Flat but Trustworthy: Security Aspects in Flattened Hierarchical Scheduling [*]

Adam Lackorzynski, Marcus Völp and Alexander Warg
Technische Universität Dresden
Computer Science, Operating Systems Group
{adam, voelp, warg}@os.inf.tu-dresden.de

Abstract

Virtualization is a well-proven technology for consolidating desktop and server applications onto the same hardware platform while maintaining their native environments. However, although embedded real-time systems start to adopt this technology, constrained resources and strict timeliness demands complicate this consolidation task, in particular if some applications are more critical than others and if the timeliness of the latter may be sacrificed for the sake of completing the former. In a previous publication, we have introduced flattening as a means to integrate mixed-criticality tasks into a single real-time system while maintaining most of their native environment as it is provided by virtual machines (VMs) and their monitors. In this paper, we focus on the security and trustworthiness aspects of flattening and on the interfaces for isolating mixed-criticality VMs on top of our microkernel for embedded real-time systems.

1 Introduction

In Lackorzynski et al. [4], we have shown that classical hierarchical scheduling, such as running virtual machines as bandwidth servers [7], cease to work when confronted with mixed-criticality task sets. As a solution, we presented flattening, that is, breaking the encapsulation of virtualized guest operating-systems (OSs) and offering them an interface through which they can inform the underlying host scheduler about the criticality of the tasks they run. The guest modifications for using this interface are negligible (between 10 lines of code (LOC) for para-virtualized FreeRTOS and 22 LOC for fully-virtualized Linux), but the additional information proved valuable in allowing the underlying scheduler to schedule task sets whose timely completion could otherwise not be guaranteed. Although in [4] we have already discussed crucial elements of this interface, in particular as far as timeliness is concerned, there are a number of trustworthiness and security concerns that must be addressed for the system to work.

After introducing mixed-criticality scheduling and summarizing the main results of [4], Section 3 gives a brief overview of Fiasco.OC and the *L4 Runtime Environment* (L4Re). Fiasco.OC offers support for para-virtualized and fully virtualized guest operating systems and deprivileged virtual machine monitors (VMMs). In Section 4, we discuss the interplay between guest and underlying host scheduler for scheduling mixed-criticality tasks in embedded real-time systems and the trust issues that arise.

2 Mixed-Criticality VM Scheduling

The mechanisms we present can be used more generally in all setups where a mode change necessitates a reconsideration of the task scheduling parameters and in particular their priorities. However, to simplify the following discussion, we restrict ourselves to mixed-criticality scheduling in the interpretation provided by Baruah et al. [1, 2] and to task sets with implicit deadline-constrained periodic tasks.

[*]This work is in part funded by the DFG through the cluster of excellence "Center for Advancing Electronics Dresden" and by the EU and the state Saxony through the ESF young researcher group "IMData".

Mixed-criticality scheduling seeks to consolidate tasks that are of different importance for the correct operation of the real-time system. Each task τ_i is categorized with a certain criticality level l_i out of the finite totally ordered set of criticality levels L. Tasks are certified at their criticality level l_i and at all lower criticality levels. A higher level means that this task is more important and that the timeliness of lower criticality tasks may be sacrificed to guarantee its completion. The certification at some level l provides an estimate $C_i(l)$ of the worst-case execution time (WCET) of τ_i that is trusted by all tasks with criticality level l. As usual, we assume that these estimates will be more pessimistic for higher criticality levels (i.e., $C_i(l) < C_i(h)$ for $l \leq h$, $l, h \in L$). Moreover, our scheduler enforces WCET budgets, which allows us to safely assume $C_i(h) = C_i(l_i)$ for all $l_i \leq h$. Besides the criticality level l_i and the vector of WCET estimates C_i, tasks are characterized by the minimal inter-release time T_i, which defines the value of the relative deadline $D_i = T_i$, and by the VM (vm_i) to which this task belongs. We write τ_i^A to denote that τ_i is a task belonging to the VM $vm_i = A$. Hence, τ_i is characterized by the triple $\tau_i^A = (l_i, T_i, C_i)$.

