Memory Protection at Option*

Michael Stilkerich Daniel Lohmann Wolfgang Schröder-Preikschat
{stilkerich,lohmann,wosch}@cs.fau.de
Friedrich-Alexander University Erlangen-Nuremberg, Germany

ABSTRACT

There is hardware- and software-based memory protection that can improve the dependability of software systems. The two variants vary in the degree of protection and the amount and sites of overhead. The decision for a particular mechanism therefore highly depends on the application and deployment scenario.

We propose a system suited for deeply embedded systems that allows to choose among no protection, software-based protection, hardware-based protection or a combination of the two without the need to change the application. In this paper, we present the current state of this work and support our claim that the best-suited memory protection type depends on the application by a preliminary evaluation.

1. INTRODUCTION

Electronic support functions in cars have rapidly developed in the past decade [2]. A modern mid-class car is equipped with about 80 electronic control units (ECUs), which communicate with each other through up to five different bus systems. This development is problematic in several aspects: the multitude of ECUs and wires that connect the ECUs with each other are costly, especially with the ever increasing copper price; the wires with about 50 kg noticeably contribute to the weight of the car; the connectors that attach the wires to the ECUs are known to be fault-prone and a major cause of hardware defects. To address these issues, the automotive industry is currently consolidating the number of ECUs in a car by replacing multiple ECUs with fewer, but more powerful microcontrollers, where multiple applications that formerly ran on a dedicated ECU now share a common microcontroller.

The coexistence of multiple applications on a microcontroller introduces the requirement to the underlying system software to enable the isolation of the applications with respect to different aspects. One of the key aspects is memory protection (MP), which is one of the points in that AUTOSAR OS [1] improves over its predecessor OSEK OS [10] in that it mandates write-protection in some scalability classes. The protection model of AUTOSAR is region-based and requires the presence of a memory protection unit (MPU) on the target microcontroller. On controllers without an MPU, MP is not supported.

Besides hardware-based MP there is software-based MP, where safety is constructively ensured by the executing program itself. Normally this is achieved by writing software in a type-safe programming language or employing run-time safety checks. Past research has shown that software-based protection based on the use of a type-safe high-level language [3, 4, 9] is superior in terms of overhead compared to lower level approaches [6], as it enables high-level static analyses that can validate many safety checks at compile time.

Both of these approaches have advantages and disadvantages compared to each other. MPU-based MP requires hardware support, has a limited set of range registers which limits flexibility, but requires only little effort to support legacy applications, is efficient in terms of space overhead and execution time of code that stays within a particular protection domain. Software-based protection introduces runtime-checks that add to the ROM size and execution time, may require applications to be (re)written in a safe language and is more susceptible to transient or permanent hardware faults than hardware-based protection, but allows very efficient domain transitions and inter-domain communication (IDC), can run on any hardware and can also detect memory access errors within a particular application. In addition, software-based MP is not limited to a particular number of memory ranges and does not require a particular placement of data in the address space. Which of the approaches is the better suited one highly depends on the application, the target platform and the environment the application is deployed in. Memory protection is not a one or the other decision and should be a configurable property of the system software.

In this paper, we present an approach, that, to the best of our knowledge, is the first system that supports both hardware-based and software-based MP as an optional and configurable non functional property. The two key contributions of our approach are (1) the ability, to directly compare the cost of MPU-based protection and the cost of software-based protection in the embedded systems domain using the same, unmodified application, and (2) the possibility to configure the level of MP as required by the application.

*This work was partly supported by the DFG under grant no. SCHR 603/4 and SCHR 603/7-1

CARS '2010, April 27, Valencia, Spain

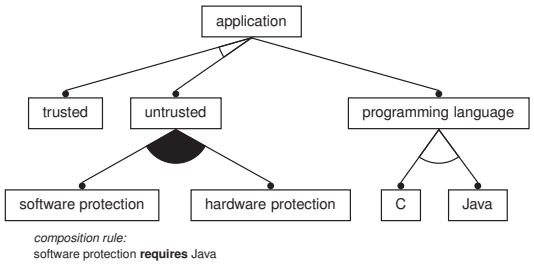

Figure 1: Memory Protection Variants

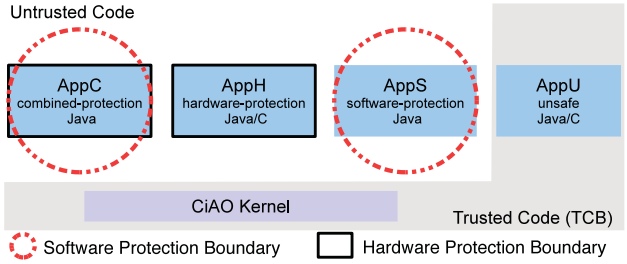

Figure 2: Application Variants

2. DESIGN

Our protection model is designed according to the needs of the domain of embedded systems and similar to that of AUTOSAR OS with some extensions. For hardware-based MP, we require the presence of an MPU that allows to restrict memory access to regions of the address space. Memory management units that manage the memory at the finer granularity of pages are found on only very few microcontrollers.

