-based memory protection features, which allow to increase the memory protection coverage at the cost of increased overhead and reduced flexibility. For example, enabling the restriction of read accesses increases the level of memory protection, as it enables the detection of unexpected read accesses, most likely program bugs, or allows applications to hide confidential data from other applications. It comes at the cost of increased overhead, as read-only system services (for example, get the identifier of the currently running task) now need a context switch to privileged mode as state-changing system services do. Or, as another example, protecting task-local data such as the task stacks from modification by other tasks in the same OS-Application, adds the ability to detect errors, such as stack overflows, but requires an additional region per task, which is not available anymore for other OS mechanisms such as shared memory areas. This constrains the flexibility of the configuration, and may even render it unfeasible for some applications.

In this section, I explore how a similar trade-off between the costs imposed to enforce memory protection and the coverage of the protection can be achieved for the software-based part of my framework, the memory protection based on the type safety of Java and the isolation established by a multi-JVM concept. To begin, I first discuss what overhead that is introduced by Java can be attributed to memory safety.

3.4.1 Memory-Protection Overhead Imposed by Java

The variability choices provided for MPU-based protection are not directly transferable to Java. A sandboxing approach applies an external barrier to an otherwise unconstrained program. This barrier can be moved (between the kernel and the OS-Applications, between the OS-Applications, between the tasks within an OS-Application) and its permissiveness can be varied (to constrain only writes, or to additionally constrain load operations or instruction fetches). For memory-safe code on the basis of a type-safe language, however, the safety is established at the level of the language, based on types and references. Types eventually define *how* the different fields of a memory fragment holding an instance of that type can be accessed. The typed reference defines *where* the application can access the memory. The created instances of types (objects) and the references that exist in the application form an object graph, in which the nodes represent objects and the directed edges represent references residing in a field of the originating object and referring to the pointee

Operation	Bytecode Instructions
	`null` Checks
Load Instance Field	`getfield`
Store Instance Field	`putfield`
Invoke Instance Method	`invokeinterface`, `invokespecial`, `invokevirtual`
Read Array Length	`arraylength`
Load Array Element	`[abcdfils]`aload
Store Reference-Array Element	`[abcdfils]`astore
Throw Exception	`athrow`
	Array-Bound Checks
Load Array Element	`[abcdfils]`aload
Store Array Element	`[abcdfils]`astore
	Subtype Checks
Cast Reference to Subtype	`checkcast`
Store Array Element	`aastore`
	Negative Array Size Checks
Create New Array	`anewarray`, `multianewarray`, `newarray`
	Division by Zero Checks
Integer Division	`idiv`, `ldiv`
Integer Remainder	`irem`, `lrem`

Table 3.3: Checked JVM Instructions

object. The object graph is rooted in external[3] reference fields. In Java, these fields are static fields of classes and slots on the stack. A Java application can only access memory in these root areas and of the objects contained in its object graph. The memory safety relies on the consistency of the object graph and control flow integrity [1]. The spatial isolation in a multi-JVM builds on the invariant that the root areas and object graphs of the isolated entities are disjoint.

These properties are largely established by the design of the language or checked by the compiler ahead of time. The runtime overhead that can be attributed to memory safety is all code, data and execution time that are spent at runtime on retaining the integrity of the object graph and the control flow. In Java, these are the following:

Safe Deallocation For applications that require the dynamic allocation and deallocation of memory, a safe deallocation mechanism is required to avoid the creation of dangling references. As discussed in Section 2.4.3.2, this requires either the use of specialized memory management mechanisms such as scopes, or a garbage collector. For the former, the runtime overhead depends on the respective technique. The RTSJ's scoped memory areas, for example, restrict the use of references to such areas and require additional runtime checks to

[3]External with respect to the object graph, as these fields are not part of any object.

enforce these restrictions. A garbage collector needs the ability to traverse the object graph at runtime, which means that it needs to be able to identify the root reference nodes, reference fields within objects, reference values on the stacks of active tasks and possibly also reference values contained only in processor registers. The root reference fields and reference fields within objects can normally be made processible by the garbage collector by a reference grouping scheme and a suitable object layout, for example a bidirectional object layout as proposed by Sable VM [39]. A grouping scheme can also be utilized for the reference variables in a stack frame, but the garbage collector still needs the ability to traverse the different frames on the stack. To enable this, a common portable technique is to manage the stack frames in a linked list as originally proposed by Henderson [50]. Such additional runtime data structures add to the memory usage and execution time spent on maintaining the data structure at runtime.

Runtime Type Information Checked downcasts (that is, casts to a subtype of the declared type of the source reference) require the ability to identify the specific type of an object at runtime, and the ability to determine whether it is a subtype of the target type of the downcast. Array accesses may need to be bound checked, for which the size of the array needs to be available at runtime. The type of an object and the array size are typically stored with the object in a header area, which adds overhead to the storage size of the object. Further runtime type information, for example dispatch tables for dynamically bound method invocations, are not related to the memory safety, but rather are cost paid for the amenities of object-oriented programming. Similar runtime type information is, for example, also used for virtual method calls in the unsafe C++ language.

Runtime Checks Some Java operations cannot always be proven safe by the ahead-of-time compiler and therefore need to be checked at runtime. Table 3.3 shows the checked operations of the JVM, except for checks and exceptions related to Java monitors and dynamic class linking, both of which are not relevant in this work. These checks increase the code size of the program, add to the execution time, and, as discussed above, may require runtime type information that also increases the memory use.

Dynamic deallocation of heap memory is a feature that is often deliberately not utilized by applications developed for a safety-critical or a hard real-time context, as it introduces new issues such as external fragmentation that the developer needs to cope with. A common alternative approach is the use of application-level pools of homogeneous objects, which avoids the fragmentation issue. If a garbage collector is needed for some reason, however, the runtime data structures required to scan the object graph are fully needed. The Java runtime environment can still perform steps to reduce the runtime overhead added by these data structures by incorporating knowledge on the system model. For the general runtime type information, the principle is grossly the same. A compiler can tailor the required data structures to the application, for example by choosing the smallest possible data types for the runtime data for that the covered value range is still sufficient for the application.

Runtime checks on the other hand can pose a notable overhead in code size and also execution time if occurring in hotspots. A decent compiler can eliminate runtime checks for operations where a static analysis proves the check to always succeed. The `null` checks do normally not require specific analyses but are eliminated by the standard dead-code elimination that incorporates the results of a data-flow analysis. Static bound-check elimination techniques are well researched [74, 45, 63]. Such analyses are simple and effective in a local scope, for example multiple operations on a reference in the same basic block or procedure. In an inter-procedural scope, the effectiveness of such static analyses depends on the assumptions that the compiler is able to make. A just-in-time compiler can utilize its volatile runtime state for effective optimizations, but as this state changes (for example, as a new class is loaded) assumptions made may become invalid and a recompilation is needed. Just-in-time (JIT) compilation is, however, problematic in hard real-time systems for the effect on deterministic execution and the jitter that JIT compilation has. Although some JIT compilers can be tuned for better determinism (for example, IBM Websphere Real Time allows to side-step JIT compilation in a low-priority thread), the reached level of determinism (that is, the accuracy of the WCET bounds) is not as high as that provided by an ahead-of-time compiler. For resource-constrained systems, an additional issue is the resource consumption of the JIT compiler, which exceeds most common applications for such platforms in complexity and resource use. Ahead-of-time compilation therefore is a more attractive choice, but for an ahead-of-time compiler a closed-world assumption is needed to be able to effectively utilize whole-program static analyses.

In the following, I discuss two approaches to complement static analyses. The first approach is to offload runtime checks to hardware exception mechanisms. This approach does not change the behavior of the program. For the second approach, I examine the necessity of the most common Java runtime checks with respect to the memory safety of the program, and how runtime checks can be selectively omitted while still providing the minimum needed isolation guarantee of fault containment. This approach departs from the Java Virtual Machine specification and reduces a type-safe program to a memory-safe one, at the benefit of reduced runtime overhead.

3.4.2 Offloading Runtime Checks to the Hardware

Many larger microcontrollers come with a trap system that can detect and signal certain error conditions in the execution of a program. In this section, I analyze which of the common runtime checks could principally be performed in hardware. A simple case of a runtime check that is usually also performed by the hardware is the detection of a division by zero (given that the instruction set provides a division instruction). Therefore, if a JVM operation that is normally guarded by a runtime check triggers a hardware exception if the checked condition is not met, the software check can be omitted and the hardware exception can be leveraged to trigger the runtime exception. This saves both the code required by the runtime check as well as the time spent on executing it[4]. In this section, I discuss which mechanisms could be used. The concrete application to Java bytecode operation follows in Section 3.4.4.

3.4.2.1 Access Errors to Unmapped Addresses

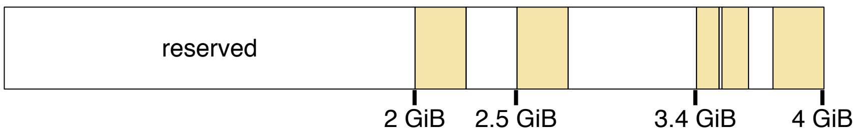
A particularly useful hardware exception mechanism is errors that are raised by the processor for accesses to address regions that are not backed by memory or mapped otherwise, for example bus access errors. Especially for microcontrollers with an address width of 32 bits, it is very common that large parts of the address space are unused, as the examples in Figure 3.5 illustrate. It shows the used (colored) and unused (white) portions of the address spaces of three microcontrollers. The shown memory maps show the designated address ranges. What is actually used depends on the actual configuration, for example optional external memories. I have included a real configuration with each example.

Infineon Tricore TC1796 On the TC1796 microcontroller [116], more than 75 percent of the address space is reserved. On any Tricore derivate, the initial eight bytes of the address space are generally reserved and the MPU provides a dedicated trap (memory protection NULL trap) for accesses to this region. This is similar to many other architectures and is for the reason that the address 0 is commonly used as a reserved address value. In addition, the entire lower two GiB of the address space are reserved on the TC1796 microcontroller, since these are used on processors of the line that are equipped with a memory management unit (MMU). Consequently, without any MPU configuration, memory accesses to the lower two GiB of the address space trigger a trap.

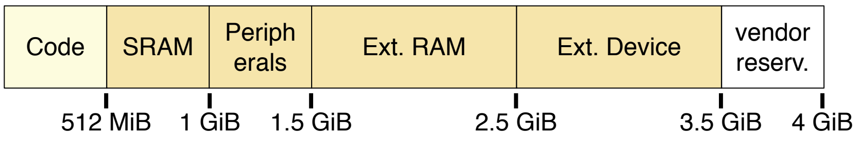
ARM Cortex-M3 On the ARM Cortex-M3 processor, the lower 512 MiB of the address space are mapped read-only to the program flash. Following is 512 MiB reserved for internal SRAM. Current implementations are equipped with memories that only back a small fraction of these designated address regions, for example Atmel's AT91SAM3U [6] line is currently equipped with at most 256 KiB of program flash and 48 KiB of SRAM.

Atmel AVR ATmega128 Atmel's AVR microcontrollers are 8-bit processors with an address width of 16 bits. One of the largest derivates of this line is the ATmega128 [7] (4 KiB SRAM, 128 KiB Flash), which is commonly found on sensor nodes such as the Crossbow MICA2. The AVR is an example of a processor without reserved regions in address space. Despite the address space being entirely used, the external

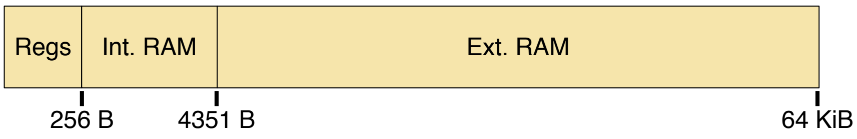
[4]The exception case will normally be slower when triggered by a hardware exception compared to a software check. As the name implies, however, exceptions are not considered to occur in the intended course of the program execution, so offloading the check to hardware optimizes the common case of a succeeding check.



(a) Infineon Tricore TC1796. Example Configuration: Infineon Triboard TC1796, 4 MiB Flash, 1 MiB SRAM



(b) ARM Cortex-M3. Example Configuration: Atmel AT91, SAM3U: 256 KiB Flash, 48 KiB SRAM



(c) Atmel ATmega128 Dataspace. Example Configuration: Crossbow MICA2: 4 KiB internal, no external RAM

Figure 3.5: Example Address Space Utilizations. White areas are unused, colored areas used regions of the address space. In an actual configuration, the regions reserved for external memories can be further reduced to the actual memory available in that setting.

RAM portion is commonly not fully utilized (for example, on the MICA2 there is no external RAM at all). AVR processors do not have a hardware exception mechanism. Writes to unmapped regions of the address space have no effect. It is therefore not possible to offload runtime checks to the hardware on AVR processors. The unused regions can still be helpful for the selective omission of runtime checks discussed in Section 3.4.4.

3.4.2.2 Utilizing an Unused MPU

For microcontrollers where the address space does not provide sufficient unused regions or an appropriate exception mechanism, for example microcontrollers with an address width of 16 bits, an MPU that is not used can be leveraged to achieve an identical behavior[5]. To achieve this, the MPU is statically configured to restrain memory accesses to the mapped regions of the address space, or only those regions that are used by all applications and the operating system. A requirement is that the MPU can be activated for code running in privileged mode. This is the case on the

[5]If the MPU isolation is used the checks can equally be shifted to the MPU, but the overhead of MPU-based isolation would also apply.

Infineon Tricore TC1796. The result is that a hardware exception is triggered on any access to unused address ranges.

3.4.3 Selective Omission of Runtime Checks

Offloading runtime checks to hardware exceptions allows saving the code and execution time of certain runtime checks without changing the behavior of the program or impacting its safety. Such hardware mechanisms are, however, not always available. In this section, I discuss a further step that can either be used in addition to hardware exceptions, or by itself if hardware exceptions are not available. Contrary to the previous approach, I now also consider a reduction of the provided safety level of the application traded for reduced code size and execution time overhead.

By omitting runtime checks based on criteria such as the type of the check and the bytecode instruction that triggers the check, a partially checked, gradually safe program can be generated. At the extreme side, all checks can be omitted, resulting in a program that does not suffer any overhead penalties but still benefits from increased dependability due to the checks successfully performed ahead-of-time compared to a program written in an unsafe language. Such a configuration would be a basis for a trusted application or a purely MPU-isolated application from the variants in Figure 3.4. A gradual selection can be based on the following criteria:

- The concrete microcontroller used in a mass product is usually the cheapest suited one from a line of functionally equivalent microcontrollers that scale in resources and price. However, even after choosing the smallest model from the line that still fulfills the requirements, the available resources are never utilized by 100 percent. Individual runtime checks need very little resources, but each check contributes to the dependability of a program. The spare resources on such a microcontroller can therefore be used to increase the dependability by adding a selected set of runtime checks to the application.

- The consequences of an omitted runtime check that would have failed are not the same for all checks. Depending on the type of check and the checked instruction, one can categorize the checks into those that only have a local impact on the current OS-Application and those with global impact. I highlight these differences using the example of the two most common types of runtime checks in Section 3.4.4.

- Bugs are not equally distributed over the code. According to McConnell [75], 80 percent of the bugs are within 20 percent of the software modules. Examples for software modules that may be more prone to faults than others are new software modules, which may be more likely to contain bugs than established existing software modules that have been used and tested by a broad user base for a longer time period, or software modules containing low-level code, for example device drivers. Software modules containing bugs are likely to contain

Operation	I4Copter	CD$_j$ 1.2	GNU Classpath 0.98
Load Primitive Field	860	1851	42545
Load Reference Field	1152	2408	98235
Store Field	782	1637	44411
Load Primitive Array Element	154	317	8263
Load Reference Array Element	20	451	9399
Store Array Element	360	662	73443
Store Reference Array Element	7	259	38443
Read Array Length	72	452	6946
Invoke Instance Method	2521	13754	292829
Throw Exception	132	605	15911
Cast Reference to Subtype	72	649	20007
Create New Array	45	190	7119
Integer Division/Remainder	39	200	2201
Null Checks	6060	22137	591982
Array-Bound Checks	534	1430	91105
Subtype Checks	79	908	58450
Negative Array Size Checks	45	190	7119
Division by Zero Checks	39	200	2201

Table 3.4: Frequency of Checked Instructions

further bugs, so runtime checks should preferably be added to such modules to help finding these bugs.

3.4.4 Impact Classification

A self-evident criterion to classify the safety value of runtime checks is the severity of the impact that the absence of such a check may cause to the system. In this section, I create a classification based on this criterion for the two most common types of runtime checks, `null` checks and array-bound checks. Table 3.4 shows how often the checked JVM instructions from Table 3.3 appear in three code bases. The first is the Java port of the I4Copter. CD$_j$ [61] is a benchmark application from the domain of real-time and embedded systems. Both of these applications are from the embedded domain and have different code characteristics than Java programs from other domains such as desktop or server environments. To complete the picture, I also added the numbers for a standard Java class library, GNU Classpath. While the standard class library shows a heavier use of downcasts than the embedded applications, the overall picture shows that `null` and bound checks are by far the most frequent check types for all three applications. I split safety checks into the following two groups, which I refer to as the *impact class* of a certain check:

Local Impact The omission of a check with the impact class *local* may result in a malfunctioning of the OS-Application in the context of which the check would normally have raised an exception. This defect does, however, not affect other OS-Applications.

Operation	**Impact Class**	**HW Offloading**		
		TC1796	**AT91SAM3U**	**ATmega128**
		`null` Checks		
Instance Field				
load reference	global	✔	✔	✖
load primitive	local	✔	✔	✖
store	local/global	✔	✔	✖
Array Operations				
load reference	global	✔	✔	✖
load primitive	local	✔	✔	✖
store	local/global	✔	✔	✖
get length	local	✔	✔	✖
Method Invocation				
dynamic binding	global	✔	✔	✖
static binding	global	➥ local	➥ local	✖
		Array-Bound Checks		
`null` array	*analogue to missing `null` check*	✖	✖	✖
valid array				
store	global	✖		
load	*analogue to missing `null` check*	✖	✖	✖

Table 3.5: Impact Classification

Global Impact The omission of a check with the impact class *global* may result in a malfunctioning of OS-Applications other than the one in the context of which the check would normally have raised an exception. The consequences of the defect may not be contained within the faulty OS-Application.

Omitting all checks of impact class local still retains fault containment and the spatial isolation of OS-Applications, a degree of protection comparable to the one provided by hardware-based memory protection with the help of an MPU.

Table 3.5 summarizes the results that I elaborate on in the remainder of this section, grouped by the type of runtime check. The first column contains a general description of the operation that causes the check; column two shows the impact class of the check; the last three columns show whether the check may be offloaded to the hardware for the three example platforms. In the following, I separately consider load, store and method invocation operations. I assume static knowledge of the address space layout of the target platform as with the three examples in Figure 3.5.

3.4.4.1 Predicting the Affected Memory Range

There are no wild pointers in type-safe languages. References or pointer values always point to an existing object, with the exception of one special `null` reference. This special value introduces the need to `null`-check operations on an object reference. If such a check is omitted, an illegal access takes place at a particular offset from the `null` reference. The actual address value of the `null` reference is known to and controlled by the compiler. It can principally be an arbitrary value that is known to never collide with a reference to a valid object. It is, however, common practice to use address 0 for the `null` reference. In contrast to unsafe languages, the absence of wild pointers enables to predict the affected address regions to a certain range. Knowing the target address of an operation is crucial for both determining if a check on a memory operation can be offloaded to hardware and determining the check's impact class.

Most such accesses address a fixed offset from the base address. This offset is statically known for each individual `null` check and the target address can exactly be predicted. These operations are accesses to instance fields and object header fields (for example, the array length or the runtime type). The maximum offset of an instance field depends on the number and types of members of the used Java classes, but is usually small in the area of approximately 20 bytes.

Operations that access array elements use a possibly computed index. The exact value set that this index may take at runtime may not be determinable at compile time. The largest possible offset is the size of the array's data type (which is statically known for each individual check site) times the maximum array index. While the Java Virtual Machine specifies array indexes of 32-bit signed integer data type, a Java implementation may provide the option to the programmer to choose a smaller data type, which would reduce not only the size of the array header but also the memory range off the `null` address that could be affected by a store following an omitted `null` check.

3.4.4.2 Load Operations

For load operations, the data type of the loaded value makes the difference between local and global impact. While the erroneous load of a primitive value only affects the computations of the running OS-Application (local impact), interpreting a random value as a reference introduces wild pointers and breaks the consistency of the object graph (global impact). Subsequent operations on a wild reference may affect data of other OS-Applications.

If the target address of the load operation can be predicted to be part of an address range where a read access triggers a hardware exception, it can be offloaded to the hardware. In the examples, this works on the TC1796 with its initial two GiB of reserved address space. On the Cortex-M3, it only works if the `null` address value if moved to an unmapped block. On the AVR, checks cannot be offloaded for the lack of a hardware exception mechanism.

3.4.4.3 Store Operations

For store operations, the target memory regions that could be modified by the store operation determine whether the operation may affect the memory of other OS-Applications. In a system that constructively isolates OS-Applications by means of software-based memory protection, the data of different OS-Applications is not necessarily physically separated from each other, but may, for example, be allocated from a common heap.

If an illegal store happens at an address that belongs to an unused portion of the address space, it does not corrupt data of the OS-Application (or, cause other defects, such as a reconfiguration of a hardware device if device registers were mapped to the affected address) and I consider the impact being of class local. On the other hand, if the store affects a used portion of the address space, the effect depends on the type of data stored there. Giving an answer to this question—if possible at all—would be a very tedious task, so I assume a global impact in such cases.

Reference Validity Checks The impact class for any `null` check of an instance store operation can be statically determined for each individual check by cross-checking with the address space of the target platform whether the destined address is within the used address space or not. Table 3.5 contains the impact class *local/global* for this operation category because the impact class depends on the target address space, the value of the `null` reference and the offset of the respective field. If a store to the affected address is discarded the impact is local. If the hardware raises an exception the check can be offloaded. For instance fields, it is very likely that a reserved block of sufficient size for the largest offset can be found. For arrays, it depends on the size of the array index. With a 16-bit index, which should be sufficient on the presented example platforms, given the typically available amount of RAM, a reserved block can be found on the 32-bit processors. For the AVR, a 16-bit index can still cover the entire address space. An 8-bit index, on the other hand, may not be sufficient for some applications.

It may, however, not be necessary to determine the affected memory range for an array store operation this way. Listing 3.1 shows a C code example for the operations that a JVM may perform internally to process an array store operation for an array of integers. The defined `int_array_t` type shows how an integer array might internally be represented in memory, while the `iastore` procedure contains the code for the store operation, including the `null` and bound checks that precede the actual operation. The bound check performs another dereference of the array reference to read the array length. Just like an instance field, the length field is stored at a fixed and known offset from the base reference. For a `null` array reference and an address space where the length field is in an unmapped address region, chances are good that reading the field would either raise a hardware exception or return the value 0. The value 0 would cause a failure of the bound check for all non-zero-size arrays. For zero-size arrays, the store would be redirected to a small and constant offset, similar to an instance field access.

Array-Bound Checks The impact of an omitted bound check is mostly the same as omitting the associated `null` check, with one exception: a missing bound check on an array *store* operation to a *valid* array certainly affects valid application data,

```c
typedef struct {
    object_hdr_t header;
    int32_t      length;
    // actually data[length]
    int32_t      data[1];
} int_array_t;

void iastore(int_array_t *arr, int32_t index, int32_t value) {
    // null check
    if(arr == NULL)
        throw_nullpointer_exception();

    // bound check
    if(index < 0 || arr->length <= index)
        throw_array_index_out_of_bounds_exception();

    arr->data[index] = value;
}
```

Listing 3.1: Example: Array Access

since the access does not happen relative to the `null` address but to the address of the array, which lies within the application heap. The bound check of an array store operation is thus more important than the `null` reference portion and should not be omitted. Keeping the bound check may also support a safe null check omission on array store operations as discussed above.

3.4.4.4 Instance Method Invocations

Instance method invocations are the third type of operation that requires a `null` check. The cases of static and dynamic binding have to be considered separately.

For a statically bound method call, the object reference is not dereferenced during the call. A JVM can normally assume the `this` reference to be `null`-checked at the call site, and therefore does not need to `null`-check dereferences of the `this` reference. To allow the omission of the check, the compiler must be aware that it cannot assume the `this` reference to always be valid in an instance method. The behavior will still not comply with the JVM specification, as a method can be invoked on a `null` reference, but it is not an issue for the program's memory safety. Consequently, the omission of the `null` check can generally have the union of effects[6] of all the other `null` checks and thus belongs to the impact class global. If all other types of `null` checks can be offloaded to the hardware (or are done in software), the impact of omitting the `null` check is additionally reduced to local.

With dynamic binding, the address of the called candidate is determined at runtime from the actual type of the object. The type is read from the object header, similar

[6]Excluding array operations, as the `this` reference is never an array reference.

to the array length that is read during a bound check. In the case of a `null` reference, an unknown, possibly random, value is interpreted as the type and used to lookup the method implementation, resulting in a loss of control flow integrity. However, if the type read were a predictable value (for example, 0), one could benefit from this knowledge by pointing this slot of the dispatch table to an error handler to detect the error, trading a few bytes of RAM for the saved execution time. If reading the type from a `null` reference triggers a hardware exception, the check can be offloaded to the hardware. Statically bound method calls do not dereference the target reference during the call.

3.4.4.5 Summary

In this section, I presented a classification of runtime checks that allows to reduce the overhead caused by runtime checks while still retaining a memory-safe (but not type-safe) program. To implement the classification, the compiler needs to be aware of the address space layout and memory bus behavior of the target. How well the approach is applicable depends on the characteristics of the target platform. For load operations, primitive values can generally be treated to have a local impact. For store operations, an unused address range of sufficient size is needed, to that store operations do not have a side effect, or raise a hardware exception. The position of this range in the address space is principally not relevant, as the compiler should be able to freely choose an address value for the `null` reference, however, there may be technical reasons that require the use of the value 0. I presented three example platforms, including a small 8-bit microcontroller. A sufficiently sized address region is available in all of these platforms, assuming that the `null` reference address can be freely chosen by the compiler and that the internal data type used for array indexing is reduced to a size that is sufficient for the typically available amounts of memory.

In addition, I discussed how hardware exception mechanisms could be used to offload runtime checks to the hardware without added cost as long as the execution stays free of exceptions. This feature removes the runtime and code overhead of qualifying checks without changing the functional behavior of the program.

3.5 Chapter Summary

In this chapter, I developed the protection model for my framework. As a starting point, I used the mandatory and optional memory protection requirements from AUTOSAR OS, which have been developed by the automotive industry considering the requirements of this application domain and can be expected to gain wide practical relevance in the coming years. I adopted these requirements as the variation features for MPU-based memory protection in my framework. I extended the protection model to include type-safe Java components at a finer granularity than the MPU-isolated OS-Applications. These components can be mapped N:1 to OS-Applications to isolate them in a shared OS-Application from the rest of the system by means of an MPU, whereas the components among each other are isolated by the type safety of the component code and the logical separation of their data implemented on the basis of a multi-JVM concept. Finally, I extended the protection model with the possibility of offloading runtime checks to the hardware to reduce the cost of software-based protection qualifying platforms, and a selective runtime check omission that reduces the cost of software-based protection at the cost of rendering a type-safe component to a memory-safe one.

4 Design A Framework that Provides Memory Protection at Option

In this chapter, I develop the design and implementation of a framework that implements the protection model from Chapter 3.

To begin, I choose an AUTOSAR OS implementation and a multi-JVM around which the framework will be built. I opted for an explicit separation of the operating system and the JVM, which facilitates the integration of native C or C++ applications. The ability to integrate native applications supports the soft migration from existing application code, one of the objectives set in Section 1.2. I chose an AUTOSAR OS for the operating system since the hardware-based part of the protection model is largely adopted from the AUTOSAR OS protection concept; consequently an AUTOSAR OS implementation already implements a good portion of the required functionality. Following the selection, I discuss how the AUTOSAR OS and the multi-JVM are combined to form the basis of the framework, which I extend to support the missing features of the protection model. I discuss how the configurability of both MPU-based and software-based protection is implemented.

The outcome of this chapter is a framework that supports the full protection model for Java applications, and additionally supports running Java applications side-by-side with native applications, although isolated with interaction limited to the basic activation and notification services provided by AUTOSAR OS.

4.1 Selection of an AUTOSAR OS Implementation

To avoid unnecessary implementation effort, I attempt to find a suitable existing AUTOSAR OS implementation to provide a base operating system and ideally a partial implementation of the MPU-based part of the protection model. A hard criterion for my selection is the availability of the source code of the AUTOSAR OS implementation, for the ability to adapt and extend it to my needs without limitations. Additionally, the implementation should be easily extendable, configurable and support memory protection.

My search for an open-source AUTOSAR OS implementation yielded three candidates: Arctic Core [5] is an open-source AUTOSAR platform that is available under both the GPL and a commercial license model. Trampoline [16] is a noncommercial open-source implementation that in its current[1] beta version 2.0b49 supports the AUTOSAR OS 3.1 interface. CiAO [70] is an academic research operating system with the primary design goal of achieving a high-level of fine-grained configurability in even fundamental architectural properties by applying aspect-oriented programming principles [62], based on AspectC++ [105]. CiAO implements large parts of the AUTOSAR OS API, including memory and timing protection.

Arctic Core does currently not provide support for memory protection, but both Trampoline and CiAO do. With respect to configurability and extensibility, Trampoline follows a traditional approach common to many OSEK/VDX and AUTOSAR OS implementations: A code generator creates a tailored OS variant from the information contained within the application description. In the case of trampoline, its generator GOIL reads a configuration file in the OSEK implementation language (OIL) format [84]. AUTOSAR OS uses an XML-based configuration format with similar content. GOIL is based on templates and similar in use to C-preprocessor-based configuration. CiAO, on the other hand, has been built with fine-grained configurability as a first-class design goal; it achieves separation of concerns and avoids code tangling by applying aspect-oriented programming. Compared to Trampoline, CiAO is configurable at a finer-grained level. In addition to the information contained within the system configuration file, CiAO uses Kconfig, the configuration tool of the Linux kernel, to provide fine-grained configuration of the operating system. Another advantage of CiAO is that it supports the Infineon Tricore architecture, which is used on the I4Copter. I therefore select CiAO as the AUTOSAR OS implementation to be used in my implementation, as it provides support for the Tricore architecture, including memory protection, and for its configurable and extensible architecture.

4.1.1 CiAO Application Model

CiAO implements the application model of AUTOSAR OS with some extensions. Trusted and non-trusted OS-Applications form the protection realms. To communicate between applications, CiAO provides a synchronous remote procedure call (RPC) [17]

[1]Current as of April, 19 2012.

mechanism, *trusted functions* and *non-trusted functions*. Only trusted functions are specified in the AUTOSAR OS specification [10].

Trusted functions are functions exported by a trusted OS-Application to be called from other (trusted or non-trusted) OS-Applications. A trusted function is executed in privileged, trusted context. With this mechanism, trusted OS-Applications can extend the system services with additional functionality. The non-trusted functions supported by CiAO are the natural extension of trusted functions to non-trusted OS-Applications. A non-trusted OS-Application can export non-trusted functions that can be called from other (trusted or non-trusted) OS-Applications. A non-trusted function is executed in the unprivileged protection context of the exporting OS-Application.

4.1.2 Introduction to AspectC++

To understand the following sections on the implementation of configurable hardware-based memory protection in CiAO, a basic understanding of the concepts and terms of AspectC++ [105] is required. In this section, I give an introduction to AspectC++. The introduction is not complete and limited to the elements of AspectC++ that are used in the following sections. A comprehensive description of AspectC++ is available in the AspectC++ language reference [121].

AspectC++ extends C++ by aspect-oriented programming [62]. AspectC++ provides a static aspect weaver, which performs a source-to-source transformation of a C++ code base, applying a collection of aspects to the target code base to produce the woven output code base. The woven source code is then compiled by a standard C++ compiler. The key concepts in AspectC++ are *join points*, *pointcuts* and *advice*. A join point is a specific position in the static structure of the program or the dynamic control flow, at which a program transformation defined by a piece of advice can be applied.

4.1.2.1 Pointcuts and Join Points

A pointcut describes a set of join points in the form of a declarative pointcut expression in the join point description language of AspectC++. There are two types of pointcuts, name pointcuts, given as match expressions, and code pointcuts, given as pointcut expressions composed of pointcut functions that are applied to name pointcuts.

A match expression is a pattern string that is matched with identifiers of the target program such as class or function names. A match expression may contain wildcard operators (`%`, `...`) for parts of the identifier, parameter lists and return types of methods. For example, the match expression

```
    pointcut pcFoo() = "int foo%(...)";
```

matches all functions whose name starts with *foo* and that return an integer, irrespective of the parameter list. AspectC++ allows to name pointcut expressions by declaring a pointcut. In the above example, a pointcut `pcFoo` is declared as a name for the example match expression.

Pointcut functions provide the set of code join points in the control flow of the program that match a provided match expression. The following pointcut functions are used in the following sections:

- `call()`: Provides the set of all call sites to functions matched by the provided name pointcut.
- `execution()`: Provides the set of all bodies of the functions matched by the name pointcut.
- `within()`: Provides all code join points within the bodies of the methods matched by the provided name pointcut.

The following example shows a use of the `call()` pointcut function on the pcFoo() pointcut declared above, which provides the code join points for all call sites to the functions in the `pcFoo()` pointcut:

```
call(pcFoo())
```

In addition, AspectC++ provides standard set operations that can be applied to sets of join points:

- Unions (operator `||`)
- Intersections (operator `&&`)
- Complements (operator `!`)

In the following example, the intersection operation is used to limit the join points from the previous example to those calls that are not within the body of the `bar()` function:

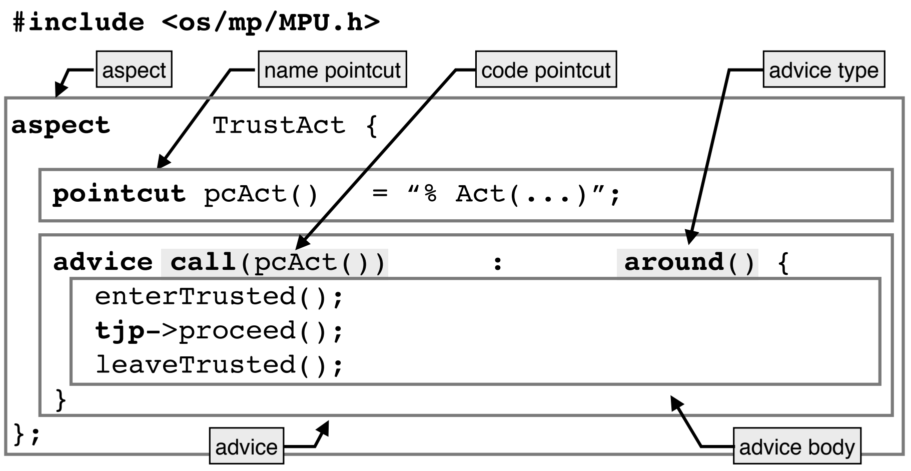
```
call(pcFoo()) && !within("void bar()")
```

4.1.2.2 Advice Code

A piece of advice defines a code transformation that is applied to a set of code join points, described by a pointcut expression. An example of a possible code transformation is the execution of the advice code in addition to (or instead of) the code join point (for example, a call to a method or the body of a method). An aspect is a program artifact similar to a C++ class, which encapsulates pointcut expressions and advice.