A task set is *mixed-criticality schedulable* if all jobs of all tasks τ_i receive $C_i(l_i)$ units time before their relative deadline D_i provided that all higher criticality tasks τ_j with $l_i < l_j$ complete before $C_j(l_i)$. We say criticality is *inverted* if a lower criticality task is prioritized over a higher criticality task and we call the point in time where it can be decided whether a task τ_i requires part of excess budget $C_i(l_i) - C_i(l)$ the *criticality decision point* of this task for criticality level l.

Fig. 1 (see [4]) shows for the task set $\mathbb{T} = \{\tau_1^A = (HI, 8, (1,4)^T), \tau_2^A = (LO, 4, (1,1)^T), \tau_3^A = (LO, 16, (4,4)^T), \tau_4^B = (HI, 16, (2,6)^T), \tau_5^B = (LO, 4, (1,1)^T)\}$ why fixed budgets for scheduling VMs breaks mixed-criticality schedulability while flattening produces a feasible hierarchical schedule. From classical scheduling theory, we know that task response times are maximal when one job of each task is released simultaneously. Fig. 1 shows a schedule for such a simultaneous release divided into four phases of 4 time units each. The ratios on top of each phase denote the budgets that are assigned to the VMs A and B. To decide whether the VMs have to schedule LO-criticality jobs or whether it is safe to sacrifice these tasks for the completion of the high criticality tasks τ_1^A and τ_4^B, VM A has to execute τ_1^A for one unit time and VM B τ_4^B for $C_4(LO) = 2$ units. Reaching both criticality decision points while guaranteeing the completion of τ_2^A and τ_5^B is not possible. In Fig. 1 we have therefore granted VM B only a budget of 2 and deferred the decision whether τ_4^B completes before $C_4(LO)$. Following the same line of argumentation, it is easy to see that all other assignments of a fixed budget per phase leads to a similar violation of the mixed-criticality guarantees. Now, if τ_1^A did not complete in Phase I, VM A needs at least a budget of 2 to guarantee the completion of this HI task. Utilizing this budget for τ_2^A and τ_5^A if τ_1^A completes latest after $C_1(LO)$ leaves a LO demand of 6 units time, which to complete τ_5^B has to be split in two budgets of 3 units for Phase III and IV. However, this leaves only 4 of the 5 units to VM B that is required to guarantee the completion of τ_5^B. On the other hand, if in Phase IV, VM A would be able to access only a budget of 1 at a higher priority than VM B and an additional 2 unit budget at a lower priority than VM B, the completion of τ_4^B could be guaranteed by granting VM B a budget of 2 in Phase IV. Flattening allows the underlying scheduler to grant these distinctly prioritized budgets to VMs and offers an interface for switching budgets and thereby informing the underlying scheduler in the host about the completion of the HI-criticality tasks.

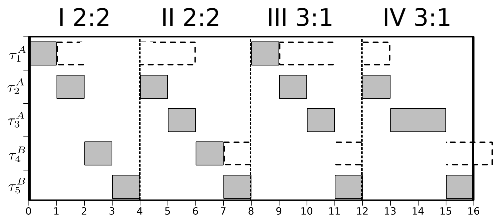

Figure 1: Critical instant schedule of the two virtual machines A and B. Filled bars show LO WCETs ($C_i(LO)$), dashed bars show the excess budget $C_i(HI) - C_i(LO)$.

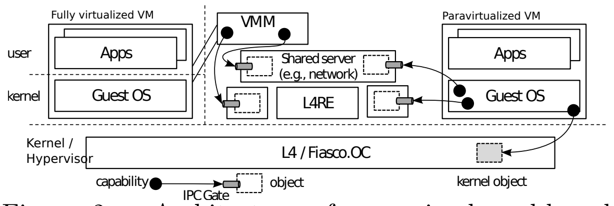

Figure 2: Architecture of our microkernel-based virtualization environment for embedded mixed-criticality systems.

3 Capability-Based Operating System

Our virtualization system consists of the microkernel Fiasco.OC and its user-level infrastructure *L4 Runtime Environment* (L4Re). Fiasco.OC provides both interfaces to run native L4 applications and para-virtualized or fully virtualized guest operating systems and their applications. For the latter, the kernel abstracts from the virtualization support of the underlying hardware platform and reflects this abstracted interface to de-privileged virtual machine monitors. By providing both native and virtualized execution environments it is possible to co-host lightweight tasks implementing the most security and real-time critical functionality with a minimal per application trusted computing base next to full featured real-time or non real-time operating systems and their applications. The trusted computing base of an application is the set of servers that must necessarily be trusted to rely on the functionality provided by this application.

