Applying Reservation-based Scheduling to a μC-based Hypervisor: An industrial case study

Dakshina Dasari, Michael Pressler, Arne Hamann, Dirk Ziegenbein
Corporate Research, Robert Bosch GmbH,
Email: {Dakshina.Dasari, Michael.Pressler,Arne.Hamann, Dirk.Ziegenbein}@de.bosch.com

Paul Austin
ETAS,
Email: paul.austin@etas.com

***Abstract*—Existing software scheduling mechanisms do not suffice for emerging applications in the automotive space, which have the conflicting needs of performance and predictability. As a concrete case, we consider the ETAS lightweight hypervisor (LWHVR), a commercially viable solution in the automotive industry, deployed on multicore microcontrollers. We describe the architecture of the hypervisor and its current scheduling mechanisms based on Time Division Multiplexing. We next show how Reservation-based Scheduling (RBS) can be implemented in the ETAS LWHVR to efficiently use resources while still providing freedom from interference and explore design choices towards an efficient implementation of such a scheduler. With experiments from an industry use case, we also compare the performance of RBS and the existing scheduler in the hypervisor.**

I. Introduction

To efficiently accommodate the increasing number of functions in today's cars, there has been a shift in the automotive architecture; as opposed to the presence of multiple single-function electronic control units (ECUs) hosted on simple microcontrollers, the trend now is towards consolidating these multiple functions onto a smaller number of domain control units (DCUs) consisting of more powerful multicore microcontrollers. DCUs have become mainstream since microcontrollers are easier amenable to safety certification compared to larger and more complex microprocessors. An example is the body domain in a car, which hosts safety critical functions like power management of the body, security critical functions like unlocking the car, and non-critical functions like interior lighting. These functions are typically developed by different suppliers and later integrated on the DCU. The challenge of consolidating different functions on these multicore microcontrollers is therefore to ensure freedom from interference between these functions, together with provisions for extensibility and composability. Hypervisors provide the much needed temporal and spatial isolation by virtue of their design and have become the de-facto solution in many embedded platforms. However, they must be further augmented by better resource management mechanisms to cater to the needs of diverse applications, and allow for better extensibility. In this work, we consider the ETAS Light Weight Hypervisor (LWHVR) [1], one of the commercially viable solutions on microcontrollers, designed to isolate applications to support mixed ASIL levels and to support integration of software from different sources and practically deployed in the automotive segment. Unlike general purpose hypervisors, the ETAS LWHVR (with only 5K bytes of code) has been successfully used for automotive embedded systems on multicore microcontrollers without hardware virtualization support.

The ETAS LWHVR scheduler, like many existing commercial hypervisors is primarily based on time-division multiplexing (TDM) to schedule the hosted virtual machines (guest partitions). TDM is simple and leads to allocating a guaranteed number of time-slices (slots) to each VM, but is inefficient due to its non-work-conserving nature Furthermore, the method works well for static periodic workloads (fitting to the traditional single ECU, fixed single function scenario), but could lead to highly sub-optimal processor utilization when integrating workloads characterized by varying execution needs or sporadic arrival patterns. Furthermore, this often leads to reconfiguration (and subsequent reverification) of the schedule table in order to efficiently accommodate additional workloads, since TDM is not inherently extensible. For this reason, there is an increased interest in using mechanisms like Reservation Based Scheduling (RBS) in hypervisors. RBS is essentially a hierarchical scheduling variant, designed to guarantee a minimum set of resources to each partition by efficiently monitoring and provisioning computing resources. It is inherently composable since applications can be designed in isolation by different suppliers, assuming the availability of a set of resources and this contract is maintained by the underlying mechanism when they are integrated with other applications on a DCU.

Contributions: In this work, we describe the architecture of the ETAS LWHVR, its current scheduling mechanisms and limitations. We next show how RBS has been implemented in the ETAS LWHVR. We also propose different design alternatives and with experiments representative of an actual use case, we evaluate key parameters like the scheduling overheads, memory footprint, context switch overheads and response time gains and show the benefits of using RBS over the existing scheduler in the hypervisor.

