An Embedded Hypervisor for Safety-Relevant Automotive E/E-Systems

Dominik Reinhardt
BMW AG
80788 Munich, Germany
Email: dominik.reinhardt@bmw.de

Gary Morgan
ETAS Ltd.
York, YO10 3JB
Email: gary.morgan@etas.com

Abstract—The number of future automotive embedded software applications and their complexity is still rising. Additional non-functional requirements such as safety, portability, maintainability and efficiency compound this trend. The AUTOSAR standard gives flexible and efficient mechanisms to build systems of software components but also involves high configuration effort. When considering safety, the standard has some weaknesses preventing the realization of full separation between software partitions of mixed integrity. Virtualisation seems to be a promising technology allowing one to merge multiple safety-relevant sub-systems onto a single hardware platform and to implement strong separation. Microkernel based hypervisors exhibit a small Trusted Computing Base and serve as the most reliable and robust component within the system. This paper describes and evaluates a microkernel approach to isolate safety-relevant automotive software virtual machines by using a Memory Management Unit less embedded hypervisor. For our analysis, safety mechanisms were implemented with a separation kernel. We present a concept, based upon the ISO26262 automotive safety standard and its safety assumptions, in order to support isolated virtual electronic control units within a real-time environment. Our final goal is to prevent virtual machines from propagating faults between each other. We evaluate our solution by porting some production automotive software to a hypervisor using a paravirtualised AUTOSAR basic software and a Real-Time Operating System. Our benchmarks are based on state-of-the-art automotive hardware and show that the approach is feasible even with less hardware support for virtualisation.

I. INTRODUCTION

Electrical and electronic (E/E) innovations within today’s cars have nearly doubled in the last five years and this trend is going to continue. Premium cars often integrate more than 70 electronic control units (ECUs), connected via different types of field buses [1]. Beside the rising complexity and volume of automotive software, the characteristics and environments of applications are also changing in order to support use cases such as highly automated driving and Car2X communication. Thus, existing applications and new development strategies and processes clash due to very different safety requirements [2], [3]. Embedded software development within the automotive industry is mainly driven by economy of scale necessitating resource-aware software design, application reuse and greater use of software modelling.

Such innovations have resulted in a complex variety of E/E-functionality which OEMs plan to reduce by consolidating ECUs on powerful multi-core platforms such as domain controller units (DCUs) or integration ECUs [4]. This move is accompanied by new domain-oriented architectures for vehicle electrical systems and is called large scale software integration [1], [5]. Automotive software systems integrate applications statically (fixed schedule, memory configuration, network connections, etc.), which introduces problems when dealing with differences in functionality (so-called variants) between vehicles. Flexible and scalable ECU variant handling, from low-end to high-end equipped cars, is very challenging. The static integration process induces high engineering efforts for software porting and additional product validation and verification. Furthermore, integrating additional safety-relevant applications into an already existing mixed integrity system is a well known but difficult problem.

The ISO26262 standard prescribes requirements for safety-relevant E/E vehicle functionality [6]. In the context of this work we are focussing on freedom from interference between applications as this is essential for the safe operation of the vehicle. We assume that a DCU will consist of software of mixed criticality levels. Therefore we are interested in ensuring separation in the presence of software faults caused by unsafe software (i.e. low criticality) as well as due to wearing effects of underlying hardware components. Safety-relevant software components must not be interfered spatially, temporally or during information exchange. AUTOSAR supports specific mechanisms for detecting and handling errors within the system [7]. Multiple error types are considered such as illegal memory accesses, exceeding time budgets or corruption of signals between components and ECU systems. These features are already used in many existing ECUs and are state-of-the-art for automotive systems. However, development and research is ongoing for full functional isolation of applications.

Another challenge is caused by the memory and power consumption constraints of embedded controllers [8]. Processing cores scale horizontally (number of cores) rather than vertically (increasing speeds) due to power consumption problems. Some hardware suppliers have already modified their products to deliver multi-core systems. Since AUTOSAR 4.x, multi-core operation is supported allowing software components to be relocated between partitions and calculation cores [9]. However, automotive software development has yet to exploit parallel processing in an efficient manner because legacy code, designed to run on single core systems, is difficult to adapt to run in parallel on multicore systems.

Virtualisation technology is already used for isolated, freely-relocatable software within data centres, small office areas and in consumer electronics [10]. In the embedded domain, an incompatible operating system (OS) and its ba-
sic software (BSW) must be consolidated without significant refactoring and porting efforts. The hypervisor encapsulates safety-relevant and secure software systems allowing them to run in conjunction with unsafe systems and giving protection from malicious accesses. Therefore they form an enabling technology for software migration and consolidation services, as well as running different OS versions and types [11], [4].

In this paper, we evaluate a research-based hypervisor, built by ETAS Ltd., that supports both non-intrusive system migration and isolation on a multi-core system, and is strictly focussed on efficient resource usage. It represents a Type-1, bare-metal hypervisor, where no hosting system is necessary. Guest systems are paravirtualised and were built on the Infineon AURIX TC27X platform. The AURIX is state-of-the-art automotive hardware, which has little hardware support for virtualisation. We want to fulfill safety requirements [6] according to ISO26262 and so separation must be guaranteed in order to isolate safety-related software systems. If several ECUs are integrated together and isolated on a single platform then there is need to share hardware resources. Therefore the hypervisor includes emulation of hardware resources in an efficient way. The communication channels between virtual machines (VMs) are also realized by the hypervisor. We focus on the design and separation features of the hypervisor.