4.1.2.3 AspectC++ Syntax Example

To illustrate the syntax, a simple AspectC++ example of a complete aspect is shown in Figure 4.1. The aspect encloses all calls to functions named `Act()` between a pair of `enterTrusted()` and `leaveTrusted()` operations.



```
#include <os/mp/MPU.h>
aspect      TrustAct {
  pointcut pcAct()    = "% Act(...)";
  advice call(pcAct())      :      around() {
    enterTrusted();
    tjp->proceed();
    leaveTrusted();
  }
};
```

Figure 4.1: AspectC++ Syntax Example

The example shows an aspect `TrustAct` with a single piece of advice. It declares a name pointcut `pcAct`, whose match expression matches all `Act()` functions in the code base, regardless of the parameter list and return types (using the `%` and `...` wildcard operators). The advice code is applied to all calls to an `Act()` function, described by using the `call()` pointcut function on the previously declared `pcAct` name pointcut. The shown advice code is of the `around` advice type, which replaces the original code join point with the advice body. AspectC++ does, however, expose the context of the affected join point. By calling the method `proceed()` on the variable `tjp` ("this join point"), which represents the affected join point, the original code of the join point can be invoked from anywhere within the advice body. In addition to the `around` advice type, there are the `before` and `after` advice types, which execute the advice code preceding or following the code of the affected code join point.

4.1.2.4 AspectC++-Prepared Codebase of CiAO

The target code base for the AspectC++ weaver does normally not need to be specially prepared, because the join points are specified declaratively. CiAO, however, has been designed to be easily adaptable by applying aspects, and therefore provides a code base with a particularly rich set of join points to aid the application of aspects. One of CiAO's concepts is *explicit join points*, empty methods that are called at certain interesting stages of the kernel. They serve the sole purpose of providing a join point for advice code. An example of such an explicit join point is `after_CPUReceive()`, which is called whenever a new task has been dispatched and allows aspects to extend the context switch with additional functionality [71].

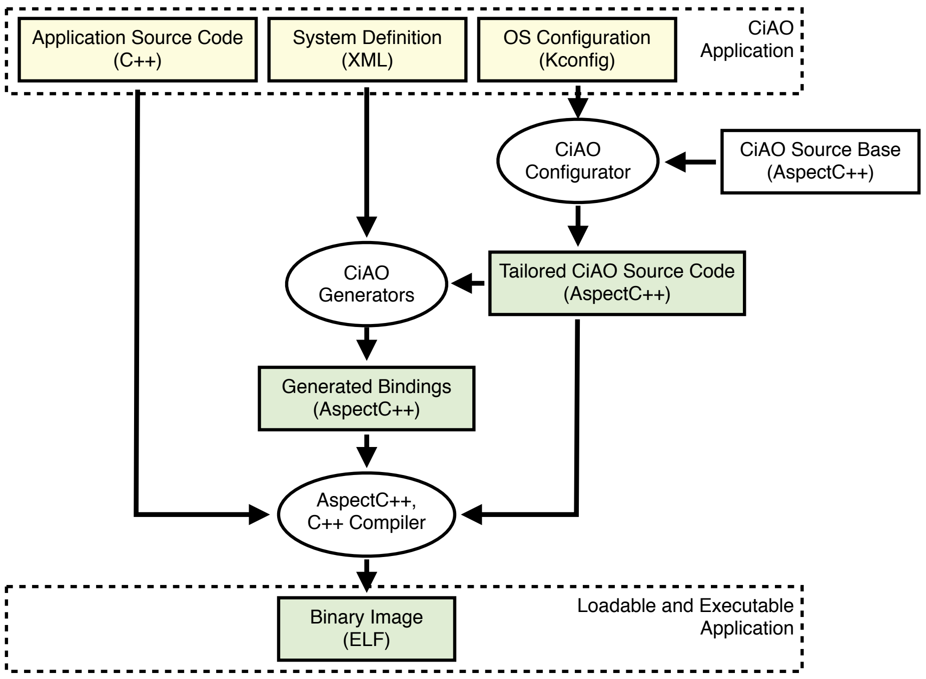


Figure 4.2: CiAO Build Process

4.1.3 CiAO Build Process

A CiAO application consists of the application's C++ source code, an AUTOSAR OS system definition in XML format, and an operating system configuration created with the Linux kernel's configuration tool Kconfig. The system definition defines the instances of AUTOSAR OS system objects (OS-Applications, tasks, etc.), their properties and relations between each other (for example, which task belongs to which OS-Application). The OS configuration defines the feature spectrum supported by the CiAO operating system.

Figure 4.2 shows a slightly simplified version of CiAO's build process. A custom Kconfig backend reads the OS configuration and creates a CiAO variant that consists of the base system and a selection of aspects that implement the additionally needed functionality selected in the OS configuration. A set of code generators, which are part of the generated OS variant, generate application-specific bindings and OS data structures tailored to the application. The bindings consist of generated pointcut expressions that are used by the aspects of the CiAO variant. As an example, the generated bindings comprise pointcut expressions that match interrupt service routines provided by the application, used to bind these handlers to the interrupt vector entries, or pointcuts that define which artifacts of the application code belong to which OS-Application, or which code artifacts are trusted code and which are not. In Section 4.4, I explain in more detail how configurable MPU-based memory

```
#include "app/componentheader.h"

// definition of component "SerialCom"
CIAO_COMPONENT (SerialCom)
public:
    // task entry function for task "SerialComTask"
    static void functionTaskSerialComTask();
    // ISR entry function for ISR "SPIRXISR"
    static void functionISRSPIRXISR();
    // can be called as non-trusted functions
    void service();
    // this (non-trusted) function does only read state
    void ro_service() const;
};
```

Listing 4.1: CiAO Component Definition

protection is realized in CiAO.

Finally, the static aspect weaver AspectC++ applies the aspects to the source code of the operating system and the application. The woven source code is then compiled by a standard C++ compiler and linked to an image that is ready to be loaded to the target and executed.

4.1.4 CiAO Components: Application Interface

For CiAO applications, the definition of the OS-Applications is a decision that is taken by the system integrator. The memory protection realms do not manifest in the source code. At the system integration stage, OS-Applications are composed of CiAO components, technically realized as C++ classes, which are assigned to exactly one OS-Application each. Tasks and ISRs are implemented by components. Listing 4.1 shows an example of a component definition, which contains the most important elements of CiAO's component interface. The macro `CIAO_COMPONENT` opens a class header and defines a basic set of functionality offered by all CiAO components. Components are statically instantiated by CiAO's build system. The defined SerialCom component provides the implementation for a task SerialComTask and an ISR SPIRXISR. The entry functions of tasks and ISRs need to follow a predefined naming scheme. Every task entry function is composed of the prefix `functionTask` followed by the name of the task. ISR entry functions use the prefix `functionISR`. Trusted and non-trusted functions are exported implicitly. All public methods of the component can be invoked from other OS-Applications as a trusted or non-trusted function. The call is treated as a (non-)trusted function call if the caller and callee components do not belong to the same OS-Application. An exported function can be declared `const` to denote that it does not change the state of the component. CiAO's memory protection subsystem can leverage this information to avoid a memory protection context switch if only write protection is selected.

4.2 Selection of a Multi-JVM

Just as with the AUTOSAR OS implementation, I build on an existing Java Virtual Machine (JVM) implementation to avoid unneeded work. The JVM implementation needs to fulfill similar criteria as the AUTOSAR OS implementation: The source code must be available, it should support spatial isolation, and it should be able to adapt the functionality provided by the JVM to scale to the requirements of a given application. Particularly, it should be possible to disable expensive mechanisms such as garbage collection if not needed. The JVM should target embedded systems: It should have been designed to economically use constrained resources such as CPU time, RAM and ROM. Following my discussion of Java compilation and execution strategies from Section 3.4.1, it should support ahead-of-time compilation, as just-in-time compilers require complex infrastructure, which exceeds most typical applications for the target domain in resource requirements, and are hardly predictable. Interpreters on the other hand impose high execution time overhead and still require a more complex runtime environment than ahead-of-time compiled code. To support low-level code, the JVM should implement the raw memory API defined in the RTSJ, or provide a similar interface that provides such functionality. Finally, the JVM would ideally allow using an external operating system for the scheduling of its threads. This simplifies the integration with an AUTOSAR OS and native applications running on top of it.

The available JVM implementations for embedded (in the sense of special purpose) systems and real-time systems are manifold, but many of these implementations have not been designed for resource-constrained targets. I restrict my following discussion to implementations that have been designed for resource-constrained systems.

4.2.1 J2ME Implementations

The most widely spread Java platform for embedded systems is the Java 2 Micro Edition (J2ME) [67]. J2ME defines two scalability concepts, configurations and profiles. Configurations define the feature set of the JVM targeting a specific class of devices, for example, a configuration typically defines a minimum amount of memory that needs to be available. There are currently two standard configurations, the Connected Limited Device Configuration (CLDC) [57], and the Connected Device Configuration (CDC) [58]. Orthogonally to configurations, J2ME defines different profiles targeting a particular application domain and mostly define the available APIs suited for the respective target domain. An example of a widespread profile is the Mobile Information Device Profile (MIDP) [59], designed for mobile devices such as cellphones or PDAs. Of the two configurations, CLDC targets more resource-constrained devices (min. 160 KiB ROM, 32 KiB RAM).

The main issue with J2ME is that it has been designed for a purpose that is different to that of my framework. It provides a portable and safe execution platform for potentially untrusted code. As a consequence of the portability, the code is distributed in the form of bytecode and interpreted or compiled just in time, and the feature set provided by the JVM is fixed as defined by the configuration. Since the code is potentially untrusted, verification of the bytecode on the target is additionally required, and low-level programming is not possible for security reasons. In contrast, I am looking for a JVM that can be tailored to a set of known, trusted (but potentially buggy) applications.

4.2.2 Commercial JVMs for Embedded and Real-Time Systems

Aicas' JamaicaVM [101], Aonix' PERC Pico [98] and Fiji Systems Inc.'s FijiVM [89] are commercial JVM implementations for resource-constrained real-time systems. All of them support ahead-of-time compilation, real-time garbage collection, and either provide support for the RTSJ or implement proprietary APIs with similar functionality. From the available information material, none of these JVMs supports spatial isolation, and the source code is not available. It is also unclear to what extent these JVMs can be tailored for a specific application and what the minimum resource requirements for the simplest configuration are.

4.2.3 Sun's SquawkVM for Sensor Nodes

SquawkVM [103] is a research project originally by Sun Microsystems of a JVM for embedded systems, primarily Sun Spot wireless sensor nodes (180 MHz ARM920T, 4 MiB ROM, 512 KiB RAM). SquawkVM is available open source, supports ahead-of-time compilation, and spatial isolation through the Java Isolate API [56]. SquawkVM can either be used as a JVM on a desktop operating system (Windows, Mac OS X or Linux), or run on bare hardware without an underlying operating system (currently on ARM-based microcontrollers). SquawkVM therefore poses a promising candidate. The downsides are that it does not run on top of AUTOSAR OS, only supports ARM microcontroller platforms, and that it can only be coarsely tailored to an application.

4.2.4 KESO

KESO [123, 113] is an academic multi-JVM implementation for OSEK/VDX platforms. In addition to spatial isolation, KESO supports ahead-of-time compilation of Java bytecode to C code, optional low-latency garbage collection, a raw memory API similar to the one in the RTSJ and an own additional mechanism of memory-mapped objects for low-level programming, and maps Java threads to tasks of the underlying OSEK/VDX operating system. KESO has been designed for static embedded systems and consequently does not provide support for dynamic class loading. What is special about KESO is that a tailored JVM is created when compiling the applications. KESO's compiler *jino* runs whole-program analyses to gather the set of features required by the application from the program code, and automatically tailors the JVM at a fine-grained level to the requirements of the application. This is in stark contrast to the execution model of J2ME: KESO does not provide a portable platform for the execution of unknown applications, but a tailored platform for the execution of known applications. This tailoring enables KESO to provide a broad spectrum of

JVM features while at the same time being able to provide a very lightweight runtime environment for applications that require few of these features[2].

Since KESO fulfills most of the requirements, it is the most suited candidate for my framework. It integrates well with OSEK/VDX operating systems, to which AUTOSAR OS is largely backwards compatible. KESO generates C code at its backend, which allows it to be easily extended to support new target platforms, as KESO itself is mostly platform independent. Most importantly, however, is that KESO adapts itself very closely to the requirements of the used application. This will allow me to evaluate the cost of software-based memory protection with little distortion by unneeded JVM features, particularly when comparing to the original C version of a program ported to Java. In the following, I will briefly introduce KESO's application model and build process, before I discuss the integration with CiAO in Section 4.3.

4.2.4.1 KESO Application Model

The architecture of a KESO system is shown in Figure 4.3. The system is structured in multiple domains, which pose the realms of isolation. Since KESO is based on OSEK/VDX, whose threading and synchronization facilities it utilizes, it uses the OSEK/VDX terms of tasks, resources and events instead of the corresponding Java terms threads and monitors[3]. To the application, each domain appears to be a JVM of its own, with a separate instance of the static class fields and a separate object heap, hence the term multi-JVM architecture. Domains provide spatial isolation, established as discussed earlier by retaining disjoint object graphs and static fields. Each OSEK/VDX system object is assigned to a domain. Each control flow executes in the context of a domain, in which it accesses that domain's instance of static fields and heap. KESO not only logically but also physically separates the heaps of the different domains. This allows statically partitioning the memory so that each domain can access a guaranteed amount of memory regardless of the amount of allocations within other domains. The memory management strategy can be chosen on a per-domain basis. KESO currently provides three memory management schemes: a simple bump-pointer constant-time allocation scheme without deallocation, a throughput optimized stop-the-domain garbage collector and an incremental low-latency garbage collector[4]. Choosing the allocation strategy per domain allows restricting garbage collection to those domains that continuously allocate memory during their execution.

Inter-Domain Communication For communication among domains, KESO provides a synchronous remote procedure call mechanism that is similar to Java remote method

[2]For a portable execution platform, a feature set that is appropriate for all targeted applications has to be predetermined and supported by the runtime environment, even if it executes an application that does not use most of these features.

[3]KESO does not currently support Java monitors as defined in the JVM specification [69], but instead provides an API to the native synchronization and notification facilities of OSEK/VDX.

[4]A survey over automatic memory management schemes is available in a publication [125] by Paul Wilson.

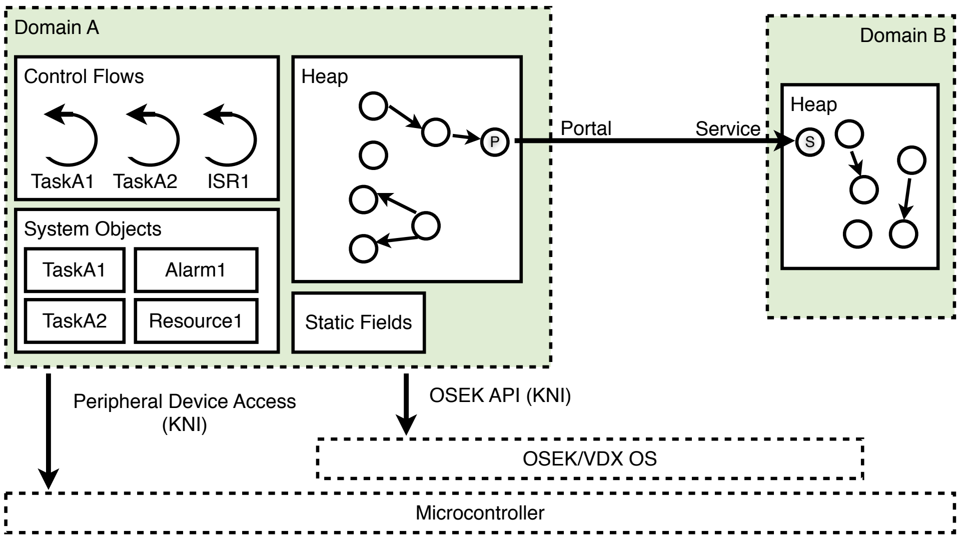


Figure 4.3: KESO: Architecture, Domains and Inter-Domain Communication

invocation [95], so-called *portals*[5]. The portal mechanism enables a task in one domain to invoke a method on a service object within a service domain by invoking a method on a proxy object (the portal) in its own domain. Data can be passed between the caller domain and the service domain as parameters to the portal call, and via the call's return value. To retain disjoint object graphs, object references must not cross domain boundaries through this mechanism. Instead, KESO creates a deep copy of the referenced part of the object graph on the service domain's heap, and replaces the reference with a reference to the clone. The same happens for returned values. KESO provides mechanisms to restrict what is deep-copied during a portal call. Deep copying is also used in the message channels defined by the Java Isolate API [56] and is a consequence of the logical heap separation. It is somewhat problematic, as it implies the need for garbage collection, because allocation happens during each call. For portal calls that take and return only primitive values, this problem does not occur. For the duration of the portal call, the calling control flow is temporarily migrated to the service domain to execute the service. This ensures that the portal mechanism does not interfere with the OSEK/VDX scheduling.

Low-Level Interface To program low-level code such as device drivers, KESO provides a raw memory API similar to the one specified in the RTSJ. This API can be used to enable Java code to access device-registers mapped to a special address range in the address space. Accessing a raw memory area is similar to accessing an array of primitive data, including bound checks for accesses to the area. For a

[5]The term *portal* was adopted from JX [41], on which KESO is conceptually based, and originates from OPAL [23].

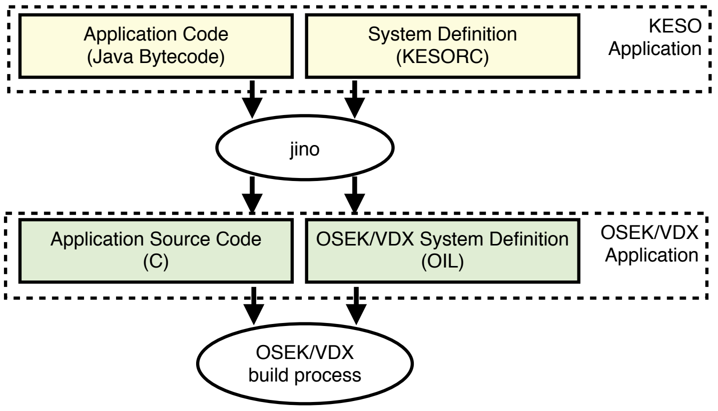


Figure 4.4: KESO Build Process

higher-level API, KESO supports an own concept that allows mapping Java objects to a raw memory area and to access the elements of the raw memory area by named fields. The concept is similar to hardware objects [97]. Using memory-mapped objects, the programmer of low-level code gets a similar experience to using C structures for accessing memory areas.

For low-level code that needs functionality beyond accessing raw memory regions, KESO provides a lightweight native interface (KNI). KNI pursues an aspect-oriented concept. KNI provides an API that allows a KNI plugin to affect the code generation of the compiler at certain points in the call graph or to extend classes with additional fields. A KNI plugin is exposed to the full internal API of jino, which renders the mechanism very powerful. KESO's raw memory API, memory-mapped objects and OSEK/VDX interface are all implemented as KNI plugins. Code generated using the KNI is potentially unsafe and the interface should therefore be used with care. In practice, I have not encountered an application that needed a custom KNI plugin beyond what is provided with KESO.

4.2.4.2 KESO Build Process

KESO's compiler jino creates an OSEK/VDX application, which can be used like a native application in the build process of the respective OSEK/VDX implementation. The build process is shown in Figure 4.4. The KESO application is provided in the form of Java bytecode and a system definition text file. The Java bytecode can be generated from Java source code by using a regular Java compiler. The system definition file contains the definitions of the used OSEK/VDX system objects (tasks, etc.) along with their properties (for tasks, for example the priority and entry functions), the definitions of the KESO entities (domains, portals and services), and the mapping of OSEK/VDX system objects to domains. The KESO application is then compiled by jino to an OSEK/VDX application, consisting of a system definition file in OIL format and the application C source code. The generated C source code not only contains the translated source code of the application, but also a Java runtime environment that is tailored towards the given application. The generated OSEK/VDX application can be used as input to the OSEK/VDX-implementation-specific build process for further processing to an executable image.

4.3 Integration of KESO with CiAO

KESO contains backends for different OSEK/VDX implementations. Common to all backends is that the produced output is the application translated to C source code and a system definition in OIL format. Since CiAO is an AUTOSAR OS-like operating system and not an OSEK/VDX implementation, integrating KESO with CiAO requires some changes to KESO beyond the addition of a new backend. The main steps towards an integrated toolchain as shown in Figure 4.5 are:

- Generate a system definition in AUTOSAR XML format
- Integrate the application model of AUTOSAR OS into KESO
- Output C++ code
- Adapt the runtime environment to support MPU-based memory protection

4.3.1 CiAO Backend

The new backend for CiAO emits an XML system definition in the format required by CiAO. To support a soft migration and a side-by-side operation of Java and native C++ applications, the system definition of KESO comprises both the Java and native C++ parts of the system. This allows KESO to fully generate the XML system definition at its backend. More importantly, it makes KESO's compiler jino aware of all operating system objects (OS-Applications, tasks, resources, etc.), which enables jino to support basic interaction between Java applications and native C++ applications, for example activating native C++ tasks or setting events for such tasks from Java application code.

To support CiAO, KESO needs to be extended to support AUTOSAR OS as the underlying operating system. AUTOSAR OS is a superset of OSEK/VDX considering the functionality that affects KESO. The main change required to enable KESO to run on top of AUTOSAR OS is the extension of its application model by the OS-Applications of AUTOSAR OS. In Section 3.3, I opted for an N:1 mapping of KESO domains to OS-Applications. I extended KESO's application model by the OS-Application container, to one of which each domain must be assigned. The system definition contains the OS-Application definitions and mappings for native application parts. This allows jino to fully generate the system definition for CiAO. In addition, jino's awareness of the OS-Application allows it to extend its language-based

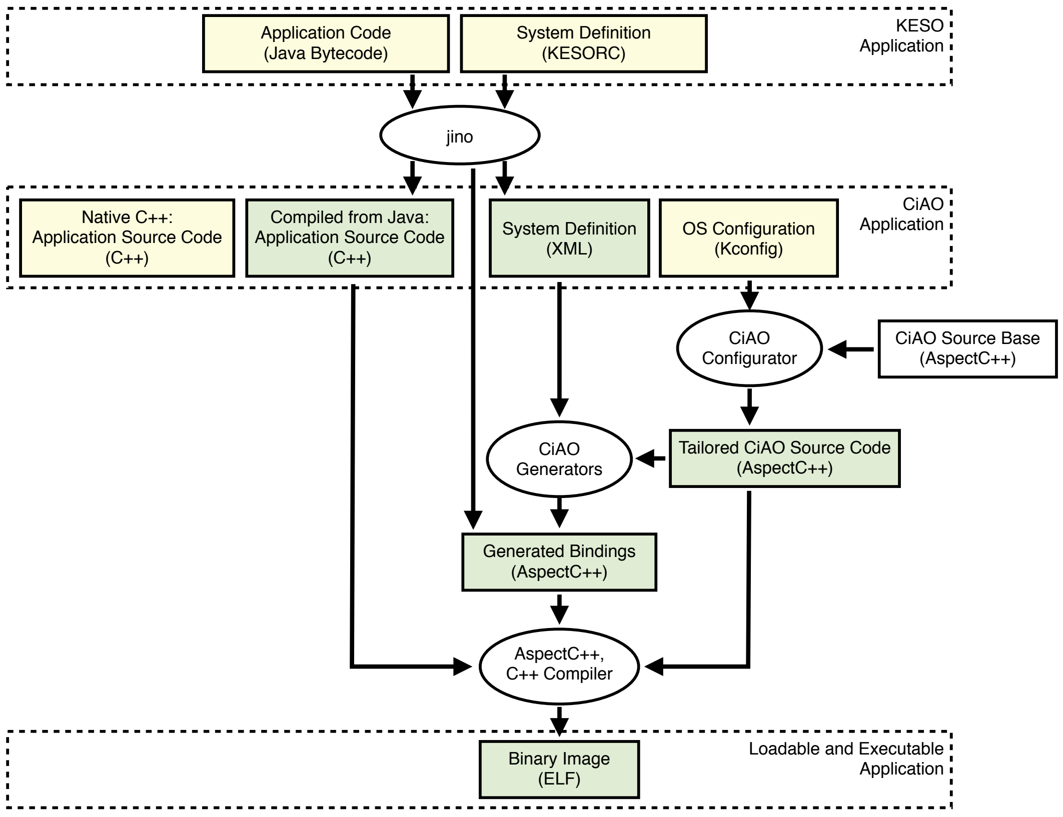

Application Code (Java Bytecode)
System Definition (KESORC)
KESO Application
jino
Native C++: Application Source Code (C++)
Compiled from Java: Application Source Code (C++)
System Definition (XML)
OS Configuration (Kconfig)
CiAO Application
CiAO Configurator
CiAO Source Base (AspectC++)
CiAO Generators
Tailored CiAO Source Code (AspectC++)
Generated Bindings (AspectC++)
AspectC++, C++ Compiler
Binary Image (ELF)
Loadable and Executable Application

Figure 4.5: KESO and CiAO Combined: KESO generates CiAO applications that are no different to CiAO than native CiAO applications.

service protection to handle operating system objects that are contained in native OS-Applications. Finally, jino needs the knowledge on the OS-Applications and the mapping of domains to OS-Applications to assist CiAO in employing MPU-based memory protection for Java applications. As shown in Figure 4.5, jino generates parts of the bindings for AspectC++. I discuss this in more detail in Section 4.6.

4.3.2 C++-Compatible Output

KESO generates C code at its backend, whereas the CiAO toolchain requires the applications to be written in C++. While C++ is for the most part a superset of C, there are some important incompatibilities. As an example, KESO widely used designated initializers for the static initialization of data structures of the runtime environment and statically allocated objects. The C code generated by KESO therefore cannot be compiled with the C++ compilers used in CiAO's toolchain. While it would be possible to compile the application parts generated by KESO with a C compiler, and combine them with the native C++ parts at the link stage, this approach would eliminate the possibility to transform the KESO-generated parts using AspectC++, and thereby prohibit the use of aspects that affect parts of the application code. A true C++ backend on the other hand would mean a significant implementation and future maintenance effort, without providing a benefit that would compensate the effort. I therefore opted to adapt KESO's C backend so that the emitted code complies with the common subset of ISO C90 and C++. The minimum C++-specific requirements that CiAO puts on its applications (Section 4.1.4) are handled in the CiAO backend (for example, all task and ISR entry functions need to be member of a C++ class).

4.3.3 MPU-based-Protection-Friendly Java Runtime Environment

KESO has originally been developed targeting OSEK/VDX systems. MPU-based memory protection was not considered, and consequently KESO's runtime environment is not prepared to support MPU-based protection. To support MPU-based protection, the Java runtime environment needs to arrange its runtime state in a manner that allows reducing the number of MPU ranges needed to grant access to the needed regions to a minimum. KESO used to organize data by kind, and used arrays to create multiple instances of domain-specific state such as the static fields. I adapted KESO to instead organize its data by the following groups:

Read-only data structures shared by multiple domains comprise for example the runtime type information on all classes and the dispatch table.

Domain-specific data structures include the heaps and static fields for each domain and related management data. For each domain, an own group for the domain-specific data is created.

Shared data writable from multiple domains is currently limited to the identifiers of the currently running task and the currently active domain. With hardware-based protection, such state must not be modifiable from non-trusted application code.

The actual placement of data in memory is managed by CiAO, however, jino assists CiAO by avoiding data belonging to different of the above groups to reside within a single object-file-level data item. In addition, jino marks the data items with their group by assigning them symbol names that follow a naming scheme mandated by CiAO, encoding the destined OS-Application or that the item belongs to the kernel realm. CiAO can use the information to group the data with data from an identical category in CiAO, so that no additional MPU regions are needed. Read-only shared data is placed in a memory region that is not writable to any OS-Application. If supported by the target platform, such data can also be placed in flash memory, which is normally a less limited resource than RAM. Domain-specific data structures are mapped to the private data segment of the containing OS-Application. Mutable state of the Java runtime environment that is accessible by multiple domains is limited to two identifiers that are only modified when a task switch occurs. KESO uses the privileged hook routines provided by AUTOSAR OS to perform this state change. CiAO can therefore group this state with similar state of its kernel.

4.4 Configurable MPU-Based Protection

In this section, I present how configurable MPU-based memory protection is achieved in CiAO. CiAO already offers configurable MPU-based memory protection [72]. For my framework, I modified and extended the existing work to suit my requirements.

To implement MPU-based memory protection, three problems need to be solved. Firstly, the memory regions that each application needs to access at runtime need to be determined. Secondly, the region for each data item in the code base must be identified and assigned to the respective region. Thirdly, the infrastructure code for maintaining and switching memory protection contexts according to the configured protection coverage needs to be added to the code base. Each of these steps varies depending on the configured type of memory protection.

4.4.1 Region Management

As discussed in Chapter 2, I aim for a static management of the MPU regions to achieve a predictable and low-overhead behavior of the memory protection subsystem. Trusted OS-Applications and the kernel are not subject to memory protection and therefore need not be considered for the region management. For non-trusted applications, the accessible data should be grouped in memory according to the access rights mandated by the configured protection model to minimize the number of needed MPU regions, as the number of regions supported by the MPU is limited.

The basic set of regions needed to implement the protection model of AUTOSAR OS is:

- CCode: Shared code that is directly executable by all OS-Applications. An example of such shared code is functions of the standard C library. If execution protection is not used, this region spans the entire code space.
- AppCode: Code that is private to the active non-trusted OS-Application. Only needed if execution protection and application isolation are enabled.
- CRData: Shared data that is read-only accessible by all OS-Applications. Examples are string constants, or, for KESO's runtime environment, the dispatch table and the runtime type information table. If read protection is not enabled, this region spans the entire data space.
- AppData: Private data segment of the active non-trusted OS-Application, readable and writable. Only needed if application isolation is enabled.

- TStack: Stack of the active control flow, readable and writable. Needed only if control-flow isolation is enabled, or if non-trusted functions or non-trusted ISRs are used.

These regions are the minimum needed set to fully support the protection model, however, an actual application may need additional regions for shared data areas, temporarily accessible memory areas or to access memory-mapped device registers. In addition, it may be sensible to split some of the above basic regions into multiple subregions, for example to place part of the code or data in special memory types such as fast on-chip memories.

On the heterogeneous MPU of the Tricore TC1796, the above base regions occupy both code regions and three of the four data regions. On the homogeneous MPU of the ARM Cortex-M3, five of eight regions are needed in the common case. In some special situations, code and data regions can be merged on a homogeneous MPU. Firstly, if only write protection is required, CCode and CRData can be merged to a single region that provides read and execute permissions on the entire address space. Secondly, if the code resides in the same memory as the data, the CCode and the CRData can be merged on a homogeneous MPU, as well as AppCode and AppData. It is more common, however, that the code resides in Flash ROM and not in the data RAM.

Stateful Library Functions

A problem arises with stateful functions of the C library such as `strtok()`. For the correct isolation, the different OS-Applications would need to have an own instance of the state each. In a system with a single shared address space, separate instances of the library state require a C library prepared for this situation. At the least, the library must not use absolute addressing to access the mutable state. Some C libraries provide solutions to this problem, for example, the Redhat newlib library groups all library state in a compound object and provides a global pointer variable that points to the currently active instance of the library state. The operating system can allocate an instance of the library state per OS-Application (or per task/ISR, to additionally provide reentrancy), and set the global pointer variable to point to the correct instance as part of the OS-Application or task switch. A problem with this solution is that the full mutable state is grouped in the compound structure, and the structure is fully allocated even if only a single element of it is used. The size of the structure is 892 bytes in newlib 1.12.0 for the Tricore architecture, which ships with the Hightec Tricore Toolchain 3.4.6. With separate instances for each OS-Application or control flow, the sizes of these instances accumulate to a notable amount considering the available memory in embedded systems. To avoid this often unneeded memory consumption, CiAO does not support the use of stateful library functions. Instead, the applications must use the reentrant versions of the library functions (for example, `strtok_r()`), for which the caller provides an instance of the needed state.

4.4.2 Region Identification and Data Mapping

With the needed regions determined, the data must be arranged in memory to form these contiguous regions. The placement of data in memory is commonly performed by a linker. For a static embedded system, a static link-and-load approach is reasonable, where all of the linking steps are performed ahead of time.

4.4.2.1 Linking Process Overview

The linker combines a group of object files produced by the compiler to an output file, in this case a linked object file that contains a memory image that can be loaded into the memory of the target. Each object file consists of different sections that contain data items categorized by purpose. For example, code (functions) is commonly contained in the `.text` section, initialized static data in the `.data` and uninitialized (or 0-initialized) data in the `.bss` section. In addition, there may be special-purpose sections such as `.rodata` containing read-only data items that enable the linker to place the contained items in special memories if appropriate. Each function or static data item is identified by a symbol (symbolic name), which is usually derived from the object's identifier at the programming language level. Each object file additionally contains a symbol table that lists the symbols defined by the object file and unresolved references to symbols used within the object file.

During the linking process, symbol references among the different input object files are resolved so that no more unresolved references remain. The code and data items from the input object files are mapped to sections in the output object file. In the simple case, the output sections are produced by merging the corresponding sections of the input files, but many linkers can be provided with a linker script to customize this mapping process. The linker script defines at the granularity of the different code and data items to which section in the output object file each will be mapped. In the binding phase, the runtime memory address of each output section is determined, and therefore an address value is bound to each symbol. Finally, the linker updates symbol references (for example, load, store or branch instructions or data initializations depending on a symbol value) to contain the value bound to the symbol (relocation).

4.4.2.2 Data-to-Application Mapping

To instruct the linker to group data items belonging to the same MPU region in memory, the issue of identifying the destination region for each data item needs to be solved. There are different approaches to achieve this, each of which places certain structural requirements on the application.

Delegate to the developer One approach is to delegate the task of grouping the data and defining the regions to the developer. In practice, the developer could place all the data belonging to the private data region of an OS-Application in an aggregate data structure at the programming language level. The corresponding region would then just contain a single data item at the linker level. Hightec's PXROS-HR [94] follows this approach. Whenever instantiating a task, the programmer explicitly needs to provide the memory ranges for all regions accessible to that task. In the I4Copter, a single aggregate data structure is used for each task, and a region is defined that spans the address range occupied by this aggregate data structure.

Mapping based on object-file sections or symbol names An alternative approach is to predefine an application structure that reflects the different protection realms in the application structure and enables a differentiation at the linking stage. This is either possible by completely separating the compilation units, so that each compilation unit (object file) of the application contains code or data items that belong to exactly one application and can be mapped section-wise to regions. Or, at a finer-grained level, a naming convention for language-level identifiers that reflects in the symbol names can be utilized, for example based on C++ namespaces. The mapping is then based on the symbolic names of the data items rather than the compilation unit, therefore data items belonging to different OS-Applications can be contained in the same object file.

The first approach leaves more freedom to the application developer, but also all the work including handling the physical arrangement of the data in memory. CiAO follows the second approach, based on its system-integration-level mapping of CiAO components (Section 4.1.4) to OS-Applications. The mapping works based on the symbol names and is independent of the compilation units.

4.4.2.3 Configuration-Independent Memory Layout

CiAO provides a generator that produces a linker script file from the system definition. The linker script contains detailed instructions for the linker on how to perform the mapping process of input data items to the output sections of the produced image. The memory layout follows the different granularity levels that the protection is available for. This memory layout is suitable for any of the configuration variants.