II. Architecture of the ETAS LWHVR:

The ETAS LWHVR, as seen in Figure 1, is used in ECUs with multicore microcontrollers where one core, the master core, runs software (called the master software) directly on the hardware. The other application cores run independent applications contained inside virtual machines (VMs). The master software contains device drivers and their interrupt handlers. VMs do not access hardware and are not directly associated with hardware interrupt handlers. Any application request for accessing a peripheral is routed via the master software on behalf of the VM hosting the application. VMs can

978-3-9819263-4-7/DATE20/©2020 EDAA

however handle asynchronous events such as timer ticks and shut-down requests using the *pseudo-interrupt* mechanism. Pseudo-interrupts are similar to hardware interrupts except they are generated and controlled by the LWHVR software. To a VM, a pseudo-interrupt looks much like a normal hardware generated interrupt.

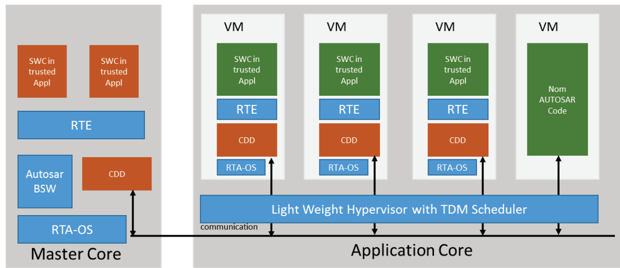

Fig. 1: The ETAS Light Weight Hypervisor with the AUTOSAR Base software (BSW), Complex Device Drivers Layer (CDD), Run-Time Environment (RTE) and underlying RTA operating system (RTA-OS)

A. Basic Time sliced Scheduler with spare slices

The ETAS LWHVR allows a single application core to host multiple VMs by sharing the core between VMs by using TDM. For each application core, the LWHVR's configuration contains a schedule table. Each entry in a schedule table defines a time-slice describing the length (duration in ticks) of the time-slice and either the identity of the VM that should run in the time-slice or an indication that the time-slice is *spare* and can be assigned to VMs dynamically. The spare slice can be used for asynchronous requests – once such a request arrives, it is serviced only at the *next available spare slice*. The LWHVR contains a scheduler for each application core that is invoked by a clock-tick interrupt. Normally a scheduler decides on the order in which to run VMs by iterating though schedule table entries in the order they occur in the schedule table and running the specified VMs for the specified durations, or idling if a time-slice is spare.

a) Dynamic Time Sliced Scheduling: In order to increase the responsiveness to sporadic requests from the VMs, ETAS LWHVR also implements the dynamic scheduler seen in Figure 2, wherein the scheduler maintains a high and a low priority queue (HPQ and LPQ). The LPQ is used for extra time requests made by VMs themselves. A VM may only be in the LPQ once (so that spare slices are shared fairly). The HPQ is used for extra time requests made by the master software for sporadic interrupts that run in a VM context. A VM may be in the HPQ multiple times. The scheduler first checks the HPQ and if there is a VM in the queue, it runs the VM at the front of the HPQ immediately, overriding and preempting the normal schedule table. Each time such a HPQ request is handled, one spare slice is skipped and is thus unavailable. In this way, we only allow the master software to borrow spare time that exists in the schedule, but in advance of when the spare time occurs in the schedule. If the HPQ is empty, the scheduler switches to "normal" scheduling. If the next time-slice is assigned to a VM, then that VM runs. If the next time-slice is spare and there is a VM in the LPQ, then the VM at the front of the LPQ runs in the next available spare slice. When considering sporadic VMs, we therefore consider the HPQ with extra-time requests coming from the master software.

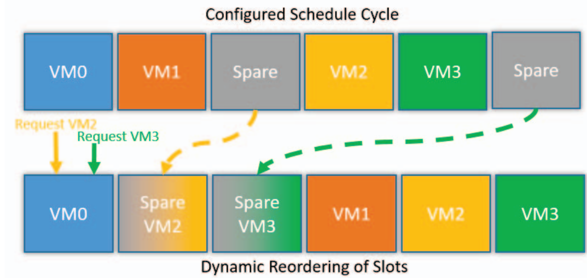

Fig. 2: Modified scheduled after VM2 and VM3 request an extra time just after tick 0. Note how the regular schedule table is shifted to immediately to execute sporadic requests directed to VM2 and VM3.

