Scheduling Policies and System Software Architectures for Mixed-criticality Computing

Soham Sinha
Computer Science Department
Boston University, Boston, USA

ABSTRACT

Mixed-criticality model of computation is being increasingly adopted in timing-sensitive systems. The model not only ensures that the most critical tasks in a system never fails, but also aims for better systems resource utilization in normal condition. In this report, we describe the widely used mixed-criticality task model and fixed-priority scheduling algorithms for the model in uniprocessors. Because of the necessity by the mixed-criticality task model and scheduling policies, isolation, both temporal and spatial, among tasks is one of the main requirements from the system design point of view. Different virtualization techniques have been used to design system software architecture with the goal of isolation. We discuss such a few system software architectures which are being and can be used for mixed-criticality model of computation.

1 INTRODUCTION

Computing in embedded systems has been increasingly moving towards a new system model, Mixed-criticality System (MCS). MCS was first introduced by Vestal in his seminal paper [53] in 2007. This new kind of system is targeted towards the new class of timing-sensitive platforms such as autonomous cars, drones, avionic systems. In MCS-es, tasks (or in usual systems terminology, processes) are divided into multiple levels of *criticality*. A task is assigned a higher criticality level when a failure to meet the deadline of that task may result in severe consequence for the whole system. For example, DO-178C document reports five levels of failure conditions from "Catastrophic" (Level A) to "No Effect" (Level E) [43]. Accordingly, tasks can be categorized to certain levels of criticality, based on their effect on the system. Additionally, a whole system can also be classified to certain levels of criticality, based on its current condition.

In higher criticality level of the system, it may not be necessary to run the lower criticality tasks as higher criticality tasks may suffer from interferences from the lower criticality tasks. Higher criticality tasks are more prone to be the deciding factor of the running condition of the whole system. For example, in a drone, a low criticality *data logging* task may be stopped in case of critical condition (e.g., thrust of wind) of the drone, when higher criticality flight controller task should not be interfered. Conversely, in lower criticality level of the system, more tasks ranging from higher to lower criticality level may all run together. To facilitate this different

model of execution, tasks of different criticality levels need to be have different set of parameters based on their criticality level. The usual parameters of real-time tasks, (Runtime, Period and Deadline) must be adjusted to suit the execution model. Additionally, we need scheduling algorithms for this new task model.

There has been numerous papers since Vestal's work, which discuss different scheduling algorithms and priority assignment strategies for such model of Mixed-criticality System. In this report, we will discuss the dominant and established fixed-priority (FP) task-scheduling model for uniprocessor scheduling. The other approaches, dynamic priority scheduling algorithms have extra runtime overhead, which are better to be avoided in time-critical computing. Additionally Baruah and Vestal have proved that the mixed-criticality model of computation does not necessarily help in schedulability of dynamic priority scheduling algorithms, as it does for FP scheduling algorithms. Therefore, we concentrate on FP scheduling algorithms.

While theoretical analysis of the scheduling algorithms for uniprocessors forms the foundation of MCS, implementation of such systems on top modern hardware also needs to be explored. This is more interesting as the underlying embedded hardware has been increasingly supporting multicore processors. Moreover, the main goal of these systems is to achieve isolation of execution, both temporally and spatially. Temporal isolation provides timing guarantee. Spatial isolation not only helps in meeting timing constraints, but also increases overall safety and security of the system. In this report, we explore the possible approaches in the architectural design of the system software for the mixed-criticality model of computation on top of modern multicore hardware.

Historically virtualization techniques have been used in systems to attain isolation. Isolation helps fault-containment to a specific component of the system and prevents spreading to the other parts of the system. This improves the safety of the system from, for example, the external environment which may corrupt a part of the stored data. Isolation also increases security so that an attacker cannot break into the whole system easily. Many approaches try to segregate the system into a number of domains of execution. In each of these domains, we may need to run the uniprocessor mixed-criticality scheduling algorithms, said earlier.

Our contributions of this report are the following:

(1) We describe different static priority assignment strategies for mixed-criticality systems. We associate

those strategies with runtime scheduling policies and discuss a number of their variants. (Section 2)

(2) We explore five system software architecture for mixed-criticality model of computation, where isolation is the primary goal: Standalone OS, Microkernel, Hypervisor, Microvisor, Partitioning Hypervisor. (Section 3)

2 MIXED-CRITICALITY TASK SCHEDULING POLICIES

The primary motivation behind the mixed-criticality systems (MCS) is to adjust the task parameters and control the tasks based on the criticality level. One of the most important task parameter in real-time systems is the execution time of a program. It is estimated by the Worst-case Execution Time (WCET) which acts as an upper bound. Traditionally, it has been hard to accurately estimate the WCET of a program [54], especially because of the complexity of modern hardware. Moreover, conservative estimation of WCET leads to under-utilization of system resources in already resource-constrained environments like embedded and real-time systems. Vestal initiated the research to develop a more practical, alternative *mixed-criticality* model for timing-sensitive systems [53]. In this section, we first elaborate Vestal (task) Model. Then, we discuss the fixed-priority scheduling algorithms around the model.

2.1 Vestal's Task Model

A task in most of the mixed-criticality taskset, following Vestal's task model or Vestal Model [53], is a 4-tuple: Computation time at different criticality levels, period, deadline, criticality level of the task. Therefore, a task τ_i can be defined by $(C_i(l), T_i, D_i, L_i)$. The task parameters are explained below:

– T_i : The period (or rate) of a task or minimum inter-arrival time.

– D_i: The deadline of a task by which it should complete its execution. Most mixed-criticality models assume implicit deadline, i.e., the task period, T_i, is equal to the deadline. Most of these models are extensible to arbitrary deadlines.

– L_i: The criticality level of a task. A system developer decides the criticality level of a task based on the task's importance to the system. For example, if the failure to meet the deadline of task results into "Catastrophic" consequence based on DO-178C standard [43], then the task could be categorized as "Level A" task. The numerical value is usually from 1, referring to "Level E (No Effect)" to 5, referring to "Level A (Catastrophic)". It is important to note that there can even more or less than five levels of criticality. When

Vestal introduced the model, he referred to only four criticality levels. However, there are other studies which considered even up to thirteen levels of criticality [7].