The next sections are organized as follows: Section II outlines the related work on embedded hypervisors for real-time systems. Section III describes the hypervisor architecture, experimental system design requirements and its layered architecture. In section IV we summarise the implementation of the experimental system. The results and performance evaluations are discussed within sections V and VI.

II. RELATED WORK

There are already existing hypervisor solutions and commercial products for embedded real-time systems. For example Green Hills’ Integrity, Freescale’s Topaz, Sysgo’s PikeOS, and the Wind River Hypervisor. Open source variants such as Xen, XtratuM [12], [13] or OKL4 from OKLabs are well known. PikeOS and OKL4 are based on L4 microkernel [14], [15].

The above hypervisors are bare metal and some use paravirtualisation techniques. Many of these products depend upon specific hardware support, such as virtualisation extensions and memory management units (MMUs). The latter are useful because they allow applications written to reside at arbitrary memory locations to be relocated into the available physical memory at run time. As non-automotive systems that use hypervisor’s load applications dynamically (i.e. applications are linked at build time) MMUs are necessary. Idiomatic automotive controllers only have memory protection units (MPUs) to detect and avoid illegal memory accesses, and the VM’s tend to be known statically - i.e. at hypervisor build time. MPUs present no possibility for virtual memory relocation and, therefore, the mapping of applications into memory is calculated offline before building the system.

Microkernel based architectures are robust because of their strict separation of CPU privileged modes. Similar approaches for AUTOSAR are already discussed and obtainable [16], [17]. With it, mechanisms such as context changes or configuring protection features can be verified more accurately and with less effort. On the other hand, they do not solve the problem of strongly coupled AUTOSAR BSW modules [18] and impede software separation.

Virtualised systems have similar advantages concerning robustness, flexibility and safety. Additionally, they exhibit capabilities that allow them to encapsulate entire ECUs. Co-hosting of diverse software systems, by using static configuration or virtualisation techniques, is an emerging use case within the automotive industry [19], [20], [21]. For example, OpenSynergy’s COQOS system is based on PikeOs and allows AUTOSAR or other systems, such as Linux, to be consolidated within an appropriate hardware platform.

In contrast to typical OS’s for PCs, embedded real-time systems are designed to meet hard deadlines based upon inputs with known timing characteristics. The supported applications typically have a deterministic processing behaviour and are closely integrated with the underlying hardware controller. Example performance evaluations of full and paravirtualised systems on powerful ARM or Intel cores are published in [15], [22], [23], [24].

If there is more than one VM per core on a multi-core system, scheduling, message passing and resource accesses will become more difficult to manage [25], [26]. A hierarchical scheduling approach for real-time Linux systems is proposed in [27], [28]. Ideally there should be optimization methods for multi-core systems [29] as well as strong models that allow the real-time properties of the system to be reasoned about prior to build time.

III. DESIGN GOALS AND SYSTEM ARCHITECTURE

Within this section we describe the overall system architecture and the role of virtual ECUs. We list key indicators for separation dealing with errors and systems failures. We also describe relevant hypervisor functionalities and how they are realized within the system. Furthermore, we discuss the privilege model supported by the AURIX hardware controller.

A. Requirements for Virtual ECUs

To design an appropriate hypervisor we identified several requirements which must be supported in order to integrate virtual ECUs. As described in [4], our use case is to merge separate ECUs into a Domain Controller Unit (DCU). The overall objective is to have one scalable E/E architecture covering cars with large and small E/E systems. In the case of lower-functionality cars, functionality is reduced and functionality that would, formerly, require its own ECU is migrated onto a comprehensive DCU. This unification introduces new requirements for virtual ECUs concerning software safety inside such clusters because, the natural separation of serial buses (such as CAN and FlexRay) is lost when one uses a shared-memory multi-core processor such as the AURIX. We will now focus on virtual ECUs and concentrate on communication concepts with hardware peripherals and interfaces to other VMs.

From the software reuse point of view, the porting effort for virtual ECUs shall be reduced to minimum, i.e. the hypervisor should be non-intrusive. The engineering effort for the unification must be lower than reconfiguring (i.e unifying) the underlying BSW and merging all the application software
systems. If the effort is not lower, there is no economy of time and costs compared to the classical AUTOSAR approach.

The determinism and functionality of systems prior to unification must be preserved taking into account the real-time behaviour and any additional timing penalties caused by emulation efforts for virtualising the system. The real-time aspect is particularly important for I/O peripherals, such as safety-relevant signals on time-triggered field buses. If it is supported by the architecture and underlying virtualisation technologies, I/O devices should be assigned directly to VMs in such a way that there are no performance penalties caused by shared devices. Such zero-penalty assignment may not always be possible with some form of emulation (such as provided by paravirtualisation) being used in its place. In this case, overheads must be known and capable of being modelled such that correct system behaviour can still be guaranteed.