To summarize, the basic TDM scheduler augmented with space slices is inherently inefficient: the spare slice is wasted when there are no asynchronous requests to use it. Additionally, it may lead to unacceptable response times for high priority asynchronous (sporadic) requests which have to wait to be serviced until the earliest spare slice is available in the schedule table. The dynamic scheduler, on the other hand, with the extra-time queue increases the responsiveness to sporadic requests but essentially breaks the time guarantees introduced by the schedule table.

III. Extending LWHVR with RBS

In order to overcome the limitations of the enhanced TDM with dynamic scheduling, the ETAS LWHVR has been modified to use RBS. As a first implementation, we consider fixed priority (FP) servers like the Deferrable Server [2] (DS) in which the servers are preassigned a unique fixed priority and a top-level scheduler schedules these servers based on their priority. As seen in Figure 3, each VM is now contained in its own DS with its guaranteed budget in each period and each VM's server has a unique priority. Applications within the VM may execute while its server has available budget and is of the highest priority. Once the server exceeds its maximum budget, its VM is suspended by the top-level server scheduler and the next highest priority VM server with active applications and available budget is executed (fixed priority pre-emptive scheduling). The top-level scheduler provides the interfaces to configure the type, priority, budget and period of each VM's server and decides which VM must execute and monitors and manages the server's budget (replenishment and consumption).

A. RBS Implementation

Each VM (contained in a DS) has an active flag that is true if-and-only-if the VM has work to do. The LWHVR receives a clock-tick interrupt at a regular interval. On each clock-tick interrupt, the LWHVR carries out the following: i) Replenishment: For each VM whose server is at the start of its period, set the server's remaining budget to its maximum

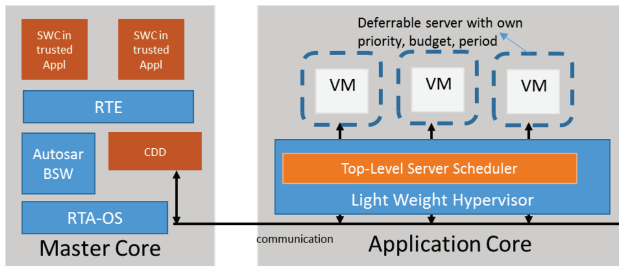

Fig. 3: Encapsulating a VM in a deferrable server

budget. ii) Periodic activation: For each VM: if the VM (its contained tasks) should be periodically activated at this time, set the VM's active flag to true. iii) Reschedule: Choose the highest priority VM with remaining server budget and whose active flag is true. This VM is executed when the LWHVR returns from the clock-tick interrupt. In case of an asynchronous event, the master core signals an asynchronous event to a VM and sets the VM's active flag to true. The VM yields the processor when it has no more active tasks and its active flag is set to false and (eventually) rescheduling occurs.

For OS'es containing periodic tasks the OS scheduler that dispatches such tasks is driven by a periodic interrupt. In the LWHVR, timer-tick pseudo-interrupts are injected into a VM to drive the OS scheduler. The LWHVR is able to activate a VM (set its active flag to true) and inject a timer-tick pseudo-interrupt at a configurable frequency to support such OS scheduling. On the other hand, in cases where the master core receives some asynchronous interrupt/signal directed towards a VM, the master core, or another VM, sets an active flag associated with the target VM to true. For the case of the RBS implementation, once the VM's active flag is true, and it is the highest priority active VM in the system with available budget, it is executed.

The master software informs the code running inside a VM that the asynchronous event has occurred by allowing the master software to set an event notification pseudo-interrupt as pending in a VM so that when the VM is next dispatched, the pseudo-interrupt will be injected into the VM. Finally, the LWHVR needs to know when an active VM no-longer has any work to do. This is done by the VM making an API call that sets the VM's active flag to false, decrements the remaining budget of the VM and then suspends the VM.

B. Representation of Time

The LWHVR receives a clock-tick interrupt at a fixed frequency; time is measured as a count of the number of times the clock-tick interrupt has occurred. The period between clock-tick interrupts is referred to as a tick. . For example, if the clock-tick interrupt arrived every 200us then each tick would represents 200us.