– $C_i(l)$: The computation time or runtime budget of a task at criticality or assurance level l. The value is usually derived by WCET estimates with varying levels of *confidence* or *assurance*. C_i increases (or stays same) with the increment of l which represents the assurance level. The probability of a task missing its deadline decreases with the increment in the assurance level, l, because C_i increases with l, which means that the task gets higher runtime to finish its computation. Assurance level thus directly relates to criticality level of the system because tasks certified with higher level of assurance have lower probability to failure. Therefore, l is taken from the set of values in criticality levels (1 to 5 in case of DO-178C).

The computation time of a task is usually assured to its own criticality level or lower criticality level than its own. Therefore, we can say that a task with $(C_i(l), T_i, D_i, L_i)$ parameters has the following property for computation time:

$$
\begin{aligned}
C_i(l) &= C(l_i) & (l_i < L_i) \\
&= C(L_i) & (\text{otherwise}) \\
&\text{and} \\
C(L_i) &\geq C(l_i)
\end{aligned}
$$

Therefore, the lowest criticality task has only one WCET estimate for computation time.

2.1.1 Burns Model: Burns model [11] relaxes the assurance levels for computation time of a task from arbitrary number of levels in Vestal Model to just two levels. He argues that it is impractical and unlikely to derive WCET estimates for task's computation time parameter with more than two levels assurance. Therefore, tasks will have two WCET estimates: 1) for *normal* mode of operation where the task is assured to the lowest criticality level. 2) for *self* mode of operation where the task is assured to the same criticality level as its own. For example, a task τ_i with criticality level 5, will have two WCET estimates: $C_i(L_1)$ - referring lowest assurance level, $C_i(L_5)$ - referring assurance level to its own criticality level. To generalize, every task's two WCET estimates will be $C(self)$ and $C(normal)$, where $C(self) \geq C(normal)$. For lowest criticality level tasks, the values would be same.

2.1.2 Concept of Criticality Mode: We will see in the next subsection that many research works utilize a concept of *criticality mode*. The criticality modes are interconnected with the criticality levels. A mixed-criticality system usually starts in the lowest criticality mode. If all tasks run according to their lowest criticality level, the system stays in the lowest criticality mode. Based on the scheduling policy, the system can be changed to a higher criticality mode when
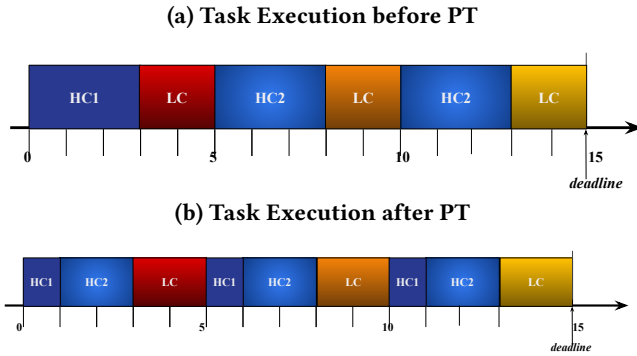
(a) Task Execution before PT

(b) Task Execution after PT

Figure 1: Period Transformation (PT) Example

Table 1: Taskset for Period Transformation Example

Task	Criticality Level	C	T	Priority
HC1	High-criticality	3	15	1
HC2	High-criticality	6	15	2
LC	High-criticality	2	15	3

some task tries executing more than their lowest criticality runtime budget. This concept of mode is applicable for a whole system. Mode can also be task-specific. There are research works going on task-specific and system-wide mode change protocols and policies. Those works are out of the scope of this report. We focus on the scheduling policies which utilize the concept of criticality mode in this report.

2.2 Fixed-priority Algorithms

If the tasks do not change their priority in their entire lifetimes, those algorithms are called fixed-priority (FP) algorithms. The priorities of the tasks are decided offline based on some strategies. We discuss these priority assignment strategies in Section 2.2.1. At runtime, there can be different scheduling policies applied for the mixed-criticality nature of the system. The foremost and predominant motivation of the scheduling policies is to ensure that more or preferably all higher criticality tasks meet their timing constrains. Some scheduling policies also try to ensure a minimum level of service guarantees for lower criticality tasks. We discuss a few important such policies in Section 2.2.2. Finally, we describe variations in Vestal Model and scheduling policies in Section 2.3.

2.2.1 Priority Assignment: We enlist the most used static priority assignment strategies.

(1) **Rate-monotonic (RM) / Criticality-monotonic (CM):** Tasks can be assigned priorities statically based on their rate or period (rate-monotonic) or criticality level (criticality monotonic). Both of these strategies show poor results, in terms of successfully scheduling randomly generated tasksets [25]. Also, unmodified RM is not optimal FP strategy for MCS, as it is for the periodic tasks. RM scheduling is a special case of deadline-monotonic (DM) scheduling where deadline is equal to period. DM scheduling also demonstrates similar results as RM scheduling, for mixed-criticality tasksets.

(2) **Period Transformation (PT):** In his first paper [53] about mixed-criticality systems, one of Vestal's suggested priority assignment strategies was Period Transformation [45]. In this approach, periods of all higher criticality tasks are first divided by an integer, so that their periods are lower than (or equal to) the comparatively lower criticality tasks. Then, tasks are scheduled according to RM priority assignment. Ultimately PT with RM scheduling becomes a criticality-monotonic priority assignment.

To better understand PT, we show an example the taskset in Table 1. We assume the priority is manually assigned for the taskset. Then, without PT, the taskset is scheduled like in Figure 1a. If we apply PT, then we need to divide the runtime and period of HC1 and HC2 by 3, so that they are at least equal to the lower criticality task LC's period 5. Therefore, the resultant taskset schedule is given in Figure 1b, where HC1 and HC2's (runtime, period) are respectively (1, 5) and (2, 5). We need to understand that each LC instance in Figure 1 is a new instance of the LC task. However, in Figure 1b, HC1 and HC2 actually are preempted and resumed multiple times.

(3) **Audsley's Algorithm:** The prevalent priority assignment strategy for mixed-criticality scheduling policies is Audsley's priority assignment algorithm [2]. We describe the algorithm in a generic sense here. We tell the difference in adoption of the algorithm wherever needed. First, a task is selected among all the tasks and assigned the lowest priority. The current task's schedulability is evaluated assuming all other tasks as higher priority tasks in a given scheduling policy. The algorithm recursively checks schedulability of remaining tasks until there are no tasks left, or no task is schedulable at some priority level when the taskset is declared not schedulable with the particular scheduling policy using Audsley's Algorithm.