The AUTOSAR OS protection model distinguishes trusted and non-trusted applications. An application comprises a number of tasks and ISRs. The data that belongs to an application is the stacks of its tasks and ISRs plus the private data segment of the application. Trusted applications run with memory protection features disabled and thus become part of the trusted computing base, whereas non-trusted applications only have write access to their own data. Read protection additionally restricts read accesses of an application to its own data and is optional in AUTOSAR. Trusted applications may provide services (*trusted functions*) to other applications, which can be used to extend the API of the OS. When a trusted function (TF) is called from a non-trusted application, it will—like an OS service—run with MP disabled. MP is restored to the restrictions that apply for the caller application upon return from the trusted function. We extend this scheme by *non-trusted functions*, that allow non-trusted applications to offer services to other applications. A non-trusted function (NTF) will be executed in the protection context of the application that offers this function, even when called from a trusted application.

2.1 Hardware-based MP

CiAO [7] (CiAO is Aspect-Oriented) is a family of operating systems for embedded applications that has been designed and developed to be highly configurable by AOP [5]. The implementation language of CiAO is AspectC++. CiAO's system design allows to configure even fundamental and highly crosscutting OS policies, thereunder hardware-based MP which we presented in a previous paper [8]. The primary target platform for CiAO is the Infineon TriCore, an architecture of 32-bit microcontrollers mostly used in the automotive industry that also serves as a reference platform for AUTOSAR. CiAO provides an API as defined by AUTOSAR OS. Since CiAO already supports optional hardware-based MP, it provides the ideal infrastructure for this project.

Protection in CiAO can be applied at two different levels: write protection only protects data from modification, whereas the optional read protection also prevents data from being read by other applications. Write protection is sufficient to ensure safety (not security) and allows read-only logical IDC operations (most notably, system services that only query state such as `GetTaskID()`) to be performed without any overhead. Read protection may be desirable in testing environments, since it may detect bugs in software that would not be detected by pure write protection.

2.2 Software-based MP

For software-based MP, we use KESO [11], an ahead-of-time Java Compiler that generates ISO C code from Java bytecode. The main goal of the KESO project is to provide software-based MP tailored towards the domain of embedded systems. KESO does not support all aspects of the Java language and the Java virtual machine and does not provide the full Java standard class library. In particular, KESO requires static applications and does neither support dynamic class loading nor Java reflection. The class library provided by KESO provides access to the system services of an OSEK/VDX or AUTOSAR OS, which is presumed as infrastructure software, and a safe and lightweight mechanism to access device registers from Java code without affecting the type-safety of the program. KESO supports optional garbage collection for applications that want to use dynamic memory allocation.

KESO is a Multi-JVM, that is, it allows tasks to be isolated in different protection domains, each of which appears as a JVM of its own from the application's point of view. This isolation is constructively ensured by preventing any shared, global data among the different domains. Initially, this is established by providing each of these domains with an own set of global data (i.e., the static fields of classes in Java), and later on sustained by preventing object references from being passed to other domain through the available IDC mechanisms. Domains in KESO are conceptually similar to applications in CiAO and are also containers for tasks and ISRs.

2.3 Putting Things Together

By running KESO applications on top of CiAO, we extend the choices of protection types for a particular application. The feature tree in Figure 1 shows the available protection variants for an application (hollow arcs describe an *alternative*, i.e., exactly one child has to be selected; filled arcs describe an *option*, i.e., at least one child has to be selected). Note that while an application that is written in Java can choose from all possible variants, an application that is written in the unsafe C language can only opt for hardware protection. It is nonetheless possible to have both Java and C applications running on the same system.