C. Per-Tick Scheduling

One design option is to invoke the top-level scheduler at every tick to decide which VM must execute. This method though simple to implement, carries a non-trivial processing overhead; At each scheduler invocation, there is a need to first save the currently running VM's processor registers, load the processor registers for the LWHVR's small-data areas and stack, carry out a plausibility check on the interrupt source, then run the scheduler, and then load the newly selected VM's processor registers. Other drawbacks of the mechanism is inefficiency: whenever a VM calls the API to deactivate itself, the system idles until the end of the current tick. Secondly, each time a VM is activated by the master software API, the VM does not start running until the start of the next tick. Increasing the frequency of clock-tick interrupts minimizes the described idling effects but increases the context-switching overhead; therefore better mechanisms are needed.

D. Mitigating the overheads of per-tick scheduling

Using a lightweight primary interrupt handler: One of the methods to overcome the drawbacks of the per-tick scheduling is invoking the scheduler only when necessary. In order to implement this, we use a very lightweight primary interrupt handler that is invoked each tick, ut carries out minimal processing (in our implementation only 13 instructions are executed). It first saves a minimum number of processor registers (only 4 in our implementation). Then, using only the registers saved, decrements a counter (that stores the next time the scheduler must run). If the counter is now 0, or the active flag is set (when some asynchronous task is triggered), it invokes the secondary interrupt handler, after which it returns from the interrupt.

The secondary interrupt handler when invoked, saves the rest of the VM state registers (around 28 of them in our case), switches to the LWHVR's stack and data areas, invokes the scheduler, restores the newly scheduled VM's state and returns from the interrupt. The secondary interrupt handler in effect, executes hundreds of instructions. The scheduler essentially selects the next VM server to run and also computes the number of ticks that must occur until the scheduler next needs to run (to handle synchronous activities) and loads a counter variable with this value. This value can be computed knowing the periodicity of tasks, priority and the remaining budget of the current VM. When a VM deactivates itself, or the call to activate a VM is done, a flag is set to indicate that the scheduler needs to run.

With the lightweight primary handler mechanism, the primary interrupt handler would still consume processor time – but it is much cheaper than invoking the scheduler on every tick. The amount of time spent in the LWHVR on a clock-tick interrupt would now vary depending on whether only the primary interrupt handler ran, or both the primary and secondary handlers ran. In this work, we implemented this lightweight interrupt handling mechanism.

IV. Experiments

Tests were run on an ST SPC58EC80 (Chorus 4M, a 32-bit PowerPC) clocked at 180Mz. The SPC58EC80 has two cores. One core was the master core while all 3 VMs (VM0-VM2) ran on the other core. Each VM runs 3 tasks and one task of VM0 is sporadic while all other tasks are periodic in nature as seen in Table II. We compare the RBS implementation against the original default LWHVR scheduler. In the RBS configuration, each VM is contained in a DS with parameters

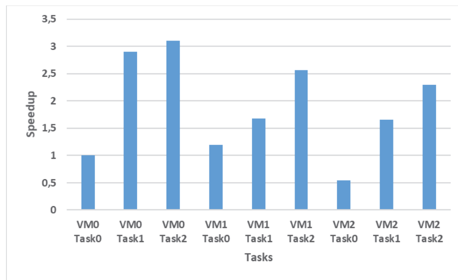

Fig. 4: MRT speed up factor with RBS across different tasks

shown in Table I. The lightweight interrupt handling mechanism was used in all cases, as the scheduling overhead of the per-tick scheduler invocation was found to be higher (by nearly a factor of 5). As a baseline, we consider the TDM dynamic scheduler of the LWHVR with 2 priority queues (we refer it simply as TDM henceforth) described earlier with Figure 2. The scheduling tick duration is 0.1ms by default, while the major cycle consists of 200 slices.

VM	Priority	Budget (ms)	Period (ms)
VM0	High	15	83
VM1	Medium	28	83
VM2	Low	40	83

TABLE I. VM characteristics

Task	Priority	WCET (ms)	Period (ms)
VM0_Task0	High	0.5	Sporadic
VM0_Task1	Medium	0.5	10
VM0_Task2	Low	1.0	17
VM1_Task0	High	1.0	10
VM1_Task1	Medium	1.0	20
VM1_Task2	Low	1.5	23
VM2_Task0	High	0.5	18
VM2_Task1	Medium	7	54
VM2_Task2	Low	12	83

TABLE II. Task characteristics

A. RBS vs Dynamic Time Sliced LWHVR Scheduler

The speed up factor is computed as the ratio of the Maximum observed response time (MRT) with TDM to that of RBS. As seen in Figure 4, all tasks benefit from the RBS scheduler as compared to TDM (note that speed up factor >1), with the exception of task 0 in VM2, which luckily benefits from its position in the constructed TDM schedule.