The isolation necessary for such a split application approach is provided by an object-oriented design of both the kernel and the user-level environment with capabilities-based access control. Capabilities in the kernel are tuples consisting of a pointer to a kernel-implemented object and access rights to limit which operations can be invoked on the referenced object. User-level capabilities provide the same functionality by referring to a special kernel object called *IPC gate*, which in turn stores a label through which servers can identify the invoked object and the authority of the invoking client on this object. For security reasons, the values stored in this tuple can only be read by the kernel. Instead, capabilities are addressed through identifiers that are local to the capability owning application or server. Figure 2 illustrates this setup.

The invocation of a capability will trigger a message transfer from the invoking client to the server or kernel which implements the object behind the capability. Because capabilities are only addressed through their local names, the invoking client cannot obtain any information about the physical identity of the server that implements the object. Likewise, upon receiving a request, the object-implementing server receives only the label stored in the IPC-gate but no information about the physical identity of the client. Instead, an implicit capability is provided to the server to reply to the client.

In a nutshell, Fiasco.OC provides the following kernel objects: A *Task* is an isolation boundary that foremost contains an address space and a capability space. *Threads* execute code and run within a task. *IRQs* provide the possibility to trigger asynchronous notifications and are used for both hardware device interrupts and software-based notifications. The kernel also contains a rudimentary *Scheduler* to schedule threads according to the scheduling parameters and with the processor affinity specified by user-level schedulers. A *Factory* creates new objects, such as tasks, threads and IPC gates and manages kernel memory in the form of quotas embedded into these factories.

There are three ways for threads in different tasks to communicate: via synchronous inter-process communication (IPC) messages sent through IPC gates, via shared memory, which has been previously established in the form of special IPC messages and via asynchronous notifications send through IRQ objects. Because all invocations (including those of kernel-implemented objects) are via messages, it is possible to transparently intercept invocations. This enables policies, filters or parameter refinement via proxies. For example, a scheduler proxy can restrict a subsystem to a defined range of scheduling priorities.

The interface provided for para-virtualization and faithful virtualization is realized as special versions of the kernel objects thread and task. Fiasco.OC implements a special form of thread — the *vCPU* — as an abstraction for the execution of a guest CPU. Essentially, a vCPU provides additional storage for the state of the guest OS and an asynchronous entry point [3]. Virtualization events triggered by a vCPU such as guest-to-host physical translation faults, hypervisor calls or trapped privileged instructions are exiting the VM and are forwarded to a user-level *virtual machine monitor* (VMM). For faithful virtualization, the guest physical address space is provided in the form of a specialized task — the *virtual machine* or *VM.* Again, VMs essentially only provide the additional state required for nested paging in guest OSs. To foster strong isolation and minimization of trusted computing bases, a common setup in our system are dedicated VMMs per VM where all security critical functionality (such as device multiplexing) is implemented by trusted servers and the VMM is only responsible of forwarding VM accesses to these native L4 servers. L4Re provides functionality to ease implementing these native servers.

4 Trust Issues in Scheduling Mixed-Criticality VMs

Several schedulers are involved in determining when and where applications and servers should run. At the very bottom of the hierarchy that these schedulers span is the rudimentary scheduler of Fiasco.OC, which follows only the settings of higher-level schedulers when deciding whether to switch to another thread or vCPU. From here on, we adopt the scheduling terminology and call a task an activity that is scheduled on a CPU and that is possibly comprised of a sequence of jobs. The kernel-level scheduler is a partitioned scheduler in the sense that it will never migrate a task on its own. However, higher level schedulers may of course do so by changing the processor affinity of a task. At the next level, there is one or more system schedulers in L4Re with complete control over all CPUs. These schedulers in turn break down this universal guarantee and pass part of it to the schedulers in the VMMs. Finally, there is the scheduler in the guest OS that decides which of its applications to run when and on which CPU.