Dorin et al. proved that the Audsley's priority assignment strategy is optimal for mixed-criticality tasksets [16]. The primary property for the optimality of Audsley's algorithm and its proof comes from the phenomenon that a task schedulable at some priority level is still schedulable at a higher priority level. The optimality of Audsley's Algorithm just

means that, if a taskset can be scheduled with any other FP scheduling algorithm, then it should also be schedulable by Audsley's Algorithm.

2.2.2 Scheduling Policy: Given a priority assignment strategy, scheduling policy for mixed-criticality systems decide how the tasks should be managed at runtime. In case, a task is overrunning its WCET estimate at some criticality level, a runtime rule is enforced. There are many runtime scheduling policies proposed over the years. We will discuss important few of those in this section and which previously discussed priority assignment strategies they utilize.

To check the schedulability of a taskset under a scheduling policy with a priority assignment strategy, many papers since Vestal's initial paper use response-time recurrence equation proposed by Joseph and Pandya [28]. Response-time of a task is the worst-case total time between the arrival of a task and its completion. It is usually calculated by summing up a task's WCET and interference by other tasks. The interference is derived by measuring the runtime of other higher priority tasks within the current task's response-time. The higher priority tasks can be invoked multiple times as well. Finally, the equation becomes a recurrence relation which is solved by iterating over multiple times, with initial response-time being equal to the task's WCET. For example, assume a task τ_i has period as T_i and WCET as C_i and its higher priority tasks are represented by a set $hp(\tau_i)$. Then, the response-time of τ_i would be, $R_i = \left(C_i + \text{Interference from } hp(\tau_i)\right) = C_i + \sum_{\tau_j \in hp(\tau_i)} \left(\left\lceil \frac{R_i}{T_j} \right\rceil \times C_j\right)$.

We now discuss different scheduling policies.

(1) Static Mixed-criticality (SMC): The first scheduling policy proposed for mixed-criticality systems was the SMC policy by Vestal [53]. In this policy, tasks are allowed to be executed for their WCETs at their own criticality levels. To be more precise, high-criticality tasks are contracted for their higher criticality level, and low-criticality tasks are contracted only for their lower criticality levels. Therefore, when the high-criticality tasks overrun their lower criticality level estimate, they can still execute up to their higher criticality execution time. Consequently, the higher criticality tasks never miss their deadlines at their criticality level. However, the lower criticality tasks may miss their deadline because of the overrunning of a higher criticality task. This behavior is fine because the lower criticality tasks are not assured to meet their deadline at a higher criticality level.

SMC can use RM scheduling, CM scheduling, PT and Audsley's algorithm as its priority assignment strategy. However, as Huang et al. showed [25] Audsley's algorithm admit more tasksets in randomly generated tasksets. An unmodified RM and CM scheduling perform worse than PT or Audsley's algorithm.

(2) Adaptive Mixed-criticality (AMC): Baruah et al. proposed AMC [10] by methodically formalizing the response-time equation. In this policy, there is a system-wide *criticality mode* which is initialized to the lowest criticality level at the beginning of the system. Whenever a higher criticality task executes for more than their lower criticality WCET estimate, the system-wide mode is changed to the next criticality level. The current higher criticality task is then allowed to be executed till the WCET at the current criticality level. Simultaneously, all lower criticality tasks than the current criticality level are dropped to minimize interference by these lower criticality tasks.

AMC uses Audsley's priority assignment strategy to decide task priorities offline. Although AMC's scheduling policy can be fused with other static priority assignment strategies, it has not been done yet, to the best of our knowledge. AMC admits more tasksets than SMC with Audsley's algorithm and PT for randomly generated tasksets [25] because of dynamically dropping all low criticality tasks in a higher criticality mode. However, Baruah et al. showed that SMC with PT and AMC with Audsley's Algorithm are theoretically incomparable because there are tasksets which are schedulable by PT but not by AMC and vice-versa [8]. They concluded that the benefit of PT is because of its inherent ability to turn a taskset into a harmonic one, resulting into better schedulability of tasksets.

(3) Slack Scheduling: Niz et al. introduced [15] a scheduling policy based on Vestal model but has similarities with Burns model. The policy is described for dual-criticality (two criticality levels) tasksets. Higher criticality tasks have two computation estimates: 1) a normal computation time (C^{normal}), 2) an overloaded computation time ($C^{overloaded}$) ($C^{overloaded} \geq C^{normal}$). With this policy, tasks are first executed with the assumption of running for their normal computation time. This mode of execution is called *normal mode*. Lower criticality tasks are prevented to run, only when the high-criticality tasks are overrunning their normal computation budget and have to be continued to meet their deadline. This restricted mode is called *critical mode*. The authors proposed an offline algorithm to calculate a time-instant, called *zero-slack instant*, by which the higher criticality tasks need to be turned on for their *critical mode* execution, and lower criticality tasks should be suspended. Overall, lower criticality tasks are essentially run in the slack time generated by the higher criticality tasks running in their *normal mode*.

Slack Scheduling uses RM scheduling to determine priority. However, Niz et al. point out that any other priority-based preemptive scheduling can be used, not only fixed-priority ones. Therefore, Earliest Deadline First algorithm can also be used with Slack Scheduling, but not yet done.

Table 2: Association between Scheduling Policies and Priority Assignment Strategies

Scheduling Policy	Implemented Priority Assignment Strategies
Static Mixed-criticality (SMC)	Rate-monotonic or Period Transformation or Audsley's Algorithm
Adaptive Mixed-crititcality (AMC)	Audsley's Algorithm
Slack Scheduling	Rate-monotonic

2.3 Variations

We have discussed the main categories of fixed-priority scheduling algorithms for Mixed-criticality Systems. We associate the scheduling policies with their respective and tried priority assignment strategies in Table 2 as a summary. We now describe some variations of the above mentioned algorithms in order to cater to different objectives, mostly related to low-criticality tasks.