Figure 4.6 shows the layout of the data memory. The code memory is correspondingly organized, or merged with the data if code and data reside in the same memory. Figure 4.6 also shows the additional symbols that are defined by the linker script at the region boundaries, and are used by the operating system code to determine the region boundary addresses that match the chosen configuration. At the first level, the trusted items, non-trusted items and shared read-only items are separated from each other. The kernel code uses known C++ namespaces, the code and data for each application can be identified based on the symbol name as described above. Any items that cannot be assigned to either an OS-Application or the kernel are considered shared library items that are readable or executable by any OS-Application, but not writable. This coarse grouping is sufficient to realize kernel protection. For the isolation of OS-Applications, the non-trusted portion is further structured to contain groups per OS-Application. The portion for each OS-Application contains

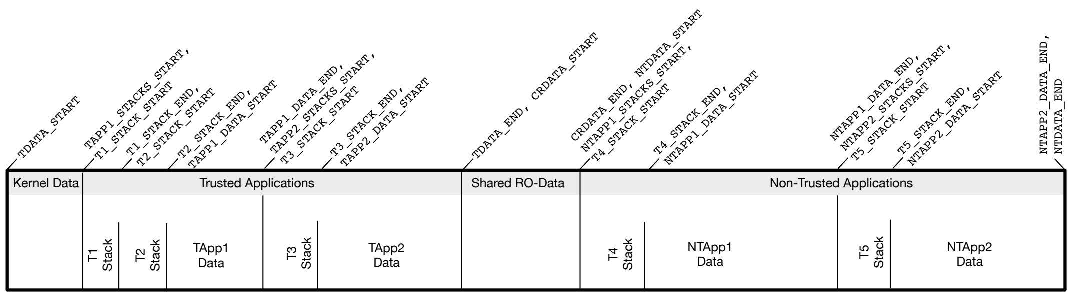

TDATA_START
TAPP1_STACKS_START, T1_STACK_START
T1_STACK_END, T2_STACK_START
T2_STACK_END, TAPP1_DATA_START
TAPP1_DATA_END, TAPP2_STACKS_START, T3_STACK_START
T3_STACK_END, TAPP2_DATA_START
TDATA_END, CRDATA_START
CRDATA_END, NTDATA_START, NTAPP1_STACKS_START, T4_STACK_START
T4_STACK_END, NTAPP1_DATA_START
NTAPP1_DATA_END, NTAPP2_STACKS_START, T5_STACK_START
T5_STACK_END, NTAPP2_DATA_START
NTAPP2_DATA_END, NTDATA_END
Kernel Data
Trusted Applications
Shared RO-Data
Non-Trusted Applications
T1 Stack
T2 Stack
TApp1 Data
T3 Stack
TApp2 Data
T4 Stack
NTApp1 Data
T5 Stack
NTApp2 Data

Figure 4.6: Configuration-Independent Memory Layout (Data Only)

Kernel Protection

CRData	CRDATA_START	CRDATA_END
AppData	NTDATA_START	NTDATA_END

Application Isolation

CRData	CRDATA_START	CRDATA_END
AppData	NTAPPx_STACKS_START	NTAPPx_DATA_END

Control-Flow Isolation

CRData	CRDATA_START	CRDATA_END
AppData	NTAPPx_DATA_START	NTAPPx_DATA_END
TStack	Ty_STACK_START	Ty_STACK_END

Table 4.1: Data Region Definitions for Read-Write Protection

the private code and data segments, and the stacks of all tasks that belong to the respective OS-Application. If only application isolation is used, a single region can span the private data segment and the task stacks of the respective OS-Application. If control-flow isolation is additionally used, one region is needed for the private data segment of the OS-Application and an additional region for the stack of the task. Table 4.1 shows the used data region boundaries for a configuration with both read and write protection for the three levels of isolation.

4.4.3 Configurable MPU-Context Switching Code

Having identified the regions required for the configured MPU protection variant and arranged and grouped all data items to reside in the proper region, the prerequisites with respect to the data representation for applying MPU-based memory protection are fulfilled. To make the protection functional, the base operating-system code has to be extended with the functionality to enable and switch the memory protection mode that corresponds to the active protection realm, which is either the kernel or

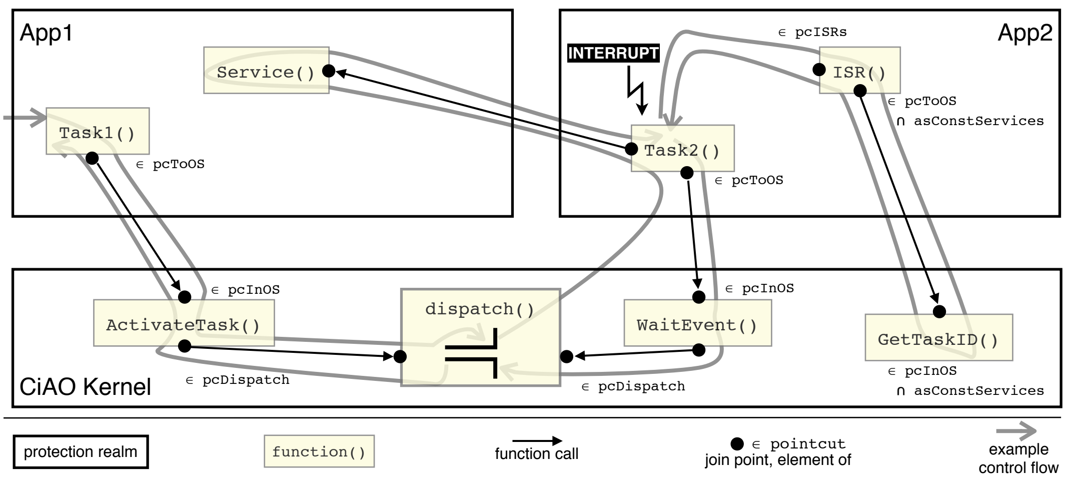


Figure 4.7: Protection Context Switching (compare [72])

an OS-Application.

Figure 4.7 shows an example control flow that passes through different protection realms. The example contains three protection realms, the two non-trusted OS-Applications App1 and App2, and the CiAO kernel. App1 contains one task Task1 and exports a non-trusted function `Service()`. App2 contains one task Task2 (of higher priority than Task1) and an interrupt service routine `ISR()`.

Initially, Task1 executes in the context of OS-Application App1. Task1 invokes the system service `ActivateTask()` to activate the higher-priority Task2. For the execution of the system service `ActivateTask()`, the protection context changes to the privileged kernel. Since the activated Task2 is of higher priority, control is transferred to it in the protection context of OS-Application App2. Task2 then invokes the non-trusted function `Service()` exported by OS-Application App1. The non-trusted function is executed in the protection context of the exporting OS-Application, wherefore Task2 is temporarily migrated to the protection context of App1. Upon return of the non-trusted function, Task2 executes back in the context of App2, and is then preempted by an interrupt service routine that belongs to the same OS-Application. The ISR queries the identifier of the currently running task. After completion of the ISR, Task2 resumes execution and blocks by invoking the `WaitEvent()` system service. The control returns to Task1.

In the basic CiAO configuration without memory protection, calls to system services of the kernel or to services of other OS-Applications are no different from regular function calls. The compiler may even inline a system call into the application code if appropriate. With memory protection enabled, switches of the protection context (that is a change of the processor privilege mode or the MPU configuration) become necessary as the control flow proceeds across the boundary of the active protection domain.

CiAO achieves configurability by means of aspect-oriented programming (AOP), where the core functionality is extended with optional features by applying aspects. In the following, I discuss how the basic CiAO code base providing no memory protection is successively extended to provide the different levels of MPU-based memory protection. The first step is to define pointcut expressions that match the join points where a change of the memory protection context occurs. The second step is to define the appropriate advice code to transform the join points to include the memory protection context switch.

4.4.3.1 Pointcuts and Generated Bindings

The pointcuts defined for the memory protection subsystem are shown in Listing 4.2. The example in Figure 4.7 shows, in which of these pointcuts each of the protection domain crossing code join points is contained. The shown pointcuts are for a configuration with write protection only, wherefore read-only system services and (non-)trusted functions declared `const` are excluded from the pointcuts that match the protection domain crossings. For a configuration that additionally uses read or execution protection, this optimization is not applied.

For the operating system, all system services of the public API are known and a pointcut expression explicitly matching each service can be created, as shown in the (incomplete) definition of the `asServices` pointcut. The entry functions of tasks and ISRs need to follow a defined naming scheme (Section 4.1.4) and can therefore also be matched by the static match expressions `pcStartFuncs` and `pcISRs`.

Pointcut expressions to match code join points for transitions between different OS-Applications (for trusted and non-trusted functions) depend on the CiAO components defined by the application and their assignment to OS-Applications. As this information varies, the pointcut expressions cannot be defined statically but are instead generated from the system definition file. Pointcut expressions are generated for each individual OS-Application, matching the methods of all components that are assigned to the respective OS-Application (`pcApplicationApp1` and `pcApplicationApp2`). In addition, pointcuts that aggregate all trusted and all non-trusted OS-Applications are generated (`pcTrustedApps` and `pcNonTrusted`).

Based on these name pointcuts, pointcut expressions that match code join points at that the different protection domain transitions are initiated can be created. For system services, the pointcut `pcInOS` comprises the implementations of all public system services themselves, whereas the pointcut `pcToOS` comprises the calls to system services from non-trusted OS-Applications. A discussion of these two alternatives follows in Section 4.4.3.2.

4.4.3.2 Kernel Protection and Trusted Functions

The most basic memory protection level provided by CiAO is kernel protection, where only the trusted protection realm containing the kernel and trusted OS-Applications and the non-trusted protection realm with the non-trusted OS-Applications exist.

```
//////////   OS INTERFACE POINTCUTS
// contains all system services
pointcut asServices() = "% AS::ActivateTask(...)"
    || "% AS::WaitEvent(...)"
    || "% AS::GetTaskID(...)";

// contains all read-only system services
pointcut asConstServices() = "% AS::GetTaskID(...)";

//////////   APPLICATION POINTCUTS
pointcut pcConstMethods() = "% ...::%(...) const";
// entry functions
pointcut pcStartFuncs() = "void ...::functionTask%()";
pointcut pcISRs() = "void ...::functionISR%()";

// per-OS-Application pointcuts (generated)
pointcut pcApplicationApp1() = "% App1Component::...::%(...)";
pointcut pcApplicationApp2() = "% App2Component::...::%(...)";

// aggregate trusted/non-trusted pointcuts (generated)
pointcut pcTrustedApps() = "";
pointcut pcNonTrusted()= pcApplicationApp1()||pcApplicationApp2();
pointcut pcNonTrustedISRs() = pcISRs() && pcNonTrusted();

//////////   PROTECTION-DOMAIN CROSSING POINTCUTS
pointcut pcOS() = asServices() && !asConstServices();
pointcut pcInOS() = execution(pcOS());
pointcut pcToOS() = call(pcOS() && within(pcNonTrusted());

// trusted and non-trusted function calls (generated)
pointcut pcToApplicationApp1() = call(pcApplicationApp1()
  && !within(pcApplicationApp1()));
pointcut pcToApplicationApp2() = call(pcApplicationApp2()
  && !within(pcApplicationApp2()));
pointcut pcToN() = pcToApplicationApp1() || pcToApplicationApp2();
pointcut pcToT() = call("");

pointcut pcNtoT() = pcToT() && within(pcNonTrusted())
    && !call(pcConstMethods());
pointcut pcNtoN() = pcToN() && within(pcNonTrusted())
    && !call(pcConstMethods());
pointcut pcTtoN() = pcToN() && within(pcTrustedApps())
    && !call(pcConstMethods());
```

Listing 4.2: Pointcuts of the Memory Protection Subsystem for Write Protection. If read or execute protection is additionally enabled, read-only system services and (non-)trusted functions are not excluded in the respective pointcut definitions.

Kernel protection requires the following basic operations:

- `enterTrusted()`: Switches the processor to supervisor mode and disables the memory protection unit.
- `leaveTrusted()`: Leaves the trusted protection realm by switching the processor to user mode and enabling the memory protection unit.
- `exportStack(func)`: Explicitly sets up the TStack region to provide access to the remaining unused portion of the currently active stack, and executes the provided function. On return of the function, the previous stack region is restored. This operation is intended to provide non-trusted ISRs and non-trusted functions with a runtime stack.

As all non-trusted OS-Applications belong to the same protection realm in this setting, the same MPU regions are used for all applications. To enable kernel protection, the `leaveTrusted()` operation needs to be invoked whenever the trusted domain is left, and conversely `enterTrusted()` needs to be invoked at all join points where a transition to trusted context happens:

Task Dispatch Upon dispatch of a task in a non-trusted OS-Application, privileged mode needs to be left (pointcut `pcDispatch`, not shown in Listing 4.2).

Non-Trusted ISRs When entering an ISR, processors typically switch to privileged mode. Upon activation of an ISR that belongs to a non-trusted OS-Application, privileged mode must be left, and reentered before returning from the ISR (pointcut `pcNonTrustedISRs`). In addition, a stack for the execution of the ISR must be made available by the `exportStack()` operation.

Non-Trusted Functions Privileged mode needs to be left when calling a non-trusted function from a trusted OS-Application, and reentered on return (pointcut `pcTtoN`).

Trusted Functions Privileged mode needs to be entered when calling a trusted function from a non-trusted OS-Application, and left on return (pointcut `pcNtoT`).

System Services When calling a system service from a non-trusted OS-Application, privileged mode needs to be entered for the call, and left before returning to the application (pointcut `pcInOS` or `pcToOS`, see below).

Pointcuts for all affected code join points are contained in Listing 4.2, except for the task dispatch, for which CiAO already provides a pointcut. Listing 4.3 exemplarily shows the advice code that implements the switch to privileged mode for system services. The advice code uses the execution code join point, which weaves within the body of the system service as opposed to the call site. The advice code is therefore

```
aspect os_mp_Trusted {
    advice pcInOS() : around() {
        bool istrusted = true;
        if(! isTrustedMPU() ) {
            istrusted = false;
            enterTrusted();
        }

        // execute the original code join point
        tjp->proceed();

        if(!istrusted) {
            leaveTrusted();
        }
    }
};
```

Listing 4.3: Aspect that Provides Privileged Mode Switching for System Services

independent of the calling context, and therefore needs to consider both trusted and non-trusted calling OS-Applications. This is implemented by a runtime check of the active privilege level of the caller. The privilege mode switches are only carried out if the caller executes in non-trusted context.

Exposure of the Privilege-Mode Switching Operations The presented implementation directly exposes the `enterTrusted()` operation to the application. A non-trusted OS-Application can use the operation anytime to elevate its privilege level. This may seem problematic at first, but is tolerable in an environment in that memory protection is used for safety, not security, purposes. The applications are assumed to not behave maliciously, are well known and the binary code can be statically checked to not contain the processor instruction that initiates the transition to trusted context. Most processors provide a designated *system call* instruction that triggers a dedicated trap to handle the system call. The `enterTrusted()` operation compiles to exactly this instruction. The trap handler disables the memory protection subsystem and resumes the caller in trusted execution context. When compiling the application code without the memory protection aspects, there should be no occurrences of the system call CPU instruction in the code of the resulting binary.

The more traditional approach is a dynamically bound system call, where the ID of the targeted system call is provided in addition to the parameters to the trap handler. The main advantage of retaining the static binding is that a uniform system call mechanism can be used for configurations with and without memory protection. In addition, the static binding is also slightly more efficient.

Call-Side versus Callee-Side Weaving AspectC++ can apply code transformations either at call sites (`call` join points) or within the method bodies (`execution` join points). Weaving at the call sites has the advantage that the respective context of each call site can be incorporated, whereas weaving in the method body avoids code duplication. The trade-off is similar to the decision on whether a method should be inlined or not.

Concerning the weaving of protection context switches, call-side weaving (`pcToOS` pointcut) allows a static distinction of trusted and non-trusted callers. Trusted callers are not affected at all, and non-trusted callers do not require the runtime check for the current privilege level that is shown in Listing 4.3. The `enterTrusted()` and `leaveTrusted()` operations consist of only few CPU operations, thus the effect on code size is little[6]. Call-side weaving is therefore preferable to realize kernel protection, and CiAO originally used call-side weaving instead of the method-body-side variant shown in Listing 4.3.

When porting the I4Copter software from its original operating system PXROS-HR to CiAO, however, a technical issue with call-side weaving appeared. The C++ parser of AspectC++ has problems with parsing template definitions and additionally does not support weaving within template code. While CiAO has been developed to work around these issues, the I4Copter code widely uses templates and cannot be parsed by AspectC++. Method-body-side weaving transforms the system call code within the operating system code and does not require weaving within the application code, evading this problem. I therefore added method-body-side weaving as an alternative to the call-side weaving, and will use it in the remainder of this thesis and particularly within the evaluation as call-side weaving does not currently work with the I4Copter code.

4.4.3.3 Application and Control-Flow Isolation

For kernel protection, all non-trusted applications share an identical MPU configuration. The MPU regions therefore only needed to be initialized once on startup. Transitions between trusted and non-trusted protection contexts are realized by enabling and disabling the memory protection unit. To isolate different non-trusted OS-Applications, a different set of regions is needed for each non-trusted OS-Application, and a new basic operation `setMPUForApplication(ToApp)` is introduced that configures the MPU regions to reflect the access permissions of the given OS-Application. The base implementation of `setMPUForApplication()` only changes the region used for the private code and data segment of the OS-Application (AppCode and AppData). The region for the shared read-only data (CRData) is identical for all non-trusted OS-Applications and needs not be changed. The base functionality is extended by further aspects to change additional regions as required by the configured functionality, for example to properly set the stack region (TStack) if control-flow isolation is enabled.

[6]On the Tricore architecture, the `enterTrusted()` operation consists of a single instruction, and the `leaveTrusted()` operation consists of five instructions.

The `setMPUForApplication()` operation needs to be issued at the following join points:

Task Dispatch Upon dispatch of a task in a non-trusted OS-Application, the MPU region needs to be configured for the OS-Application in the context of that the dispatched task currently executes (pointcut `pcDispatch`).

Non-Trusted ISRs Upon activation of an ISR in a non-trusted OS-Application, the protection context of the interrupted control flow needs to be preserved. The MPU is reconfigured to the OS-Application the ISR belongs to. Upon termination of the ISR, the preserved protection context is restored (pointcut `pcNonTrustedISRs`).

Non-Trusted Functions Upon start of a non-trusted function, the protection context that the running task executes in needs to be preserved. The calling protection context is saved on the stack and the MPU is reconfigured to match the context of the application exporting the non-trusted function. On return of the non-trusted function, the preserved protection context is restored (pointcut union of `pcTtoN` and `pcNtoN`). With the isolation of non-trusted applications, non-trusted functions additionally require the `exportStack()` operation to provide the non-trusted function with access to the remaining portion of the runtime stack of the calling task, which it could otherwise not access from within the context of the callee application.

Trusted Functions On call of a trusted function from a non-trusted OS-Application, the protection context of the caller needs to be preserved before entering trusted execution mode. Upon return from the trusted function, the original context is restored (pointcut `pcNtoT`).

4.4.3.4 Mapping Portals to Non-Trusted Functions

KESO's portal mechanism is very similar to non-trusted functions in CiAO. A portal call in the presence of hardware-based memory protection requires the same changes to the MPU configuration that a non-trusted function requires. KESO needs to additionally update the identifier of the currently active domain and set a mark on the stack that tells the garbage collector that the following stack partition belongs to a different domain. KESO generates pointcut definitions that identify the portal proxy methods. The CiAO aspects for non-trusted functions can be applied to these pointcuts.

4.5 Determining Domain Reachability for Java Code

For Java applications, the same principle issue of grouping code and data to consecutive memory regions for each OS-Application needs to be solved as for native CiAO applications. KESO's domain concept greatly simplifies the issue for data items, however. With the physical separation of the heaps and static fields, the domain-specific data can easily be identified and colocated in a consecutive memory region for each domain, fitting the memory layout required by CiAO. Hardware-based memory protection with respect to read and write accesses can thus easily be supported by KESO.

To support execution protection, the code of applications and libraries needs to be identified to belong to a particular domain (and consequently an OS-Application), or to be shared code. Execution protection is a measure that allows the detection of additional errors in the program execution, but it is not needed to provide fault containment. For example, an overflow of a stack-allocated array (in a C application) could corrupt the return address, and upon return from the function the execution would resume at the corrupted address. With execution protection, the error would be detected unless the corrupted address still points to the executable code regions of the active OS-Application. The distinction of shared functions and private functions needs not be optimal, but the more precise the assignment of private functions to private data segments is, the more effective execution protection will be in detecting violations of the control-flow integrity.

In CiAO, the assignment of private code to OS-Applications is performed explicitly by the system integrator at the granularity of C++ classes that are assigned to OS-Applications. Any code that is not explicitly assigned to an OS-Application this way, and that is not in the known namespaces of the kernel code, is treated as shared code. This approach works well for small applications. With the port to CiAO of the I4Copter as the first larger CiAO application, however, the manual partitioning turned out to be tedious for the multitude of C++ classes, and the assignment unit of C++ classes showed to be too coarse grained, as functionality of many classes is frequently used by multiple OS-Applications. The consequence was that the code assignment was practically limited to the classes containing the entry functions of the control-flows of the respective OS-Applications, and the remaining code was treated as shared code. This effectively constrains execution protection to the boundary between kernel and application code.

With the whole-program static analyses performed by jino, an automated and finer-grained solution is viable, which has the major benefit that no changes to KESO's programming model are needed and existing KESO applications can benefit from hardware-based memory protection without changes to the existing code. The idea is to identify for each basic block the set of domains from that it is reachable. For the problem of determining Java methods exclusively used from one domain and those shared by two or more domains, the set of domains from that each method is reachable is that of the method's entry basic block. This information is, however, valuable for other problems and optimizations as well, for example:

- Static service protection enforcement: Service protection means that the parameters to certain system services are restricted to subsets that depend on the calling OS-Application. For example, a task may activate a task in a different OS-Application only if the system definition explicitly allows it to do so. With reachability information at the basic block level, dynamic parameter checks can partially be replaced by compile time checks. In addition, inconsistencies between the code and the system definition can partially be detected at compile time.

- Absolute addressing of static fields: As discussed earlier, static fields need to be accessed by an indirection to enable shared code between multiple domains. This indirection can be removed if the access is within a basic block that is reachable from a single domain only.

- Per-application configurability of software-based protection (Section 4.6.2)

4.5.1 Overview of the Reachability Analyses in Jino

The core of jino's middle end is an iterative SSA-based [31, 109] work-list algorithm that combines data-flow and 0CFA [99] control-flow analysis. To determine the domain reachability for each basic block, the middle end was adapted to separately process and collect the information for each domain. A key issue to achieve a satisfactory level of preciseness in determining the reachable code for each domain is the devirtualization [2, 115] (that is, the static binding) of virtual method calls, given that all regular method calls are virtual in Java.

The devirtualization of a virtual method call is possible if the runtime type of the target reference can be determined at compile time specifically enough, so that only a single candidate method remains in the resulting class sub hierarchy. Jino combines two analyses to statically determine the type. The first is a data-flow analysis, which propagates the types from allocation sites to all variables in the program. Where paths with differing types flow together, the most-specific common supertype is remembered for the variable. The second analysis is a rapid type analysis [14] (RTA), which determines the set of live types for each domain, based on all allocation operations in the reachable code of the respective domain. The analyses affect each other and are applied iteratively. The actual set of type candidates for a virtual method call site is determined by removing from the class sub hierarchy determined by the data-flow analysis all classes, which are not in the set of allocated types as determined by the RTA, except for those that implement a method candidate not overridden in an instantiated subclass. A class hierarchy analysis [32] of the remaining sub-hierarchy determines the possible candidate methods. If the result is a single method, the call can be statically bound.

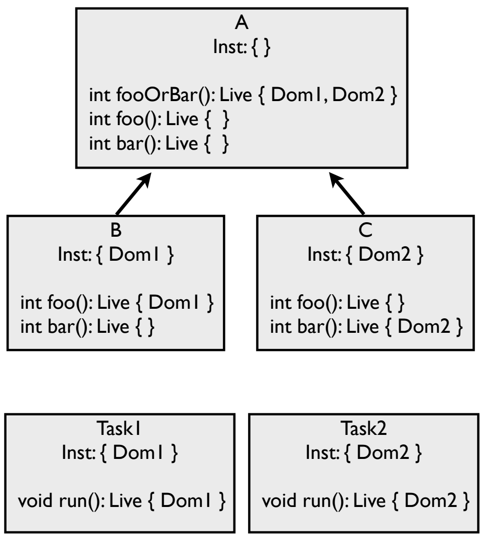
A detailed description of the static analyses in jino is available in the dissertation of Christian Wawersich [123]; the redesign of the analyses to independently process the different domains is covered in Christoph Erhardt's diploma thesis [35].

4.5.2 Domain Reachability Example

Figure 4.8 shows a simple example that illustrates the reachability information collected by jino. The example contains a configuration with two domains, each

```
class A {
  static int fooOrBar(A inst,
      boolean runFoo) {

    if(runFoo)
      return inst.foo();
    return inst.bar();
  }

  int foo() { return 1; }
  int bar() { return 3; }
}

class B extends A {
  int foo() { return 2; }
  int bar() { return 2; }
}

class C extends A {
  int foo() { return 3; }
  int bar() { return 1; }
}
```

(a) Class Hierarchy



(b) Code and Type Reachability

```
// belongs to Domain Dom1
class Task1 {
  void run() {
    A.fooOrBar(new B(),true);
  }
}
```

(c) Task 1 Entry (Domain Dom1)

```
// belongs to Domain Dom2
class Task2 {
  void run() {
    A.fooOrBar(new C(),false);
  }
}
```

(d) Task 2 Entry (Domain Dom2)

Figure 4.8: Domain-Specific Rapid Type Analysis and Reachability Analysis

containing one task. The code base contains a simple class hierarchy consisting of the class `A`, which implements the two instance methods `foo()` and `bar()`, and two subclasses of `A`, `B` and `C`, which override the two methods with own implementations. In addition, the `A` has a static method that either calls `foo()` or `bar()` on a provided instance. The first task calls this method with an instance of `B` and asks it to call `foo()`, the second task calls it with an instance of `C` and asks it to call `bar()`.

Figure 4.8(b) shows the results of the reachability analysis for this example at the class and method level. For each class, the set of domains, in which instances of the class exist, is collected (*Inst*). For each method, the set of domains from that the method is reachable is collected (*Live*). Initially, jino knows from the system definition that `Task1`/`Task2` are instantiated from only DOM1/DOM2, and that the task entry methods `run()` are respectively reachable. Both entry functions invoke `A.fooOrBar()`, which consequently is reachable from both domains at the method granularity. At the basic block level, however, the if-case of the method body is only reachable from DOM1, whereas the implicit else-case is reachable only from DOM2. This information, together with the information that `B`/`C` are only instantiated from DOM1/DOM2 enable jino to devirtualize the two method calls and identify a single live candidate for `foo()` and `bar()`.

Concerning the problem of identifying private and shared code, `Task1.run()` and `B.foo()` can be mapped to the private code segment of the OS-Application that DOM1 is assigned to, and respectively `Task2.run()` and `C.bar()` for DOM2. Only `A.fooOrBar()` is put into the shared code segment. The other candidates are dead code and eliminated.

4.5.3 Reachability Results for the I4Copter Codebase

Table 4.2 shows the results of the domain reachability analyses for the Java port of the I4Copter application. The numbers include all used Java library code. The majority of methods and basic blocks can be identified to be reachable from a single domain only. Consequently, 64 % of the methods can be placed in the private code segments of the respective OS-Application. As a further measure to reduce the amount of code in the shared code segment, a part of the methods reachable from only two domains could be placed in an overlapping part of the two domains' private code segments. However, given the low amount of methods that are reachable from exactly two domains, I have not further pursued this technique for the placement of methods in this thesis.

4.6 Configurable Software-Based Protection

The graduations for software-based protection developed in Section 3.4 are based on the selective omission of runtime checks. I categorized the most-common runtime checks into the impact classes local and global. Only the checks of the impact class global are required to retain the spatial isolation of software-isolated applications.

Domain	**Classes**	**Methods**	**BBs**
FlightControl	36 (6, 17 %)	76 (27, 36 %)	952 (607, 64 %)
CopterControl	68 (11, 16 %)	166 (40, 24 %)	1190 (612, 51 %)
SerialCom	30 (3, 10 %)	78 (30, 38 %)	339 (241, 71 %)
SignalProcessing	91 (33, 36 %)	232 (103, 44 %)	2068 (1463, 71 %)
Ethernet	31 (9, 29 %)	56 (27, 48 %)	291 (222, 76 %)

(a) Instantiated Classes and Reachable Methods and Basic Blocks per Domain. The numbers in brackets show the amount exclusively used by the respective domain.

# of domains	**Classes**	**Methods**	**BBs**
1	62 (52 %)	227 (63 %)	3145 (84 %)
2	13 (11 %)	52 (14 %)	225 (6 %)
3	25 (21 %)	55 (15 %)	325 (9 %)
4	7 (6 %)	13 (4 %)	35 (1 %)
5	13 (11 %)	12 (3 %)	26 (1 %)
Total	120	359	3756

(b) Distribution of Classes and Basic Blocks

Table 4.2: Domain Reachability in the I4Copter Application for Allocated Classes, Reachable Methods and Basic Blocks (BBs)

The cost of software-based memory protection can be reduced by omitting runtime checks. Firstly, static analyses can prove the checked condition to always hold at compile time, and therefore safely eliminate the runtime check. Secondly, I discussed how the memory characteristics of the target system, optionally with the help of a memory protection unit, could be leveraged to offload some runtime checks to hardware exception mechanisms. Both techniques achieve a reduction of code size and execution time without impairing the safety of the program. Thirdly, the runtime checks with local impact can be omitted for a further cost reduction, at the price of losing the detection of some memory access errors that only affect the containing program. Finally, the remaining checks of the global impact class can be omitted. The result is an unsafe program similar to a program written in a language such as C. The program can be isolated using hardware mechanisms, or be accepted as part of the trusted computing base.

For the offloading of runtime checks to the hardware, jino needs to be aware of the memory characteristics and memory bus behavior of the target platform. This information may also allow more runtime checks to be assigned to the impact class local. With this knowledge, the compiler can perform the impact classification and omit runtime checks according to the configured safety level, by applying the rules from Section 3.4. For the mixed-mode operation of software- and hardware-isolated applications, this configuration should be possible on a per-domain basis in KESO. In the following, I present how the memory characteristics are provided to jino and how the individual configurability of the level of software-based protection on the basis of domains is enabled.

```
MemoryDescription {
    # accessing 0x0 to 0x7 generates a MPN trap
    # accessing 0x8 to 0x7fffffff generates a bus error
    reserved_virtual_address_space = {
        origin = 0;
        length = 0x80000000;
        read = "trap";
        write = "trap";
    }

    # accessing 0xf8800000 to 0xffffffff generates a bus error
    reserved_space_high = {
        origin = 0xf8800000;
        length = 0x07800000;
        read = "trap";
        write = "trap";
    }
}
```

Listing 4.4: TC1796 Memory Characteristics Description

4.6.1 Incorporating Memory Characteristics into the Compiler

To provide the memory characteristics and memory bus behavior to jino, KESO's configuration was extended by a new block `MemoryDescription`, which defines the memory characteristics for different regions of the address space. Listing 4.4 shows the most important address regions for the memory description for the TC1796 microcontroller.

For each defined region, the start address (origin), length, and the behavior of load (read) and store (write) instructions targeting addresses within the region are defined. Possible behaviors are:

Regular An address region backed by memory or mapped to device registers, which may be accessible to an OS-Application. The compiler makes no assumptions on the region, except that load accesses to the region do not cause side effects. For loads, the compiler assumes that random values are returned. For stores, the compiler assumes that the region holds data that must not be modified in an uncontrolled manner. This is also the default for all regions that are not explicitly defined in the memory description.

Trap An access of the respective type targeting an address within the region is signaled by a hardware exception.

Volatile (loads only) Load accesses targeting an address within the region provide random values and may additionally cause side effects. An example is a memory-mapped shift register through which a hardware buffer can be accessed, and where reading the register removes a value from the buffer.

Const (loads only) A load access targeting an address within the region reads a compile-time constant value. This setting is provided with a byte string, which is repeated by the compiler to the size of the region. An example where this setting is usable is vendor reserved address regions, which often simply read as zero (as specified in the processor manual).

Ignore (stores only) A store to the addresses within the region has no effect. This setting is mainly useful on processors without a hardware exception mechanism, for example the AVR architecture.

The memory description is provided with KESO for supported targets. It is part of the regular application configuration file. The developer can either include the descriptions provided with KESO, or easily define a custom memory description for a particular development board. The memory description needs not be complete. For jino, it is sufficient to know a region of sufficient size to that accesses trigger a hardware trap, or to that store accesses are ignored. The region definitions for the TC1796 processor shown in Listing 4.4 are sufficient for jino for the currently applied rules.

4.6.2 Per-Application Configurability

To support a mixed-mode operation as motivated in Section 1.2, where different spatial isolation schemes can coexist in a system configuration, the configuration of software-based protection needs to be possible on a per-domain basis. In combination with the possibility to disable and enable hardware-based memory protection by setting OS-Applications as trusted or non-trusted, all isolation variants (as developed in Section 3.3) can be realized and coexist side-by-side.

The software-based protection in my implementation is enabled, disabled, and graduated by varying the amount of runtime checks in the generated application code. For differing configurations for different applications, jino needs to identify the domain that a checked operation is reachable from and compile the code according to the setting made for this domain. The reachability analyses outlined in Section 4.5 provide this information at the granularity of basic blocks and enable jino to differentiate the runtime check emission at this granularity level.

4.6.2.1 Shared Code: Specialization versus Generalization

An obvious issue that arises is basic blocks that are reachable from multiple domains with differing settings for the level of software-based protection. For the I4Copter application, 16 % of the basic blocks are reachable from two or more domains[7].

There are two principal approaches to handle this situation, specialization and generalization. With specialization, different variants of the containing method of the basic block could be created, each of which reflects one of the participating

[7]That is, jino was not able to prove the contrary.

settings. With generalization, the safest of all participating settings could be used to compile the basic block. The compiled basic block is safe to use in all settings. Specialization may save execution time in domains with less strict protection settings, but on the other hand increases the code size and has further implications on the runtime environment. Specialization introduces new method candidates that need to properly be selected. Dynamically bound calls need changes to the dynamic dispatch mechanism, for example dispatch tables that are (partially) domain specific.

In my current implementation, I opted for the approach of generalization for its simplicity. Basic blocks that are reachable from multiple domains are compiled using the safest from the set of protection settings in question. It is questionable whether the execution time saved by specialized method variants weighs in for the added footprint and increased complexity of the runtime environment. I have not addressed this question in my thesis.

4.6.2.2 Incorporating Combined Protection

The system definition provided to KESO contains information on which domains are mapped to a hardware-isolated OS-Application. This knowledge can be utilized by jino in combination with a memory description that defines the address regions destined for regular memory. This may allow the omission of additional runtime checks on platforms, where no reserved region of sufficient size to that accesses trigger a hardware exception exists.