As we have seen in Section 2, it must be possible to assign different sets of scheduling parameters to the same thread or vCPU. In our example, the task set $\mathbb{T}$ became feasible after we have assigned the vCPU of VM A both a higher prioritized budget than VM B with 1 unit time and an additional 2 unit budget at a lower priority. Of course one possible way to implement this functionality would be to invoke the VMM scheduler and in turn all lower level schedulers down to the system level to change the budget of VM A once it has executed for one unit time. However, the overhead of these upcalls would be significant in particular since we have to reflect asynchronous notifications of a VM such as injected interrupts at a high priority budget since we do not know the internal structure of the guest and in particular which of its tasks is released or unblocks upon reception of this interrupt. Our solution is therefore to split scheduling parameters from the other state that is kept in the thread control block (TCB) of a thread or vCPU. Instead, we keep these parameters in a kernel object called *scheduling context* (SC) [5] and allow multiple of these contexts to be associated with a TCB.

The usage model of SCs is that guest schedulers can select among their available SCs and thus define how their vCPUs are scheduled on the host. The selection typically happens in the context switching path of the guest OS that is called right after a scheduling decision is made. At this point the new task to be run has already been chosen and the OS can determine its "importance" by selecting a corresponding SC. A practical option to select this SC can be an OS-internal priority. This way, the guest does not need to know about other VMs and their internal structure nor must it rely on their correctness to guarantee the completion of higher criticality tasks. Notice though that for lower criticality tasks, the trust dependency on the correct execution of higher criticality tasks remains in the sense that if one of these tasks exceeds their lower criticality execution budget, the completion of these low criticality tasks is no longer guaranteed.

As already mentioned, the described mechanisms of voluntarily changing SCs works only when switching from a higher priority SC to a lower priority one but not in the opposite direction when some event releases a task eligible of running at a higher priority. The reason for this is that the underlying schedulers do not know about this task unless of course we would completely expose all internal scheduling decisions which at the same time negates all arguments for a scheduler hierarchy in the first place. As a consequence, they may prioritize other SCs thereby inverting priorities because the VMs guest scheduler would not receive time to make its scheduling decision. Our solution to this problem is to allow SCs to be bound also to IRQ objects with the semantics that if such an interrupt or notification is posted to the vCPU, the bound-to SC is activated automatically. The system-call to switch SCs is realized as a normal system-call for paravirtualized guests and as a hypervisor-call for fully virtualized guests. The latter is restricted to the guest kernel.

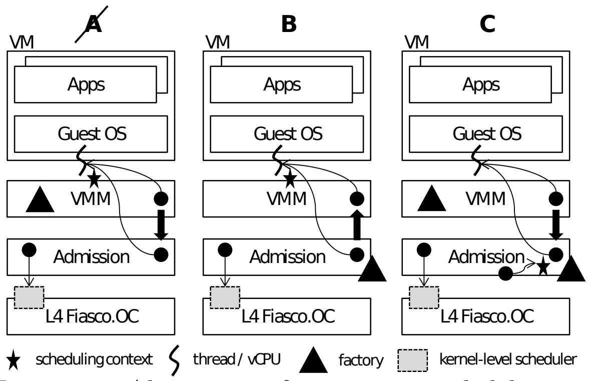

Figure 3: Alternatives for creating and delegating scheduling contexts.

4.1 Managing Scheduling Contexts

Additional scheduling contexts must be created and destroyed, equipped with scheduling parameters and finally selected. Throughout our system different user-level entities are responsible for each of the steps. The schedulers must be able to set and in particular reset these parameters at any time. The range of parameters that they are allowed to set is thereby the more restrictive the higher in the hierarchy such a scheduler is located. Unless all schedulers are fair share, approaches such as lottery scheduling [6] that are based on splitting the guarantees that a scheduler receives from its underlying layer are not as easily realizable. In lottery scheduling, a fixed amount of tickets is passed to higher level schedulers, which in turn may split these tickets before passing them along. Tasks receive a fraction of the CPU time that is proportional to their share of tickets. With priorities and enforced budgets in the form of C_i units time every T_i and up to an explicitly or implicitly given relative deadline D_i, such a split is not as easily done. In particular it is not possible to deduce how much time can be guaranteed when reducing the priority of such an SC or when increasing the period T_i without knowing the decisions made by other schedulers. For this reason, we have constructed our interface such that scheduling parameters have to be validated by all lower level schedulers before they are enforced in the kernel. Because all system-calls have message semantics, invocations of the scheduling system-calls can easily be intercepted by the lower-level schedulers and their functionality can easily be virtualized by invoking the same system-call after validating or changing the passed down parameters using the scheduler's own scheduling-object capability.