If ECUs are already configured statically, as supported by the AUTOSAR standard, their former existing architecture will not be changed, including interfaces and task scheduling.

B. Safety-Relevant Software Clusters

Safety-relevant virtual ECUs inherit the same safety requirements as before integration onto a single DCU. The pre-integration ECUs interacted exclusively using a common vehicle field bus (such as CAN or FlexRay) and have no other physical connection. This limits the possibility for a failing ECU to interfere with an ECU that is operating correctly. Virtual ECUs are integrated into a common hardware platform and probably share memory, peripherals, common crossbar switches and maybe even calculation cores. Within safety-relevant systems, freedom from interference [6] between software components is an important requirement and must be guaranteed in time, in space and during information exchange.

To increase the system’s reliability and safety, all run-time failures must be treated correctly in order to prevent hazards to vehicle passengers. As depicted in figure 1, we differentiate between faults, errors and failures according to ISO26262.

A fault is a non-predictable, abnormal condition, which could have influence on the system behaviour [6]. Typically, some faults can be avoided at design phase by using adequate module and system tests. We concentrate on sporadic faulty states, which were not detected and could appear during system runtime. These are often caused by hardware, but erroneously-engineered software can also be a source.

Similar to a fault, an error represents abnormal system behaviour or value and is in conflict with the system specification. An error can arise due to a fault. Similarly to faults, errors can be detected by given safety mechanisms, either supported by the underlying hardware (e.g. Memory Protection Unit or Watchdog) or due to software features (e.g. AUTOSAR End-to-End Protection).

A failure represents a concrete loss of functionality, where an error could not be removed and the system is not in a safe state. This can arise due to an error. To mitigate existing failures, supporting methods from an additional control mechanisms or the application software layer can be used. Cascading or propagating failures within the system must be prevented. Alternatively, they might be tolerated internally to the system.

Finally, a failure might result in a hazard for passengers inside the vehicle or for other people outside it. A hazard might lead to an accident, and must be avoided.

Therefore, we encapsulate errors and failures of ECU systems within VMs. A microkernel-like approach is chosen, which acts as the most Trusted Computing Base (TCB) within the system. It is a small piece of software which is well suited for safety-qualification purposes [16], [17]. The hypervisor configures the spatial and temporal protection mechanisms of each VM and controls the communication flow between them.

C. Layered Architecture and System Structure

For many years (and this is still, by far, the most common practise) automotive systems have run entirely in supervisor mode. Historically the hardware did not support a distinction between user mode and supervisor. Now that many processors do support this distinction, most software is not structured to use it. However, performance is also an issue because moving between user and supervisor modes (and vice versa) takes time but does not contribute the functionality of the ECU. For example, to achieve faster timing behaviour, large parts of automotive systems (e.g. device drivers and even application code) operate in supervisor mode. In microkernel based approaches, by comparison, only small parts of the OS are trusted and allow to operate in supervisor mode. Type-1 hypervisors run directly on the hardware and use supervisor mode themselves, and user mode for the unprivileged guest systems.

RTA-HV represents a native, bare-metal virtual machine monitor produced by ETAS Ltd. Its architecture is shown in figure 2. It is a kind of hybrid hypervisor, combining attributes of a small microkernel (nanokernel) and a virtual machine monitor. There is no hosting OS necessary, RTA-HV runs on bare metal. Logically, it is integrated above the hardware and abstracts devices for all VMs. RTA-HV runs in supervisor mode and has full access to core registers. All VMs run at a lower privilege level (user mode) and have restricted accesses and cannot change privileged or sensitive states.

Every VM is mapped uniquely onto a calculation core. From a real-time perspective this is a proper solution because no hierarchical scheduling is necessary; every VM has its own independent core, including all core registers and core local memory. Therefore, interrupt latency (which is a key performance indicator in this domain) is reduced. The source code of the hypervisor is integrated once within the system and is shared between all cores. The hypervisor itself is aware of this sharing and uses knowledge of the currently executing core and semaphores, to protect its own integrity.
Virtual Device Emulators (VDEs) act as programmable interfaces for sharing devices. They are under the control of the hypervisor and run in supervisor mode. VDEs exist to abstract hardware devices, such as I/O peripherals, and arbitrate access to them between VMs. Because VDEs can handle interrupts and trigger interrupts on other cores, VDEs can also be used to implement inter-VM communication mechanisms. A synchronous and an asynchronous transmission protocol was implemented as part of this work. A VDE exists once within the system and all cores can have access to it. Therefore, the source code is re-entrant and the developer must be aware of possible race-conditions. If more than one core accesses a VDE there must be code to identify the calling context because different cores provide different parts of a VDE’s functionality.

IV. IMPLEMENTATION

In this section we describe an example port of an AUTOSAR System onto a hypervisor. We first describe the underlying AURIX hardware controller and its core architecture. Then we provide an overview of our demonstrator and show where we had to paravirtualise the parts of the software. Furthermore, we explain how to ensure spatial and temporal separation and temporal correctness.

A. Overview of the AURIX Architecture

The experimental system is based upon an Infineon AURIX processor. This is a typical automotive part. We provide an overview of the processor cores, memory and exception mechanism as these are the most relevant in implementing the hypervisor. Individual peripherals are of less interest as they tend to be in the domain of the application in the VM.