2.3.1 Protection Variants

Figure 2 shows an example with the four protection vari-

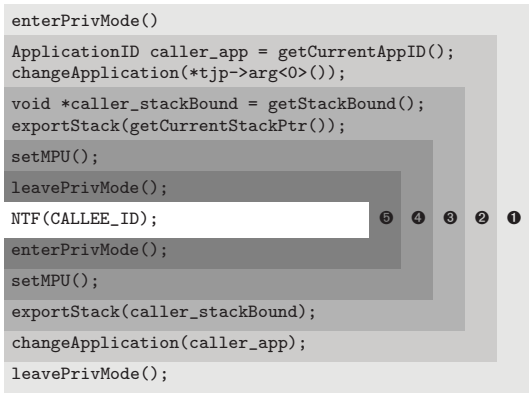

Figure 3: IDC Context Switch Operations

ants that are possible in our system.

No protection (AppU) is an unsafe configuration in which we disable MP in the CiAO OS and additionally instruct the KESO compiler not to generate any runtime safety checks. This configuration does not infer any MP runtime overhead to the application, but still offers the added value of static safety checks compared to an unsafe C(++) application. Since such an application is not subject to any memory access restrictions, it becomes part of the trusted code base and is what is called a trusted application in AUTOSAR.

Software-based Protection (AppS) relies purely on constructive MP, wherefore the application needs to be written in the safe Java language. From the view point of the operating system, such an application is seen as a trusted application that is not subject to hardware-based memory protection, however, logically the application is non-trusted code in the sense that constructive means ensure that memory accesses are limited to the memory regions belonging to the application.

Hardware-based Protection (AppH) uses only CiAO's MPU based MP while no runtime checks are generated by the KESO compiler. This configuration is comparable to containing unsafe applications in hardware-enforced memory regions but again has the added robustness that stem from the use of a safe language with soundness partially checked at compile time.

Combined Protection (AppC) uses both hardware- and software-based MP to achieve maximum robustness for highly critical environments. It combines the strengths and costs of both variants and offers the fine-grained protection provided by software-based MP with the robustness with respect to hardware failures (e.g., caused by EMC influence) of hardware-based protection.

In the following, we discuss the operations taken upon the invocation of a non-trusted function to give an insight into an implementation detail.

2.3.2 Interdomain Communication

Our primary interdomain communication (IDC) mechanism is control-flow-oriented services; these services are explicitly exported by the respective applications and thus statically known to our toolchain. Depending on whether the exporting application is trusted or untrusted, these services correspond to AUTOSAR's TF or NTF. System services are special in that they are implicitly known to our toolchain but do not otherwise differ from TF. Upon invocation of such a service, the control-flow temporarily changes its protection context (the application) to that of the callee application. The operations that need to be performed in order to switch the protection context depend on the protection types of the caller and the callee application. The KESO compiler is able to statically identify the call sites and the callee domain.

	AppU	AppS	AppH	AppC	Caused by
`GetTaskID()`	0.05	0.05	0.05	0.05	
`ActivateTask()`	0.44	0.60	1.76	1.78	❶, ❷, RT-Check
`Service()`	1.26	1.40	5.66	5.77	❶–❺, RT-Check
Sensor App	22.73	30.79	22.73	31.21	RT-Checks

Table 1: microbenchmark results (runtimes in μs)

Figure 3 shows a simplified version of the operations that can surround the call of a non-trusted function `NTF()`. Each service call site is affected by nested pairs of operations that perform elements of the protection context switch before and after the actual execution of the NTF. These operations pairs are in detail:

❶ `enterPrivMode()` switches the CPU to supervisor mode and disables the MPU to be able to perform the necessary changes on system data structures and MPU registers. The corresponding final `leavePrivMode()` re-enables the MPU and returns to user mode. Applies to: caller application of type AppH or AppC.

❷ Backup the ID of the caller application on the stack, and set the application of the control-flow to the ID of the callee application. After the service, restore the saved ID. `*tjp->arg<0>()` hereby returns the first argument to the NTF call, `CALLEE_ID`. Applies to: any service call.

❸ Backup the previous stack bound, and set it to the current value of the stack pointer, which makes the rest of the stack accessible to the callee application for the duration of the service call. Applies to: callee application of type AppH or AppC.

❹ Reconfigure the MPU registers to the regions of the callee application, and restore them to the caller application after the service call. Applies to: callee application of type AppH or AppC.

❺ Enable hardware protection for the execution of the service itself, and return to supervisor mode after the call has finished. Applies to: callee application of type AppH or AppC.

For the reader, who is familiar with AOP: we use AspectC++ *around advice* to technically implement the above.

3. PRELIMINARY EVALUATION

This first evaluation is meant to show that the decision, which type of MP is the better suited one for a particular application, does not only depend on the differing degrees of protection but also on the imposed runtime overhead. For this, we chose four microbenchmarks that show the cost of different types of inter-domain communication as well as an application that rarely communicates with other domains.