2.3.1 Variable Task Period: Most of the scheduling policies are based on multiple estimates of WCET of a task. However, in real-world, the dynamic behavior of a task may not only affect its execution time budget, but also the period or rate at which the tasks are needed to be executed. For example, when an autonomous car suddenly discovers an object in front, it may need to communicate to the driving controller more frequently than normal. Then, the tasks related to the driving controller should change its rate of execution, instead of their computation budget. There are research works which change the Vestal Model to allow multiple values of periods (minimum inter-arrival time) at different criticality levels.

Baruah first introduced the concept of multiple periods or T in the mixed-criticality task model [5] in order to comply with the requirements of the Certification Authorities (CAs) like Federal Aviation Administration in the USA. A conservative smaller value of period ($T(HI)$) is added in the task model in addition to a *normal* value for period ($T(LO)$). In this case, $T(HI) \leq T(LO)$. For low-criticality tasks, the two values of periods are the same. In this extended model, each task is allowed to complete within $T(HI)$ which is its implicit deadline. However, a task's inter-arrival time may still be higher than $T(HI)$ i.e, $T(LO)$ in the LO-criticality mode. It is assumed that a CA will have restricted smaller value of period like $T(HI)$, while a system designer would try to maximize the utilization of the system by a relaxed value of period with $T(LO)$. Later, Baruah extended the model [4] to allow multiple periods to satisfy the basic functioning of the system such as the driving control example given above.

Both these works suggest algorithm based on dynamic priority Earliest Deadline First algorithm, rather than our current topic of Fixed-priority algorithms. Nevertheless, Baruah and Chattopadhyay have done [9] a response-time analysis for multiple estimations of periods with fixed-priority strategy, by using response-time equations found in their work on AMC [10]. They again use Audsley's Algorithm to determine fixed priorities of the tasks offline.

Another task model, named Elastic Mixed-criticality (E-MC), also explores the idea of variable periods to improve the service of the low-criticality tasks [51]. In most mixed-criticality scheduling policies, low-criticality tasks are abandoned whenever there is a *change of mode* in the system. E-MC allows variable periods for low-criticality tasks where largest period is derived from a minimum service requirement of a low-criticality task. Additionally, E-MC also proposes an EDF-based Early-Release (ER-EDF) algorithm which can release low-criticality tasks early in the slack generated by high-criticality tasks, to improve the service of the low-criticality tasks. However, the low-criticality tasks may also be constrained by minimum rate of execution where an early release would not be possible. To accommodate the issue, E-MC defines a set of *early-release* points in the model parameter itself. Furthermore, Su et al. used the model for dual-criticality systems as Dual-criticality Mixed-criticality model (DR-MC) [50] and posited the scheduling problem in the context of fixed-priority scheduling as non-linear optimization problem [49]. Another technique uses a similar and related concept of extending the deadline of low-criticality tasks by including a predefined *stretching factor* (for the periods) in the task model [27].

2.3.2 Managing Low-criticality Tasks: High-criticality tasks in mixed-criticality systems are most important and always needed to be finished within their deadlines. However, low-criticality tasks have some flexibility in its execution requirement. There are research works which try to exploit this flexibility to experiment with the execution of the low-criticality tasks. Burns and Baruah discussed a few ways to manage the low-criticality tasks, so that they have some level of service guarantee, even in the high-criticality mode [13]. One of the proposed topic is similar to the E-MC task model [51] described earlier. Among others, they also propose using a smaller runtime budget for low-criticality tasks at higher criticality level. Therefore, for low-criticality tasks, $C(LO) \geq C(HI)$. They present a modified response-time analysis with the new task model and AMC scheduling policy, assuming Audsley's priority assignment algorithm. In later research works, this model is also called Imprecise Mixed-criticality or IMC model [36]. However, no evaluation has been done for the new model for fixed-priority schemes.

Most of the mixed-criticality systems drop all the low-criticality tasks once the *system-level mode* changes to HI. We have described a few modified task models and scheduling policies above (e.g., IMC/E-MC) which allow the low-criticality to degrade its service when high-criticality tasks overrun their LO-criticality budget. Low-criticality tasks can be managed in various other ways as well. For example, MC-ADAPT develops a dynamic task model where only a few low-criticality tasks are dropped in case high-criticality tasks overrun their LO-criticality budget [32]. As the authors extend the dynamic task priority EDF-VD algorithm [6] for MC-ADAPT, they use a utilization bound test online to determine the low-criticality tasks to drop, starting from the one with higher utilization. However, there has been no work on adaptively dropping selected few low-criticality tasks for fixed-priority scheduling algorithms, to the best of our knowledge.

3 SYSTEM SOFTWARE ARCHITECTURES FOR MIXED-CRITICALITY SYSTEMS

With the rise of multicore architectures in the embedded systems space, there have been many algorithms for multicore mixed-criticality systems. Most of the algorithms are extensions of the uniprocessor scheduling that we have mentioned above. Rather than focusing on the theoretical side of these algorithms, we will now look at more practical implementations of mixed-criticality systems. Many of the implementations separate executions into multiple sandboxes to provide spacial and temporal isolation to these isolated domains of executions. These sandboxed domains can be defined in terms of criticality, security, importance, relevance to the system or some combination of them. Notwithstanding, these isolated domains still need the notion of mixed-criticality task execution because of the nature of the system. Therefore, previously discussed FP scheduling policies could be implemented in these sandboxes.

Isolation in terms of time and space is one of the key driver of the mixed-criticality systems implementations. Traditionally, virtualization has been used by the systems research community to provide isolation of execution within a single system. Additionally, the philosophy of microkernel, where the kernel only provides the most essential functionalities like address space, threads of execution and interprocess communication, has been also used to provide isolation in systems. We discuss few of system software design paradigms which can be and are being considered for mixed-criticality computing. Table 3 provides a summary of all these architectures.

3.1 Standalone OS

Huang et al. implemented a few fixed-priority mixed-criticality scheduling policies in user-space Linux to measure the overhead of specific approaches objectively [24, 25] and provide such comparative evaluations for the first time. They discovered that the overhead (in terms of increment in busy period) of runtime policy enforcement for Slack Scheduling and AMC is bounded by 3% for 32 tasks. Moreover, Period Transformation priority assignment strategy which increases the number of context switches because of short bursts of task-dispatches, incur only a ~2% overhead for 20 tasks, but increases quickly after that. They used signals and timers for the accounting of task budgets and periods. The authors conclude that the real-world tasksets which can be theoretically proven to be schedulable by simple strategies like SMC should be using those relatively straightforward runtime scheduling policies.