B. Context Switches, code footprint, scheduler overheads

Mechanism	Code (bytes)	Constants (bytes)	RAM (bytes)	CS
TDM	5376	2296	1780	78273
RBS	5832	1764	2024	5187

TABLE III. Coot Footprint of the LWHVR along with TDM/RBS and Context Switches (CS) over 10s

Table III shows the code footprint of the LWHVR along with TDM/RBS. RBS uses slightly more Flash memory for code since the scheduling logic is a lot more complicated than TDM and more RAM because there is more dynamic bookkeeping needed for scheduling. However, TDM uses more Flash memory for constant data for storing the schedule table.

Context switches between VMs are very expensive, especially in microcontrollers without hardware virtualization support. We compared the number of context switches needed by the VMs over a span of time (multiple hyperperiods) over 10s and observe in Table III that RBS greatly reduces the number of context switches needed among the VMs.

We also compared the scheduler overheads over different tick sizes.As expected the scheduling load decreases as the clock-tick length increases. RBS has a lower scheduling overhead because far fewer of the clock-tick interrupts result in the scheduler being called because the lightweight interrupt handler only calls the scheduler when a context switch or budget refresh is needed.

V. Related Work and Conclusions

Most hypervisors cater to the microprocessor space, meant for generic applications and are essentially too heavyweight for the resource constrained microcontroller embedded application space. The task model for RT-Xen [3], a hypervisor for microprocessor, only assumes periodic tasks The PikeOS hypervisor [4] uses a time driven and priority driven scheduler and is thereby different from our approach. The QNX hypervisor [8] uses adaptive partitioning across its guest OSes differs from the proposed reservation based scheduler. Cucinotta [5] et al. use the KVM hypervisor, with a Constant Bandwidth Scheduler algorithm as the global scheduler. Hierarchical scheduling is also employed in the L4/Fiasco. However they also cater to periodic tasks only like the RT-Xen hypervisor. The ETAS LWHVR unlike other commercial or academic hypervisors specifically caters to industry-grade multicore microcontrollers (even without hardware virtualization support) and has a very low footprint. We demonstrate in this paper the benefits of the RBS scheduler over the existing TDM scheduler and that it can be efficiently implemented making it viable, with a low scheduling overhead and low memory footprint.

References

[1] ETAS, "ETAS RTA Lightweight Hypervisor, User Manual (SPC58ECxxGHS)", Available at {https://www.etas.com/download-center-files/products_RTA_Software_Products/Lightweight_Hypervisor_User_Manual_R01_EN.pdf}

[2] J. Strosnider, L. Sha and J. Lehoczky, "The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments" in IEEE Transactions on Computers, vol. 44, no. 01, pp. 73-91, 1995

[3] S. Xi, J. Wilson, C. Lu and C. Gill, "RT-Xen: Towards real-time hypervisor scheduling in Xen," 2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT), Taipei, 2011, pp. 39-48.

[4] SYSGO White Paper, "PikeOS Safe Real-Time Scheduling, Adaptive Time-Partitioning Scheduler for EN 50128 certified Multi-Core Platforms", Available at: {http://nohau.eu/wp-content/uploads/sysgo-safe-realtime-scheduling.pdf}

[5] Cucinotta, Tommaso; Anastasi, Gaetano and Abeni, Luca. (2008), "Real-Time Virtual Machines".

[6] Arne Hamann, Dakshina Dasari, Jorge Martinez, and Dirk Ziegenbein. 2018. "Response Time Analysis for Fixed Priority Servers". In Proceedings of the 26th International Conference on Real-Time Networks and Systems (RTNS '18). ACM, New York, NY, USA, 254-264.

[7] Jungwoo Yang, Hyungseok Kim, Sangwon Park, Changki Hong, and Insik Shin. 2011, "Implementation of compositional scheduling framework on virtualization". SIGBED Rev. 8, 1 (March 2011), 30-37.

[8] BlackBerry, "QNX Hypervisor", Available at https://www.qnx.com/content/dam/qnx/products/hypervisor/hypervisor-product-brief.pdf