4.7 Chapter Summary

In this chapter, I presented the core of my framework composed of the AUTOSAR OS-like operating system CiAO and the self-tailoring multi-JVM KESO. CiAO provides the MPU-based protection part of the protection model developed in Chapter 3, and KESO implements software-based isolation for Java applications. I presented a toolchain in that the applications output by KESO are indistinguishable to CiAO from native C++ applications for CiAO, and therefore no special KESO support needed to be added to CiAO. This approach additionally enables the coexistence of Java applications and native C++ applications. I adapted KESO's runtime environment to support MPU-based protection. Most notably, I reorganized the data structures of the runtime environment so that data that belongs to different domains is never part of a single data structure.

I presented how the configurable MPU-based protection is realized in CiAO. The approach is aspect-oriented. Join points in the code at that a change of the protection context takes place are described by pointcut expressions. These pointcut expressions are used in aspects that can be accumulated to provide the memory protection variants of the protection model matching the configuration.

At the programming interface, CiAO defines a component-based application model that maps to C++ language constructs. The protection realms are defined independently of the code by assigning components (C++ classes) to OS-Applications. I discussed the practical shortcomings of the approach that showed up when adapting the I4Copter application to this component model. In particular, the static partitioning of the code base at the granularity of C++ classes turned out to be tedious, and the class granularity too coarse grained. As a consequence, I pursued a different automated approach based on static code reachability analyses in jino, which showed to be able to determine a single domain for the larger part of the I4Copter code base and generates according pointcut expressions for CiAO's memory protection subsystem. I showed how the same analyses are used to provide per-application configurability of the software-based isolation provided by KESO.

At this point, the framework is complete for Java applications that use KESO's native portal mechanism to communicate. The framework fulfills the goals of fine-grained configurability and the mixed-mode operation of different spatial isolation mechanisms (Section 1.2). It also enables the coexistence of native CiAO applications side-by-side with the Java applications, but communication is limited to the basic activation and notification services provided by the operating system. In particular, no mechanism to exchange data between native applications and Java applications has been defined so far. In the following chapter, I introduce such communication mechanisms with the aim of providing a soft migration strategy.

5 Component-Wise Soft Migration

The framework developed in Chapter 4 provides configurable memory protection with variable protection degrees for applications and application components with differing characteristics and requirements, but requires the applications to be written in Java. Java currently is, however, rather exotic in the domain of statically configured, deeply embedded systems. On the other hand, a huge base of legacy code predominantly written in C exists. Abandoning or porting all the existing code is not feasible for both cost and time reasons. In this chapter, I present an extension to the framework that allows the interaction of native C or C++ components with Java components, isolated in separate OS-Applications. The extension enables the co-existence and cooperation of C or C++ applications with Java applications. In addition, it provides the possibility of a component-wise migration from C or C++ to Java, to open the full spectrum of software-based memory protection to these application parts.

In the following, I first discuss why software components pose a well-suited unit for a soft-migration approach, and why CiAO and KESO both can be considered component systems. The main part of the chapter is concerned with the presentation of system abstractions that enable the cooperation of C or C++ and Java components. Based on these abstractions, I conducted a full, component-wise port of the I4Copter C++ application to Java.

5.1 Migration Granularity: Software Components

A software component [76] encapsulates a set of related software functions and interacts with other software components by well-defined interfaces. Key properties of software components are the reusability in multiple contexts and the replaceability by other software components that implement the same interface. Software components have long existed and are a well-established software structuring technique. Because of the strong encapsulation and the property of replaceability, they pose an appropriate granularity for a soft-migration approach.

With software components as the migration unit, different migration approaches are possible. For example, individual components can be ported to Java and be used as a replacement for the original component in combination with the other legacy components. Or, alternatively, existing components can be retained and continued to be used, and only newly developed software components are added as Java components. In the long run, the legacy components can be gradually phased out.

5.1.1 Considering CiAO and KESO as Component Systems

Both CiAO and KESO are component systems. In CiAO, a component is represented as a C++ class, where the public interface of the C++ class represents the interface of the component, implicitly exported as an external interface to other components as (non-)trusted functions. In KESO, a domain can be considered a component, whose interface consists of the interfaces of the exported services.

To enable the communication of native CiAO components with Java components, a bridge between KESO's portal mechanism and CiAO's (non-)trusted functions could be created. Interface description languages provide a language-independent way to describe a component interface. Based on such a language-independent description, data conversion code could be generated that converts the passed data items from the data representation of C++ to Java's representation and vice versa.

RPC-like interfaces such as portals and (non-)trusted functions are, however, not commonly found in operating systems for my target domain and therefore are of limited use to provide a soft-migration path for legacy components that do not use such interfaces. Therefore, I have not pursued this approach and instead focused on two more common communication mechanisms, shared memory and message ports. In this chapter, I present how I extended CiAO and KESO by these communication mechanisms to enable the data exchange between native and Java components.

5.1.2 I4Copter Component Interfaces

The I4Copter software is based on the CoSa (**Co**mponent architecture for **Sa**fety-critical embedded systems) framework, a user-level component framework, which provides a uniform C++ API for exchanging and sharing data among components, the so-called *connectors*. The framework provides two implementations of the connector API, one that is mapped to the mailbox messaging mechanism of the PXROS-HR operating system, and one that merely passes a pointer to an area of shared memory. Message channels can be established dynamically in PXROS-HR, whereas CiAO's static model requires all communication channels to be explicitly defined in the system definition at system creation time. The shared memory areas in the CoSa framework are not known to the operating system. PXROS-HR leaves the definition of the protection regions to the application, and the I4Copter application is developed

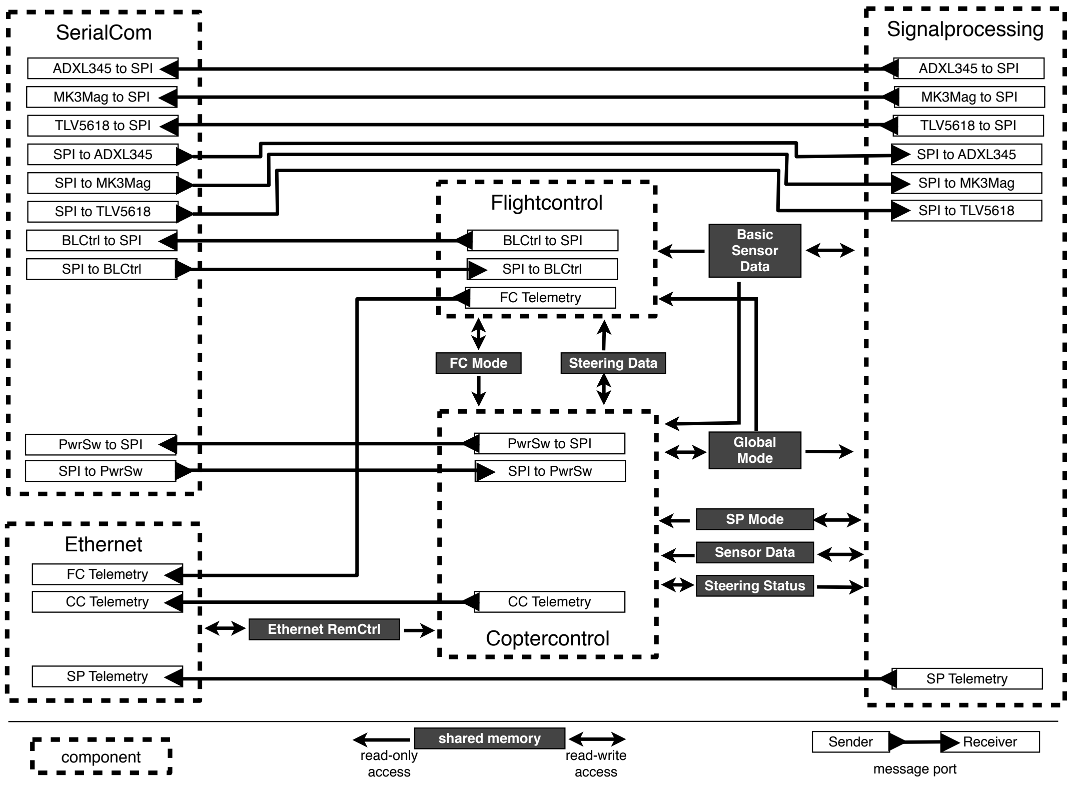


Figure 5.1: I4Copter Component Interfaces

to work with write-only protection and assumes global read permissions for all applications. All shared memory regions in the I4Copter have a single producer component, that accesses the shared memory area in read-write mode, and one or more consumer components that can only read from the shared memory area. The shared memory area is located in the private data segment of the producer component. The consumers are provided with a pointer by the CoSa API, and can read from the shared memory area because global read access is presumed.

For the strictly static model of CiAO, I extracted the component interfaces from the code of the I4Copter application, that is, the existing shared memory regions and message channels, and which components use which of these communication channels, and in which role (producer or consumer of a shared memory region, sender or receiver of a message connection). The result is depicted in Figure 5.1. I extended CiAO with suitable shared memory and messaging mechanisms to enable the I4Copter software to run on top of CiAO, and developed a safe Java API that enables Java applications to utilize these mechanisms. I present these extensions in the remainder of this chapter. Based on these extensions, I ported the I4Copter software component by component to Java. The outcome is a C++ and a Java variant of each component, which can be combined arbitrarily to a fully functioning control software.

5.2 Shared Memory Extension for CiAO

A shared memory area is a typed area of memory that is made accessible to multiple components; a global variable that is accessed by multiple components is an implicit form of shared memory. A shared memory area is accessed using regular memory operations. The operating system only ensures that the authorized components are able to access the shared memory area. As opposed to the message-based communication, no particular access protocol is defined for the use of shared memory.

Shared memory is a controversially viewed mechanism, because it implies a number of issues: Since the shared memory area is accessible from multiple components at the same time, the accesses must be synchronized as required on the application level. The implicit use of shared memory weakens the encapsulation of components. Still, shared memory is found in many existing applications (for example, use of global variables by multiple components), and to support a soft migration I include an explicit form of shared memory in the extension of my framework. In the following, I present the shared memory extension.

5.2.1 Definition of Shared Memory Areas

Shared memory regions are globally defined in the system definition, and so are the uses and access modes on the shared memory area. This form of shared memory is explicit, because all uses are documented in the system definition and therefore are a visible part of the component interface. Listing 5.1 shows an excerpt from a KESO system definition for one shared memory area in the I4Copter application. Each shared memory area is described by the following attributes:

- A system-widely unique name that identifies the shared memory area. This name is used in the C++ and Java programming interfaces to access the shared memory area from the program code.
- A C data type for the memory area.
- Optionally, the filename of a C header file containing the definition of the type. Shared memory instances are statically allocated in a generated separate compilation unit, wherefore the type definition must be available. The information could also be used to create a Java version of the type for the use with memory-mapped objects in KESO, but this process is currently manual.

In addition to the global definition, each component[1] that accesses a shared memory region must declare this use in the system definition, including the information on whether the shared-memory area is accessed read-only or read-write.

[1] Shared memory uses are declared at the domain level in KESO and at the level of OS-Applications in CiAO.

```
System(I4Copter) {
    SharedMemory(SteeringData) {
        ctype = "basicFlightCtrlData_t";
        cheaderfile = "FlightControlData.h";
    }

    Domain(CopterControl) {
        UseSharedMemory = SteeringData {
            mode = "rw";
        }
    }

    Domain(FlightControl) {
        UseSharedMemory = SteeringData {
            mode = "r";
        }
    }
}
```

Listing 5.1: Shared Memory Definition

5.2.2 Shared Memory Placement

The operating system has to ensure that the components that declared the use of a shared memory region are able to access the shared memory region at runtime. In Chapter 2, I decided against a virtualization of the MPU regions, which makes memory regions accessible on demand but suffers from predictability issues. To avoid the need for an extra MPU region for each shared memory area that a component needs to access, the shared memory areas should be placed in an address region that is already accessible, if possible.

5.2.2.1 Common Case: No Read Protection, Single Writer

For the common case of write-only protection and shared memory areas with a single writer component, this task is simple. The shared memory area is placed within the private data segment of the producer component's OS-Application. With global read access, it is automatically readable by all reader components.

My current prototype only covers this case. Nevertheless, I discuss the ramifications with respect to the region requirements for settings where hardware-based read protection or multiple writer components are used. My discussion assumes the following simplifications, which do not cause a loss of generality:

- Uses of a shared memory area by trusted OS-Applications are ignored, since trusted OS-Applications are able to both read and write a shared-memory area without special arrangements.

		Ethernet	Coptercontrol	Flightcontrol	Signalproc.
S1	Ethernet RemCtrl	read-write	read-only		
S2	Steering Status		read-write		read-only
S3	Global Mode		read-write	read-only	read-only
S4	Steering Data		read-write	read-only	
S5	FC Mode		read-only	read-write	
S6	Basic Sensor Data		read-only	read-only	read-write
S7	SP Mode Sensor Data		read-only		read-write

Table 5.1: Shared Memory Uses in the I4Copter

Ethernet	Coptercontrol			Flightcontrol	Signalprocessing	
S1	S2	S3	S4	S5	S6	S7

Table 5.2: Shared Memory Placement in the I4Copter

- Shared memory areas with identical participants and use pattern can be treated as a single area and be co-located in memory.

5.2.2.2 Read Protection

If read protection is enabled, the reading components of a shared-memory area must be provided with read access to the shared memory area. As all private data regions (AppData, TStack) provide read-write access, but there is no private data region that provides read-only access, it is sensible to co-locate the shared memory region with the private data of the writer component. Therefore, the use of hardware-based read protection in combination with shared memory requires an additional MPU region in the reader components for each shared-memory area.

As an example, I illustrate the additionally required MPU regions to enable shared memory use in the presence of hardware-based read protection in the I4Copter. Table 5.1 summarizes all shared memory uses from the I4Copter application, as shown in Figure 5.1. For brevity, the shared memory areas are enumerated from S1–S7. S7 comprises the two areas SP Mode and Sensor Data, which were grouped according to the above simplification for their identical uses. The shared memory regions are placed next to the private data segment of the respective writer component, as shown in Table 5.2. With this placement, the following additional regions are needed to enable read access:

- Coptercontrol, three additional read-only regions
  - `start(S1)` – `end(S1)`

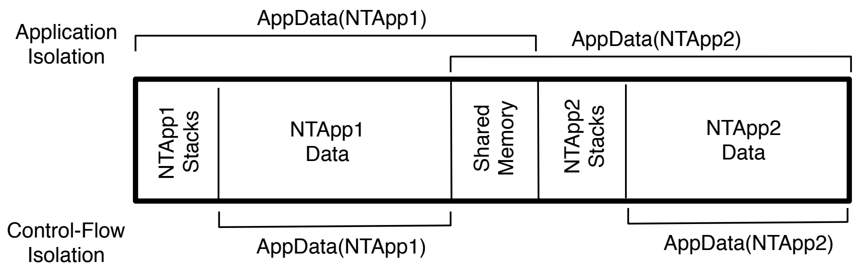


Figure 5.2: Shared Memory Placement by Overlapping Private Data Segments

  - `start(S5) – end(S5)`
  - `start(S6) – end(S7)`

- Flightcontrol, two additional read-only regions
  - `start(S3) – end(S4)`
  - `start(S6) – end(S6)`
- Signalprocessing, one additional read-only region
  - `start(S2) – end(S3)`

The region assignment already includes an optimization that subsumes multiple neighbored shared memory areas in a single region, for example S6 and S7 for the Coptercontrol component. The example illustrates that hardware-based read protection for the I4Copter is not possible on the Tricore TC1796 processor, where up to three of the total four data regions are already being used by the base configuration (Section 4.4.1), but providing read access to the shared memory areas would require up to three additional MPU regions. For the message-based communication discussed below, which is used by all components, additional regions may be required to provide temporary memory access at runtime.

5.2.2.3 Multiple Writers

The placement becomes more complicated if multiple writers exist for a shared memory area. For two writers, the occupancy of an additional MPU region can be avoided by placing the shared memory area between the two private data segments of the writing non-trusted OS-Application, and having the private data segment regions overlap on the shared memory area, as shown in Figure 5.2. It is discernible in the lower part of the figure that the technique can only be applied in the absence of control-flow isolation with the current memory layout; if control-flow isolation is enabled, a separate region (TStack) is used to grant access to the stack, and the AppData region does not include the task stacks and therefore cannot be extended to include a shared memory area shared with the preceding application[2]. For more than two writers, it is inevitable to use additional regions for all but possibly two writers, for which the overlapping placement can be used.

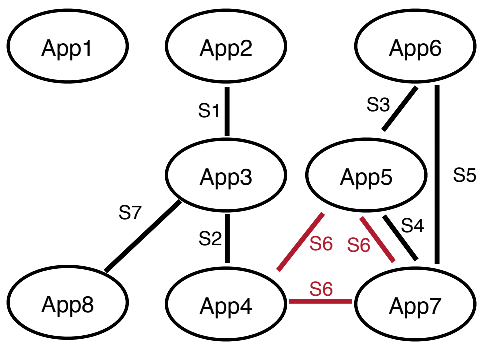
To utilize the overlapping technique, a linear order of arrangement needs to be found for the OS-Applications. This can be well visualized in an undirected graph, in which the vertices are OS-Applications and the edges labeled with a shared memory area indicate that two OS-Applications both write to that shared memory area. Figure 5.3 shows a fictive example in this graph form (the situation of multiple writers for one shared memory area does not occur in the I4Copter). A linear application order for the placement of the private data region in memory can be derived from the graph when it meets the following properties:

1. no more than one edge with a particular label
2. acyclic
3. all vertices are of degree two or less

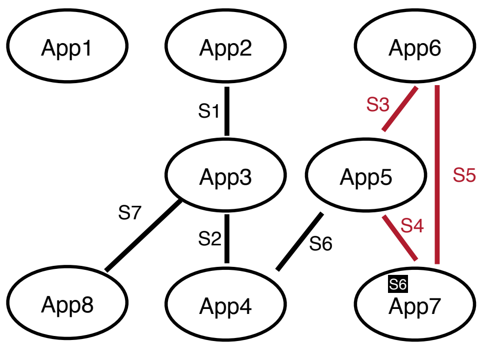
The graph can be transformed to meet the above properties by allocating extra MPU regions to the applications. Figures 5.3(b)–5.3(d) show how this is performed for the example graph.

1. Unique Edge Labels Shared memory areas with three or more writers lead to multiple edges with the same label in the graph. The overlapping technique can only be applied for two writers (that is, one edge). By assigning one of the writer OS-Applications an extra MPU region to access the shared memory area, $n-1$ edges with the shared memory area's label can be removed from the graph for a shared memory area accessed by n writers. The remaining uses correspond to an area with $n-1$ writers. The step can be repeatedly applied until only two writers remain. In the example, the area S6 is accessed by three writers, one of which needs to be assigned an MPU region to access the area. When choosing the OS-Application to receive an MPU region, it should be avoided to reduce the degree of one of the participating vertices below two, since this may eliminate a possible use of region overlapping. In the example, the region can either be assigned to App4, App5 or App7. Choosing App4 would reduce its degree to one. For the other two candidates, the choice may depend on other criteria such as the already used extra regions or the need for temporary regions (for example, some OS-Applications may use message-based communication, others may not). In the example, I assigned an extra region to App7 to access the area S6. The resulting graph is shown in Figure 5.3(b). After all edge labels are unique, multiple edges between the same two nodes can be merged, as the represented shared memory areas can be placed next to each other and be treated as a single one, corresponding to the second of the simplifications from Section 5.2.2.1.

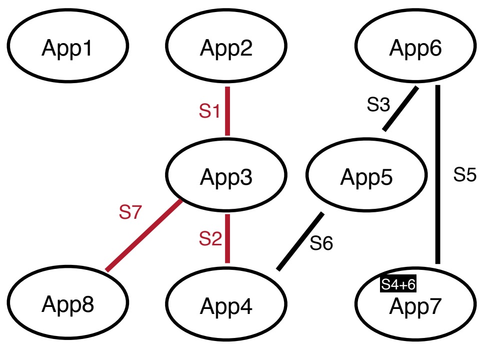
[2]The memory layout could, however, be easily modified to place the stacks separately from the private data segments in the case of control-flow isolation to solve the applicability issue, but the memory layout would no longer be uniform for all configuration options.



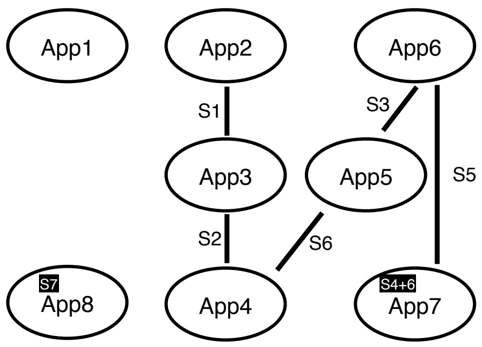
(a) Initial State



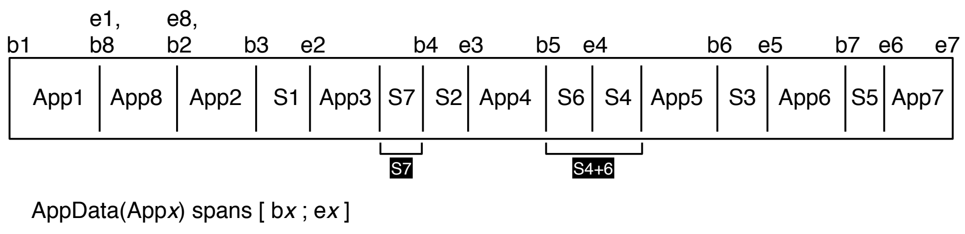
(b) Unique Edge Labels



(c) Cut Cycles



(d) Reduce Vertices' Degrees Greater than Two



(e) Application Order, Shared Memory Placement and Extra Regions

Figure 5.3: Example: Shared Memory Dependencies Among OS-Applications

2. Cut Cycles Cycles in the graph represent a shared-memory use constellation that cannot be solved by a linear order and overlapping private data regions. To resolve the situation, the cycle needs to be cut by realizing the shared memory access using an extra MPU region for one of the OS-Applications in the cycle. In Figure 5.3(b), there is a cyclic constellation between the OS-Applications App5–App7. Any of these three candidates can be chosen to remove one of the edges to break the cycle. In the example, App7 poses the best candidate and was chosen. It already has an extra MPU region to the shared memory area S6; after removing the edge S4, the degree of App5 is reduced to two, and therefore the area S6 can be placed in an overlapping region between App4 and App5. Cutting the edge S4 leaves the shared memory area S4 in the private data segment of App5, where it can be placed next to S6, and the already used extra region can be expanded to span both shared memory areas, avoiding the need for an additional region. Figure 5.3(c) shows the situation after the cycle has been cut by removing the edge S4.

3. Reduce Vertices' Degrees To Two Finally, in the acyclic graph with unique edge labels, the degree of all vertices with a degree higher than two needs to be reduced to two. In Figure 5.3(c), only App3 has a degree higher than two. Any of the edges that include App3 can be removed by assigning an extra MPU region to either App3 or the OS-Application at the other end of the edge to enable access to the shared memory area represented by the edge. In the example, it does make no difference which edge is removed, and which of the two OS-Applications is assigned the extra region. In other constellations, factors such as the degree of the OS-Application on the other end of the edge, mergeability with already assigned extra regions (as in the cycle cut), or the differing needs of different OS-Applications for leftover temporary regions may be incorporated in the decision. In Figure 5.3(d), the edge S7 was removed and App8 was assigned an extra region to access S7, which will be placed with the private data of App3.

The graph in Figure 5.3(d) fulfills the required properties to read a suitable application order. Each path through the graph shows a series of OS-Applications that should be placed in the order corresponding to the path in memory. Figure 5.3(e) shows one possible application order and shared memory placement for the example. It also shows the ranges for the AppData region of each OS-Application, and for the two assigned extra regions.

5.2.3 Shared Memory Conclusions

For the simple case of a single writer and in the absence of read protection, shared memory can be combined well with a static MPU region definition that provides the using OS-Applications with permanent access to the shared memory areas. Although I have not implemented the placement of shared memory and the region determination for the situations of read protection or multiple writers, the above discussion shows that these can quickly render a static region definition impractical. I showed that read protection for the shared memory constellation in the I4Copter application cannot be realized using a static region assignment with the four available data regions of the Tricore TC1796. This only leaves the options of MPU virtualization – with the intrinsic impact on predictability and execution time overhead – or changes to the interfaces of the application components.

Read protection is a not essential feature to provide spatial isolation and fault containment and only enhances the capability of detecting errors in the program. Software-based protection provides read protection at no additional cost, and can be used in place of hardware-based read protection to avoid the issue. This is in line with my argument in Section 2.5 to use MPU-based protection as a safety net for the safety-relevant parts, and to leverage the enhanced but not safety-critical error-detection facilities of the Java language and the multi-JVM concept.

For the case of multiple writers to a shared memory area, I sketched a technique that avoids the occupancy of extra MPU regions for the case of two writers by overlapping the private data regions of the two writing OS-Applications. The approach determines a suitable application order and assigns extra regions to access shared memory where it cannot be solved by overlapping data regions. It must be noted that such an application ordering may be in conflict with other goals that also require a particular application order. For example, on the Cortex-M3 microcontroller, MPU regions need to be of a size that is a power of two, and the start address needs to be aligned by the size of the region. To minimize the external fragmentation caused by these hardware requirements, CiAO implements an algorithm that orders the applications so that the external fragmentation is minimized. The results of this ordering are likely to conflict with the results of the placement that aims at reducing the number of needed MPU regions to access shared memory areas.

5.3 Message Ports for CiAO

Messages are a data-flow-oriented[3] approach for communicating among different protection realms in a system, or even on different systems in a network. As opposed to shared memory, sending and receiving messages requires the use of primitives provided by the operating system. Messaging mechanisms are found in many systems; examples in my target domain are OSEK COM [85] and the mailboxes in PXROS-HR [94].

For my framework, I designed a simple message mechanism for the communication among OS-Applications on a single node, although the mechanism could be extended to inter-node communication without changing the programming interface, and hence the application code. My message mechanism is optimized for the common case of write-only protection, for which it provides copy-free messaging without the need of protection context switches for the basic variant. The message mechanism is non-blocking and unidirectional, with an asymmetric relationship of the sender side and the receiver side. I refer to these two ends of a message port as the sender port and the receiver port. Message ports are 1:1 communication links; for each message

[3]As opposed to control-flow-oriented remote procedure call mechanisms.

```
System(I4Copter) {
    Domain(CopterControl) {
        SenderPort(EnginePowerSwitchSPICom) {
            ctype = "spiDeviceData_t";
            cheaderfile="System/CommonFiles/SPIDeviceData.h";
            msgcount = "2";
        }
    }

    Domain(SerialCom) {
        ReceiverPort(EnginePowerSwitchSPICom) { }
    }
}
```

Listing 5.2: Message Port Definition

port, there exists exactly one sender and one receiver port. Messages are received at the receiver port in sending order. All messages on a message port are of the same data type, and the number of concurrently live messages is statically limited for each message port.

Message ports communication links are statically defined in the system definition as shown in Figure 5.2. Each message port is defined by a system-widely unique name. For each message port, there must be one definition of a sender port and a corresponding receiver port, which are defined at the level of KESO domains or CiAO OS-Applications. The named sender and receiver ports are part of the external component interface. The `ctype` and `cheaderfile` attributes correspond to the respective shared memory attributes and define the type of message of the message port. The `msgcount` attribute defines the capacity of messages of the message port, that is, the maximum number of messages that can exist on the message port at a time.

5.3.1 Message Protocol

The use of message ports follows a predefined protocol. At the sender-port side, the following operations are provided:

- `allocate()`: Allocates a message buffer. The operation returns `null` if no more message buffers are available. On success, access to the allocated message buffer is granted to the caller and a handle to the message buffer is returned. The message buffer can now be filled with the message data to be sent.

- `send()`: Transmits a message. Messages on a sender port are transmitted in the order of allocation, that is, if the caller currently holds references to multiple message buffers of the same sender port, the buffer that was first allocated is used for the transmission. After transmitting the message, the sender must no longer access the buffer. The operation does not block. There is currently no way for the sender to check if a specific message has been received; such a primitive could be easily provided with the current implementation, however.

At the receiver-port side, the following operations are defined:

- `receive()`: Receives a message from the receiver port. The operation returns `null` if no unreceived messages are available at the receiver port. On success, the reader is granted read-only access (optionally, read-write access) to the message, and the operation returns a handle to the message.
- `release()`: Releases the oldest received message and makes the message buffer available for `allocate()` operations on the associated sender port. After releasing a message, the caller must no longer access the message buffer.
- `peek()`: Checks if one or more messages are available for reception on the receiver port, without changing the state of the receiver port.

The lifetime of a message starts with its allocation by the `allocate()` operation at the sender port, and ends with the messages destruction, which is carried out by the `release()` operation at the receiver side. The capacity of a message port bounds the number of life messages on a message port at a time.

As opposed to shared memory areas, which are concurrently accessible by multiple components, the message protocol ensures that any message is accessible by a single component only at a given time, as the sending and receiving phase do not overlap in a message's lifetime. The compliance to the message protocol is not enforced in the most basic message port variant, but available at additional cost as an option.

5.3.2 Placement and Implementation

The message port implementation is based on a simple bounded-buffer design, where the data structures of the bounded buffers are split to form the sender port and receiver port. The split is performed so that no data item of the bounded buffer needs to be written by both the sender and receiver side. Figure 5.4 shows the separation and placement of the data structures. The message pool that contains the message buffers and the position of the sender are stored with the sender; only the position of the receiver is stored with the receiver port. To distinguish the situations of a full and an empty pool, a simple trick is used to avoid the maintenance of a separate fill variable that needs to be writable by both the sender and the receiver: The index variables of the receiver and the sender count to twice the size of the pool. The pool is empty when both indexes are equal; when the pool is filled completely, the distance of the two indexes is equal to the capacity of the pool. To access messages in the pool, a modulo operation by the capacity of the pool is applied to the reader or writer index to compute a valid index into the message pool.

The data structure split allows the use of the message mechanism without the need of protection context changes in a system where the memory protection policy only restricts write accesses. Because the sender port data structures are placed in the

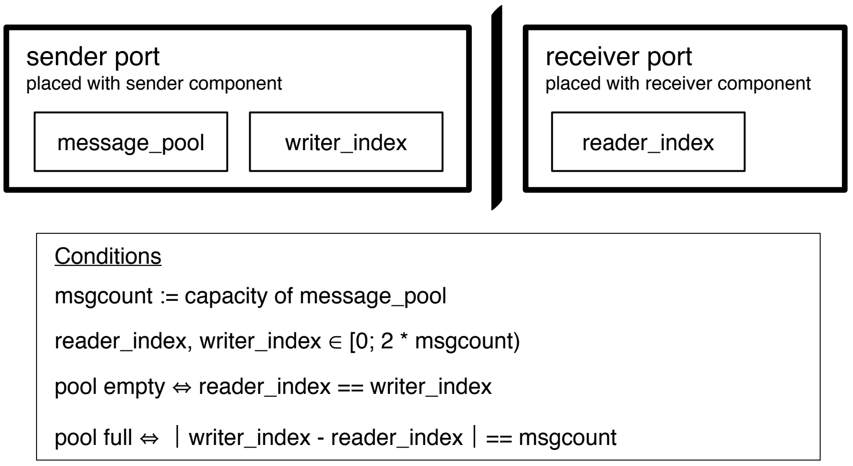

sender port
placed with sender component
message_pool
writer_index
receiver port
placed with receiver component
reader_index
Conditions
msgcount := capacity of message_pool
reader_index, writer_index ∈ [0; 2 * msgcount)
pool empty ⇔ reader_index == writer_index
pool full ⇔ | writer_index - reader_index | == msgcount

Figure 5.4: Basic Message Port Implementation and Placement

private data segment of the sender component's OS-Application, the sender has write access to all message buffers in the pool and the writer index into the buffer. To check the condition of a full buffer, the sender only needs to read the index of the reader component. Conversely, the reader component only needs write access to the reader index. Read access to the messages in the pool is possible without further action.

5.3.3 Implications of Read Protection

If the protection policy includes read protection, the `allocate()`, `receive()`, `peek()` and `release()` operations become privileged operations. The `allocate()` operation reads the buffer index of the receiver side to determine whether the message pool contains free message buffers. The `receive()` and `peek()` operations likewise read the position of the sender to determine whether the message pool contains unreceived messages. To enable the receiver component to read received messages, at least one temporary MPU region is required. The access is granted in `receive()` and revoked in the `release()` operation, which need to execute in privileged context to be able to perform these operations.

5.3.4 Variants

The basic message port variant has the restrictions that the message protocol is not enforced on the sender side, and that the receiver only gets read-only access to the memory of received messages. Both of these restrictions are for efficiency reasons, as they enable to perform all operations of the message protocol without requiring a change to the memory protection context. At the price of losing this efficiency benefit, I provide two options that can be individually enabled for each message port if desired:

Enforced Message Access Revocation on `send()` This option enforces the revocation of access to a sent message according to the global memory protection setting. If only write protection is used, the sender loses write access to the message, if read protection is additionally used, the sender also loses read access upon send. The option is realized by placing the sender port in a separate memory region that is not part of any OS-Applications memory. Access to the message is explicitly enabled by using a free temporary MPU region in the `allocate()` operation, and revoked upon `send()` by releasing the MPU region.

Write Access to Received Messages For receiver ports for that the application requires write access to the received messages, this option uses a spare temporary MPU region to enable write access to the region in the `receive()` operation. Write access is revoked in the `release()` operation by releasing the MPU region. This option does not add additional cost if read protection is enabled in the global memory protection setting.

5.3.5 Message Ports Conclusions

The presented message ports provide a very basic, but efficient way of inter-domain communication. The above protocol is suited for the periodic polling of receiver ports for new messages in a time-triggered system. The basic primitives could be extended by a user-level library that provides a blocking variant of the `receive()` operation, particularly useful to event-triggered systems. The library only needed to provide wrappers around the `receive()` and `send()` operations, which use the AUTOSAR OS `WaitEvent()` primitive to block on empty receiver ports, and the `SetEvent()` primitive in the `send()` operation to notify and wake a blocked receiver. Another conceivable extension is the provision of 1:N message ports, which could internally be mapped to N 1:1 links, hidden behind the same programming interface. For the I4Copter, the basic primitives showed to be sufficient, wherefore I have not currently implemented any extension to the basic mechanism.

5.4 Safe Java Interface

The presented interfaces for shared memory and message ports are both C++ programming interfaces. The C++ API simply provides the address to the shared memory area or the messages of a message port in the form of a typed pointer. To enable the interaction with Java components, a Java interface is required that provides Java components with safe access to shared memory areas and messages. Since these memory areas are no regular Java objects managed by the Java runtime environment, regular Java references are not suited for this purpose.

5.4.1 KESO Abstractions for Accessing Raw Memory Areas

KESO provides the abstractions of raw memory and memory-mapped objects for the purpose of accessing non-managed memory areas. Raw memory allows accessing

```
typedef struct {
    private:
        float distance_m;
        bool  reliable;

    public:

        inline bool getReliable() {
            return reliable;
        }
        // ... more getters/setters ...
} proximityData_t;
```

(a) C++ Definition

```
import keso.core.*;

final class proximityData_t
      implements MemoryMappedObject {
    private MT_FLOAT   distance_m;
    private MT_BOOLEAN reliable;

    public boolean getReliable() {
        return reliable.get();
    }
    // ... more getters/setters ...
}
```

(b) Java Definition

Figure 5.5: Example: Mapping C Data Types to KESO's Memory-Mapped Objects

a non-managed area similar to an array of primitive data. The raw memory has a base address and a size, and all accesses incorporate an offset into the area and a data type (such as 8/16/32-bit integer or float) to read from the offset. Computed offsets may be used, and accesses are subject to bound checking as are regular arrays. Memory-mapped objects, on the other hand, allow the definition of a Java class with special fields that are mapped to off-heap locations. Besides these special fields, the Java class can contain all elements of a regular Java class, including heap-allocated fields. The mapped fields of a memory-mapped object are mapped to a base address. The size of the mapped area is defined by the number and size of the individual mapped fields, and the offset of a mapped field to the base address is determined by the sizes of previously declared mapped fields in the class definition. For accesses to mapped fields, a statically known offset is used. Therefore, a bound check is only required when initially establishing the mapping, to check whether the target memory area is of sufficient size for all mapped fields, or not.