The AURIX, as depicted in figure 3 [20], consists of three TriCore processors. Each processor has some local RAM that can be used for code and some that can be used for data. The processors were not designed to support virtualisation. However, they do support a distinction between supervisor mode and user mode.

Each processor has internal state that is accessible from its own address space. Such state includes the program status word (PSW) and the MPU. In order to write this state a special instruction is used (MTCR) and this can only be executed in supervisor mode. In user mode it causes a trap. Processor state can always be read (using an MFCR instruction) as this has no side-effects.

Each processor contains its own MPU. The MPU makes a distinction between code and data spaces and has a different configuration for each. Up to four MPU configurations can exist at the same time. One is used to be selected by a field in the processor’s state. Memory is divided into regions with base/limit register pairs and each region can be assigned read, write or execute permission. The MPU configuration is only accessible from supervisor mode. There is a limited number of base/limit pairs.

The MPU has a granularity of 8 bytes. This is important for automotive applications which are characterised partly by relatively small amounts of data for regions that need to be protected. When programming any protection system two adjacent regions that do not belong to the same partition or VM must be separated such that the granularity constraints are not violated. Otherwise, the boundaries of protected regions merge into each other and protection is lost at these boundaries.

Each processor supports traps (synchronous exceptions) and interrupts (asynchronous exceptions). Traps are generated in response to illegal instructions, privilege violations, MPU protection errors, intentional traps, etc. When the MPU detects an erroneous access, a trap is generated. Intentional traps are used by the VM to make hypervisor calls. The processors also support fully vectored and prioritised interrupts. Traps and interrupts each have their own vector tables, which can be moved around in memory, pointed to by a base register in each case.

A peripheral device that can cause an interrupt has one control register per interrupt source, that determines the core to which the request is directed as well as the request priority.

It must be appreciated that this is a fairly large device for automotive applications. However, there is a key observation that should be made about the relative amounts of FLASH and RAM. If we only consider shared RAM then FLASH dominates RAM with a ratio of about 120 to 1. If we take into account core local memory then FLASH dominates RAM by a factor of about 9 to 1 (this also depends upon the specific device chosen). This is completely different to systems that
use hypervisors. Typically, RAM dominates completely and it also necessitates some different programming techniques. For example, heavy use is made of constant data tables, as these use FLASH rather than RAM; and algorithms that require FLASH are preferred to those that use RAM.

B. Implementation of privileged functionality

Because Infineon’s AURIX TC27X has limited hardware support for virtualisation, we use paravirtualisation (similar to Xen) as well as emulation of privileged instructions. Figure 4 shows basic mechanisms of the hypervisor where traps and interrupts are handled. When a VM’s application wishes to access some privileged state (i.e. that normally only accessible from supervisor mode) the application executes the appropriate MTCR instruction which causes a trap and the emulation of that instruction. Emulation involves checking the value to be set, discarding certain bits that can damage the integrity of the hypervisor and then performing the required operation.

Examples of such operations are:

- Modifying the MPU settings. Any unallowed attempt to access that state will restart the VM.
- Modifying the PSW and interrupt control register

When using paravirtualisation traps are used by the VM to make requests to the hypervisor. These traps allow the simulation of operations such as:

- Reading and writing areas in the address space that are configured for a VM, were insufficient ranges in the MPU exists. This means that the simulation code has to check the address and operation validity.
- Putting the VM into trusted mode. This involves checking the legality of the MPU state before trusted mode can be granted. If the VM fails this test it is unconditionally restarted.
- Changing the processors priority.
- Calling VDEs to execute extra functionality. In this case parameter checking is performed in the VDE with support from the hypervisor.

The hypervisor only needs one interrupt per core for itself, and that is connected to a timer used for temporal separation. All other interrupts can be configured to be forwarded to the VM or to a VDE. When an interrupt connected to a VM occurs, it is first handled by the hypervisor and then forwarded to the VM using the VM’s vector table. When an interrupt arrives in a VM the processors priority is set to the maximum of all interrupts used by the VM. This is different to the normal hardware behaviour which would be to turn off all interrupts at this point. Interrupts for VDEs are forwarded directly to the VDE because the VDE uses the same vector table as the hypervisor. Also, a VDE can generate an interrupt in a VM thereby giving it the ability to simulate peripheral hardware.

Traps that are MPU-related must be handled by the hypervisor. However, other traps can be configured to be forwarded to the VM. The trap mechanism and address policing incur time penalties at run time, and so these operations are significantly longer than the equivalent instructions in the bare-metal system. However, the hypervisor’s build-time configuration allows the integrator to choose which addresses are mapped into the I/O space. Either by hardware (i.e. fast) or those where hypervisor intervention is required (i.e. slow) so that the maximum efficiency can be obtained.

The OS also needs porting to the hypervisor. A standard production OS was used (the ETAS RTA-OS product, which is an implementation of the AUTOSAR OS specification) and ported to the hypervisor. The port was greatly eased by an existing port to that hardware and compiler. Changes were needed in the handling of trusted modes (the distinction between user and supervisor mode was faked using the MPU) and in the handling of processor priorities and interrupts.