The results in Table 1 show the average over 1000 iterations taken on a Tricore TC1796 controller clocked at 50 MHz. The programs were compiled with tricore-gcc 3.4.5 (`-O3`) and loaded to the internal no-wait-state RAM. The times were determined using a Lauterbach hardware trace analyzer.

`GetTaskID()` is an AUTOSAR system call that determines the ID of the currently running task. This is an example of a read-only system call that does not change any system state and—due to our design decision to only provide write protection—does not require a change to the processor mode or the MPU configuration, wherefore none of the MP variants introduces any overhead to this call.

`ActivateTask()` is a state changing AUTOSAR system call, which—in the case of hardware MP—requires to enter supervisor mode and disable the MPU. This results in a 300% runtime overhead compared to the actual cost of the service (unsafe). The use of software MP in our implementation requires a null reference check, since the task's AUTOSAR id needs to be determined from a task object, which explains the differences between unsafe/software and hardware/combined.

`Service()` is an example of a NTF. With software MP, this service invocation requires a null pointer check. The use of hardware MP requires two MPU reconfigurations and four CPU mode switches (Section 2.3.2), which lead to a 350% runtime overhead.

Obviously, hardware MP performs worse than software MP for applications that frequently perform state changing IDC operations. To give an example for an application type where software MP infers the higher overhead we chose a small application that collects sensor data in regular intervals, and, at less frequent intervals, computes the average of the values collected during the last collection interval and passes them as input to a control application (requiring a NTF operation, which is not part of the measured time). Collection and computation in this application requires multiple null reference and array bounds checks when software MP is being used, which introduce a overhead of 35%, whereas hardware MP does not introduce any added cost since all needed memory operations are to regions of the active application.

4. CONCLUSION

In this paper we presented first results of an approach that enables to choose the type of memory protection transparent to the application. To achieve this, we combined the CiAO operation system, which provides the option of hardware MP through aspect-oriented implementation techniques with KESO, an ahead-of-time Java compiler with the ability to optionally disable unsafe code to remove the overhead of software MP. This combination enables the developer to freely choose among four types of memory protection for an application, unsafe, software-only, hardware-only and hardware-and-software-combined MP. We showed that the decision for the best suited type of protection highly depends on the particular application with respect to both the offered degree of protection and the runtime overhead. The decision may also depend on the type of deployment, for instance, software MP is helpful to detect bugs in a program that would not be detected by hardware MP, wherefore it could be used during the testing phase but be disabled in the shipped product.

5. FUTURE WORK

In our current implementation the choice of MP can only be done for all applications in the system, however, in most cases the desired MP type will not be the same for all applications in the system. We are therefore working on supporting mixed-mode operation where the MP type becomes an individual option for each application. We also want to support applications written in unsafe languages such as C in the system, where MP can only be provided by hardware. This is already possible, but we do not yet have a communication mechanism that would allow communication between C and Java domains.

Another direction that we are currently working on is to provide fine-grained configurability of the various MP variants, that allow making a more distinctive tradeoff between degree of protection and costs. We also seek to exploit more the available ahead-of-time knowledge (e.g., properties of the hardware that could be used to reduce the costs of the different variants).

We are currently porting a complex control application for a quadrocopter to CiAO/KESO. This application comprises multiple protection domains of great variety with respect to the amount of IDC performed. We hope that this application will provide a real-world scenario to demonstrate the usefulness of a mixed-mode operation and to provide a direct comparison of the costs of hardware versus software MP.

6. REFERENCES

[1] AUTOSAR. Specification of operating system (version 2.0.1). Technical report, Automotive Open System Architecture GbR, June 2006.

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

[3] J. Condit, M. Harren, Z. R. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In R. D. Nicola, editor, *ESOP*, volume 4421 of *LNCS*, pages 520–535. Springer, 2007.

[4] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *2002 USENIX TC*, pages 275–288, Berkeley, CA, USA, 2002. USENIX.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *11th Eur. Conf. on OOP (ECOOP '97)*, volume 1241 of *LNCS*, pages 220–242. Springer, June 1997.

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

[7] D. Lohmann, W. Hofer, W. Schröder-Preikschat, J. Streicher, and O. Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *2009 USENIX TC*, pages 215–228, Berkeley, CA, USA, June 2009. USENIX.

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

[9] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *POPL '02: 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 128–139, New York, NY, USA, 2002. ACM.

[10] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, Feb. 2005. `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`, visited 2009-09-09.

[11] 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*, IFIP International Federation for Information Processing, pages 85–96, Boston, 2007. Springer.