5.4.2 Using Memory-Mapped Objects to Resemble C Data Types

Memory-mapped objects are a suitable abstraction to recreate the memory layout of a C data structure in the Java code. Figure 5.5 shows the C++ original and the equivalent memory-mapped KESO class for the proximity sensor data in the I4Copter. To distinguish mapped fields from heap-allocated fields, the mapped fields are not declared using the standard primitive types but specially treated *memory types* provided by KESO's class library as a match for various primitive data types. For example, there are types for integer values of varying sizes (for example, `MT_U32` and `MT_U16`), floating point values or a type that matches the C++ `bool` type. Each of the types defines getters and setters to read or write suitable Java primitive values to mapped fields. There are also read-only variants of all memory types, which lack the modifying methods, and gap-filling types of various sizes that can be used to fill gaps in the memory layout.

While accesses to mapped fields appear to be virtual method calls at the source code level and the Java bytecode level, the virtual methods of memory types are specially treated by jino, and the fields are directly accessed without incurring a method call in the generated output. For memory-mapped objects with no heap state, jino may even be able to entirely avoid the allocation of an object instance, if it is able to statically determine the base address of the mapping for all possible uses of the object. This works particularly well if the memory-mapped object is created and used only in a local scope that it does not escape, for example within a single method.

5.4.3 Possible Issues with Mapping C Types to Mapped Objects

When creating a class with memory-mapped fields that resembles a C data type, one may stumble across some constructs that cannot directly be mapped with the currently provided feature set of KESO's memory-mapped objects. These are:

Nesting of Memory-Mapped Classes KESO does not currently support the nesting of classes with memory-mapped fields to create a new memory-mapped class. This situation occurs with C structures that contain fields that are themselves of a structure type. The `proximityData_t` type from Figure 5.5 is actually part of an aggregate type `basicSensorData_t` that subsumes the sensor data of all sensors and is used as the type of the Sensor Data shared memory area from Figure 5.1. The issue can be manually worked around by manually expanding the fields of the member classes into the aggregate class. This solution may require fields to be renamed to avoid name clashes (for example, if a nested type is included multiple times in the aggregate type, as is the case when there are multiple instances of a specific sensor type), and implies that the aggregate type needs to be kept in sync with the member types manually.

Union Types C supports untagged union types, which provide differently typed views on the same piece of memory. A common use of union types in embedded programming is types for device registers consisting of multiple sub values, where a union type can provide a convenient way to both access the register as a whole as well as accessing sub values of the register by name without requiring manual bit operations. There is currently no way to resemble union types with memory-mapped objects other than creating specialized variants for the needed combinations. In the I4Copter application, union types do not appear in the data structures used for messages and shared memory areas, and therefore this issue does not occur.

Mapped Array Fields Possibly the most severe restriction is that currently no arrays of mapped fields can be defined to match arrays in the original C data type. In the I4Copter, such arrays occur for example in the `spiDeviceData_t` type, which is used as the message type of all message ports to the SerialCom component, and which contains an array of eight bytes to hold the actual payload to transmit over the SPI bus to the slave device. An array of n elements cannot simply be mapped to n simple fields of the element type, since simple fields need to be accessed by their individual names and cannot be accessed using a computed index, for example to iterate all array elements in a loop.

5.4.4 Raw Memory as a Base Abstraction

These shortcomings of KESO's memory-mapped objects are planned to be resolved in a future release. In the current state, particularly the missing support for mapped array fields prevented me from choosing memory-mapped object as the immediate data access mechanism for the KESO interfaces to the shared memory and message port mechanisms of CiAO. Instead, I used the more generic raw memory abstraction discussed above, which makes it possible to access shared memory areas and messages using computed offsets. It is still possible to use memory-mapped objects, because raw memory areas can be used as a target area to establish a memory-mapped object. It is even possible to use memory-mapped objects to access the simple fields of a shared memory area or a message, and to use the raw memory abstraction only to access those parts of the area which require computed offsets.

5.5 Port of the I4Copter Application

I practically tested the feasibility of the soft-migration approach at the example of the I4Copter application, which I ported component by component to Java. The C++ and Java implementations of the different components can be arbitrarily combined and produce a functioning system. The component granularity showed to be a significant facilitation for porting a larger piece of software such as the I4Copter application. Firstly, a newly ported component can be combined and tested with the existing components that are known to work. This reduces the scope of possible errors and aids debugging. Secondly, it allows the early evaluation of the ported component to determine the quantitative impact of the change of the programming language on the component.

5.6 Chapter Summary

In this chapter, I presented an extension of my framework that supports the soft migration of a C or C++ application to Java at the migration granularity of software components. I chose software components as the migration unit for they have been a well-established technique in software development for many years, and their properties of encapsulation and replaceability make them a well-suited migration unit. Both CiAO and KESO can be considered component systems, in which the (non-)trusted functions in CiAO and the portal mechanism in KESO represent the external component interfaces. Instead of building my soft-migration approach on these control-flow-oriented communication mechanisms, which are little spread in the domain of deeply embedded systems, I extended CiAO by the more commonly used communication idioms of shared memory and message-based communication.

The shared memory and message mechanism I designed are optimized for a protection policy that only includes hardware-based write protection, which is the needed minimum to provide fault containment. This basic protection policy can be well supported by a message mechanism without strict enforcement of the protocol and shared memory with a single writer, without requiring the use of additional MPU regions. I also discussed the implications of more comprehensive memory protection policies (read and execute protection) for shared memory and message ports and more general use scenarios (shared memory with multiple writers, enforced message protocol) with respect to the use of the available MPU regions. The conclusions are that such extensions of the basic protection quickly lead to MPU region requirements that exceed the typically available number of regions; for the I4Copter application on the Tricore TC1796 microcontroller, I showed that shared memory regions cannot be made available by a static MPU configuration in the presence of read protection. It is therefore feasible to utilize MPU protection at the basic level needed to provide fault containment; more comprehensive – but not safety-critical – coverage such as read protection can be well provided by software-based protection.

6 Quantitative Evaluation

The framework presented in the previous chapters fulfills the first three goals set in Section 1.2: It provides fine-grained configurability for both hardware- and software-based memory protection, allows different protection realms of the application software to be established by different protection mechanisms, and, with the extension from Chapter 5, provides a soft-migration path for existing code at a manageable granularity. In this chapter, I address the fourth goal, the easy quantitative evaluation of the imposed protection cost for a given application. In addition, I also evaluate different aspects of the framework itself. The questions that the evaluation in this chapter addresses are:

- What is the overhead of using Java instead of C or C++ in static embedded applications? (Section 6.4)
- What is the individual cost of the basic primitives and operations needed to enforce memory protection? (Section 6.5)
- Does the framework fulfill the goal of supporting the easy quantitative comparison of hardware- versus software-based memory protection for a given application? (Section 6.6)

6.1 Test Setup

Before I address these questions, I describe the test setup, the applications used for the evaluation, and the metrics and method of measurement.

6.1.1 Evaluation Platform

The target platform for all tests is the Tricore TC1796 microcontroller. The microcontroller is equipped with 48 KiB program scratchpad memory and two MiB of program flash as code memories, and 56 KiB local data RAM and 64 KiB of internal SRAM as data memories. My evaluation board is an Infineon Triboard TC1796, which is equipped with one MiB of external SRAM.

For my macrobenchmarks, the CPU is run at the maximum possible clock of 150 MHz, with a 75 MHz system clock. For the microbenchmarks, I use a slower CPU clock of 50 MHz, which allows running the system clock at the same frequency as the CPU and enables more precise runtime measurements using the system timer. Except for CD_x, all benchmarks are executed from the internal first level memories. This minimizes the impact of memory access latencies on the measurements. CD_x has memory requirements that exceed the internal memories and is executed from the external memory. The 16 KiB instruction cache of the TC1796 is enabled in all my measurements.

6.1.2 Used Compilers and Tools

The following tools and options are used in the I4Copter experiments and microbenchmarks:

- CiAO, subversion revision 1666 (2012-07-01)
- KESO, subversion revision 2853 (2012-07-01), with the following additional compiler options:
  - `omit_fields`: Removes unused static fields to reduce the data memory consumption.
  - `production`: Removes verbose exception messages to reduce the footprint.
- AspectC++ 1.1, AG++ 0.8 (2012-03-02)
- Hightec Tricore toolchain 3.4.6 (based on GCC 3.4.5)

The used test applications are available in the CiAO (C++ variants) and KESO (Java variants) repositories. Except where otherwise noted above, the compiler options used are those in the build files found with the applications in the above repositories. The CD_x measurements are based on an earlier version and taken from a published paper [113].

6.2 Test Applications

The main application used in my evaluation is the already introduced control software of the I4Copter. It is an example of a hard real-time application that does not require dynamic memory allocation (and therefore garbage collection) and uses few of Java's high-level language features. For my comparison of C and Java, I complement the I4Copter application with the Collision Detector [61] Java benchmark, which does dynamic memory allocation and utilizes more of Java's features, such as generic collection types.

6.2.1 I4Copter

For the I4Copter, the original C++ version and my Java port are used. The Java port is very closely oriented at the original code to ensure comparability. Two examples for C constructs that could not directly be mapped to Java code are stack-allocated objects and template classes. Stack allocation has been substituted for heap allocation in my Java port; to avoid the need for a garbage collector, all heap allocations are performed in the initialization phase; references to the objects are retained throughout the entire runtime of the program. Templates are widely used in the I4Copter for the static configuration, for example in hardware driver classes. Since most of these templates take parameters of primitive types, they cannot be mapped to Java generics. Instead, I created classes that contain the template parameters as regular object fields. Consequently, where multiple parameter sets for a template class are used in the I4Copter, a generic version for all parameter sets exists in my Java port, whereas in the C++ variant a specialized variant is generated for each parameter set. When the template is only instantiated with a single set of parameters, the compiler optimizations in jino will create a similarly specialized variant for this parameter set.

6.2.1.1 Reproducible Test Runs

The code paths executed in the different subsystems depend on the operating mode of the I4Copter. The operating mode is globally provided by the Coptercontrol subsystem, although the Flightcontrol and Signalprocessing subsystems influence the decision by suggesting changes of the operating mode. To allow the comparison of execution times of different variants, it is therefore required that all variants execute with the same input data.

Since operational mode changes are mostly triggered by input data, such as sensor values exceeding a certain threshold or specific remote control commands entering the system, running two variants in sequence will never result in the same internal application flow. To work around this issue, I created standalone variants of the four most interesting components (Coptercontrol, Flightcontrol, SerialCom and Signalprocessing), which can be provided with sensor and steering data logged during a flight. These standalone variants can be executed multiple times with the same recorded input data, resulting in identical application flows.

I instrumented the drivers for the analog-to-digital converters and the radio remote control. The values are logged and replayed directly at the source where read from the hardware devices. The remainder of the measured code remains unaffected. Input read from shared memory areas or received from message ports is also logged and replayed. Each standalone setting contains the regular main task, with instrumented hardware drivers as described above, and a second task that updates the input data for the main task by updating the shared memory areas or sending the appropriate messages. The two tasks run alternatingly, so that after each job of the measured task the input data is updated with the next set of logged data. The logged data and measured times are placed in the external RAM. The measured times are downloaded from the target using the debug connection after the benchmark has been fully executed.

6.2.1.2 Evaluated Application Parts

For the static evaluation of the memory footprint, I use the original and uninstrumented images of the software. For the execution time measurement, I only evaluate the runtimes of the Coptercontrol, SerialCom, Signalprocessing and Flightcontrol subsystems. The Ethernet subsystem is mostly concerned with copying data to the network protocol stack. The evaluated parts are all that is needed to operate the aircraft. The evaluated parts perform the following operations:

Coptercontrol consists of two main parts, a state machine that manages the global operation mode of the aircraft, and the driver for the radio remote control. The task is activated periodically each 21 milliseconds. It reads the frames received from the radio remote control since the last execution, sensor values such as the battery voltage, and the possible operational mode suggestions from the Flightcontrol and Signalprocessing components. The state machine that determines the operational mode is then executed and may change the operational mode based on the input data. The state machine includes emergency mode management that handles situations such as the loss of connection to the remote control or low battery voltage.

Flightcontrol consists mainly of the flight attitude controller that computes the engine thrust levels with a period of nine milliseconds. An execution of the Flightcontrol task consists of three phases. In the first one, the input data for the controller (steering commands and sensor values) are read from the corresponding shared memory areas and copied into the controller. In the second phase, the controller is executed. In the third phase, the new thrust levels are sent to the engine controllers as messages to the SerialCom subsystem.

Signalprocessing collects and preprocesses the values from the sensors with a period of three milliseconds. The sensor values are published to other components by copying the values to shared memory areas. In addition, the Signalprocessing subsystem contains a state machine that manages a local mode, based on the state of the different sensor drivers. This mode is the basis for suggested global operational mode transitions. This state machine mainly covers the initialization and calibration phases of the sensors and ensures that no flight mode is entered before all sensors have finished initialization and calibration.

SerialCom collects messages, writes the content to the SPI bus, and sends the answer of the SPI slave device back to the sender of the original message. The interaction with the SPI bus contains some active wait loops and is otherwise dependent on the slave device. For my standalone benchmark, I removed the SPI bus portions from the component. What remains is the code that sends and receives messages. The standalone variant can therefore also be considered a benchmark of the message port performance.

6.2.2 Collision Detector Benchmark Family

As an example of a more fully-fledged Java application I use Collision Detector (CD$_x$) [61], an open-source benchmark family that is available in a C (CD$_c$) and a Java (CD$_j$) version with almost equivalent algorithmic behavior. This benchmark complements the I4Copter with its quite diverse code characteristics and more dynamic nature. It is one of few available benchmarks for real-time Java that also comes in a C variant for comparison. The benchmark has gained some acceptance in the real-time Java community [89, 91, 93, 46].

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 two-dimensional space ignoring the z coordinate (altitude) to reduce the complexity for the second stage (detector phase), in which a full three-dimensional collision detection is performed (*detected collisions*). A detailed description of the benchmark is available in [61]. Since CD$_j$ allocates temporary objects and uses collection classes of the Java library, it requires the use of a garbage collector in KESO.

6.2.2.1 Benchmark Version

I use version 1.2 of the CD$_j$ benchmark and a bug-fixed version of the CD$_c$ version 1.2 benchmark. The modifications that I made to CD$_c$ are the following:

- Use the same hash function for two-dimensional vectors as CD$_j$
- Remove duplicate collisions in the detector's result set as CD$_j$ does
- Fixed a bug in the hash map implementation where a hash key collision was not detected correctly while adding a new element to the map and the existing element was consequently replaced in the map

With my changes, both CD$_j$ and CD$_c$ compute the same amount of suspected and detected collisions. This is not the case with the released version of the benchmark. Since the runtime of the different iterations is heavily influenced by the number of suspected and detected collisions, the execution times of CD$_c$ and CD$_j$ are not comparable without the above modifications.

6.2.2.2 Used CD$_x$ configuration

The CD$_x$ benchmark supports different configurations for different runtime environments. The radar frames may be generated online, synchronously or concurrently to the detector, by a separate air traffic simulator task. For avoiding any dependencies between the detector and the air traffic simulator tasks, the radar frames can also be pre-simulated and stored in a buffer of sufficient size. For the Tricore TC1796 platform, I can neither use an online simulation of the air traffic using the air traffic simulator nor pre-generate the frames due to memory constraints. Instead, I use a simplified online radar frame computation that is included in the C version of the benchmark. This configuration is labeled *on-the-go frame generation* in the benchmark. I transferred this frame computation to CD$_j$.

My benchmark configuration uses six airplanes. I run the collision detector on 10,000 radar frames (that is, 10,000 iterations). The collision detector task is periodically released with a period of 40 milliseconds. CD$_j$ contains two additional tasks, which mainly perform initialization work and output the benchmarks results upon completion of the benchmark. Both of these tasks are blocked during the entire execution of the benchmark. The garbage collector task is configured with the lowest priority in the system and uses the slack time for garbage collection that the detector task leaves in each period. I run the benchmark with both garbage collection strategies in KESO, and without a garbage collector to get an impression of the allocation overhead when using a garbage collector compared to the fast bump-pointer allocation provided by the pseudo-static allocator.

6.3 Metrics and Method of Measurement

In this section, I present the metrics used to assess the cost of a test candidate, and how I determine these metrics. Some metrics are determined from the static program image; others are dynamically determined during the execution of the candidate. For the latter, the candidates need to be modified to contain additional code that carries out the measurement. This modification naturally affects the results of the measurement. I instrument the application code so that this impact is kept at a minimum. Normally, the measured code sections are only modified to contain a short code fragment in the beginning and at the end that takes a snapshot of the system time. For statically determined metrics, I always use a program image that does not contain the modifications needed for the runtime measurements.

6.3.1 Statically Determined Metrics

The following metrics are determined statically from a program image, using the `nm`, `size` and `objdump` utilities of the GNU binutils tool collection.

- Code size: Size of the `text` section, which contains all executable code.

- Initialized data, except if initialized to zero: Size of the `data` section, which contains all statically allocated data items that are initialized with values other than zero.
- Uninitialized data or data initialized with zero: Size of the `bss` section, which contains statically allocated data items that are initialized to zero. At the programming-language level, static data items that are not initialized also end up in this section.

6.3.2 Runtime Measurements

At runtime, I determine the execution times for portions of the code and the maximum amount of dynamically allocated memory:

- Execution time: I determine the runtimes of the measured code sections by reading the value of the Tricore's free-running system timer immediately before and after the measured code section, and compute the difference. In my setup, the system timer counts at the maximum possible frequency of 75 MHz. The timer value is stored in a 56-bit register. I only use the lower 32 bits of the register to determine the value using a single read access. At 75 MHz, these 32 bits are capable of capturing intervals of up to approximately 57 seconds, which is sufficient for all measured code sections. Since the system timer only counts at half of the CPU frequency, an inaccuracy of one cycle exists. This inaccuracy is insignificant for all my measurements and within the measurement precision.
- Heap usage (CD_x only): For the CD_x benchmark, I determine the amount of heap used during an iteration, by reading the amount of free heap memory before and after the measured code section. Since garbage collection in KESO is only performed during slack time, no heap memory is reclaimed during the measurement and therefore the difference of the taken values is the amount of memory allocated from the heap during the measurement. The I4Copter only allocates heap objects during the initialization phase; therefore this metric is not of interest. The heap is included in the uninitialized `data` section and thereby accounted for.

6.4 Cost of Using Java Instead of C or C++

In this section, I determine the overhead caused by using Java as a programming language in place of C++, without assessing the cost for memory protection. The cost for memory protection is separately evaluated in Section 6.6 for the I4Copter, using only the Java variant without the impact of implementation differences and differences in programming languages. Hardware-based memory protection is enabled (control-flow isolation, write-only protection) in all measurements within this section.

6.4.1 I4Copter

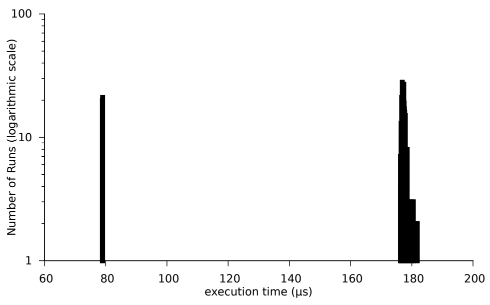
6.4.1.1 Execution Time

The results of the execution time comparison are shown in Figure 6.1 (Flightcontrol), Figure 6.2 (Coptercontrol), Figure 6.3 (Signalprocessing) and Figure 6.4 (SerialCom). Each figure contains histograms of the execution times of the C++ and the unsafe Java variant, a line diagram that shows, for an excerpt of 100 iterations, how the execution of the C++ and the Java code correlates, and a table with the worst observed execution time, the median of the measured execution times, and the relative overhead to the C++ version. In this section, I focus on the comparison of the unsafe Java version with the C++ version, as these differences reflect the programming language differences. I postpone the discussion of the overhead caused by runtime checks to Section 6.6.

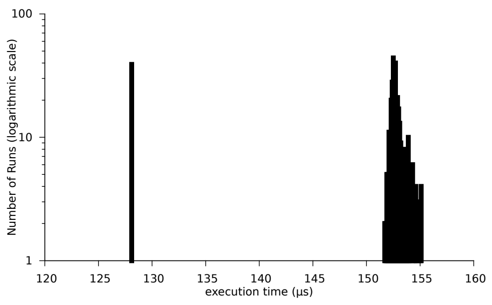
A commonality that can be observed for all components is the strong correlation of the execution times of the individual iterations. The execution times of the different jobs are almost constant in all components. In the Coptercontrol component, there are some peaks in the graph, which are caused by state transitions of the internal state machine. Because all variants process the same input data, these peaks occur in the same iteration for each variant. In the Flightcontrol component, the execution time variances of the Java variant do not quite match those of the C++ variant. The reason here is that the Java variant uses the C math library only for expensive operations (for example, the computation of the arc sine), but uses KESO's internal math functions for simpler operations (for example, floor or sine). The runtime of these operations partly depends on the input values, and the algorithmic behavior differs from the versions in the C library, wherefore the variations in the execution time do not correlate. Considering the total execution time of the controller, however, these variations are minor. The strong correlation shows that the use of Java does not introduce indeterministic behavior. With the exception of the garbage collectors, allocation operations from garbage-collected heaps and portal calls that require the cloning of parts of the object graph, none of which are used in the I4Copter, all of KESO's internal operations are of constant complexity.

Flightcontrol A job of the Flightcontrol task can be divided into three phases: In the input phase, the input data (steering commands and sensor data) are copied from shared memory areas to the controller. In the controller phase, the controller is executed. Finally, the new thrust levels determined by the controller are actuated to the engine controllers by sending messages to the SerialCom component.

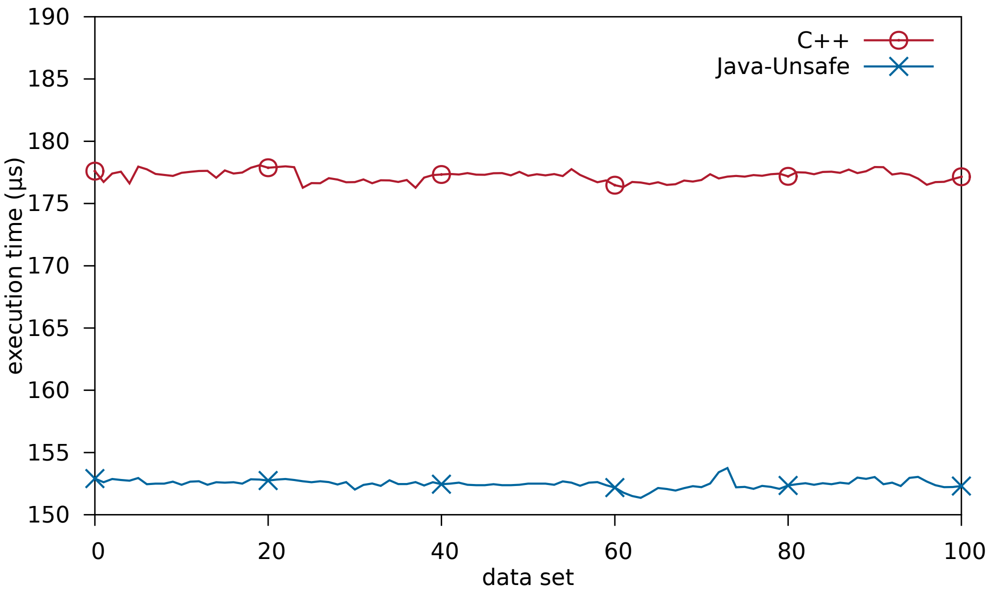
In the input and output phases, the Java version needs approximately 1.5x (input, 0.85 versus 1.25 microseconds) and 2x (output, 6.5 versus 13.6 microseconds) the time compared to the C++ version. This is due to the use of the Java compatibility interface to the shared memory and message mechanisms, which are more expensive in use than their native C++ counterparts. I discuss this in more detail below. The execution time of the Flightcontrol task is, however, dominated by the controller phase,



(a) C++



(b) Java (unsafe)



(c) Correlation of 100 Executions

	Worst	Median
C++	183.9 µs	177.3 µs
Java-Unsafe	156.5 µs (-14.9 %)	152.4 µs (-14.1 %)

(d) Execution Times (% overhead to C++)

Figure 6.1: Flightcontrol: Execution Time Comparison C++ Versus Java

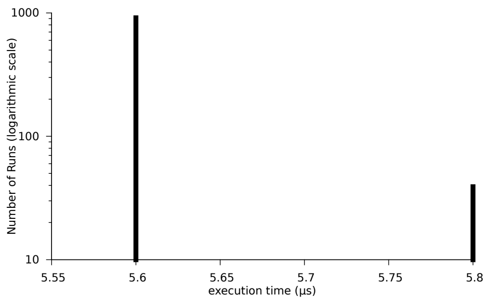
wherefore these differences are insignificant in the total execution time of the task. In the controller phase, the Java port outperforms the original variant by 14.1 %. The major reason for the faster execution is the more efficient implementations of simple operations of the math library in the KESO class library than their counterparts in the C library, which above were already identified as being responsible for the execution times of the Java and C++ versions not strictly correlating. KESO can be configured to use the C math library for all operations in the `Math` class, in which case the controller shows a very similar behavior to the C++ version.

As a side note, the original variant of the Java controller outperformed the C++ controller by approximately 30 %. Investigating the differences, I found that the C++ version used the double precision variants for some math operations instead of the single precision variants (`fmod()` and `fabs()` instead of `fmodf()` and `fabsf()`), although the controller operates in single precision. Besides the more expensive operations, this also caused the conversion of the parameters to double precision and the conversion of the return value to single precision. In the Java version, the single precision parameter caused the automatic selection of the single precision variants. I changed my Java port to explicitly cast the input values at the affected operations to double precision to show the same behavior.

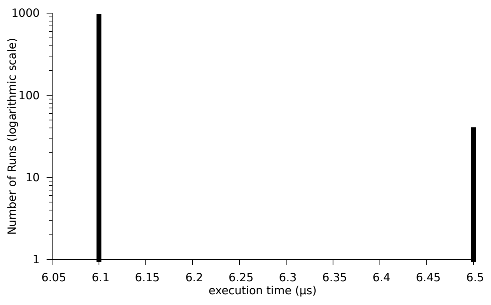
Coptercontrol The Coptercontrol task processes incoming commands from the radio control and runs through a simple state machine. Depending on the reached state, the task activates the beeper or changes the state of the four signal LEDs to give audible or visual feedback to the user. The regular peaks in the processing graph represent jobs, in which the state of the LEDs was changed. In flight mode, which I used in the simulation, the I4Copter shows an LED blink pattern, which is discernible in the regularly recurring peaks in the execution time graph.

The Java variant is 8.6 % slower in the median than the C++ variant. Considering the absolute difference of 0.5 microseconds, the effect is minor. The causes are some missing opportunities for inlining in the Java code of a method used to calculate the port and pin number from a GPIO pin number. The method is called with constant pin numbers and the computation could be carried out at compile time, which is what happens in the C++ version. Most of the shared memory accesses in the Coptercontrol are optimized by jino to be as efficient as their C++ counterparts. Other minor factors are the more efficient representation of aggregate data structures in C++, which are represented as pointer-linked separate objects in Java that require some additional instructions to access nested objects.

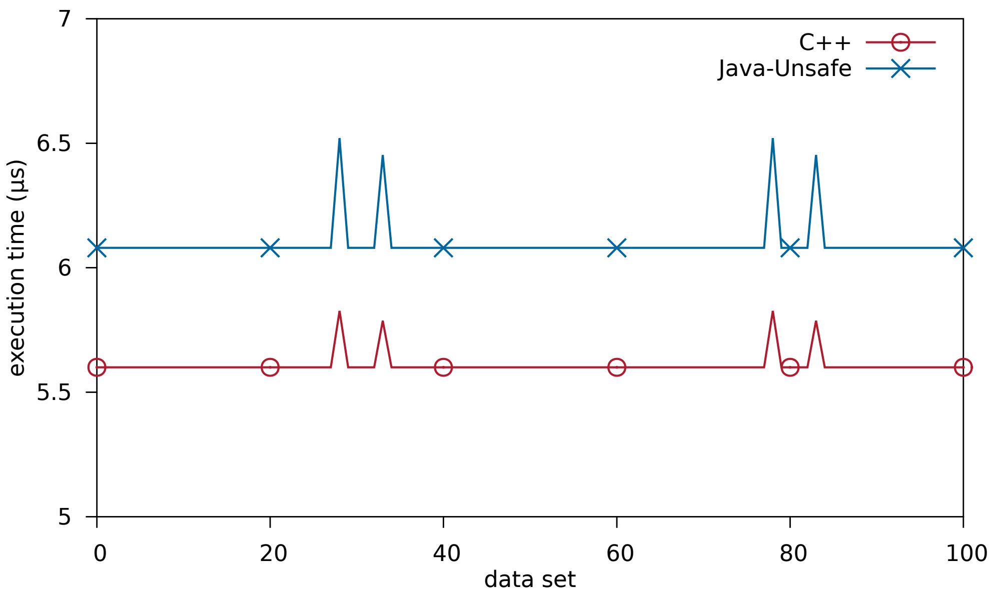
Signalprocessing The Signalprocessing task periodically activates the device drivers for the different sensors, performs simple preprocessing and filtering and makes the current sensor values available to the other components in shared memory areas. In my standalone component, only the three gyroscope drivers are enabled. In the used version of the I4Copter software, these are the only sensor values that influence the flight attitude controller. The Signalprocessing component also contains an internal



(a) C++



(b) Java (unsafe)

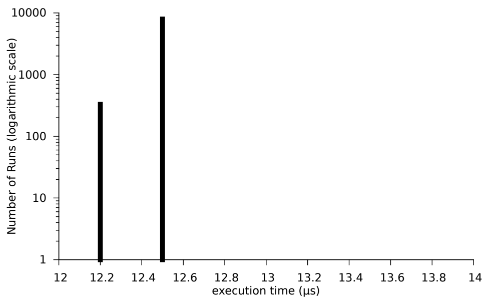


(c) Correlation of 100 Executions

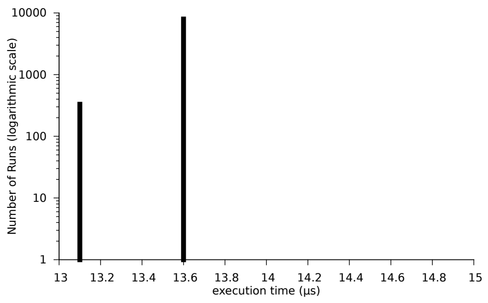
	Worst	Median
C++	5.8 µs	5.6 µs
Java-Unsafe	6.5 µs (11.9 %)	6.1 µs (8.6 %)

(d) Execution Times (% overhead to C++)

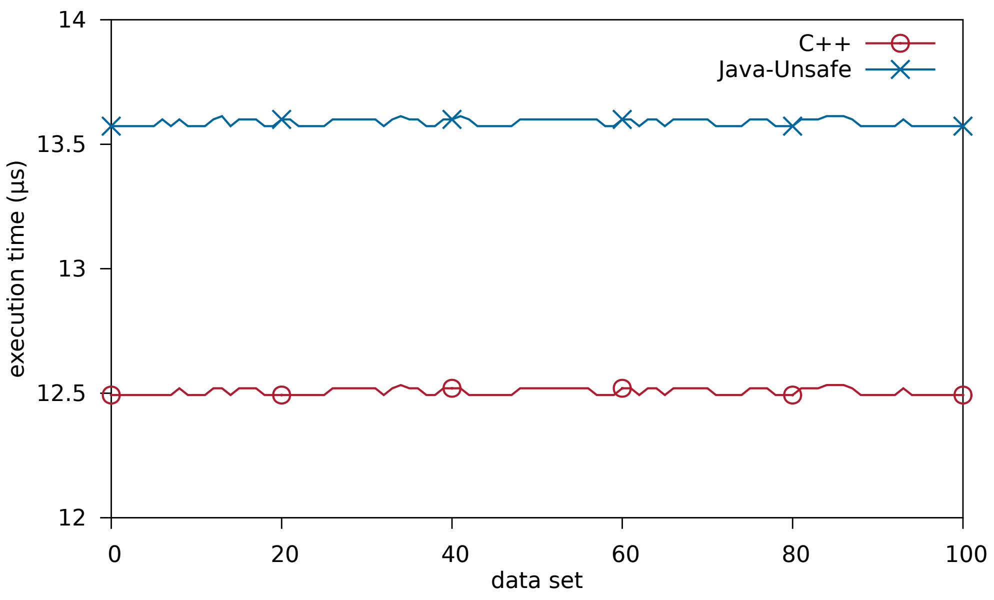
Figure 6.2: Coptercontrol: Execution Time Comparison C++ Versus Java



(a) C++



(b) Java (unsafe)



(c) Correlation of 100 Executions

	Worst	Median
C++	13.9 µs	12.5 µs
Java-Unsafe	14.9 µs (7.0 %)	13.6 µs (8.8 %)

(d) Execution Times (% overhead to C++)

Figure 6.3: Signalprocessing: Execution Time Comparison C++ Versus Java

mode state machine, but it is only relevant during the initialization and calibration phases of the I4Copter.

The Java variant of the Signalprocessing component with 8.8 % shows the highest overhead in comparison to the original C++ variant of all measured components. The cause of this overhead is the use of a generic driver class in the Java variant for all analog-to-digital converter channels, where the C++ version uses template specialization for the static configuration of the different device drivers. Particularly the base driver for the analog-to-digital converters is a code hotspot. In the specialized variant, some computations that depend on the template parameters can be performed at compile time, which are computed at runtime in the generic Java code. Because these additional computations are needed for every used analog-to-digital converter channel, the runtime computations cause a notable overhead compared to the C++ version.

SerialCom The SerialCom component is mostly sending and receiving messages. The message primitives dominate the execution time. The Java version is 3.8 % slower than the C++ variant due to the overhead of the Java API to the message port mechanism.

6.4.1.2 Memory Footprint

Table 6.1 shows the footprints of the three variants for the complete system. All three images are standard configurations of the I4Copter software without any modifications. The `bss` section includes the stacks of the different application tasks. The C++ and Java variants are configured to use the same stack sizes for the various tasks.

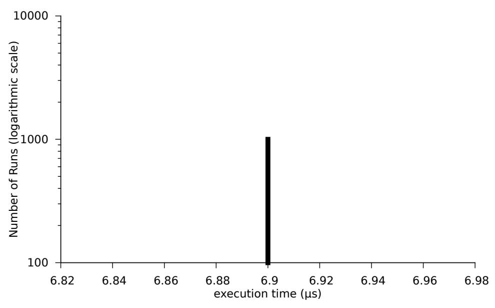
All sections are further categorized into the following categories:

CiAO Code and data belonging to the CiAO operating system. These data items are identified based on the namespaces used by the operating system (`hw`, `os` and `ipstack`).