C. Spatial and Temporal Separation

The hypervisor also provides limited virtualisation of the MPU so that an AUTOSAR VM can implement its own trusted and non-trusted sub-systems (these are called partitions, in AUTOSAR terminology). The hypervisor is decoupled from a VM, but within a VM the AUTOSAR OS can enforce, with the MPU, its own standardised and independent distinction between trusted and non-trusted partitions.

1) Spatial Separation: Our MPU virtualisation scheme is a compromise based upon the known behaviour of the OS we used. On bare metal the OS makes a distinction between trusted and non-trusted partitions by running trusted partitions in supervisor mode and non-trusted partitions in user mode, with appropriate adjustments to the writeable memory. However, the hypervisor needs to use supervisor mode leaving only user mode for the VM to implement both trusted and untrusted partitions.

The AURIX hardware allows four sets of protection regions to be specified. We virtualise the MPU in the following steps:

- Protection register sets are allocated such that: 0 is always the hypervisor running in supervisor mode, 1 is always the VM’s trusted mode running in user mode, and 2 is always the VM’s untrusted mode also running in user mode.
- Protection region set 1 is set up by the hypervisor to enable access to the entire memory of the VM.
- Instructions to reprogram the MPU are only allowed to access protection region 2 and address ranges could...
only be changed to be inside the VM’s statically configured boundaries.

- Hypervisor calls are added to move the VM into trusted mode (i.e. user mode with protection register set 1) or untrusted mode (i.e. user mode with protection register set 2).

Although this cannot be considered to be full virtualisation of the MPU, it is adequate to implement the AUTOSAR partitioning scheme.

2) Temporal Separation: Every VM has an own calculation core. There are no other VMs integrated onto the core which could cause blocking of the VM. However, any VDEs are at a higher interrupt priority than the VM, and this temporal interference must be taken into account in order to provide separation.

Temporal separation is achieved using two methods:

- Firstly the processor’s priority space must be partitioned so that any interrupt handled by a VDE must be of a higher priority than those handled by a VM. This is to prevent a VM raising its priority and starving a VDE of time and, thereby, also starving VDE clients on other cores.

- The second method is to monitor interrupt arrival times in VDEs and throttle those which rose too high. This is to prevent the case where a VM erroneously makes too many VDE calls thereby starving another VM of processor time.

Each AURIX peripheral device that can interrupt has a register that determines the destination processor and priority of the interrupt. In a bare-metal system this register is under the control of AUTOSARs Microcontroller Abstraction Layer (MCAL). In the hypervised system these registers are a problem because they are still under the control of the MCAL and, if incorrectly programmed, they permit an erroneous or malicious processor to direct a stream of interrupts to any other processor thereby breaking temporal separation. This is the most significant problem with respect to interference.

D. Research-based demonstrator build-up

The demonstrator represents two independent automotive software systems, as shown in figure 5. Both use a close to development AUTOSAR BSW and a paravirtualised AUTOSAR OS. The first application layer includes parts of BMW’s AUTOSAR Core (BAC4), representing platform software of several automotive system functions. The second system includes an example powertrain application. All application software is placed above the RTE and are connected to each other and to underlying BSW modules.

Both software clusters have their own BSW and OS. Every BSW is connected to its MCAL. Shared devices must be arbitrated by using a VDE or, if possible, reconfigured to fit into project configuration. According to our studies of configured hardware, core local failures cannot cross VM borders.

Inter-virtual-machine communication is implemented using global data structures within shared RAM. Each VM connects to a common VDE, routing diagnostic messages between each participant. This VDE distinguishes between its calling core and controls the communication flow. To interface the AUTOSAR system to the VDE, a complex device driver was integrated within both AUTOSAR stacks, invoking the VDE using hypercalls.

The OS was paravirtualised before starting to build the demonstrator and was used as an off-the-shelf system component.

Some challenges were caused by microcontroller initialization. Because certain common devices (such as the clock tree) are initialized only once by the first master VM, successive slave VMs must be modified so as not to attempt such operations. So, the steps for configuring the hardware after core startup had to be excluded. Within another scenario, those startup routines could be virtualised and initialisation of peripherals take place within the hypervisor. Other hardware requirements such as the core timer and watchdog (compare figure 2 and 3) are available per core and can be configured exclusively for each VM.

E. Basic Porting Overhead

The AURIX hardware chosen for the hypervisor is only capable of supporting paravirtualisation thereby implying some porting effort in the VM’s software.

Porting was required in the following areas:

- The startup code: This needed small amounts of code removing due to some initialisation being performed by the hypervisor’s startup code.

- The OS: The VMs were designed to support the AUTOSAR OS in all its scalability classes [9]. The OS needs to interact with the hardware to change interrupt priorities and processor trust levels; and these mechanisms changed with the hypervisor requiring some porting overheads.

- A virtual device and associated drivers were needed in order to support safe inter-VM communications.