LITMUS^{RT}, based on Linux, was one of the first full-fledged OS-es to propose using mixed-criticality scheduling [1]. The authors extended the work to include different scheduling policies to be applied to tasks of different criticality levels [22]. The latter work tries mitigating more issues such as scheduling overhead, interrupt handling. For example, interrupts are redirected to only one core to reduce interference to the other cores. However, using a monolithic complex kernel like Linux is an issue. Such a bloated kernel leads to high overhead due to contention to the shared resources, poor security and safety, a significant concern in mixed-criticality systems.

HIgh Performance Parallel Embedded Real-time Operating Systems (HIPPEROS) [40] thus takes a microkernel based approach to develop an OS for mixed-criticality systems [41]. They separated out the lightweight tasks of the kernel such as simple system calls and context switching to one core while another core handles heavyweight part like scheduling, resource handling. They implemented E-MC [51] with simple provisions for mode-changing. The work is still going on in HIPPEROS.
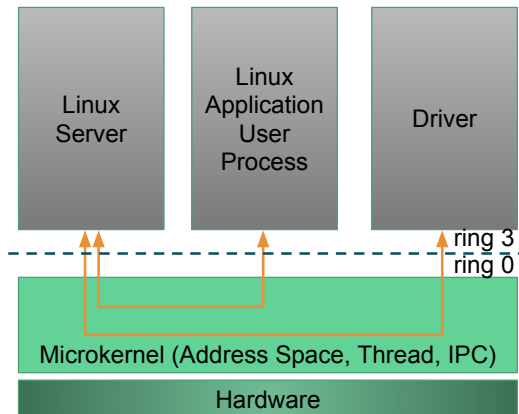
3.2 Microkernel

In an effort to increase the security and safety of the system, the componentization of certain codebase was tried out. Microkernels came out of such necessity of bringing isolation as one of the key characteristic of system development. Microkernel thus reduces the Trusted Computing Base (TCB) of the system because it minimizes the amount of lines of code (LOC) run at the highest privilege level to access all the hardware capabilities. Only most essential services such as address space management, IPC and thread management are done in the microkernel and run in kernel space (ring 0

Table 3: Comparison between different System Software Architectures for Mixed-criticality Computing

Name	Philosophy (Size of TCB)	Architecture	Scheduling	Interrupt Handling	Inter-domain Communication
LITMUS^{RT}	Standalone OS (Full Kernel)	Tasks run in *containers* as budgeted servers.	P-EDF and G-EDF in servers.	non-real-time	Linux IPC Mechanisms
HIPPEROS	OS with micro-kernel flavor (Full Kernel)	Master core is heavyweight (scheduler, handling shared resource) and Slave core runs applications and handles lightweight kernel functionalities.	E-MC and a fixed-priority scheduler with budgeted servers	Master core handles.	Message passing and shared memory
PikeOS	microkernel (essential microkernel services)	Tasks, drivers run as services in user-space	Priority-based + time-partitions	Based on assigned priority of interrupt-task.	Message passing
RT-Xen (or XtratuM)	hypervisor (Xen hypervisor layer with Dom0)	Paravirtualized Guest OS-es running on top of bare metal hypervisor.	Preemptive fixed-priority scheduling of guest OS-es through different classes of servers.	"Split-driver" model where all interrupts are mediated by Dom0 with lightweight event mechanism.	As network packets and shared memory.
OKL4 (or NOVA)	microvisor (thin microvisor layer and some dependency on the VMM and device drivers)	3-layered where microhypervisor is a thin layer on top of hardware; a VMM, device driver layer on top of it; finally guest OS-es on top of VMM layer; each VM gets a separate VMM	Preepmtive fixed-priority round robin scheduling for guest OS-es.	Coordinated by the VMM and co-located device drivers to the guest VMs through virtual interrupts.	microvisor and VMM coordinated message passing with capability controls.
Quest-V	separation kernel and partitioning hypervisor (thin Quest-V VMM)	Quest-based sandbox directly on one or more cores. Paravirtualized Linux directly on other cores. A thin VMM for every sandbox.	Quest - RMS and AMC [39]. Other guest OS-es own scheduling policies.	Mapped to specific cores. Quest - priority inherited interrupt handling. Otherwise, based on guest OS policy.	Shared memory
Jailhouse	partitioning hypervisor (the thin hypervisor layer and parts of the Linux kernel)	Linux-based one sandbox directly on one core. Other guest OS-es directly on other cores.	Based on Linux and other Guest OS-es.	Mapped to specific cores and their guest OS-es.	Shared memory

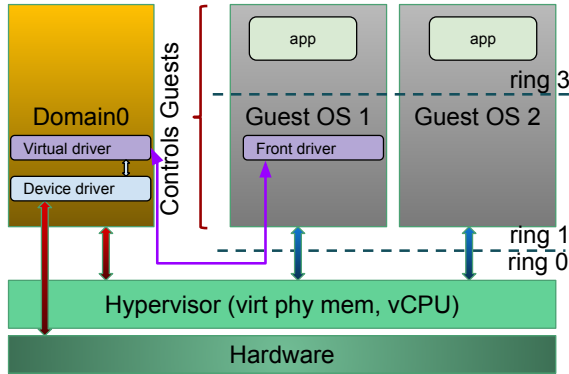**Figure 2: Architecture of Microkernel Philosophy**

protection mode in x86). Rest of the functionalities including device drivers are executed in user-space (ring 3 in x86 protection mode). The interactions between the components are done via message passing (IPC) through the microkernel. Figure 2 portrays the philosophy behind microkernel. Because of smaller TCB, seL4 microkernel was one of the first kernels to be formally verified [31]. seL4, implemented in C, was based on the L4 microkernel family [34, 35]. L4 microkernel already demonstrated desired performance in terms of limiting the overhead and minimizing IPC cost with a *paravirtualized* Linux running on top of L4 [19].