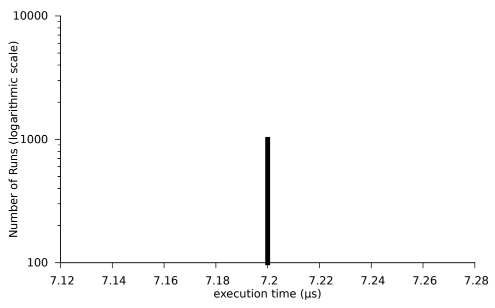
KESO Code and data that can be attributed to KESO's runtime environment, such as the memory management routines and the dispatch table.

By OS-Application The code and data that can be identified as private items only used by a single OS-Application. The information that is also available to the memory protection subsystem is used for the categorization. For the C++ application, the information is solely based on the information provided in the system configuration file. For the Java applications, the output of the reachability analysis is used. Statically allocated objects of the KESO runtime (for example, proxy objects to shared memory areas or message ports) are also assigned to the respective OS-Application that uses the objects.

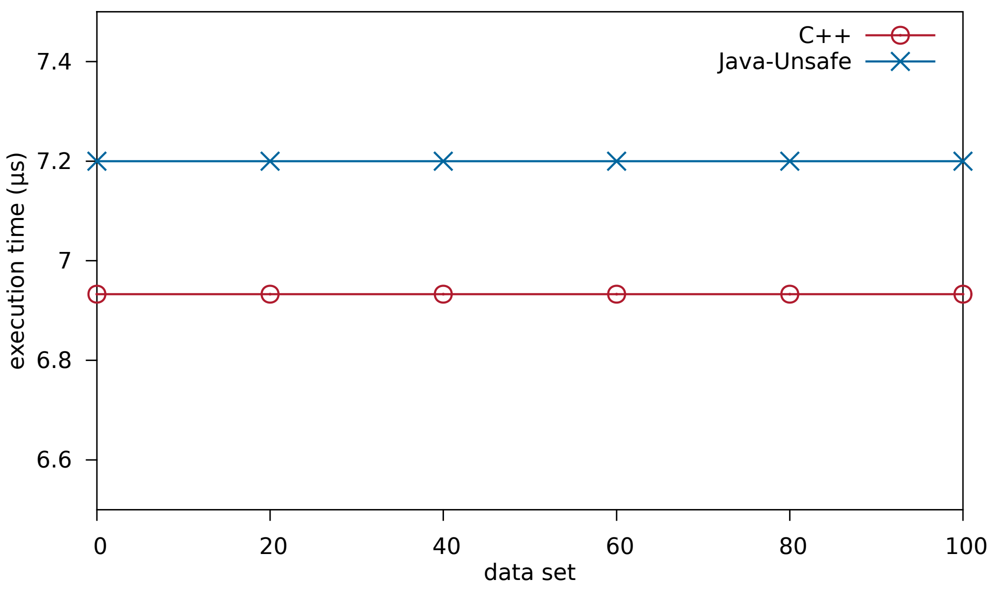
Shared Application items that are identified to be used by multiple applications. For the C++ application, only application hook routines specified in the configu-



(a) C++



(b) Java (unsafe)



(c) Correlation of 100 Executions

	Worst	Median
C++	6.9 µs	6.9 µs
Java-Unsafe	7.2 µs (3.8 %)	7.2 µs (3.8 %)

(d) Execution Times (% overhead to C++)

Figure 6.4: SerialCom: Execution Time Comparison C++ Versus Java

	bss	data	text
CiAO	2047 (13.9 %)	537 (5.7 %)	16218 (20.5 %)
CopterControl	850 (5.8 %)	2144 (22.8 %)	332 (0.4 %)
Ethernet	5472 (37.2 %)	24 (0.3 %)	362 (0.5 %)
FlightControl	1714 (11.6 %)	2138 (22.8 %)	320 (0.4 %)
InitTask	128 (0.9 %)	0 (0.0 %)	118 (0.1 %)
SerialCom	520 (3.5 %)	360 (3.8 %)	854 (1.1 %)
SignalProcessing	2962 (20.1 %)	2320 (24.7 %)	290 (0.4 %)
Shared	0 (0.0 %)	0 (0.0 %)	294 (0.4 %)
Other	1036 (7.0 %)	1873 (19.9 %)	60178 (76.2 %)
Total	14729	9396	78966

(a) C++

	bss	data	text
CiAO	2047 (13.0 %)	549 (5.2 %)	16282 (26.4 %)
KESO	44 (0.3 %)	320 (3.0 %)	298 (0.5 %)
CopterControl	1305 (8.3 %)	2436 (23.0 %)	5690 (9.2 %)
Ethernet	5648 (35.8 %)	194 (1.8 %)	2166 (3.5 %)
FlightControl	1925 (12.2 %)	2388 (22.5 %)	5452 (8.9 %)
InitTask	128 (0.8 %)	28 (0.3 %)	118 (0.2 %)
SerialCom	640 (4.1 %)	728 (6.9 %)	2862 (4.6 %)
SignalProcessing	4017 (25.5 %)	2696 (25.5 %)	10870 (17.7 %)
Shared	0 (0.0 %)	84 (0.8 %)	4240 (6.9 %)
Other	12 (0.1 %)	1169 (11.0 %)	13596 (22.1 %)
Total	15766	10592	61574

(b) Java (unsafe)

Table 6.1: I4Copter Memory Footprint: C++ Versus Java (sizes in bytes, % of the total section)

ration are attributed to this section. For Java applications, the results of the reachability analysis are used.

Other Everything left that does not fall into any of the above categories, for example functions of the C library or platform support of the compiler. The items in this category are treated in the same way as shared items by the system. I differentiate them here to show what could actually be identified as really shared data items, and what could just not be automatically assigned. No functions from a Java application are ever assigned to this category.

The size of CiAO is dominated by the network protocol stack. It can be seen that the runtime environment of KESO is very low in code and data memory requirements. Since the Java port does not use garbage collection, which is the most complex part of KESO's runtime environment, it only contains a very simple allocator. The data memory used by KESO is dominated by the dispatch table (316 bytes) and an index that maps CiAO task identifiers to Java `Thread` objects (41 bytes). The class store that contains runtime type information was completely dropped by the linker because there exist no references in the code. The code of the runtime environment comprises mainly the allocators and exception handling routines.

Looking at the assignment of the application code and data to the private memories of the different OS-Applications, it is obvious that the manual code partitioning approach on the granularity of classes used by CiAO is too coarse-grained and elaborate for the developer. The original C++ variant of the I4Copter is almost entirely considered shared code, except for the entry functions of the different control flows. With the analyses-based approach in KESO, on the other hand, all but 6.9 % of the application code could be uniquely identified as being used by a single OS-Application only. For the application data, only 0.4 % of the application items need to be considered shared data. This is, of course, a direct consequence of the multi-JVM architecture in KESO, where all data is strictly separated. The shared items represent constant, statically allocated objects, which are shared by multiple domains – a controlled exception to the rule of the absence of shared references in different domains that KESO makes for select objects such as string constants. With these precise identification of private code and data items, read and execution protection are possible with high effectiveness.

The `data` and `bss` sections of the Java version are 9.3 % larger than those of the C version. This is partially caused because the `bss` section includes the heaps of the respective domain, each of which exceeds the actual memory requirements by a threshold of 100–200 bytes. In addition, Java objects are slightly larger than C++ objects, because they carry runtime type information (4 bytes per regular object, 8 bytes per array on the Tricore architecture). Other reasons for additional memory requirements are the separate instances of static fields in each domain (80 bytes per domain) and the less efficient representation of aggregate complex types as multiple linked objects.

In the code size, the unsafe Java version is 22 % smaller than the C++ version. This can almost completely be ascribed to the static configuration using templates, which

Driver	Java	C++
Analog-to-Digital Converter Base Driver	798	0 (inlined)
Type 1 Gyroscope Driver	370	1130
Type 2 Gyroscope Driver (Variant 1)	520	2018
Type 2 Gyroscope Driver (Variant 2)	n/a	2008
Gyroscope Aggregation Driver	736	692
Total	2424	5848

Table 6.2: Specialization Versus Generalization: Gyroscope Driver in the I4Copter (sizes in bytes)

causes a separate variant of all template methods for each parameter combination. In the execution time of the Signalprocessing task, this specialization showed to be beneficial for the execution time; the increased code size is the downside of this specialization.

6.4.1.3 Discussion of the Major Factors of Diversity

In the comparison of the C++ and the unsafe Java variant, I identified the following aspects to cause the main differences in the resource requirements, which I now discuss in more detail.

Static Configuration via Templates In the Signalprocessing task, it was observable that the Java variant performed slower, but also was considerably smaller in code size. The reason for this is the broad use of C++ templates for the static configuration of device drivers throughout the I4Copter, resulting in specialized variants of the respective base abstraction classes.

An example in the codebase is the gyroscope driver. The I4Copter is equipped with three gyroscopes of two types. In total, these gyroscopes are connected to eight analog-to-digital converter channels of the microcontroller, each of which is driven by a specialized driver variant. Table 6.2 shows the code size for the generated variants. In total, eight specialized variants of the analog-to-digital converter base driver and two variants of the type 2 sensor drivers exist in the C++ version. The code of the base driver is inlined into the gyroscope drivers. It is discernible that the specialized variants of the base driver are smaller than the generic variant in the Java port – otherwise, the gyroscope drivers would be significantly larger in size. Still, the driver code for the gyroscopes in the C++ variant is 2.4x the size of the generic Java variant. This scheme is found for all driver code in the I4Copter.

The specialized variants are smaller and more efficient than the generic variant because the code can be simplified based on the actual values of the template parameters at compile time. In the analog-to-digital converter base driver, for example, the IO port and pin for the channel are computed from the channel number. This channel number is constant in the specialized variants, which causes the computation to be carried out at compile time.

```java
static final int OFS_MYFIELD = 12;

static int accessSHM() {
    RawMemory shm = SharedMemoryService.getSharedMemoryByName("MySHMArea");
    int value = shm.get32(OFS_MYFIELD);
    return value;
}
```

Listing 6.1: Shared Memory Access (Java Code)

```c
jint accessSHM() {
    // the null check was eliminated by static analyses
    KESO_CHECK_BOUNDS(sizeof(basicFlightCtrlData_t), 15);
    jint value = *(volatile jint *) ((char *) &MySHMArea_SharedMemory + 12);
    return value;
}
```

Listing 6.2: Shared Memory Access (Generated C Code)

Compatibility Interface to Access Shared Memory and Message Ports The second aspect where the Java version showed notable overhead over the C++ version is the Java interface to CiAO's shared memory and message port mechanisms, which is based on `RawMemory` objects. In the C++ version, shared memory areas are directly accessed, whereas `RawMemory` introduces an indirection, `null` checks and bound checks.

To optimize accesses to shared memory areas and message ports, jino attaches the name of the shared memory area or the message port to the data flow information when a `RawMemory` object is initially retrieved from these interfaces. Listing 6.1 shows what such a code snippet could look like in the Java code for the access to a shared memory area. Raw memory areas are often accessed using a constant offset, 12 in the example. Listing 6.2 shows the code that jino generates for this access in the optimal case, with the following optimizations applied:

- The `null` check required on the `RawMemory` object was eliminated. This is possible if the call to the `getSharedMemoryByName()` service is reachable only from domains that are allowed to access the shared memory area specified in the parameter string[1]. In this case, `getSharedMemoryByName()` is known to never return `null`.
- Jino cannot directly eliminate the bound check, because it does not know the size of the shared memory area. It can, however, aid an optimizing C compiler in removing the check, if the C data type of the area is known for the object: Instead of reading the size of the shared memory area from the `RawMemory` object, the `sizeof` operator is used to insert the area size as a compile time constant into the check. If the offset is also known, the C compiler removes the check.

[1]This string always needs to be a compile time constant, it does not exist at runtime.

- Finally, if the exact shared memory area is known, the generated code accesses the shared memory directly, avoiding the indirection over the `RawMemory` object.

When shared memory areas are used in a local scope as in the example, the access to shared memory is no more expensive than in a native C application. If the `RawMemory` object escapes the local scope, the optimizations that can still be applied depend on where the object is stored. In many cases, the optimizations can still be applied in this case. For messages, only the `sizeof` trick is used. Allocating or receiving a message can always return a `null` reference. While the indirection over the memory object could be avoided by combining the `RawMemory` object and the message buffer in a single structure, allowing to access the message buffer relative to the reference to the `RawMemory` object, this is not currently implemented.

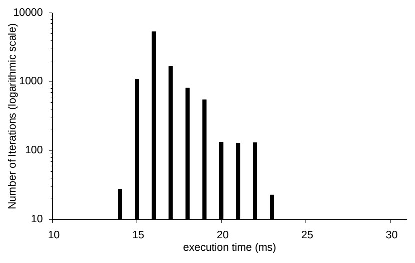
6.4.2 Collision Detector

The second example application is CD$_x$, an embedded real-time Java benchmark, which is more dynamic than the I4Copter. The shown numbers for the Java variants include runtime safety checks.

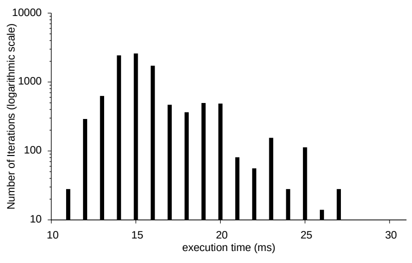
6.4.2.1 Execution Times

Figure 6.5 shows the distribution of execution times over the 10,000 iterations of the collision detector for CD$_c$ and the two garbage-collected KESO variants. CD$_j$ is 6 % faster than CD$_c$ in the median execution time when using the stop-the-world garbage collector (GC), despite the overhead of runtime checks, virtual method calls and overhead to maintain the runtime data structures of the runtime environment such as the linked stack frames [50]. The observed worst-case time is, however, 17 % higher than that of CD$_c$. The incremental collector has a higher overhead for the mutator (6 % on average, 30 % in the observed worst case), which is caused by the added overhead of write barriers and due to the complex linked list implementation used to manage the free memory that allows a list traversal to be interrupted by higher-priority mutators. Disabling the synchronization code in the incremental collector shows indeed a very similar allocation performance to that of the stop-the-world GC.

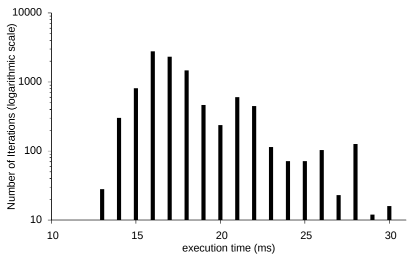
Figure 6.6 shows a more detailed view on the execution time correlation of the different versions for the first 100 detector iterations. To determine the cost caused to the mutator by the use of the GCs, I also started the test with a version that does not use a GC; it runs out of memory after 24 iterations. The graph shows a clear correlation between the execution times of the C and the Java version of the benchmark, although the observed execution time peaks tend to be stronger in the Java version (hence the relatively higher observed worst-case overhead), while CD$_j$ tends to have a better baseline performance. The execution time of the detector task varies mainly depending on the number of suspected collisions that are detected in the reducer phase. More suspected collisions require more computations in the following detector phase and also more memory allocations. Comparing the pseudo-static allocation variant that provides short, constant allocation times and does not



(a) CD_c



(b) Stop-the-World GC



(c) Incremental GC

	Worst	Median
CD_c	23 ms	16 ms
Stop-the-World GC	27 ms (17 %)	15 ms (-6 %)
Incremental GC	30 ms (30 %)	17 ms (6 %)

(d) Execution Times (% overhead to CD_c)

Figure 6.5: Collision Detector: Distribution of Iteration Execution Times

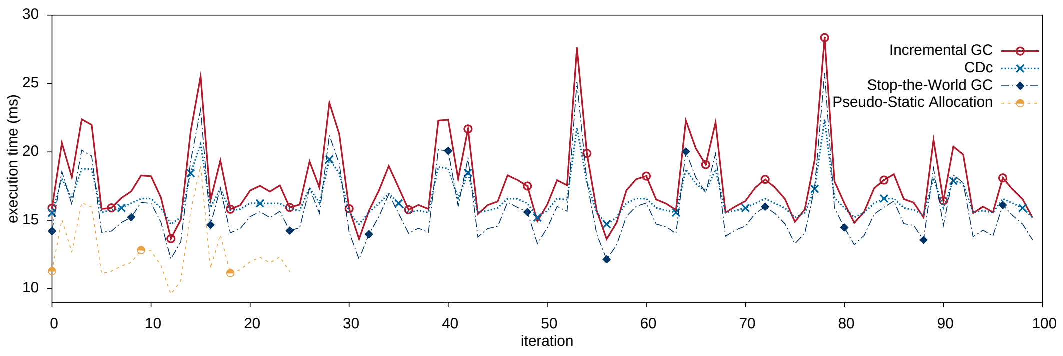
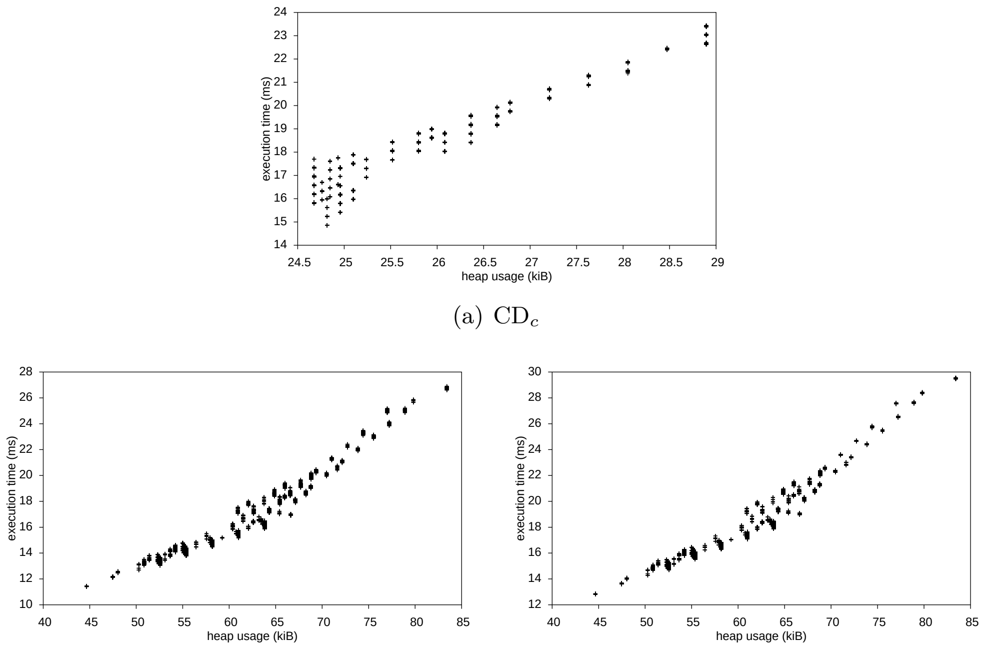


Figure 6.6: Collision Detector: Execution Times for 100 Iterations



(a) CD$_c$

(b) Stop-the-World GC

(c) Incremental GC

Figure 6.7: Correlation Between Heap Usage and Execution Time

require the maintenance of linked stack frames, with the garbage-collected variants shows, that the mutator overhead caused by using a GC is significant. Figure 6.7 shows that there is an almost linear correlation between the execution time and the amount of memory allocated by the corresponding iteration. On the other hand, CD$_c$ uses `malloc()` for memory allocation, which has an allocator with linear allocation complexity as well, and is additionally penalized by `free()` operations that are not done in the Java code. It is particularly notable that CD$_c$ allocates less memory during the iterations and that the variance in the amount of memory allocated during the different iterations is significantly less than in CD$_j$ (5.5 KiB in CD$_c$ vs. 40 KiB in CD$_j$). The cause for this is that CD$_c$ and CD$_j$ turned out to be quite different in their allocation behavior: While CD$_c$ preallocates most of the memory (378 KiB in the used configuration), CD$_j$'s allocation is driven by the actual demand. As a concrete example, CD$_c$ preallocates a configurable number (default 300) of array-backed lists with a fixed size (statically configured to the anticipated maximum, default 300), *prior to the first iteration*. CD$_j$ on the other hand allocates these lists as required in *each iteration*. Furthermore, CD$_j$ uses Java's standard collection classes, which have a smaller initial capacity (array list: 10 elements) and are enlarged on demand. This explains that CD$_c$ has an overall smaller allocation amount measured during the individual iterations, since most of the needed memory is preallocated, as well as the higher variance in the amount of allocated memory of CD$_j$, which allocates memory as required by each individual iteration.

CD_x Variant	`text`	`data`	`bss`
CD_c	39194	2075	930307
Pseudo-Static Allocation	46422	1820	827647
Stop-the-World GC	50450	1880	833140
Incremental GC	56506	1890	833136

Table 6.3: CD_x Footprint (sizes in bytes)

The pseudo-static allocation variant shows that the peaks in the execution time tend to be higher relative to the baseline performance for the Java version even when using fast bump-pointer allocation, wherefore the higher execution time variance (Figure 6.5) is apparently not caused by memory allocations. While other Java-specific overheads such as safety checks and virtual method calls occur more often in iterations with more suspected collisions, separate measurements (not depicted) showed that the overhead caused by safety checks is minor for this benchmark. I could trace the causes for CD_j having a better baseline performance but suffering more from collision intensive iterations to some algorithmic differences in the two versions of the benchmark: The used hash map implementations differ. CD_c uses a linked hash map that allows a faster iteration of the map's elements. Also, the Java hash map has an initial capacity of 11 buckets (load factor 0.75) whereas the C version starts with a capacity of 1024 buckets (load factor 1). While the initial allocation of the large hash map may be more expensive and penalize CD_c in iterations with few or no collisions, the Java version will need to reallocate and rehash the map for frames with more collisions. Another source of added overhead is a hotspot method that is only called in the detector phase (that is, if suspected collisions are present). In this method, four vectors are cloned (that is, allocated and copied) needlessly, whereas the CD_c just uses the vectors by reference (CD_j could do so as well). These differences in the two benchmarks explain the causes of the higher execution time variance of CD_j.

6.4.2.2 Footprint

Table 6.3 shows the footprint of the different variants. In the code size, the Java code is 18 % larger than CD_c. Besides CD_j containing some additional code surrounding the benchmark that is not part of CD_c, the Java version's `text` section contains the dispatch table (3.7 KiB) and the type information table (900 bytes), which account for most of the differences. The garbage-collected variants additionally contain the code for the respective GC, which adds approximately four KiB to the `text` section. The incremental GC besides having a more complex implementation of the actual collector also introduces write barriers in the application code. The initialized data use is similar for all variants. Slight differences in the CD_j variants are caused by differences in the management data of the different heap implementations. As to the uninitialized data (`bss`), the C version required a heap of 900 KiB whereas the CD_j versions were only given a heap of 800 KiB size. At runtime, most of the heap space is preallocated by CD_x for storing the benchmark results. CD_j has at most 326 KiB of heap space left when the detector starts.

6.4.3 Conclusions

The comparison of the C and C++ variants of CD$_x$ and the I4Copter with the Java variants shows that KESO is able to offer competitive performance. In the CD$_x$ benchmark, the whole-program analyses performed by jino are able to more than compensate the overheads of Java, such as dynamic method binding, short-lived objects, and the maintenance of runtime data structures that enable garbage collection. Even for a fully static application as the I4Copter, the overheads of using Java (without runtime safety checks) are below 10 %. These were mainly caused by the use of generic driver code where the C++ version used specialized variants derived from C++ templates (trading faster execution time for increased code size), and the overhead of Java interfaces to the shared memory and messaging mechanisms of CiAO.

Concluding, Java can offer comparable performance as C or C++, but heavily relies on a good compiler and a slim runtime environment to be able to do so. C and C++ provide the programmer with more explicit control over the language features used by the program, which in Java often are under control – and cost therefore depends on the quality – of the compiler. For example, in C++, the use of dynamically bound methods is explicit and can fully be avoided by the programmer. In Java, on the other hand, most instance method calls are dynamically bound, and the quality of the compiler's devirtualization has a high impact on the efficiency of the resulting program.

Jino has improved a lot in the area of static program analyses and compiler optimizations in the last years, but there is still room for further improvement. One such area of improvement is the native interfaces. For example, `RawMemory` objects are currently not as well handled by the static analyses as regular Java arrays. This also reflects in the performed benchmarks, where applications that make little use of the native interfaces (Flightcontrol and CD$_j$) showed better performance than those with a higher use of these mechanisms.

6.5 Microbenchmarks: Individual Costs of Basic Operations

In this section, I evaluate the cost of the different base primitives used to enforce memory protection. I also measure the runtime of some common operations of the CiAO operating system in the presence of different levels of hardware-based memory protection. The actual cost for an application depends on the extent to that the respective primitives and operations occur in the application code, which varies from application to application. In Section 6.6, I investigate the effects at the example of the I4Copter application. For these measurements, the system timer runs at the same 50 MHz frequency as the CPU. The measured values are cycle accurate.

	CPU Time (cycles)	Code Size (bytes)	RAM Accesses
`null` Check	1	8	0
Bound Check	5	14	1
Type Check (Specific)	4	14	1
Type Check (Range)	8	8	1
`enterTrusted()`	26	4	3
`leaveTrusted()`	3	20	0

Table 6.4: Costs of Basic Protection Primitives (TC1796, 50 MHz)

6.5.1 Basic Protection Primitives

Table 6.4 shows the cost of the different basic protection primitives. For software-based protection, these are the most commonly encountered runtime checks. For hardware-based protection, these are the operations that switch the execution context to trusted mode and back to non-trusted mode. Code and data are run from internal scratchpad memories of the CPU, which can be accessed in a single cycle. For each operation, I also provide the number of bytes added for each operation to the application code, and the number of load or store accesses to the memory that occur in the respective operation. If the data resides in external memory, these operations require more CPU cycles. It should also be noted that the actual code generated for a check depends on the C compiler and the context in which the check occurs. Deviations from the values in Table 6.4 are therefore possible. Still, the values provide a sufficiently accurate indicator of the cost of the respective operation.

6.5.1.1 Runtime Checks

All runtime checks consist of a condition that is checked and a call to an error handler if the condition fails. The shown execution times are for the case where the check succeeds. KESO emits hints that tell the C compiler that the success case is expected for the check. For the Tricore architecture, the C compiler consequently selects a conditional 16-bit forward branch instruction, which according to the static branch prediction of the Tricore architecture does not cause a pipeline stall in the case that the branch is taken.

For a `null` check, the by far most frequent runtime check, a single conditional branch instruction is often sufficient. The reference is loaded for the checked operation anyway. With the Tricore's branch prediction, the cost in the success case is a single CPU cycle. For bound checks, different variants exist in KESO, depending on whether the size or the used index is already known statically. The bound check measured in Table 6.4 is a generic one, where neither the used index nor the array size are statically known. The size of the array commonly needs to be loaded from RAM. The cost for the check is five cycles. The third type of check is a type check, which

	Task Activation	Task Switch	NTF Invoke	NTF Return
No Protection	71	181	10	1
Kernel Protection	120	240	101	61
Application Isolation	120	286	111	78
Control-Flow Isolation	120	298	95	64

Table 6.5: Runtime of System Operations with Different Levels of MPU Protection (CPU cycles, TC1796, 50 MHz)

occurs on downcasts in the class hierarchy, for example, or implicit with the use of Java generics. There are two kinds of type checks in KESO. The first one checks for a specific type, which corresponds to an integer comparison. This check is selected by jino if the target type contains no further (instantiated) subtypes. The second kind checks for a range of subtypes, which with KESO's class numbering scheme requires integer comparisons with a lower and an upper bound. This code is not inlined, wherefore the more expensive check adds less code to the program than the check for a specific subtype. The runtime cost is four cycles for the fast variant and eight cycles for the generic one. In both cases, the type identifier needs to be read from memory. The identifier of the target type or the target range's bounds are statically known and inserted as constants into the code.

6.5.1.2 Protection Mode Switch Operations

The lower part of Table 6.4 shows the cost for CiAO's primitives to enter and leave trusted protection context. The `enterTrusted` operation consists of a single `syscall` CPU instruction at the call side, which triggers a trap. The trap handler switches the region set of the Tricore MPU to the supervisor mode region set[2] and returns to the caller. The total cost of the operation is 26 cycles. The return to non-trusted context does not require a trap and costs only three cycles.

6.5.2 Costs of Common System Operations

Hardware-based memory protection also increases the cost of many system operations (Section 4.4). Table 6.5 shows the runtime of some common operations in CiAO with differing levels of memory protection.

The first is a call to the `ActivateTask()` service without triggering a task switch. In the presence of memory protection, this service is wrapped with `enterTrusted()` and `leaveTrusted()` operations. These operations do not vary with the differing protection level; the cost for the system service increases by 51 cycles (69 %). This may be surprising because the combined cost of `enterTrusted()` and `leaveTrusted()` is only

[2]On the Tricore, the MPU is active even in supervisor mode.

29 cycles according to Table 6.4. The difference is caused by the code transformations applied by the AspectC++ weaver, which are not fully optimized by GCC.

The second column shows the cost of a task activation followed by a task switch. The measured time interval is from the call to `ActivateTask()` to the first instruction of the activated task. If there is only kernel protection, besides the task activation itself becoming more expensive (as above), the scheduler must check on dispatch of a task whether the application of that task is trusted, or not. For application isolation and control-flow isolation, the MPU regions need to be additionally set up for the target application. With control-flow isolation, more regions are needed than for application isolation, wherefore the task switch is slightly more expensive. In total, the overhead for a task switch with memory protection ranges from 59 cycles (33 %) to 117 cycles (65 %).

The last two columns show the cost for the invocation of a non-trusted function, and the return from a non-trusted function. For the invocation, the time from the call to the non-trusted function to the first instruction of the non-trusted function is measured. For the return, the time from the last instruction of the non-trusted function to the instruction following the non-trusted function call in the caller OS-Application is measured. Besides switching to the memory protection context of the respective target OS-Application, the unused portion of the task stack needs to be made available in the callee OS-Application (Section 4.4). For a non-trusted ISR, similar actions are taken on entry and exit. Compared to a regular function call in the absence of memory protection, the costs increase up to a factor of eleven for the invocation, and a factor of 78 for the return. The reason why the overhead is smaller with control-flow isolation is that handling the stack region is cheaper with control-flow isolation, where a separate region is used for the stack anyway. With lower isolation levels, enabling access to the task stack is a special case that requires some additional instructions.

6.5.3 Conclusions

Protection mode changes in the presence of hardware-based memory protection showed to increase the cost of different system operations significantly. In comparison, the costs for the common runtime checks of Java are rather low. While protection mode switches with hardware-based protection occur only when the control-flow needs to cross the boundary of its protection context, however, the runtime checks with software-based protection are spread throughout the application code and commonly occur at much higher frequency. The actual overhead thus strongly depends on the respective application. In the next section, I analyze the actual overhead for an application at the example of the I4Copter.

6.6 Costs of Memory Protection in the I4Copter

While microbenchmarks provide a measure for the cost of individual system operations, they are of limited significance for an actual application. Cost differences of an order of magnitude as seen for non-trusted function calls, for example, may be acceptable to an application if such operations rarely occur. On the other hand, individually inexpensive operations such as bound checks may turn out to significantly increase the execution time if occurring in hotspots. In this section, I compare the costs of using hardware-based protection, software-based protection, and the combination of the two for the four components in the I4Copter software. The baseline of the comparison is a configuration that uses neither hardware- nor software-based protection.

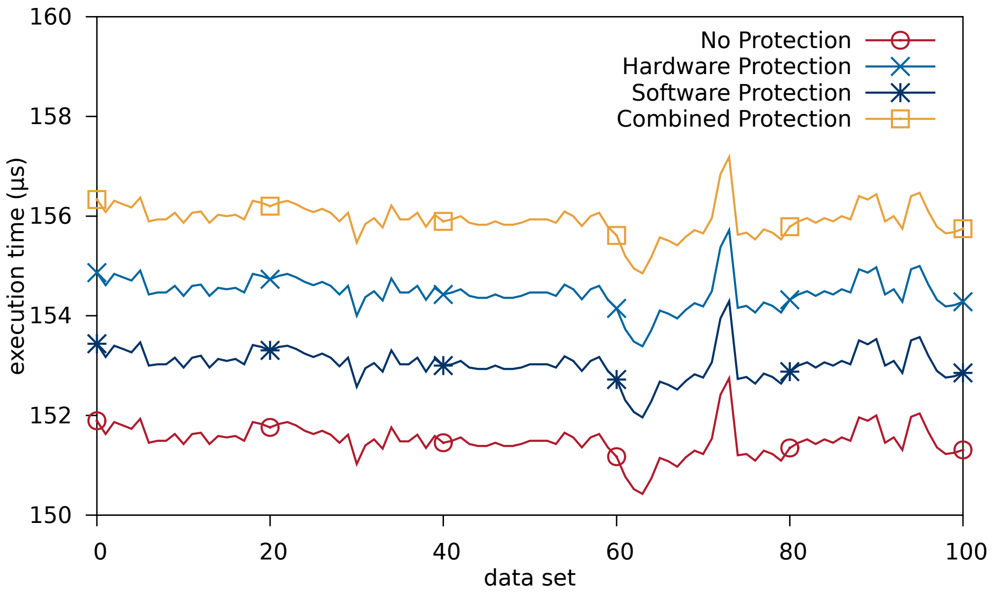
For this comparison, I only use the Java port of the I4Copter components. The code of all variants is therefore identical, except for the changes imposed by the respective protection mode. The configuration of hardware-based memory protection is identical to my previous setting in the language comparison (write-only protection, control-flow isolation), except for the message ports. Message ports in this section use the variants where hardware-based protection ensures that messages can only be written between the `receive()` and `release()` primitives on the receiver side, and the `allocate()` and `send()` operations on the sender side. These four primitives therefore become operations that trigger changes to the MPU configuration.

6.6.1 Execution Times

The results of the execution time comparison are shown in Figure 6.8 (Flightcontrol), Figure 6.9 (Coptercontrol), Figure 6.10 (Signalprocessing) and Figure 6.11 (SerialCom). Each figure contains a line graph that shows the correlation of execution times for 100 iterations and a table with the worst observed execution time and the median of execution times, including the relative overhead over the baseline of an unprotected execution. The third table shows the number of `null`, bound and type checks in the variants with software protection, split into checks that need to be performed at runtime and those that could be performed at compile time and are therefore not contained in the generated code anymore. These check numbers represent the static number of checks in the image, not the actual number that each of these checks is triggered at runtime. They provide a basic measure for the expected cost for software-based protection for the respective part, and in particular show to what extent jino was able to prove checks to always succeed at compile time.

6.6.1.1 Flightcontrol

In the Flightcontrol task, one message is received and one message is sent in each iteration. In addition, there is one call to `SetEvent()`. These are all operations for which hardware-based protection incurs additional cost. For software-based protection, more than 95 % of the runtime checks of each category could be proven to succeed by jino at compile time. In total, the protection cost is low for both hardware- and software-based protection in relation to the total execution time of the component.