Although we also expected to have to expend porting effort porting the MCAL [30], this was unnecessary because we had sufficient capacity in the hardware MPU to map all required peripherals into the VM’s address space. Porting effort in the MCAL was only required to cope with different compiler versions. It is more than likely that other applications will
exceed the MPU’s capacity for mapping I/O devices into the VM’s address space. In this case the hypervisor provides calls, accessed via traps, that allow the VM code to perform I/O-oriented operations on arbitrary addresses. The ranges of addresses that can be accessed by a VM are set at build time and policed by hypervisor code at run time.

V. Results

We now present an overview of our measurements on the Infineon AURIX TC27X in combination with RTA-HV and RTA-OS. We measured the spatial and temporal virtualisation overheads caused by the hypervisor and its emulation code.

A. Memory Consumption

As explained previously, the hypervisor uses the idiomatic method of generating code and configuration data from a configuration file at build time. This means that it is difficult to provide figures for the sizes of code and data because these depend very strongly on the configuration of the system. However, in porting our example application, we found that the spatial overheads introduced were small enough to make the port possible without undue worry.

The sizes we measured with the ETAS hypervisor are similar to other approaches based on embedded microkernels or hypervisors [13], [31]. Within our studies, we measured a program size of 45504 bytes (Text), the initialised variables were 48 bytes (Data) and the uninitialised variables used a space of 264 bytes (BSS). A big part of the accumulated space was caused by the interrupt and the trap vector tables for each core. For example, the interrupt vector table is always needed and has a size of 2048 bytes per core. The trap table is always needed as well, and has a size of 256 bytes per core.

We observed that the hypervisor as the only trusted instance has a small memory footprint. The TCB is manageable and there is potential for optimisation. It is notable that this approach has no further scheduling for additional VMs per core included. The hypervisor routes interrupts and controls traps (see figure 4).

B. Temporal Measurements

Overhead measurements were performed on AURIX hardware. A program was written that used an internal hardware counter to measure the length of time of certain instruction sequences. The program was compiled both with and without the hypervisor, so there are two sets of measurements presented.

All VMs got direct access (i.e. not via the hypervisor) to the hardware counter so that the overheads of accessing the counter, in both cases, should be the same. Also, before the measurements started, the time to get the counter value was measured so that it could later be subtracted from the measured times in order to remove any overheads caused by the measurement method. We measured the times taken for operations that we expected to be modified by the hypervisor as well as some that we expected to remain the same. All measurements were executed several times to gain stable results. A subset of the results is presented in table I.

The ”Virtualised” column of table I refers to the time taken with the hypervisor in place i.e. including any hardware protection and hypervisor overheads. The ”Native” column is the time taken without the hypervisor and protection turned off.

An example of the confounding effects can be seen in the time taken for “two nops”. This measured time, surprisingly, went up a little bit when operating with the hypervisor (possible due to measurement accuracy and resolution).

The time for setting the BIV refers to the time to set the VM’s interrupt vector table virtual base address. Normally this is a very fast instruction as it refers to a register inside the processor. However, in this case it is much slower due to paravirtualisation. Typically this and similar operations are only performed at startup. Therefore the overheads are not too noticeable.

The manipulation of the PSW (and the related interrupt control register) takes much longer and is typically in critical OS code. The low performance here is due to the amount of checking that the hypervisor needs to perform due to the chosen processor not being designed to support virtualisation. Some other instruction sequences (for example, MPU virtualisation and priority manipulation) also suffer from this problem.

The time to perform HV_ReadUI() and HV_BitSetUI() are interesting because they give an indication of the penalty incurred when performing I/O via the hypervisor (i.e. when it is not possible to map a device into the VM’s address space). These times are long due to the amount of address range checking that needs to be performed; and this is entirely configuration dependent.

A similar performance problem arises when programming the MPU from the VM as many addresses need to be checked for validity.

We also put some effort into making hypervisor calls, as far as possible, straight line code. By this we mean that an operation always takes the same time to execute in the case where there are no errors. This is to make modelling the timing properties of the hypervisor more feasible in the future.

For one of our key application domains short interrupt latency is a key performance indicator. The interrupt entry time is clearly considerably increased. However, the time reported

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Virtualised</th>
<th>Number of Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>two nops</td>
<td>3</td>
<td>50%</td>
</tr>
<tr>
<td>setting BIV</td>
<td>211</td>
<td>+2914%</td>
</tr>
<tr>
<td>four writes to the PSW</td>
<td>469</td>
<td>+23350%</td>
</tr>
<tr>
<td>Read from STM_CLC (HV_ReadUI())</td>
<td>164</td>
<td>+3180%</td>
</tr>
<tr>
<td>Set bits in the interrupt register (HV_BitSetUI())</td>
<td>163</td>
<td>+1381%</td>
</tr>
<tr>
<td>retrieving priority</td>
<td>13</td>
<td>+160%</td>
</tr>
<tr>
<td>setting priority</td>
<td>166</td>
<td>+8200%</td>
</tr>
<tr>
<td>write to interrupt register (HV_WriteUI())</td>
<td>153</td>
<td>+2450%</td>
</tr>
<tr>
<td>return from interrupt</td>
<td>27</td>
<td>+107%</td>
</tr>
<tr>
<td>interrupt entry</td>
<td>105</td>
<td>+3400%</td>
</tr>
<tr>
<td>entering trap</td>
<td>100</td>
<td>+284%</td>
</tr>
<tr>
<td>return from trap</td>
<td>13</td>
<td>+18%</td>
</tr>
<tr>
<td>Set untrusted mode</td>
<td>1503</td>
<td>+15030%</td>
</tr>
</tbody>
</table>
does not take into account any OS overheads and, therefore, is proportionally reduced.