PikeOS [29] is another microkernel which has evolved from the concepts and goals of L4. However, L4 or the L⁴Linux [19] was not geared towards the real-time systems where time is a valuable resource. In PikeOS, time is accurately accounted. Therefore, there are supports for event-triggered (e.g., interrupt-driven) and time-triggered (e.g - periodic) applications. Tasks in PikeOS can have priorities, and time is accounted for those tasks. However, a higher priority task can still exhaust and block other tasks in the system. Therefore, tasks priorities need to be assigned properly so that interrupts are processed in a timely manner. However, PikeOS is not entirely suitable to provide temporal isolation

Figure 3: Architecture of Xen Hypervisor Philosophy

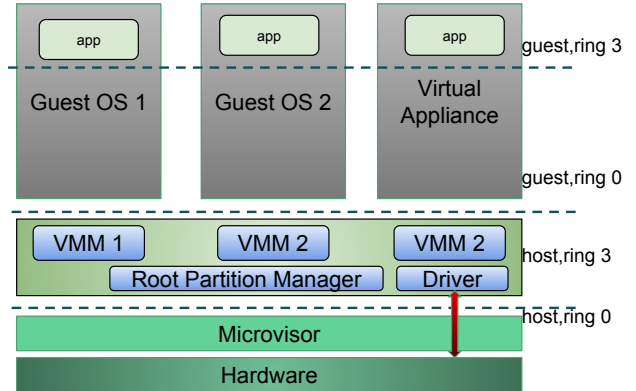**Figure 4: Architecture of the Microvisor Philosophy**

to tasks, which is essential to mixed-criticality systems. Although PikeOS is not fully tuned to support mixed-criticality workload, Vanga et al. demonstrated how a mixed-criticality system can be designed with PikeOS [52]. They mainly tried to support low latency low-criticality tasks with their scheduling policy. In addition, they port the PikeOS scheduling architecture to the LITMUS[RT] [1] because of the proprietary nature of PikeOS.

3.3 Hypervisor

XtratuM is a minimized hypervisor which was initially designed as a *nanokernel* [37]. Linux was essentially used to boot the system up, after which a XtratuM kernel module would take over the system. It was very similar to Linux Kernel-based Virtual Machine (KVM) [30] where Linux worked as the host OS, while QEMU [12] and KVM could work as the hypervisor. However, XtratuM was redesigned as a full-fledged bare-metal hypervisor [38]. Paravirtualized guests such as Linux or their own in-house LithOS [42] can be run on top of the hypervisor. The hypervisor uses a fixed cyclic scheduling to schedule each domain which is called "partition" by the authors. The interrupts are virtualized and managed by the hypervisor. Therefore, a high-priority task's interrupt may be deferred because of the virtualization. Interestingly, the authors enable only the interrupts associated with the currently executing partition. The hypervisor detects the other pending interrupts and defer them.

RT-Xen [55] is another full-fledged hypervisor like XtratuM, but based on the Xen virtualization infrastructure [3]. There is much in common between RT-Xen and Xtratum, as the authors of RT-Xen admit as well. The general philosophy of Xen-based hypervisor is captured in Figure 3. The hypervisor layer sits on top of hardware and virtualizes the CPU, physical memory for the guest OS-es. But in contrast to XtratuM, Xen has a special *Domain0* as the main hardware controlling authority and mediator between the guest

domains and the hypervisor. Device interrupts are routed through Domain0 in a "split-driver" model. Additionally, RT-Xen has multiple inter-domain communication mediums, originally offered in Xen. RT-Xen also offers many classes of server containers such as Periodic Server, Deferrable Server. Xen (and consequently, RT-Xen) also has wider community support. Although both XtratuM and RT-Xen can be used in mixed-criticality context, handling interrupts in a timing predictable manner is an issue, because interrupts are virtualized and queued in event notification channel.

3.4 Microvisor

The systems research community has been debating [18, 21, 23] over advantages and disadvantages of microkernel and hypervisors as virtual machine monitors (VMMs). Researchers have noted many common goals of microkernels and hypervisors [20]. Among them, reducing the TCB is one of the primary goals. There has been efforts to combine these streams of research together to build a unified model named *microvisor*, coined by Gernot Heiser [20]. It is essentially taking the idea of microkernel and applying to the hypervisor layer, instead of the OS. We now discuss about the major efforts related to microvisors and how this approach can be beneficial to mixed-criticality systems. Figure 4 encapsulates the basic idea of a microvisor based architecture.

OKL4 [20] was one of the first efforts to combine the two separate research streams together to show the viability of the microvisor approach. Microvisors in general have a three-layered architecture where the thin microvisor is the most privileged layer just on top of the hardware. Multiple VMMs managing Virtual Machines (VMs), device drivers and a policy enforcement component as *root partition manager* sit on top of the microhypervisor. Each Guest VM sits on top of a VMM and interacts with the VMMs when necessary through message passing. The microvisor approach separate policy
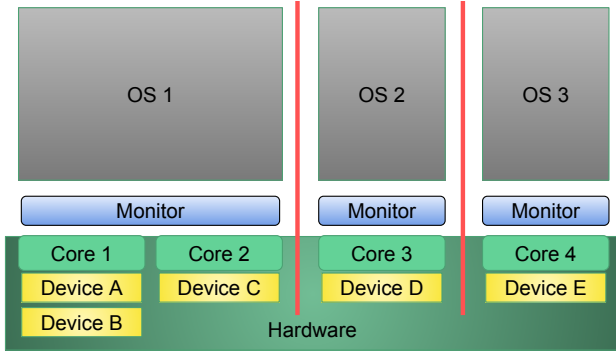
Figure 5: Architecture of the Partitioning Hypervisor Philosophy

and mechanism to different layers. Instruction execution and memory are virtualized through a virtual CPU (vCPU) and virtual MMU just like the hypervisor. Hardware support like Intel VT can help in doing some of the virtualizations to be done in hardware. NOVA [48] is another microvisor which functionally decomposes the hypervisor into minimal components. The most privileged component runs essential hypervisor-related services like scheduling, communication channel establishment and results into just 9k LOC. Thus, the microvisor architecture is a significant step in isolating major components of the kernel and also providing strong isolation to the guest OS-es. However, virtualizing interrupts, CPUs and drivers may incur cost significant to the timing-sensitive system. Additionally, the VMM layer (host, ring 3) is significantly bloated and privileged. Therefore, the trade-off between isolation, security and performance needs to be weighed.