Creation and deletion of SCs is by itself not security critical because non-configured SCs (with no parameters set) convey no time on any CPU. However, the questions arise on which quota to account the memory for created SCs and whether SCs should be first class objects with capabilities referring to them. Figure 3 illustrates the difficulties and solutions that may arise. In configuration A, the VMM creates vCPUs on behalf of the guest together with first or second class scheduling contexts. At this point, the vCPU cannot run because unconfigured SCs convey no time on physical CPUs. Therefore, the VMM invokes the scheduling interface of the admission component passing it the thread capability or the SC capability if SCs are first class. Admission in turn invokes the kernel-level scheduler to configure the parameters of the SC (i.e., to set its priority, execution budget and period). At this point the thread may run and scenarios like the one in Section 2 be implemented, provided the admission component must never change the parameters it has assigned to the vCPU (e.g., to lower its priority). The reason for this inverted trust relationship is that the vCPU or SC capability may later on be revoked by the VMM leaving Admission no mean to change the parameters it has assigned. There is no capability or other identifier left in Admission to address the vCPU / SC. Therefore time once granted may never be reclaimed unless all VMMs turn out to be trustworthy at the highest criticality level, or are destroyed.

Scenario B presents a first solution to this problem. Rather than creating vCPUs in the VMM, the creation is done by Admission using a factory and the therein embedded quota that is part of the trusted computing base of this necessarily trustworthy server. The thread capability is then passed to the VMM which may create and attach the SCs. Because the Admission has created the vCPU, it has a non-revocable identifier — the vCPU capability — for as long as the vCPU lives. In this setting it is unproblematic that the VMM creates or destroys SCs because there is always a way for Admission to set and reset their parameters by addressing SCs indirectly through the vCPU.

In Scenario B, the entire memory for vCPUs has to origin from trusted factories and their quotas. In particular for heavily multi-threaded servers and guests with large numbers of CPUs, this allocation may place burden on Admission to properly manage the kernel resources they require. Scenario C offers an alternative where Admission in only responsible for scheduling-related object, that is, the scheduling contexts but not the much larger TCBs of threads and vCPUs. Like in Scenario A, vCPUs are created by the VMM and passed down in a revocable fashion to Admission. However, instead of creating a second class SC, which can only implicitly be addressed through threads, Admission now creates a first class SC with its own capability. Therefore, even if the VMM later on revokes the thread/vCPU capability, an identifier to the created SCs remain within Admission to change or reset scheduling parameters.

For the above reasons, we target first class scheduling contexts to facilitate trustworthy scheduler hierarchies for flattened mixed-criticality VM scheduling.

5 Conclusions

In this paper we have discussed trust and security concerns in the design of a scheduling interface for mixed-criticality virtual machines in embedded real-time systems. By separating first class scheduling contexts from the actually executing entities (threads and vCPUs), user-level schedulers maintain control over all scheduling related concerns without having to manage the execution resources they schedule.

Fiasco.OC and L4Re available from http://os.inf.tu-dresden.de/L4Re/.

References

[1] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. In *Mathematical Foundations of Computer Science 2010*, volume 6281 of *LNCS*, pages 90–101. Springer Berlin / Heidelberg, 2010.

[2] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:13–22, 2010.

[3] A. Lackorzynski, A. Warg, and M. Peter. Generic Virtualization with Virtual Processors. In *Proceedings of Twelfth Real-Time Linux Workshop*, Nairobi, Kenya, October 2010.

[4] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig. Flattening Hierarchical Scheduling. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '12, pages 93–102, Tampere, Finland, 2012.

[5] U. Steinberg, J. Wolter, and H. Härtig. Fast Component Interaction for Real-Time Systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, Palma de Mallorca, Balearic Islands, Spain, July 2005. IEEE.

[6] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11, Monterey, CA, USA, November 1994.

[7] S. Xi, J. Wilson, C. Lu, and C. Gill. RT-Xen: Towards Real-time Hierarchical Scheduling in Xen. In *Proceedings of the 11th International Conference on Embedded Software*, EMSOFT, Oct. 2011.