Figure 6 shows the pathway that an interrupt takes, through critical code, before it can be handled by the application ISR. The measurement starts when the interrupt is received by the hypervisor and ends after the VM’s OS has called the applications handler. The overall time was 9.5μs. This is an example where less virtualisation support adversely affects execution time. The clock frequency we used was 80 MHz.

For the hypervisor to forward the interrupt to the VM takes 3μs. Within the VM’s OS a BISR instruction must be executed in order to save state and manipulate the processors priority. However, this is a privileged instruction and is trapped and emulated within the hypervisor. Emulating the BISR takes additional 4.5μs and forms the largest component of the interrupt latency. The BISR instruction is difficult to virtualise thereby resulting in a time-consuming implementation. Without a hypervisor there is no need to forward interrupts to partitions and the BISR modifications were less complicated. Therefore, the runtime to reach the interrupt processing subroutine will take far less than 1μs.

C. Separation and Isolation

For the purposes of this work separation was divided into spatial and temporal aspects. We showed that these can largely be effectively implemented, although caveats exist. The demonstration system was “well behaved” in the sense that it produced neither spatial nor temporal errors, as one would expect from well tested applications.

Due to the way that the interrupt priorities are organised, it is not necessary to patrol temporal separation in the VM, although this must be monitored in the VDEs. Ideally, VDEs should be avoided due to their additional complexity and overheads; and we achieved this in the demonstrator apart from one very simple VDE for inter-core communications.

The overheads introduced due to spatial separation arise from two sources: initial programming of the MPU at system startup, and re-programming of the MPU by a VM that wishes to implement protection internally using the AUTOSAR protection scheme.

The initial MPU programming is a very short amount of code that runs once when the hypervisor starts up. The startup code overheads are dominated by other activities and, therefore, we saw little value in measuring the MPU programming overheads.

More expensive operations are caused when a VM that wishes to implement the AUTOSAR memory protection scheme uses the virtualised MPU. Small code changes (outside the OS) were necessary in the VM to implement this scheme. Setting the MPU registers is a relatively fast hypercall. The overheads, when the hypervisor is instructed to move the partition from its trusted mode to its untrusted mode (see I, “Set untrusted mode”). The large overhead here are caused by the hypervisor verifying the entire setup of protection register set 2 within the MPU (see section IV-C).

VI. DISCUSSION

From a technical point of view, the hypervisor has been shown to work with a successful port of some production automotive software. However, showing that it works is not the entire story. The automotive market is very cost-sensitive and so a clear economic advantage must also be evident. At the moment, this is far from clear and almost certainly this must be evaluated on a per-project basis. There are multiple reasons for this, which we will now explore.

A. How Much Does Virtualisation Cost?

We investigated the use case of integrating multiple ECUs into a single multi-core platform. There are currently two main sub-use cases: integration of multiple AUTOSAR systems and integration of diverse (e.g. a mixture of AUTOSAR and non-AUTOSAR) systems. Additionally, it is expected that the two systems need to be separated either because they have different Automotive Safety Integrity Levels (ASILs) or because of warranty or similar reasons.

In each use case we are balancing two sets of costs: the costs of unifying two sets of software on one hand, and the cost of porting to the hypervisor on the other.

In the case where we are unifying two AUTOSAR systems, the main work takes place in reconfiguring the RTE, the BSW and the MCAL. These are significantly complex pieces of software and reconfiguring them is far from trivial. But it probably saves RAM and ROM costs due to the reuse of a comprehensive AUTOSAR BSW.

In the case where we are unifying AUTOSAR and non-AUTOSAR systems we assume that the unified system will be based upon AUTOSAR. Therefore, the cost is in porting the existing software to use AUTOSAR interfaces and in reconfiguring the RTE, BSW and MCAL.

In the case where we are porting to a hypervisor the effort that needs to be expended in porting legacy systems is relatively small. For AUTOSAR a new OS port is required that uses the hypervisor’s paravirtualisation. The hypervisor needs to be configured, the MCAL needs modifications to reflect any use of paravirtualisation. VDEs need to be acquired and configured correctly. Here, there is no reuse of the AUTOSAR BSW and its static RAM and ROM needs are multiplied.

The amount of work to port the OS is relatively small (assuming that an OS for that target hardware already exists), as is the MCAL work. Changes to the MCAL, should they be required, are typically simple and mechanical due to the paravirtualisation functions in the hypervisor closely reflecting the way that drivers access hardware. The costs for the VDEs depend heavily upon what is required from it.
Section V-C describes the main overheads in implementing spatial separation when the VM wishes also to implement its own spatial separation scheme. These overheads are very unpleasant due to their size and also because they are on the critical path when switching the VM into untrusted mode. Ideally we would prefer some better hardware support for operations such as this.

One must also take into account the financial cost of the hypervisor; and this is currently not determined, particularly as the hypervisor must be engineered to the highest ASIL of the VMs in the system. Within a safety-relevant software system the hypervisor will probably inherit the highest ASIL of its integrated applications.