3.5 Partitioning Hypervisor

Quest-V [33] is developed with the philosophy of a *separation kernel* [44]. It segregates execution in multiple spatially and temporally isolated domains in different sandboxes. Each sandbox is statically given dedicated number of CPU cores and hardware devices, as it can be seen in Figure 5. Because of static partition, the architectural design is called Partitioning Hypervisor. The philosophy of a separation kernel moves one step ahead and omit any side channel between the sandboxes. Quest-V has been designed and is being developed to achieve the goal of a separation kernel.

Quest-V is built upon the Quest real-time operating system (RTOS) [14] which can reside in one of the Quest-V sandboxes, providing real-time execution guarantee, even for the I/O. Quest-V recommends grouping tasks based on their criticality and importance into different sandboxes. As Quest-V sandboxes can communicate between themselves via shared memory, tasks in different sandboxes can have some relations

and dependability. However, the communication between the sandboxes need to be explicitly provisioned. Indeed, vLibOS [56], a virtualization paradigm based on Quest-V, presents an idea of *dual-mode* applications transcending multiple sandboxes. In this paradigm, well-supported, legacy library services can be provided by a well-known, established OS like Linux, and real-time guarantees can be serviced by Quest. Quest-V can thus bring timing guarantee to Linux-based sandbox through its accurate timing-budget management by Quest. This could be very useful in embedded systems which have more I/O-related activities, where Linux may not perform as expected. Quest-V (with support from Quest) provides predictable I/O communication, unlike RT-Xen [55] whose interrupt handling is not accurately time-budgeted according to task priorities. Thus, Quest-V can be utilized in multiple ways for mixed-criticality computation, tasks of different criticalities mapped to different sandboxes or with some other mapping of tasksets.

Jailhouse [46] also provides similar facilities as Quest-V of statically partitioning hardware resources. Jailhouse uses Linux to boot up and initialize all the hardware drivers. Then the Jailhouse hypervisor is loaded as a firmware image by the Jailhouse kernel module. After that, any number of sandboxes can be created up to the limit of the number of physical cores. However, one of those sanboxes need to be Linux which act as a *root cell*. The other sandboxes are *non-root cell*. The *root cell* acts like a *Dom0* in Xen, except interrupts are not virtualized through root cell. Rather devices are directly attached to the particular sandboxes, with the help of the feature like *interrupt remapping* in x86. A complete OS or a bare-metal binary can be run on non-root cells. The Jailhouse hypervisor does not need to be intervened with much of the functioning of the system, except some privileged instruction (e.g. - cpuid) and illegal memory accesses. Consequently, the hypervisor is only involved in management of the cells and the few occurrences of trap, which makes the surface of the hypervisor small. However, as Linux in root cell still holds many controlling capabilities of the system, a compromised Linux can potentially harm the whole system. On the contrary, Quest-V copies the VMM in every sandbox, and the failure of one sandbox can be contained within that particular affected cell. There is no such *root* cell in Quest-V, although the Quest RTOS is the primary OS in Quest-V. The root cell increases usability of Jailhouse, at the expense of some security and safety compromises.

4 CONCLUSION AND FUTURE WORK

In this report, we have presented the scheduling policies and system software designs for mixed-criticality model of computation. We have introduced the widely-used mixed-criticality task models, sprung out from the Vestal model [53].

We have explained some static priority assignment scheduling algorithms for uniprocessors. We have identified that most of the earlier scheduling policies focused on meeting the timing constraints of the higher criticality tasks. Recently, there has been efforts with newer scheduling policies to maintain a *tolerable* service for the lower criticality tasks while meeting all the deadlines of the high criticality tasks. We have also worked on such a strategy, named PAStime [47], where CPU time of a high criticality process is dynamically adjusted at runtime based on its execution progress. The research community needs to investigate whether there are possibilities of integrating these new policies with the previous ones such as AMC [10].

Furthermore, we have described different system software architectures for mixed-criticality systems on top of modern multicore processors. This is an ongoing area of active research with no clear winner [17, 26]. Primary driver of the system design is to provide temporal guarantee to the more critical tasks and spatial isolation of applications. Isolated domains of executions are created in different ways with multiple objectives including separating critical tasks, fault-containment. Although different virtualization models are being explored in this research space, key factors seem to be the reduction of Trusted Computing Base, minimizing overhead and predictable computation for one or more sandboxed domain(s). In that case, microvisor and partitioning hypervisor seem to be ahead of others. Hardware virtualizations techniques such Intel VT are also helping this endeavor. In future, more research focus needs to be given to the communication models, programming paradigm for applications to be developed in this new virtualization infrastructures, where applications do not necessarily run on a single domain.

REFERENCES

[1] James H Anderson, Sanjoy K Baruah, and Bjorn B Brandenburg. 2009. Multicore Operating-System Support for Mixed Criticality. In *Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*. 11.

[2] N.C. Audsley. 5/2001. On Priority Assignment in Fixed Priority Scheduling. *Inform. Process. Lett.* 79, 1 (May 5/2001), 39–44.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, New York, NY, USA, 164–177.

[4] Sanjoy Baruah. 2016. Schedulability Analysis of Mixed-Criticality Systems with Multiple Frequency Specifications. In *Proceedings of the 13th International Conference on Embedded Software (EMSOFT '16)*. ACM, New York, NY, USA, 24:1–24:10.

[5] S. Baruah. June 2012. Certification-Cognizant Scheduling of Tasks with Pessimistic Frequency Specification. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*. 31–38.

[6] S. Baruah, V. Bonifaci, G. DAngelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. July 2012. The Preemptive Uniprocessor Scheduling of Mixed-Criticality Implicit-Deadline Sporadic Task Systems. In *2012 24th Euromicro Conference on Real-Time Systems*. 145–154.

[7] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'angelo, Haohan Li, Alberto Marchetti-Spaccamela, Suzanne Van Der Ster, and Leen Stougie. May 2015. Preemptive Uniprocessor Scheduling of Mixed-Criticality Sporadic Task Systems. *J. ACM* 62, 2 (May May 2015), 14:1–14:33.

[8] Sanjoy Baruah and Alan Burns. 2013. Fixed-Priority Scheduling of Dual-Criticality Systems. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems (RTNS '13)*. ACM, New York, NY, USA, 173–181.