(a) Correlation of 100 Executions

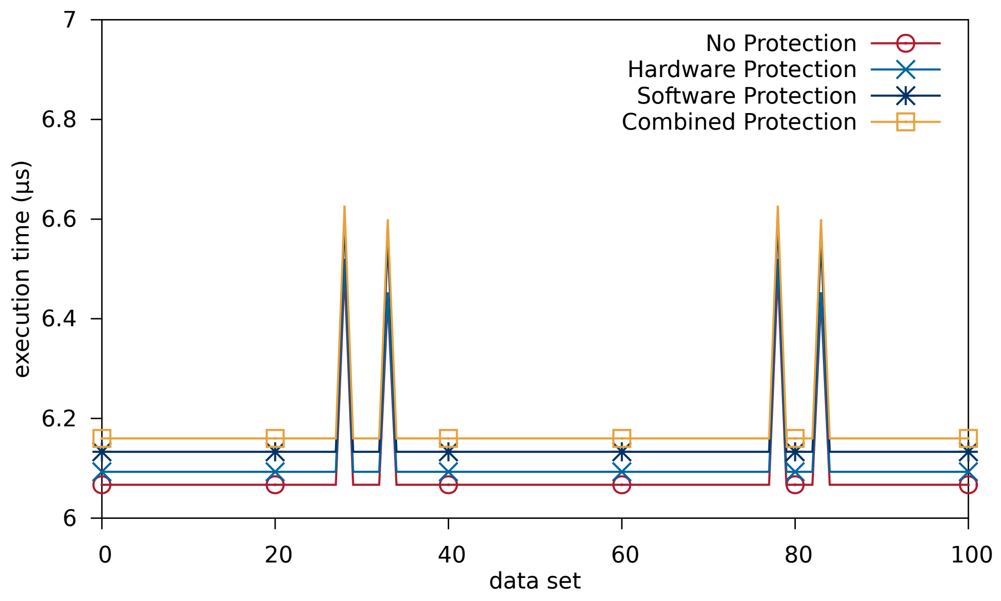
	Worst	Median
No Protection	155.4 µs	151.4 µs
Hardware Protection	158.6 µs (2.1 %)	154.4 µs (2.0 %)
Software Protection	156.9 µs (1.0 %)	152.9 µs (1.0 %)
Combined Protection	160.1 µs (3.0 %)	155.8 µs (2.9 %)

(b) Execution Times (% overhead to no protection)

	Runtime	Ahead-of-Time Success	Total
`null` checks	52 (4 %)	1120 (96 %)	1172
bound checks	6 (2 %)	270 (98 %)	276
type checks	0	0	0

(c) Runtime Checks

Figure 6.8: Flightcontrol: Comparison of Protection Variants



(a) Correlation of 100 Executions

	Worst	Median
No Protection	6.5 µs	6.1 µs
Hardware Protection	6.5 µs (0.4 %)	6.1 µs (0.4 %)
Software Protection	6.6 µs (1.6 %)	6.1 µs (1.1 %)
Combined Protection	6.6 µs (2.1 %)	6.2 µs (1.5 %)

(b) Execution Times (% overhead to no protection)

	Runtime	Ahead-of-Time Success	Total
`null` checks	44 (3 %)	1230 (97 %)	1274
bound checks	54 (31 %)	120 (69 %)	174
type checks	0	0	0

(c) Runtime Checks

Figure 6.9: Coptercontrol: Comparison of Protection Variants

6.6.1.2 Coptercontrol

In Coptercontrol, the used system services per iteration are principally identical to the Flightcontrol task. Messages are sent and received to change the power state of the engine controllers. This power state is, however, only changed in the initialization phase and during emergency shutdown. Since my measurements only simulate flight mode, no messages are sent or received in the measured code. The added cost for hardware-based protection is therefore only caused by the single call to `SetEvent()` and is negligible (0.4 %) in relation to the total execution time. For software-based protection, jino was not able to eliminate a considerable number of bound checks at compile time; none of these checks occurs in a hotspot location, however, so the total relative overhead of software-based protection is little (1.1 %).

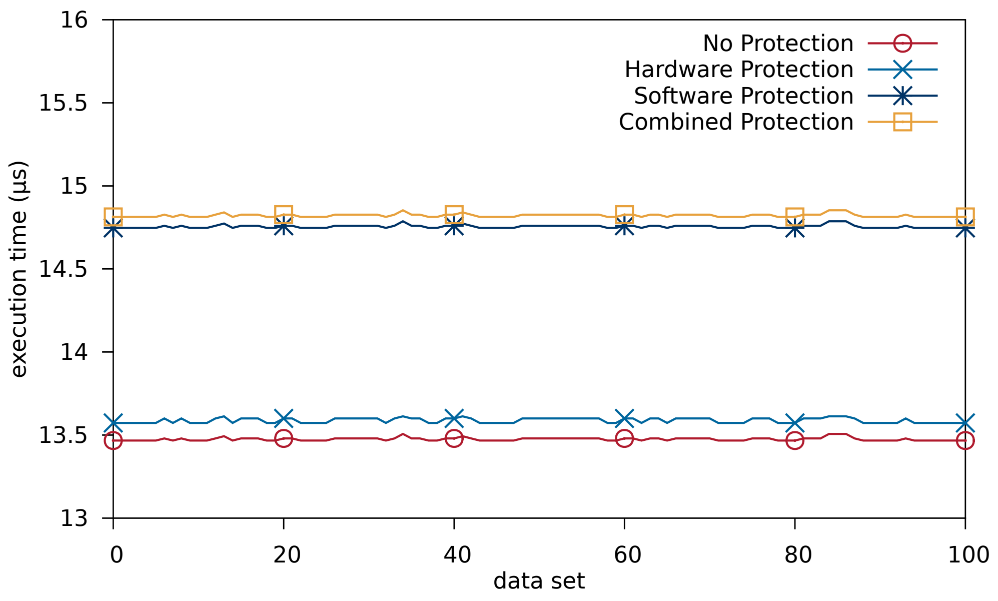
6.6.1.3 Signalprocessing

The Signalprocessing component with 9.5 % shows the highest overhead of all components for software-based protection. I could identify two bound checks in a hotspot code location that are the source for the largest part of this overhead. The bound checks occur in the prefiltering code that is used for most analog-to-digital converter channels. The code computes the average sensor value of a per-channel configurable number of last samplings of the channel. These numbers are kept in an array, which is read and updated on each sampling. Jino is currently only able to optimize bound checks if the size of the array is a compile time constant. It is not currently able to evaluate dependencies between the used index and a non-constant array length. For example, a loop such as the following is commonly used to iterate an array:

```
for(int i=0; i<array.length; i++) { // array is of type int[]
    array[i] = 0;
}
```

Jino can only eliminate the bound check required for the array store operation in the above example if `array.length` is a compile-time constant for all arrays that the reference could possibly point to. In the analog-to-digital converter prefiltering code, the length of the array is constant for each channel, but not the same for all channels. Consequently, jino fails to remove these bound checks. With these two bound checks manually removed, the overhead of software-based protection drops to 1.5 %. This example shows how few checks in hotspot locations can incur a significant overhead for the program.

Hardware-based protection should not incur any overhead to the Signalprocessing component. My standalone variant only contains the gyroscope drivers, which do not use the message port mechanism. There are also no other system calls affected by hardware-based protection in the measured code. Yet the measurement shows an overhead of 0.8 % for hardware-based protection. A comparison of the generated machine code shows no differences in the measured functions. The difference is within the accuracy of the measurement.



(a) Correlation of 100 Executions

	Worst	Median
No Protection	14.9 µs	13.5 µs
Hardware Protection	14.9 µs (0.3 %)	13.6 µs (0.8 %)
Software Protection	16.1 µs (8.6 %)	14.8 µs (9.5 %)
Combined Protection	16.2 µs (9.1 %)	14.8 µs (10.0 %)

(b) Execution Times (% overhead to no protection)

	Runtime	Ahead-of-Time Success	Total
`null` checks	24 (3 %)	836 (97 %)	860
bound checks	44 (49 %)	46 (51 %)	90
type checks	0	0	0

(c) Runtime Checks

Figure 6.10: Signalprocessing Comparison of Protection Variants

6.6.1.4 SerialCom

The SerialCom standalone component is essentially a benchmark of the messaging performance. The runtime of the task with hardware-based protection is three times the runtime of the task without hardware-based protection. This is fully a consequence of the overhead added to enforce the message protocol, which adds overhead to all message primitives except for `peek()`. Without enforcement of the protocol, the message mechanism is completely orthogonal to hardware-based memory protection, and hardware-based memory protection can be used without added cost for the SerialCom component. The software-protected variants only include a few `null` checks, which are negligible in relation to the total execution time. The slight speedup that the software-protected variant shows over the unprotected variant is within the accuracy of the measurement.

6.6.2 Memory Footprint

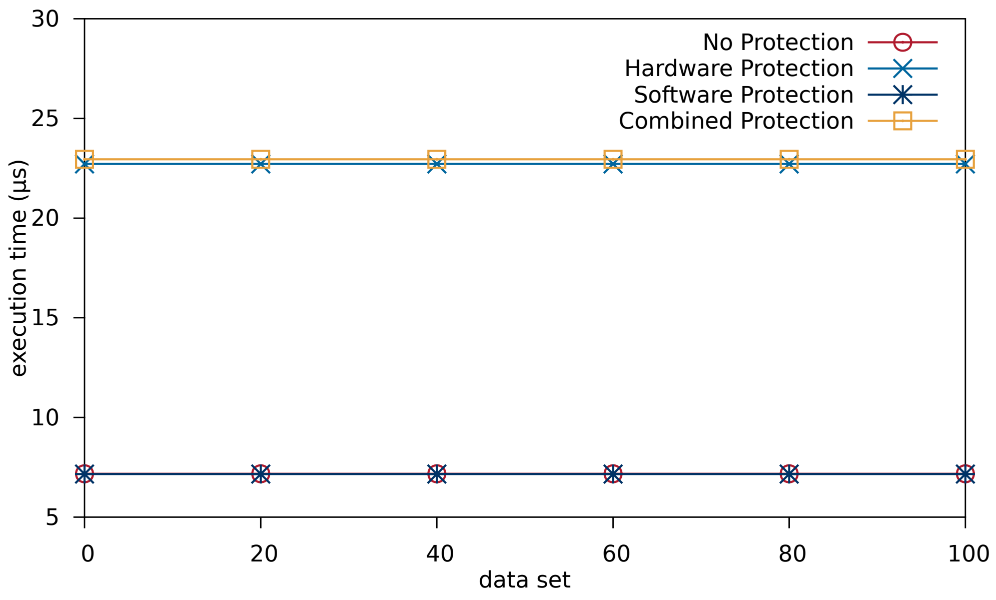
Table 6.6 shows the effect on the footprint of the program for the different protection variants. The used images are the standard images of the I4Copter software, not the standalone variants. The shaded cells in Table 6.6(b) and Table 6.6(c) highlight the differences to the unprotected variant.

6.6.2.1 Hardware-based Protection

The numbers for hardware-based protection (Table 6.6(b)) differ from those of the unsafe Java variant in the language comparison (Table 6.1(b)) because the former uses enforcement of the message protocol, whereas the latter uses the simple message port variant without enforcement. In the operating system, hardware-based memory protection introduces new data structures for the management of OS-Applications and extends existing ones such as the task control blocks with, for example, the mapping to the respective OS-Application or task-specific memory regions. In total, the data memory requirements for the CiAO OS itself increase by 198 bytes (8 %) in the I4Copter. The code of the OS is extended by the MPU driver, and various operations of the OS, such as the context switch, are extended to switch the memory protection context. With the callee-side weaving (Section 4.4.3.2), the application interface to system services is also extended to include privilege mode changes on the OS side. In total, the code of the OS grows by 830 bytes (5 %).

The code of KESO's runtime environment increases by 72 bytes (32 %) because it contains a call to CiAO's `ShutdownOS()` service in the error handling routines, which is extended by the memory protection aspects and inlined in KESO's code.

In the application data, a shift of data from the private application categories to the *other* category stands out. The reason for this shift is the use of the protocol-enforcing message ports, for which the pool with the message buffers is stored outside the private data segments of the OS-Applications and message buffers are only temporarily made available to the OS-Application between the `allocate()`/`send()` and the `receive()`/`release()` pairs. The total data consumption remains the same.



(a) Correlation of 100 Executions

	Worst	Median
No Protection	7.2 µs	7.2 µs
Hardware Protection	22.7 µs (216.3 %)	22.7 µs (216.7 %)
Software Protection	7.2 µs (-0.4 %)	7.2 µs (-0.2 %)
Combined Protection	23.0 µs (219.7 %)	22.9 µs (219.9 %)

(b) Execution Times (% overhead to no protection)

	Runtime	Ahead-of-Time Success	Total
`null` checks	28 (37 %)	48 (63 %)	76
bound checks	0 (0 %)	6 (100 %)	6
type checks	0	0	0

(c) Runtime Checks

Figure 6.11: SerialCom Comparison of Protection Variants

	bss	data	text
CiAO	2034 (12.9 %)	364 (3.5 %)	15228 (25.9 %)
KESO	44 (0.3 %)	320 (3.1 %)	226 (0.4 %)
CopterControl	1305 (8.3 %)	2436 (23.4 %)	5314 (9.0 %)
Ethernet	5648 (35.9 %)	194 (1.9 %)	2030 (3.4 %)
FlightControl	1925 (12.2 %)	2388 (22.9 %)	5352 (9.1 %)
InitTask	128 (0.8 %)	28 (0.3 %)	68 (0.1 %)
SerialCom	640 (4.1 %)	728 (7.0 %)	2536 (4.3 %)
SignalProcessing	4017 (25.5 %)	2696 (25.9 %)	10406 (17.7 %)
Shared	0 (0.0 %)	84 (0.8 %)	4122 (7.0 %)
Other	12 (0.1 %)	1169 (11.2 %)	13596 (23.1 %)
Total	15753	10407	58878

(a) No Protection

	bss	data	text
CiAO	2047 (13.0 %)	549 (5.2 %)	16058 (23.1 %)
KESO	44 (0.3 %)	320 (3.0 %)	298 (0.4 %)
CopterControl	1305 (8.3 %)	284 (2.7 %)	5870 (8.4 %)
Ethernet	5648 (35.8 %)	194 (1.8 %)	2644 (3.8 %)
FlightControl	1925 (12.2 %)	236 (2.2 %)	5632 (8.1 %)
InitTask	128 (0.8 %)	28 (0.3 %)	118 (0.2 %)
SerialCom	640 (4.1 %)	344 (3.2 %)	4934 (7.1 %)
SignalProcessing	4017 (25.5 %)	352 (3.3 %)	11046 (15.9 %)
Shared	0 (0.0 %)	84 (0.8 %)	9342 (13.4 %)
Other	12 (0.1 %)	8201 (77.4 %)	13596 (19.6 %)
Total	15766	10592	69538

(b) Hardware Protection

	bss	data	text
CiAO	2034 (12.9 %)	364 (3.5 %)	15228 (24.5 %)
KESO	44 (0.3 %)	320 (3.1 %)	306 (0.5 %)
CopterControl	1305 (8.3 %)	2436 (23.4 %)	5702 (9.2 %)
Ethernet	5648 (35.9 %)	194 (1.9 %)	2494 (4.0 %)
FlightControl	1925 (12.2 %)	2388 (22.9 %)	5632 (9.1 %)
InitTask	128 (0.8 %)	28 (0.3 %)	68 (0.1 %)
SerialCom	640 (4.1 %)	728 (7.0 %)	2952 (4.7 %)
SignalProcessing	4017 (25.5 %)	2696 (25.9 %)	11190 (18.0 %)
Shared	0 (0.0 %)	84 (0.8 %)	5044 (8.1 %)
Other	12 (0.1 %)	1169 (11.2 %)	13596 (21.9 %)
Total	15753	10407	62212

(c) Software Protection

Table 6.6: I4Copter Memory Footprint: Different Memory Protection Variants (sizes in bytes, % share of the total section)

The code size for the application code increases by 9758 bytes (33 %), caused by the inlined use of system services affected by the memory protection aspects. The high overhead is mainly caused by the protocol-enforcing message ports as a consequence of the size increase of the individual primitives, multiplied by template specialization (14 variants) and inlining. This effect is the same to what I earlier observed with the static driver configuration based on C++ templates in the I4Copter application.

In total, the data memory consumption increases by 185 bytes (less than 1 %), and the code size increases by 18 %.

6.6.2.2 Software-based Protection

For software-based protection, the data memory use does not change. Only the code is expanded with the runtime checks. The size of the KESO runtime increases by 80 bytes, caused by the function that performs type-range checks. The size of the application code increases by 3254 bytes (11 %).

In total, the data memory consumption is unaffected and the code size increases by 3334 bytes (6 %) compared to the unprotected variant. The overhead could be further reduced with the graduations of software-based protection presented in Section 4.6. By offloading checks to the hardware, the total size can be shrunk to 61454 bytes, reducing the code size overhead to 4 % without reducing the level of protection provided. Reducing the level of protection to fault-containment further shrinks the code to 61300 bytes; this small gain does, however, hardly compensate the loss in the provided protection level.

6.6.3 Conclusions

The comparison of the cost of memory protection in the I4Copter showed mixed results for different components. In the Flightcontrol and Coptercontrol tasks, the cost for both hardware- and software-based memory protection is very low. Given the low cost in execution time of 2.9 % in Flightcontrol and 1.5 % in the Coptercontrol added by combining hardware- and software-based protection, this combination seems feasible. In the other two components, the picture was more diverse. In the Signalprocessing component, the runtime cost of software-based protection was relatively high with 9.5 %, mainly a consequence of jino not being able to statically eliminate bound checks where the index correlates to an array length that is not a compile-time constant. Hardware-based protection, on the other hand, incurred no further cost. The SerialCom component showed the inverse picture: Hardware-based protection triples the execution time of the task, which is caused by the heavy use of the expensive messaging mechanism. Software-based protection is very low in overhead in the component, on the other hand. It should be pointed out, however, that the cost of hardware-based protection could be reduced to a similar level as in the other components by using the simple message port variant that does not enforce the message protocol.

In the footprint, the effect on the RAM usage was insignificant. With respect to code size, however, software-based protection added significantly less (6 %) to the code than hardware-based protection (18 %). This is not least due to the runtime check elimination of jino, which showed to be highly effective throughout all components. Except for the SerialCom component, more than 95 % of the `null` checks could be proven to succeed at compile time. Even the elimination of bound checks with the deficiencies described above is fairly effective for a static embedded application like the I4Copter. The `RawMemory` objects used to access memory-mapped registers, shared memory areas or messages are commonly of a fixed and known size, even though jino has to resort to the `sizeof` trick for the latter two. The used indexes – with few exceptions – are also compile time constants or can be identified to be within a safe value range. Due to this high effectiveness, the overhead of software-based protection is low and I therefore did not analyze the further potential of gradual software-based protection in detail – the possible gains are little.

6.7 Chapter Summary

In this chapter, I evaluated quantitative aspects of my framework and addressed the remaining open goal of the support for an easy quantitative evaluation of memory protection costs for a given application.

6.7.1 Overhead of Using Java as a Language Compared to C/C++

The first question addressed by the evaluation is the overhead imposed by using Java as a language instead of the more spread C and C++ languages. I used two diverse applications from the domain of embedded real-time applications to evaluate this aspect: On the one end is the I4Copter as a very static application that includes low-level driver code, uses the shared memory and message port facilities to interact with C++ components and uses no dynamic memory allocation. On the other end of the spectrum is CD$_x$, which in its Java variant uses many of Java's high level features, including garbage collection.

In these comparisons, the code produced by jino showed competitive performance for (mostly) pure Java applications – CD$_j$ and the Flightcontrol component of the I4Copter performed better than their C++ counterparts, mainly a result of the whole-program analyses performed by jino. In the I4Copter components that use low-level memory interfaces, the Java variants showed higher execution times, mainly caused by the overhead of the native interface abstractions, but also by the consequences of the generalization approach used in Java and the specialization approach used in C++ (templates). Yet the overhead stayed below 10 % in the median of execution times for these components. The code size showed the downside of the specialization, where the Java variant of the I4Copter was more than 20 % smaller than the C++ variant. The data use of the Java variant increased over the C++ version. The main cause is KESO's multi-JVM concept, which replicates the static fields, and proxy objects for low-level memory accesses. The single domain CD$_j$ application showed no overhead in data use to the C version CD$_c$.

The bottom line of these comparisons is that Java can offer competitive performance to C++ in the field of static embedded applications. In particular, the comparison straightens some common misbeliefs on Java: Firstly, the Java versions showed to behave as deterministic as the C++ versions. Secondly, KESO's self-tailoring approach [113] showed how small the size of the Java runtime environment can be for applications that do not use most of Java's features – in the I4Copter, KESO's runtime environment is below 500 bytes respectively in code and data. On the other hand, the self-tailoring approach does not impose as strict fixed limits as other approaches, for example J2ME configurations. Features such as garbage collection are available for applications such as CD_j that actually need these features, but only paid for if actually used.

6.7.2 Costs of Basic Protection Primitives

The second question addressed is the cost of the individual basic protection primitives. To answer this question, I conducted microbenchmarks on the cost in code size and data use of the individual software- and hardware-based protection primitives. I also investigated the cost added to internal operating system operations such as a context switch in the presence of different levels of hardware-based memory protection, an aspect that is not measured in my later application benchmarks anymore. Summarized, the software-based protection primitives showed to be far less expensive than the hardware-based ones, but can be expected to occur in higher volumes in the code of an actual application. This aspect is addressed by the third question.

6.7.3 Comparison of Protection Mechanisms for an Application

In the third part, I practically evaluated if the framework meets the goal of providing support for the quantitative evaluation of the protection cost for a specific application by performing such an evaluation on the I4Copter. To enable comparability of the execution times down to the level of the individual jobs of each task, I created standalone variants of each task that could be run with logged input data to achieve reproducible executions. For a practical estimate of the cost of memory protection, however, such a detailed job-by-job comparison is not required to determine the cost of memory protection. In such cases, the framework freely allows to switch between different memory protection variants and measure the cost. I therefore believe that the framework fulfills this goal.

The case study on the I4Copter also showed the cost of hardware-based versus software-based memory protection for a static embedded real-time application. For two components, both protection variants showed little overhead and could be used in combination with added cost below 3 %. For the other two components, one showed to be cheaper with hardware-based protection, whereas the other showed significant overhead (200 %) with hardware-based protection. The first was caused by bound checks in a hotspot location that could not be eliminated at compile time by jino. The second was mainly caused by the protocol-enforcing message variants.

Summarized, the evaluation showed that MPU-based protection can provide a basic level of fault-containment at very little cost. Enforcing more elaborate policies such as the message protocol quickly results in increased cost, however. Software-based protection is much more flexible in this regard. Although software-based protection theoretically incurs a high number of runtime safety checks, jino showed that the largest part of these checks can be eliminated at compile time, particularly in static applications such as the I4Copter.

7 Summary, Conclusions, and Outlook

In this thesis, I developed a framework that allows to choose between hardware- and software-based memory protection for the domain of statically-configured, deeply-embedded systems. The main goal was to show the limitations of the widely-adopted memory protection based on the use of an MPU, and to investigate software-based protection as both an alternative and a complement that is better suited for many applications. The developed framework provides an infrastructure to easily switch and combine these mechanisms, evaluate the quantitative cost of each for a given application, and based thereon supports the decision making for the best suited memory protection mechanism.

7.1 Summary

This work was motivated by the recent trends in the automotive – or more generally the embedded – industry to integrate multiple software-functions on a single control unit, raising among other requirements the need for spatial isolation. With the AUTOSAR OS specification, the automotive industry included region-based memory protection as a requirement for hardware platforms. MPU-based protection is, however, not optimal for all application scenarios. Besides limiting the hardware options in cost-sensitive markets to the higher-end derivates of microcontroller product lines, MPU-based protection is subject to limitations imposed by the small number of available memory regions. Software-based memory protection provides benefits over hardware-based protection in many aspects, but has so far received little attention in this application domain. This issue motivated this thesis, in which I aimed to show the potential of software-based protection as an alternative or a complement to hardware-based protection.

I performed a thorough review of the state of the art in both hardware- and software-based memory protection with the goal to select two mechanisms, one hardware- and one software-based one, that complement each other so that the strengths of each address deficiencies of the other. A combination of the two is therefore able to provide a comprehensive level of memory protection. The two mechanisms that I opted for are MPU-based protection and a language-level approach based on the type-safe Java language in combination with a multi-JVM architecture.

Based on the configurable operating system CiAO and the multi-JVM KESO, I composed a framework that fulfills the following goals:

Fine-grained Configurability: For both hardware- and software-based protection, the framework supports varying cost versus protection-level trade-offs, including completely disabling either mechanism. For hardware-based protection, I used aspect-oriented programming to integrate the necessary operations into the OS and the application to establish the configured level of memory protection. For software-based protection, the configurability is based on the static analyses in jino.

Mixed-Mode Operation: The type of memory protection can be individually selected for each protection realm. Thereby, the best suited mechanism can be used for each application in the system, or even for different components of the same application.

Soft Migration: The framework requires the applications to be written in Java to utilize the full feature spectrum. Because Java is not widely adopted, I developed an extension of the framework by communication mechanisms that enable Java components to interact with C or C++ components. These interfaces enable a soft migration of existing code to Java on a per-component basis. I practically tested this soft migration approach at the example of a control application for a quadrotor helicopter.

Support Quantitative Evaluation: With the possibility to change the memory protection policy by simply adapting the system configuration, it is easy to evaluate the quantitative cost imposed by different memory protection policies for a given application, aiding the decision for the best-suited mechanism from a cost perspective.

7.2 Conclusions

The following are the main conclusions that can be drawn from the findings made in this thesis:

MPU-based Protection is Inflexible and does not come for free: In this thesis, I opted for a very static variant of MPU-based protection to provide a safety net for protection in the areas where software-based protection has deficiencies. This basic implementation can be complemented by software-based protection. My implementation and system abstractions are designed according to this principle, resulting in a highly efficient basic protection level provided by MPU-based protection. I did, however, also show the limitations of this basic protection. My analysis of the region requirements of shared memory areas with multiple writers or in the presence of read protection showed that this simple mechanism quickly becomes unfeasible. A more dynamic access granting to shared memory would be possible, however, my evaluation of the protocol-enforcing message ports showed that such more elaborate policies can lead to significant cost for MPU-based protection.

Managed Languages Facilitate MPU-Based Protection: MPU-based protection has strict requirements on the physical locations of data items in memory. C and C++ provide a large degree of freedom to the developer in how data is represented; this also means, however, that the full responsibility of appropriately arranging the data stays with the developer. The I4Copter application showed that this task is tedious and only the minimum required effort was taken to enable hardware-based memory protection. In a managed language such as Java and a multi-JVM concept with strict separation of the data of different applications, the data representation required to enable MPU-based protection can be carried out by the compiler automatically, without requiring the developer to structure the application according to the requirements of MPU-based protection.

Java can be as Efficient as C/C++: It is still a widely-spread preconception that Java is prohibitively expensive for the use in resource-constrained embedded systems. My evaluation of both a fully static and a more dynamic application from this target domain showed, that a suitable Java implementation can be as efficient as C/C++. The key technologies for achieving competitive performance in this domain are ahead-of-time compilation, JVM-tailoring to achieve a slim runtime that fits the feature requirements of the application, and incorporation of the system model into the compiler.

7.3 Contributions

This thesis advances the state of the art in several aspects, which are summarized in the following:

- The first framework that provides the choice between and the combination of memory protection based on an MPU and software-based memory protection
- The control-flow-specific static analyses, which provide the infrastructure for the configurability of software-based protection levels down to the granularity of basic blocks, and the automatic identification of private application code and data that vastly increases the effectiveness and usability of hardware-based memory protection in CiAO

- A soft migration strategy to help clear the hurdle of existing legacy code, a major barrier to the introduction of modern languages in the embedded domain
- The easy evaluation of the cost imposed by hardware- versus software-based memory protection, enabled by making the change of memory protection policies a matter of adapting the configuration
- The first comparison of MPU- versus software-based memory protection for a real-world safety-critical real-time application, which shows the practicability of software-based protection and the use of Java as a language in this domain
- Advances to the KESO project; to the best of my knowledge, KESO is the smallest and most resource-efficient JVM for embedded systems. KESO evolved to an open-source project and is freely available to the research community.

7.4 Ideas for Future Work

I conclude this thesis with some sketches of possible future work.

Hardening of the JVM Against Transient Faults One drawback of memory safety based on the use of Java is its susceptibility to transient hardware faults [24, 25]. Since the memory safety relies on the integrity of references in the object graph, bit errors in reference values and type identifiers have the potential to compromise the memory safety of the program, and cause the error to spread to other protection realms. Other parts of the Java runtime environment, for example the garbage collector, rely on the consistency of the object graph as well.

With shrinking structure sizes, soft errors can be expected to become more frequent in the future [79]. Chen at. al [26] proposed a fault-tolerant JVM, but their technique requires a memory management unit, which is not commonly available in deeply embedded systems. An alternative could be purely compiler-based techniques to replicate critical data structures (for example, triple each reference) or to use unused bits (for example, the least-significant bits in references are often clear for alignment reasons) to store parity information. There are many interesting open questions to be addressed, for example when to check the consistence of a reference to achieve an acceptable trade-off of cost and dependability.

Automated Replication of Application Parts Similar to the previous idea of hardening the Java runtime system, the application or critical parts of the application itself can be hardened against soft errors. A common software-based technique is triple-modular redundancy (TMR), where three instances of the critical part of the application are executed with identical input data and a majority voter decides over the results to detect the correct result and failed replicates. While the topic of TMR has been well-researched, a multi-JVM such as KESO provides the infrastructure to deal with a number of commonly encountered problems in an automated way: The domains with fully separated data and clearly defined external interfaces pose a suited unit for automatic replication and recovery on failure. A commonly encountered problem with TMR is the deterministic execution of the replicas, which is complicated by the use of non-idempotent operations. This problem exists in Java as well, but any source of indeterminism requires the use of interfaces provided by the runtime environment, which can be helpful in finding checkpointing locations. Some other sources of indeterminism within the application, for example reading memory that has not been initialized, do not exist in Java. Finally, the JVM has all the runtime state that is needed to fully automatically recover a faulty domain from the state of a consistent one, or compare to not only vote over the immediate results but over the entire state of the replicate. While there are many open questions, the available information on the structure of the application state provides promising potential to advances in this area.

Use of Runtime Data Structural Information There are other areas where the ability to inspect the structure of the application state is helpful. For example, there are different scenarios in the domain of wireless sensor networks where the application state needs to be identified and replicated. Such situations can occur when updating the software of a sensor node without losing precious state built over a period of time, or when migrating functionality to a different node, for example because the current one is running out of battery power. KESO is small enough to run on such platforms and has been successfully used on tiny 8-bit AVR microcontrollers with less than one KiB of RAM. A main problem when trying to preserve or move the state of the application is the identification of pointer values, which may need to be adjusted when the state is moved in memory. Commonly, the task of saving and restoring the state is delegated to the application. With KESO, all the required information needed to perform this operation automatically is available.

Better Java Support for Embedded Programs Java has originally not been designed for embedded systems and lacks some desirable features for this domain. The real-time specification for Java (RTSJ) [55] brought advances in this area, but open issues still remain. I sketch two ideas where Java could be improved to make it a better candidate for embedded programming.

One issue is that there is no way of defining a constant object, most notably arrays of constant data, in Java. Java only allows references to objects be declared as non-modifiable (`final`), but not the objects themselves. Better support for such objects has many advantages: On some architectures, such objects could be stored in ROM to save precious RAM. Even if this is not possible, the code size of the program can be reduced if such constant objects are statically allocated – for an array that is initialized with a static initializer of constant values, the Java compiler creates code in the class initializer (or constructor) that initializes the object. This code is much less compact than the actual data itself. Korsholm [64] presented an annotation-based approach to address this problem. In the static target environment for KESO, however, such objects could be identified fully automatically by the compiler, wherefore the expected gains can be expected to exceed those of a manual approach.

A second issue is creating temporary objects without the need for a garbage collector. For reasons of cost and determinism, it is often desirable to avoid the need for garbage collection. A safe equivalent to allocating such temporary objects on the stack would be desirable. While the RTSJ defines a conceptually similar concept, scoped memory, these memory areas impose restrictions on the use of references to such areas, which make the use difficult and error-prone. Compliance to the rules is checked at runtime and adds new exceptions, which are undesirable in a safety-critical system. An alternative, which is also supported by KESO, is performing a static escape analysis on instantiated objects, and allocating them on the stack if the reference does not escape the local scope. The problem is that the programmer does not reliably know whether the compiler will allocate a particular temporary object on the stack. A simple and pragmatic solution to this issue could be an annotation that enables the programmer to mark objects that must be allocated on the stack. If the escape analysis fails to prove the stack allocation to be safe, the compiler can raise an error. This simple approach would enable programmers to confidently allocate temporary objects on the stack without introducing any new runtime overhead or exceptions, yet the approach can be expected to be sufficient for most common cases.

Bibliography

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. „Control-Flow Integrity: Principles, Implementations, and Applications.“ In: *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS '05)*. (Alexandria, VA, USA). New York, NY, USA: ACM Press, 2005, pp. 340–353. ISBN: 1-59593-226-7. DOI: 10.1145/1102120.1102165 (cited on pages 15, 20, 45).

[2] G. Aigner and U. Hölzle. „Eliminating Virtual Function Calls in C++ Programs.“ In: *Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP '96)*. (Linz, Austria). London, UK: Springer-Verlag, 1996, pp. 142–166. ISBN: 3-540-61439-7 (cited on page 87).

[3] 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*. (San Jose, CA, USA). New York, NY, USA: ACM Press, 2006, pp. 1–10. ISBN: 1-59593-578-9. DOI: 10.1145/1178597.1178599 (cited on pages 10, 27, 43).

[4] A. W. Appel, J. R. Ellis, and K. Li. „Real-Time Concurrent Collection on Stock Multiprocessors.“ In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '88)*. (Atlanta, GA, USA). New York, NY, USA: ACM Press, 1988, pp. 11–20. DOI: 10.1145/53990.53992 (cited on page 29).

[5] *Arctic Core: Open Source AUTOSAR platform. Homepage.* http://arccore.com/, visited 2012-03-12 (cited on page 60).

[6] *AT91SAM ARM-based Flash MCU, SAM3U Series.* 6430E–ATARM–29-Aug-11. Atmel Corporation. Aug. 2011 (cited on page 48).

[7] *ATmega128 8-bit AVR Microcontroller with 128KBytes In-System Programmable Flash.* Rev. 2467X–AVR–06/11. Atmel Corporation. June 2011 (cited on page 48).

[8] J. Auerbach, D. F. Bacon, R. Guerraoui, J. H. Spring, and J. Vitek. „Flexible Task Graphs: A Unified Restricted Thread Programming Model for Java.“ In: *Proceedings of the 2008 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '08)*. (Tucson, AZ, USA). New York, NY, USA: ACM Press, 2008, pp. 1–11. ISBN: 978-1-60558-104-0. DOI: 10.1145/1375657.1375659 (cited on page 27).

[9] J. Auerbach, D. F. Bacon, D. Iercan, C. M. Kirsch, V. T. Rajan, H. Röck, and R. Trummer. „Low-Latency Time-Portable Real-Time Programming with Exotasks.“ In: *ACM Transactions on Embedded Computing Systems (TECS)* 8.2 (2009), pp. 1–48. ISSN: 1539-9087. DOI: 10.1145/1457255.1457262 (cited on page 27).

[10] AUTOSAR. *Specification of Operating System (Version 5.0.0).* Tech. rep. Automotive Open System Architecture GbR, Nov. 2011 (cited on pages 3, 35, 61).