B. Shortcomings and Limitations

The main shortcoming of our work so far is that, due to the hardware design, complete temporal separation is not possible. There are two sources of interference: contention for shared resources (such as memory and peripherals) can slow down accesses, and the ability of an erroneous processor to direct interrupts to another can be problematic. The latter is particularly serious.

Another shortcoming is that, in order to reduce interrupt latency as much as possible, only a one-to-one mapping of VMs to cores is supported. In the design we evaluated a number of mechanisms to provide more than one VM per core with low latency and temporal predictability, but we were unable to find a suitable scheme. There were multiple reasons for this failure:

- We had to run on very limited hardware. So we could not use any mechanisms that forwarded interrupts directly from the hardware to the VM. We always had to go via the hypervisor.
- We excluded all type 2 hypervisors as, without hardware support, interrupts are first handled by the OS and also the host system’s scheduler interfered with the VM’s.
- We excluded ARINC 653-style schedulers because, although predictable, their overheads are very large when trying to achieve low latency.
- The scheduler used in PikeOS [32] looked very interesting but is patented, and runs on non-idiomatic hardware.

We are currently devising an alternative in a research project in order to address this shortcoming in future versions of the hypervisor.

Regarding the demonstrator we built, and its hypervisor, interrupts are disabled for code executed within or in the context of the hypervisor (in VDEs for example). During that time, no interrupts will be processed and incoming requests must be stored for later execution. If the VM frequently services interrupts (and this is typically the case for engine control) the additional latency could cause severe timing problems. The worst-case response time for interrupts will be increased to the maximum dwell time within hypervisor source code.

In the future there are more IO devices per VM and per core, there will be the need for more MPU protection region ranges and sets to enable more complex protection strategies.

C. Further Advantages of Hypervisors

Regarding safety-relevant systems in general, there are already microkernel-based approaches available within the automotive E/E sector. Due to a smaller TCB, these approaches better support system qualification, because only a small piece of code is responsible for executing integral functionality. Providing guaranteed context switches and a safe MPU configuration are examples of such pieces of source code.

Another factor that needs to be taken into account is the ability of the hypervisor to provide better isolation than AUTOSAR. In particular, under the hypervisor, a complete VM can be terminated and restarted, if certain errors are detected, without affecting other VMs. This is superior to what can be achieved in AUTOSAR.

We determined, during the demonstrator build-up, that initializing the system could cause interference or further modifications within a partition. The solution depends on the strategy for the integration of the MCAL layer and other drivers; either within the hypervisor or within the VM. Several other possibilities or hybrid approaches are already used within other virtualisation systems. Keeping all drivers within the hypervisor will probably solve some problems for shared devices, but it will expand the TCB of the hypervisor. Keeping the layer of hardware drivers in one place or core is a similar approach realized within AUTOSAR and its MCAL.

The current AUTOSAR specification arbitrates between multiple cores sharing I/O devices by forcing all requests onto a single core. The process of adapting code to run on a hypervisor can be improved if explicit support were to be provided in AUTOSAR by, for example, adapting the MCAL. This would allow for better interaction between VMs that run in parallel, or concurrently, and any hardware that must be shared. Also, the hypervisor naturally allows interrupts to be shared between cores.

Paravirtualisation is a possible method for meeting real-time requirements and is efficient enough to match automotive requirements. For our purpose, it is the best trade-off between engineering effort and meeting efficient real-time capabilities. However, paravirtualisation does not enable our system to produce faster interrupt routing or to lower timing overheads caused within context switches. Here, better hardware support would result in better system performance. Within a fully virtualised system we would need more detailed information about traps which occur within a VM. Improving the interrupt handling would reduce critical paths through the hypervisor. Clearly an additional hypervisor mode would help as well to keep old privilege levels within VMs.

VII. CONCLUSION AND FUTURE WORK

We have demonstrated that a hypervisor can be built to run on a relatively simple processor that was not designed for this use case. The processor is idiomatic for the chosen application domain, and so the ability to run a hypervisor on it, without the need to move onto a completely different processor type,
is very important. The hypervisor allows multiple VMs to run in parallel, with spatial and temporal separation (as far as the hardware supports), and enables complete AUTOSAR partitions with memory protection inside the VM. We were able to measure the key performance overheads of the hypervisor (in time and space). Although, certain key operations are longer, they are not as long as to rule out the use of this technology. However, such economic and performance judgements can only be made on a per-project basis.

A hypervisor approach can be a possible solution to the problem of consolidating disparate systems on the same ECU, and keeping them separate. It is a useful technique if the costs of merging or porting the legacy systems is high or the risk of violating safety and security requirements is unacceptable.

ACKNOWLEDGMENT

The authors would like to express their gratitude to Dirk Kaule and Hans-Ulrich Michel from BMW Research and Technology for their cooperative work and reviewing the paper. The authors also thank Andreas Graf of BMW AG and Technische Universität München for his discussions and reviews.

This work was funded within the project ARAMIS by the German Federal Ministry for Education and Research with the funding IDs 01IS11035. The responsibility for the content remains with the authors.

REFERENCES