[11] G. Back and W. C. Hsieh. „The KaffeOS Java Runtime System.“ In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27.4 (2005), pp. 583–630. DOI: 10.1145/1075382.1075383 (cited on page 27).

[12] G. Back, W. C. Hsieh, and J. Lepreau. „Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java.“ In: *4th Symposium on Operating System Design and Implementation (OSDI '00).* (San Diego, CA, USA). Berkeley, CA, USA: USENIX Association, 2000, pp. 333–346 (cited on page 27).

[13] D. F. Bacon, P. Cheng, and V. T. Rajan. „The Metronome: A Simpler Approach to Garbage Collection in Real-Time Systems.“ In: *Proceedings of the OTM Workshops: Workshop on Java Technologies for Real-Time and Embedded Systems.* (Catania, Sicily). Ed. by R. Meersman and Z. Tari. Vol. 2889. Lecture Notes in Computer Science. Springer-Verlag, Nov. 2003, pp. 466–478 (cited on page 29).

[14] D. F. Bacon and P. F. Sweeney. „Fast Static Analysis of C++ Virtual Function Calls.“ In: *ACM SIGPLAN Notices* 31.10 (1996), pp. 324–341. ISSN: 0362-1340. DOI: 10.1145/236338.236371 (cited on page 87).

[15] H. G. Baker. „The Treadmill: Real-Time Garbage Collection without Motion Sickness.“ In: *ACM SIGPLAN Notices* 27.3 (1992), pp. 66–70. ISSN: 0362-1340. DOI: 10.1145/130854.130862 (cited on page 29).

[16] J.-L. Béchennec, M. Briday, S. Faucou, and Y. Trinquet. „Trampoline: An OpenSource Implementation of the OSEK/VDX RTOS Specification.“ In: *IEEE Conference on Emerging Technologies and Factory Automation, 2006. ETFA '06.* (Prague, Czech Republic). Washington, DC, USA: IEEE Computer Society Press, Sept. 2006, pp. 62–69. ISBN: 0-7803-9758-4. DOI: 10.1109/ETFA.2006.355432 (cited on page 60).

[17] A. D. Birrell and B. J. Nelson. „Implementing Remote Procedure Calls.“ In: *ACM Transactions on Computer Systems* 2.1 (Feb. 1984), pp. 39–59. ISSN: 0734-2071. DOI: 10.1145/2080.357392 (cited on page 60).

[18] S. Biswas, T. Carley, M. Simpson, B. Middha, and R. Barua. „Memory Overflow Protection for Embedded Systems Using Run-Time Checks, Reuse, and Compression.“ In: *ACM Transactions on Embedded Computing Systems (TECS)* 5 (4 Nov. 2006), pp. 719–752. ISSN: 1539-9087. DOI: 10.1145/1196636.1196637 (cited on page 21).

[19] H.-J. Boehm and M. Weiser. „Garbage Collection in an Uncooperative Environment.“ In: *Software: Practice and Experience* 18.9 (1988), pp. 807–820. ISSN: 0038-0644. DOI: 10.1002/spe.4380180902 (cited on page 29).

[20] M. Broy. „Challenges in Automotive Software Engineering.“ In: *Proceedings of the 28th International Conference on Software Engineering (ICSE ’06).* (Shanghai, China). New York, NY, USA: ACM Press, 2006, pp. 33–42. ISBN: 1-59593-375-1. DOI: 10.1145/1134285.1134292 (cited on page 2).

[21] S. Chandra and T. Reps. „Physical Type Checking for C.“ In: *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering.* (Toulouse, France). PASTE ’99. New York, NY, USA: ACM Press, 1999, pp. 66–75. ISBN: 1-58113-137-2. DOI: 10.1145/316158.316183 (cited on page 23).

[22] Y. Chang and A. Wellings. „Hard Real-Time Hybrid Garbage Collection with Low Memory Requirements.“ In: *Proceedings of the 27th IEEE International Symposium on Real-Time Systems (RTSS ’06).* (Rio de Janeiro, Brazil). Washington, DC, USA: IEEE Computer Society Press, Dec. 2006, pp. 77–88. ISBN: 0-7695-2761-2. DOI: 10.1109/RTSS.2006.25 (cited on page 29).

[23] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. „Sharing and Protection in a Single-Address-Space Operating System.“ In: *ACM Transactions on Computer Systems* 12.4 (1994), pp. 271–307. DOI: 10.1145/195792.195795 (cited on page 69).

[24] D. Chen, A. Messer, P. Bernadat, G. Fu, Z. Dimitrijevic, D. J. F. Lie, D. Mannaru, A. Riska, and D. Milojicic. „JVM Susceptibility to Memory Errors.“ In: *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium.* (Monterey, CA, USA). Berkeley, CA, USA: USENIX Association, Apr. 2001, pp. 67–78. ISBN: 1-880446-11-1 (cited on page 156).

[25] G. Chen, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. „Analyzing Heap Error Behavior in Embedded JVM Environments.“ In: *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS ’04).* (Stockholm, Sweden). New York, NY, USA: ACM Press, 2004, pp. 230–235. ISBN: 1-58113- 937-3. DOI: 10.1145/1016720.1016775 (cited on page 156).

[26] G. Chen and M. Kandemir. „Improving Java Virtual Machine Reliability for Memory-Constrained Embedded Systems.“ In: *Proceedings of the 42nd annual Design Automation Conference.* (Anaheim, CA, USA). DAC ’05. New York, NY, USA: ACM Press, 2005, pp. 690–695. ISBN: 1-59593-058-2. DOI: 10.1145/1065579.1065761 (cited on page 156).

[27] J. Condit, M. Harren, Z. R. Anderson, D. Gay, and G. C. Necula. „Dependent Types for Low-Level Programming.“ In: *ESOP*. Ed. by R. D. Nicola. Vol. 4421. Lecture Notes in Computer Science. Heidelberg, Germany: Springer-Verlag,

2007, pp. 520–535. ISBN: 978-3-540-71314-2. DOI: 10.1007/978-3-540-71316-6_35 (cited on page 25).

[28] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. „CCured in the Real World.“ In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’03).* (San Diego, CA, USA). New York, NY, USA: ACM Press, 2003, pp. 232–244. ISBN: 1-58113-662-5. DOI: 10.1145/781131.781157 (cited on page 22).

[29] K. D. Cooper, M. W. Hall, and K. Kennedy. „Procedure Cloning.“ In: *Proceedings of the 1992 International Conference on Computer Languages.* (Oakland, CA, USA). Washington, DC, USA: IEEE Computer Society Press, Apr. 1992, pp. 96–105. DOI: 10.1109/ICCL.1992.185472 (cited on page 22).

[30] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. „Efficient Memory Safety for TinyOS.“ In: *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems.* (Sydney, Australia). New York, NY, USA: ACM Press, 2007, pp. 205–218. ISBN: 978-1-59593-763-6. DOI: 10.1145/1322263.1322283 (cited on page 26).

[31] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. „Efficiently Computing Static Single Assignment Form and the Control Dependence Graph.“ In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.4 (Oct. 1991), pp. 451–490. DOI: 10.1145/115372.115320 (cited on pages 21, 22, 87).

[32] J. Dean, D. Grove, and C. Chambers. „Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis.“ In: *Lecture Notes in Computer Science* 952 (1995), pp. 77–101 (cited on page 87).

[33] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. „HardBound: Architectural Support for Spatial Safety of the C Programming Language.“ In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’08).* (Seattle, WA, USA). New York, NY, USA: ACM Press, 2008, pp. 103–114. ISBN: 978-1-59593-958-6. DOI: 10.1145/1346281.1346295 (cited on page 16).

[34] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. „Memory Safety Without Garbage Collection for Embedded Applications.“ In: *Transactions on Embedded Computing Systems* 4.4 (1 Feb. 2005), pp. 73–111. ISSN: 1539-9087. DOI: 10.1145/1053271.1053275 (cited on pages 23, 28).

[35] C. Erhardt. „A Control-Flow-Sensitive Analysis and Optimization Framework for the KESO Multi-JVM.“ Diplomarbeit. Friedrich-Alexander University Erlangen-Nuremberg, Mar. 2011 (cited on page 87).

[36] C. Erhardt, M. Stilkerich, D. Lohmann, and W. Schröder-Preikschat. „Exploiting Static Application Knowledge in a Java Compiler for Embedded Systems: A Case Study." In: *JTRES '11: Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems.* (York, UK). New York, NY, USA: ACM Press, 2011, pp. 96–105. ISBN: 978-1-4503-0731-4. DOI: 10.1145/2043910.2043927 (cited on page 7).

[37] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. „XFI: Software Guards for System Address Spaces." In: *7th Symposium on Operating System Design and Implementation (OSDI '06).* (Seattle, WA, USA). Berkeley, CA, USA: USENIX Association, 2006, pp. 75–88. ISBN: 1-931971-47-1 (cited on page 20).

[38] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. „Language Support for Fast and Reliable Message-Based Communication in Singularity OS." In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06).* (Leuven, Belgium). New York, NY, USA: ACM Press, 2006, pp. 177–190. ISBN: 1-59593-322-0. DOI: 10.1145/1217935.1217953 (cited on page 27).

[39] E. M. Gagnon and L. J. Hendren. „SableVM: A Research Framework for the Efficient Execution of Java Bytecode." In: *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium.* (Monterey, CA, USA). Berkeley, CA, USA: USENIX Association, Apr. 2001, pp. 27–40. ISBN: 1-880446-11-1 (cited on page 46).

[40] D. Gay, R. Ennals, and E. Brewer. „Safe Manual Memory Management." In: *ISMM '07: Proceedings of the 5th International Symposium on Memory Management.* (Montreal, Quebec, Canada). New York, NY, USA: ACM Press, 2007, pp. 2–14. ISBN: 978-1-59593-893-0. DOI: 10.1145/1296907.1296911 (cited on page 26).

[41] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder. „The JX Operating System." In: *Proceedings of the 2002 USENIX Annual Technical Conference.* Berkeley, CA, USA: USENIX Association, June 2002, pp. 45–58. ISBN: 1-880446-00-6 (cited on pages 27, 69).

[42] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. „Region-Based Memory Management in Cyclone." In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02).* (Berlin, Germany). New York, NY, USA: ACM Press, 2002, pp. 282–293. ISBN: 1-58113-463-0. DOI: 10.1145/512529.512563 (cited on page 25).

[43] L. Gu and J. A. Stankovic. „t-kernel: A Naturalizing OS Kernel for Low-Power Cost-Effective Computers." In: *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05).* (Brighton, UK). New York, NY, USA: ACM Press, 2005. ISBN: 1-59593-079-5. DOI: 10.1145/1095810.1118588 (cited on page 20).

[44] L. Gu and J. A. Stankovic. „t-kernel: Providing Reliable OS Support to Wireless Sensor Networks." In: *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems.* (Boulder, CO, USA). New York, NY, USA: ACM Press, 2006. ISBN: 1-59593-343-3. DOI: 10.1145/1182807.1182809 (cited on page 20).

[45] R. Gupta. „A Fresh Look at Optimizing Array Bound Checking." In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '90).* (White Plains, NY, USA). PLDI '90. New York, NY, USA: ACM Press, 1990, pp. 272–282. ISBN: 0-89791-364-7. DOI: 10.1145/93542.93581 (cited on page 47).

[46] G. Haddad, F. Hussain, and G. T. Leavens. „The Design of SafeJML, a Specification Language for SCJ with Support for WCET Specification." In: *JTRES '10: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems.* (Prague, Czech Republic). New York, NY, USA: ACM Press, 2010, pp. 155–163. ISBN: 978-1-4503-0122-0. DOI: 10.1145/1850771.1850793 (cited on page 119).

[47] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. „A Dynamic Operating System for Sensor Networks." In: *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services (MobiSys '05).* (Seattle, WA, USA). New York, NY, USA: ACM Press, June 2005, pp. 163–176. ISBN: 1-931971-31-5. DOI: 10.1145/1067170.1067188 (cited on page 20).

[48] B. Hardung, T. Kölzow, and A. Krüger. „Reuse of Software in Distributed Embedded Automotive Systems." In: *Proceedings of the 4th ACM Conference on Embedded Software (EMSOFT '04).* (Pisa, Italy). New York, NY, USA: ACM Press, Sept. 2004, pp. 203–210 (cited on pages 2, 5).

[49] C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. „Implementing Multiple Protection Domains in Java." In: *Proc. of the 1998 USENIX Annual Technical Conference.* 1998, pp. 259–270 (cited on page 27).

[50] F. Henderson. „Accurate Garbage Collection in an Uncooperative Environment." In: *ISMM '02: Proceedings of the 3rd International Symposium on Memory Management.* (Berlin, Germany). New York, NY, USA: ACM Press, 2002, pp. 150–156. ISBN: 1-58113-539-4. DOI: 10.1145/512429.512449 (cited on pages 46, 133).

[51] G. C. Hunt and J. R. Larus. „Singularity: Rethinking the Software Stack." In: *ACM SIGOPS Operating Systems Review* 41.2 (2007), pp. 37–49. ISSN: 0163-5980. DOI: 10.1145/1243418.1243424 (cited on page 27).

[52] B. L. Jacob and T. N. Mudge. „A Look at Several Memory Management Units, TLB-Refill Mechanisms, and Page Table Organizations." In: *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII).* (San Jose, CA, USA). New

York, NY, USA: ACM Press, 1998, pp. 295–306. ISBN: 1-58113-107-0. DOI: 10.1145/291069.291065 (cited on page 17).

[53] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. „Cyclone: A Safe Dialect of C.“ In: *Proceedings of the 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2002, pp. 275–288. ISBN: 1-880446-00-6 (cited on pages 24, 25).

[54] M. S. Johnstone. „Non-Compacting Memory Allocation and Real-Time Garbage Collection.“ Supervisor: Paul R. Wilson. PhD thesis. 1997. ISBN: 0-591-77415-1 (cited on page 29).

[55] *JSR 1: Real-time Specification for Java*. Sun Microsystems JCP. Palo Alto, CA, USA, May 2006. URL: http://jcp.org/en/jsr/detail?id=1 (cited on pages 24, 27, 157).

[56] *JSR 121: Application Isolation API Specification*. Sun Microsystems JCP. Palo Alto, CA, USA, June 2006. URL: http://jcp.org/aboutJava/communityprocess/final/jsr121/ (cited on pages 27, 67, 69).

[57] *JSR 139: Connected Limited Device Configuration 1.1*. Sun Microsystems JCP. Palo Alto, CA, USA, Mar. 2003. URL: http://jcp.org/aboutJava/communityprocess/final/jsr139/ (cited on page 66).

[58] *JSR 218: Connected Device Configuration (CDC) 1.1*. Sun Microsystems JCP. Palo Alto, CA, USA, Aug. 2005. URL: http://www.jcp.org/en/jsr/detail?id=218 (cited on page 66).

[59] *JSR 271: Mobile Information Device Profile 3*. Sun Microsystems JCP. Palo Alto, CA, USA, Dec. 2009. URL: http://www.jcp.org/en/jsr/detail?id=271 (cited on page 66).

[60] *JSR-302: Safety Critical Java Technology Specification (Version 0.78)*. Oracle JCP. San Francisco, CA, USA, Oct. 2010. URL: http://jcp.org/en/jsr/detail?id=302 (cited on page 27).

[61] 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: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*. (Madrid, Spain). New York, NY, USA: ACM Press, 2009, pp. 41–50. ISBN: 978-1-60558-732-5. DOI: 10.1145/1620405.1620412 (cited on pages 51, 117, 119).

[62] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. „Aspect-Oriented Programming.“ In: *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*. Ed. by M. Aksit and S. Matsuoka. Vol. 1241. Lecture Notes in Computer Science. Springer-Verlag, June 1997, pp. 220–242 (cited on pages 60, 61).

[63] P. Kolte and M. Wolfe. „Elimination of Redundant Array Subscript Range Checks." In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '95)*. (La Jolla, CA, USA). PLDI '95. New York, NY, USA: ACM Press, 1995, pp. 270–278. ISBN: 0-89791-697-2. DOI: 10.1145/207110.207160 (cited on page 47).

[64] S. Korsholm. „Flash Memory in Embedded Java Programs." In: *JTRES '11: Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*. (York, UK). New York, NY, USA: ACM Press, 2011, pp. 116–124. ISBN: 978-1-4503-0731-4. DOI: 10.1145/2043910.2043930 (cited on page 157).

[65] R. Kumar, E. Kohler, and M. Srivastava. „Harbor: Software-based Memory Protection for Sensor Nodes." In: *IPSN '07: Proceedings of the 6th International Conference on Information Processing in Sensor Networks*. (Cambridge, MA, USA). New York, NY, USA: ACM Press, 2007, pp. 340–349. ISBN: 978-1-59593-638-X. DOI: 10.1145/1236360.1236404 (cited on pages 15, 20).

[66] R. Kumar, A. Singhania, A. Castner, E. Kohler, and M. Srivastava. „A System for Coarse Grained Memory Protection in Tiny Embedded Processors." In: *Proceedings of the 44th annual Design Automation Conference*. (San Diego, CA, USA). New York, NY, USA: ACM Press, 2007, pp. 218–223. ISBN: 978-1-59593-627-1. DOI: 10.1145/1278480.1278534 (cited on pages 15, 17, 20).

[67] *J2ME Building Blocks for Mobile Devices — White Paper on KVM and the Connected, Limited Device Configuration (CLDC)*. May 2000. URL: http://java.sun.com/products/cldc/wp/KVMwp.pdf (cited on page 66).

[68] C. Lattner and V. Adve. „LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. (Palo Alto, CA, USA). Washington, DC, USA: IEEE Computer Society Press, Mar. 2004 (cited on pages 21, 22).

[69] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. 2nd. The Java Series. Addison-Wesley, 1999. ISBN: 0-201-43294-3 (cited on page 68).

[70] 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: *Proceedings of the 2009 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, June 2009, pp. 215–228. ISBN: 978-1-931971-68-3 (cited on page 60).

[71] D. Lohmann, O. Spinczyk, W. Hofer, and W. Schröder-Preikschat. „The Aspect-Aware Design and Implementation of the CiAO Operating-System Family." In: *Transactions on AOSD IX*. Ed. by M. Haupt and E. Wohlstadter. Lecture Notes in Computer Science. (To Appear). Springer-Verlag, 2012, pp. 1–49 (cited on page 63).

[72] D. Lohmann, J. Streicher, W. Hofer, O. Spinczyk, and W. Schröder-Preikschat. „Configurable Memory Protection by Aspects.“ In: *Proceedings of the 4th Workshop on Programming Languages and Operating Systems (PLOS '07)*. (Stevenson, WA, USA). New York, NY, USA: ACM Press, Oct. 2007, pp. 1–5. ISBN: 978-1-59593-922-7. DOI: 10.1145/1376789.1376794 (cited on pages 74, 79).

[73] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. „Simics: A Full System Simulation Platform.“ In: *IEEE Computer* 35.2 (Feb. 2002), pp. 50–58. ISSN: 0018-9162. DOI: 10.1109/2.982916 (cited on page 16).

[74] V. Markstein, J. Cocke, and P. Markstein. „Optimization of Range Checking.“ In: *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*. (Boston, MA, USA). SIGPLAN '82. New York, NY, USA: ACM Press, 1982, pp. 114–119. ISBN: 0-89791-074-5. DOI: 10.1145/800230.806986 (cited on page 47).

[75] S. McConnell. *Code Complete*. Second. Microsoft Press, 2004. ISBN: 0-7356-1967-0 (cited on page 50).

[76] M. D. McIlroy. „Mass-Produced Software Components.“ In: *Proceedings of the 1st International Conference on Software Engineering*. (Garmisch Pattenkirchen, Germany). 1968, pp. 88–98 (cited on page 95).

[77] *Microcontroller Pocket Guide*. Infineon Technologies AG. St.-Martin-Str. 53, 81669 München, Germany, Feb. 2010 (cited on page 17).

[78] J. Mössinger. „Software in Automotive Systems.“ In: *Software, IEEE* 27.2 (Mar. 2010), pp. 92–94. ISSN: 0740-7459. DOI: 10.1109/MS.2010.55 (cited on page 2).

[79] S. Mukherjee. *Architecture Design for Soft Errors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 978-0-12-369529-1 (cited on page 156).

[80] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. „SoftBound: Highly Compatible and Complete Spatial Memory Safety for C.“ In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. (Dublin, Ireland). New York, NY, USA: ACM Press, 2009, pp. 245–258. ISBN: 978-1-60558-392-1. DOI: 10.1145/1542476.1542504 (cited on page 22).

[81] G. C. Necula, S. McPeak, and W. Weimer. „CCured: Type-Safe Retrofitting of Legacy Code.“ In: *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. (Portland, OR, USA). New York, NY, USA: ACM Press, 2002, pp. 128–139. ISBN: 1-58113-450-9. DOI: 10.1145/503272.503286 (cited on page 22).

[82] K. Nilsen. „Ada-Java Middleware for Legacy Software Modernization.“ In: *JTRES ’10: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems.* (Prague, Czech Republic). New York, NY, USA: ACM Press, 2010, pp. 85–94. ISBN: 978-1-4503-0122-0. DOI: 10.1145/1850771.1850785 (cited on pages 27, 33).

[83] Y. Oiwa. „Implementation of the Memory-Safe Full ANSI-C Compiler.“ In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’09).* (Dublin, Ireland). New York, NY, USA: ACM Press, 2009, pp. 259–269. ISBN: 978-1-60558-392-1. DOI: 10.1145/1542476.1542505 (cited on page 21).

[84] OSEK/VDX Group. *OSEK Implementation Language Specification 2.5.* Tech. rep. http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf, visited 2009-09-09. OSEK/VDX Group, 2004 (cited on page 60).

[85] OSEK/VDX Group. *OSEK/VDX Communication 3.0.3.* Tech. rep. http://portal.osek-vdx.org/files/pdf/specs/osekcom303.pdf. OSEK/VDX Group, July 2004 (cited on page 105).

[86] OSEK/VDX Group. *Operating System Specification 2.2.3.* Tech. rep. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, visited 2011-08-17. OSEK/VDX Group, Feb. 2005 (cited on page 3).

[87] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard. „Stopless: A Real-Time Garbage Collector for Multiprocessors.“ In: *ISMM ’07: Proceedings of the 5th International Symposium on Memory Management.* (Montreal, Quebec, Canada). New York, NY, USA: ACM Press, 2007, pp. 159–172. ISBN: 978-1-59593-893-0. DOI: 10.1145/1296907.1296927 (cited on page 29).

[88] F. Pizlo, E. Petrank, and B. Steensgaard. „A Study of Concurrent Real-Time Garbage Collectors.“ In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’08).* (Tucson, AZ, USA). New York, NY, USA: ACM Press, 2008, pp. 33–44. ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375587 (cited on page 29).

[89] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. Vitek. „High-Level Programming of Embedded Hard Real-Time Devices.“ In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2010 (EuroSys ’10).* (Paris, France). New York, NY, USA: ACM Press, Apr. 2010, pp. 69–82. ISBN: 978-1-60558-577-2. DOI: 10.1145/1755913.1755922 (cited on pages 67, 119).

[90] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. „Schism: Fragmentation-Tolerant Real-Time Garbage Collection.“ In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’10).* (Toronto, Ontario, Canada). New York, NY, USA: ACM Press, 2010, pp. 146–159. ISBN: 978-1-4503-0019-3. DOI: 10.1145/1806596.1806615 (cited on page 29).

[91] A. Plsek, L. Zhao, V. H. Sahin, D. Tang, T. Kalibera, and J. Vitek. „Developing Safety Critical Java Applications with oSCJ/L0." In: *JTRES '10: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems.* (Prague, Czech Republic). New York, NY, USA: ACM Press, 2010, pp. 95–101. ISBN: 978-1-4503-0122-0. DOI: 10.1145/1850771.1850786 (cited on page 119).

[93] W. Puffitsch, B. Huber, and M. Schoeberl. „Worst-Case Analysis of Heap Allocations." In: *Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part II.* (Heraklion, Crete, Greece). Heidelberg, Germany: Springer-Verlag, 2010, pp. 464–478. ISBN: 3-642-16560-5, 978-3-642-16560-3 (cited on page 119).

[94] *PXROS-HR User's Guide (Version 1.1).* HighTec EDV-Systeme GmbH. Feldmannstraße 98, 66119 Saarbrücken, Germany, Jan. 2011 (cited on pages 42, 77, 105).

[95] *Java RMI — Distributed Computing for Java.* White Paper, Sun Microsystems Inc. (Cited on page 69).

[96] S. G. Robertz and R. Henriksson. „Time-Triggered Garbage Collection: Robust and Adaptive Real-Time GC Scheduling for Embedded Systems." In: *Proceedings of the 2003 Joint Conference on Languages, Compilers and Tools for Embedded Systems & Soft. and Compilers for Embedded Systems (LCTES/SCOPES '03).* (San Diego, CA, USA). New York, NY, USA: ACM Press, 2003, pp. 93–102. ISBN: 1-58113-647-1. DOI: 10.1145/780732.780745 (cited on page 29).

[97] M. Schoeberl, S. Korsholm, C. Thalinger, and A. P. Ravn. „Hardware Objects for Java." In: *Proceedings of the 11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '08).* (Orlando, FL, USA). Washington, DC, USA: IEEE Computer Society Press, 2008, pp. 445–452. ISBN: 978-0-7695-3132-8. DOI: 10.1109/ISORC.2008.63 (cited on page 70).

[98] T. Schoofs, E. Jenn, S. Leriche, K. Nilsen, L. Gauthier, and M. Richard-Foy. „Use of PERC Pico in the AIDA Avionics Platform." In: *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems.* (Madrid, Spain). New York, NY, USA: ACM Press, 2009, pp. 169–178. ISBN: 978-1-60558-732-5. DOI: 10.1145/1620405.1620429 (cited on page 67).

[99] O. Shivers. „Control Flow Analysis in Scheme." In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '88).* (Atlanta, GA, USA). PLDI '88. New York, NY, USA: ACM Press, 1988, pp. 164–174. ISBN: 0-89791-269-1. DOI: 10.1145/53990.54007 (cited on page 87).

[100] F. Siebert. „Realtime Garbage Collection in the JamaicaVM 3.0.“ In: *JTRES '07: Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems.* (Vienna, Austria). New York, NY, USA: ACM Press, 2007, pp. 94–103. ISBN: 978-59593-813-8. DOI: 10.1145/1288940.1288954 (cited on page 29).

[101] F. Siebert and A. Walter. „Deterministic Execution of Java's Primitive Bytecode Operations.“ In: *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium.* (Monterey, CA, USA). Berkeley, CA, USA: USENIX Association, Apr. 2001, pp. 18–18. ISBN: 1-880446-11-1 (cited on page 67).

[102] M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. „Coping with Type Casts in C.“ In: *Proceedings of the 7th European Software Engineering Conference.* (Toulouse, France). ESEC/FSE-7. Heidelberg, Germany: Springer-Verlag, 1999, pp. 180–198. ISBN: 3-540-66538-2. DOI: 10.1145/318773.318942 (cited on page 23).

[103] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. „Java™ on the Bare Metal of Wireless Sensor Devices: The Squawk Java Virtual Machine.“ In: *Proceedings of the 2nd USENIX International Conf. on Virtual Execution Environments (VEE '06).* (Ottawa, Ontario, Canada). New York, NY, USA: ACM Press, 2006, pp. 78–88. ISBN: 1-59593-332-6. DOI: 10.1145/1134760.1134773 (cited on pages 27, 67).

[104] M. Simpson, B. Middha, and R. Barua. „Segment Protection for Embedded Systems Using Run-Time Checks.“ In: *Proceedings of the 2005 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES '05).* (San Francisco, CA, USA). New York, NY, USA: ACM Press, 2005, pp. 66–77. ISBN: 1-59593-149-X. DOI: 10.1145/1086297.1086307 (cited on page 21).

[105] O. Spinczyk, D. Lohmann, and M. Urban. „AspectC++: An AOP Extension for C++.“ In: *Software Developers Journal* 5 (May 2005), pp. 68–76. URL: http://www.aspectc.org/fileadmin/publications/sdj-2005-en.pdf (cited on pages 60, 61).

[106] D. Spoonhower, J. Auerbach, D. F. Bacon, P. Cheng, and D. Grove. „Eventrons: A Safe Programming Construct for High-Frequency Hard Real-Time Applications.“ In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06).* (Ottawa, Ontario, Canada). New York, NY, USA: ACM Press, 2006, pp. 283–294. ISBN: 1-59593-320-4. DOI: 10.1145/1133981.1134015 (cited on page 27).

[107] J. H. Spring, F. Pizlo, R. Guerraoui, and J. Vitek. „Reflexes: Abstractions for Highly Responsive Systems.“ In: *Proceedings of the 3rd USENIX International Conf. on Virtual Execution Environments (VEE '07).* (San Diego, CA, USA). New York, NY, USA: ACM Press, 2007, pp. 191–201. ISBN: 978-1-59593-630-1. DOI: 10.1145/1254810.1254837 (cited on page 27).

[108] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. „Streamflex: High-Throughput Stream Programming in Java.“ In: *Proceedings of the 22nd ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '07).* (Montreal, Quebec, Canada). New York, NY, USA: ACM Press, 2007, pp. 211–228. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297027.1297043 (cited on page 27).

[109] 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.* (Venice, Italy). SAS '99. Heidelberg, Germany: Springer-Verlag, 1999, pp. 194–210. ISBN: 3-540-66459-9 (cited on page 87).

[110] M. Stilkerich, D. Lohmann, and W. Schröder-Preikschat. „Gradual Software-Based Memory Protection.“ In: *Proceedings of the Workshop on Isolation and Integration for Dependable Systems (IIDS '10).* (Paris, France). New York, NY, USA: ACM Press, 2010. ISBN: 978-1-4503-0120-6 (cited on page 7).

[111] M. Stilkerich, D. Lohmann, and W. Schröder-Preikschat. „Memory Protection at Option.“ In: *Proceedings of the 1st Workshop on Critical Automotive Applications: Robustness & Safety.* (Valencia, Spain). New York, NY, USA: ACM Press, 2010, pp. 17–20. ISBN: 978-1-60558-915-2. DOI: 10.1145/1772643.1772649 (cited on page 7).

[112] M. Stilkerich, J. Schedel, P. Ulbrich, W. Schröder-Preikschat, and D. Lohmann. „Escaping the Bonds of the Legacy: Step-Wise Migration to a Type-Safe Language in Safety-Critical Embedded Systems.“ In: *Proceedings of the 14th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '11).* (Newport Beach, CA, USA). Ed. by G. Karsai, A. Polze, D.-H. Kim, and W. Steiner. IEEE Computer Society Press, Mar. 2011, pp. 163–170. ISBN: 978-0-7695-4368-0. DOI: 10.1109/ISORC.2011.29 (cited on page 7).

[113] M. Stilkerich, I. Thomm, C. Wawersich, and W. Schröder-Preikschat. „Tailor-Made JVMs for Statically Configured Embedded Systems.“ In: *Concurrency and Computation: Practice and Experience* 24.8 (2012), pp. 789–812. ISSN: 1532-0634. DOI: 10.1002/cpe.1755 (cited on pages 7, 27, 67, 116, 151).

[114] M. Stilkerich, C. Wawersich, W. Schröder-Preikschat, A. Gal, and M. Franz. „OSEK/VDX API for Java.“ In: *Proceedings of the Linguistic Support for Modern Operating Systems ASPLOS XII Workshop (PLOS '06).* (San Jose, CA, USA). New York, NY, USA: ACM Press, Oct. 2006, pp. 13–17. ISBN: 1-59593-577-0. DOI: 10.1145/1215995.1215999 (cited on page 7).

[115] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. „Practical Virtual Method Call Resolution for Java.“ In: *ACM SIGPLAN Notices* 35.10 (2000), pp. 264–280 (cited on page 87).

[116] *TC1796 User's Manual (V2.0).* Infineon Technologies AG. St.-Martin-Str. 53, 81669 München, Germany, July 2007 (cited on page 48).

[117] D. Tennenhouse. „Proactive Computing.“ In: *Communications of the ACM* (May 2000), pp. 43–45 (cited on page 1).

[118] 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: Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems.* (York, UK). New York, NY, USA: ACM Press, 2011, pp. 87–95. ISBN: 978-1-4503-0731-4. DOI: 10.1145/2043910.2043925 (cited on page 7).

[119] I. Thomm, M. Stilkerich, C. Wawersich, and W. Schröder-Preikschat. „KESO: An Open-Source Multi-JVM for Deeply Embedded Systems.“ In: *JTRES ’10: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems.* (Prague, Czech Republic). New York, NY, USA: ACM Press, 2010, pp. 109–119. ISBN: 978-1-4503-0122-0. DOI: 10.1145/1850771.1850788 (cited on page 7).

[120] 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).* (TaiChung, Taiwan). New York, NY, USA: ACM Press, 2011, pp. 380–396. ISBN: 978-1-4503-0113-8 (cited on page 39).

[121] M. Urban and O. Spinczyk. *AspectC++ Language Reference (Version 1.7).* http://www.aspectc.org/fileadmin/documentation/ac-languageref.pdf, visited 2012-04-19. pure-systems GmbH. Agnetenstr. 14, 39106 Magdeburg, Germany, Apr. 2011 (cited on page 61).

[122] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. „Efficient Software-Based Fault Isolation.“ In: *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP ’93).* (Asheville, NC, USA). New York, NY, USA: ACM Press, 1993, pp. 203–216. ISBN: 0-89791-632-8. DOI: 10.1145/168619.168635 (cited on page 19).

[123] C. Wawersich. „KESO: Konstruktiver Speicherschutz für Eingebettete Systeme.“ Dissertation. Friedrich-Alexander University Erlangen-Nuremberg, Oct. 2008 (cited on pages 67, 87).

[124] C. Wawersich, M. Stilkerich, and W. Schröder-Preikschat. „An OSEK/VDX-Based Multi-JVM for Automotive Appliances.“ In: *Embedded System Design: Topics, Techniques and Trends.* (Irvine, CA , USA). IFIP International Federation for Information Processing. Boston: Springer-Verlag, 2007, pp. 85–96. ISBN: 978-0-387-72257-3 (cited on page 7).

[125] P. R. Wilson. „Uniprocessor Garbage Collection Techniques.“ In: *Proceedings of the International Workshop on Memory Management (IWMM ’92).* (St. Malo, France). Ed. by Y. Bekkers and J. Cohen. Vol. 637. Lecture Notes in Computer Science. Heidelberg, Germany: Springer-Verlag, Sept. 17–19, 1992, pp. 1–42. ISBN: 3-540-55940-X (cited on page 68).

[126] E. Witchel, J. Cates, and K. Asanović. „Mondrian Memory Protection.“ In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’02)*. (San Jose, CA, USA). New York, NY, USA: ACM Press, 2002, pp. 304–316. ISBN: 1-58113-574-2. DOI: 10.1145/605397.605429 (cited on page 14).

[127] E. Witchel, J. Rhee, and K. Asanović. „Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection.“ In: *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP ’05)*. (Brighton, UK). New York, NY, USA: ACM Press, 2005, pp. 31–44. ISBN: 1-59593-079-5. DOI: 10.1145/1095810.1095814 (cited on page 15).

[128] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. „SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques.“ In: *7th Symposium on Operating System Design and Implementation (OSDI ’06)*. (Seattle, WA, USA). Berkeley, CA, USA: USENIX Association, 2006, pp. 45–60 (cited on page 26).