[9] S. Baruah and B. Chattopadhyay. August 2013. Response-Time Analysis of Mixed Criticality Systems with Pessimistic Frequency Specification. In *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*. 237–246.

[10] S. K. Baruah, A. Burns, and R. I. Davis. November 2011. Response-Time Analysis for Mixed Criticality Systems. In *2011 IEEE 32nd Real-Time Systems Symposium*. 34–43.

[11] Sanjoy K. Baruah, Liliana Cucu-Grosjean, Roabert I. Davis, and Claire Maiza. 2015. Mixed Criticality on Multicore/Manycore Platforms (Dagstuhl Seminar 15121). (2015).

[12] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. (2005), 6.

[13] Alan Burns and Sanjoy Baruah. 2013. Towards a More Practical Model for Mixed Criticality Systems. In *Workshop on Mixed-Criticality Systems (Colocated with RTSS)*.

[14] M. Danish, Y. Li, and R. West. April 2011. Virtual-CPU Scheduling in the Quest Operating System. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '11)*. 169–179.

[15] Dionisio de Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. 12/2009. On the Scheduling of Mixed-Criticality Real-Time Task Sets. In *2009 30th IEEE Real-Time Systems Symposium*. IEEE, Washington DC, USA, 291–300.

[16] François Dorin, Pascal Richard, Michaël Richard, and Joël Goossens. 2010-12-01. Schedulability and Sensitivity Analysis of Multiple Criticality Tasks with Fixed-Priorities. *Real-Time Systems* 46, 3 (Dec. 2010-12-01), 305–331.

[17] Zonghua Gu and Qingling Zhao. 2012. A State-of-the-Art Survey on Real-Time Issues in Embedded Systems Virtualization. *Journal of Software Engineering and Applications* 05, 04 (2012), 277–290.

[18] Steven Hand, Andrew Warfield, Keir Fraser, Evangelos Kotsovinos, and Dan Magenheimer. 2005. Are Virtual Machine Monitors Microkernels Done Right?. In *HotOS*. 5.

[19] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. 1997. The Performance of μ-Kernel-Based Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*. ACM, New York, NY, USA, 66–77.

[20] Gernot Heiser and Ben Leslie. 2010. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proceedings of the First ACM Asia-Pacific Workshop on Workshop on Systems (APSys '10)*. ACM, New York, NY, USA, 19–24.

[21] Gernot Heiser, Volkmar Uhlig, and Joshua LeVasseur. January 2006. Are Virtual-Machine Monitors Microkernels Done Right? *SIGOPS Oper. Syst. Rev.* 40, 1 (Jan. January 2006), 95–99.

[22] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson. April 2012. RTOS Support for Multicore Mixed-Criticality Systems. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*. 197–208.

[23] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. 2004. Reducing TCB Size by Using Untrusted Components: Small Kernels Versus Virtual-Machine Monitors. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop (EW 11)*. ACM,

New York, NY, USA.

[24] H. Huang, C. Gill, and C. Lu. April 2012. Implementation and Evaluation of Mixed-Criticality Scheduling Approaches for Periodic Tasks. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*. 23–32.

[25] Huang-Ming Huang, Christopher Gill, and Chenyang Lu. April 2014. Implementation and Evaluation of Mixed-Criticality Scheduling Approaches for Sporadic Tasks. *ACM Trans. Embed. Comput. Syst.* 13, 4s (April April 2014), 126:1–126:25.

[26] Asif Iqbal, Nayeema Sadeque, and Rafika Ida Mutia. 2009. *An Overview of Microkernel, Hypervisor and Microvisor Virtualization Approaches for Embedded Systems.*

[27] Mathieu Jan, Lilia Zaourar, and Maurice Pitel. 2013. Maximizing the Execution Rate of Low-Criticality Tasks in Mixed Criticality Systems. *WMC, RTSS* (2013), 6.

[28] M. Joseph and P. Pandya. 1986/01/01. Finding Response Times in a Real-Time System. *Comput. J.* 29, 5 (Jan. 1986/01/01), 390–395.

[29] Robert Kaiser and Stephan Wagner. 2007. Evolution of the PikeOS Microkernel. In *First International Workshop on Microkernels for Embedded Systems*. 50.

[30] Avi Kivity, Yaniv Kamay, and Dor Laor. 2007. Kvm: The Linux Virtual Machine Monitor. In *Linux Symposium*. 8.

[31] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 207–220.

[32] Jaewoo Lee, Hoon Sung Chwa, Linh T. X. Phan, Insik Shin, and Insup Lee. September 2017. MC-ADAPT: Adaptive Task Dropping in Mixed-Criticality Scheduling. *ACM Trans. Embed. Comput. Syst.* 16, 5s (Sept. September 2017), 163:1–163:21.

[33] Ye Li, Richard West, and Eric Missimer. 2014. A Virtualized Separation Kernel for Mixed Criticality Systems. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '14)*. ACM, New York, NY, USA, 201–212.

[34] J. Liedtke. 1995. On Micro-Kernel Construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM, New York, NY, USA, 237–250.

[35] Jochen Liedtke. 1996-9-1. Toward Real Microkernels. *Commun. ACM* 39, 9 (Sept. 1996-9-1), 70–77.

[36] D. Liu, N. Guan, J. Spasic, G. Chen, S. Liu, T. Stefanov, and W. Yi. July 2018. Scheduling Analysis of Imprecise Mixed-Criticality Real-Time Tasks. *IEEE Trans. Comput.* 67, 7 (July July 2018), 975–991.

[37] M Masmano, I Ripoll, and A Crespo. 2005. An Overview of the XtratuM Nanokernel. *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)* 18, 1 (2005), ix.

[38] M Masmano, I Ripoll, A Crespo, and J J Metge. 2009. XtratuM: A Hypervisor for Safety Critical Embedded Systems. *Real-time Linux Workshop* (2009), 9.

[39] E. Missimer, K. Missimer, and R. West. July 2016. Mixed-Criticality Scheduling with I/O. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*. 120–130.

[40] Antonio Paolillo, Olivier Desenfans, Vladimir Svoboda, Joël Goossens, and Ben Rodriguez. 2015. A New Configurable and Parallel Embedded Real-Time Micro-Kernel for Multi-Core Platforms. *OSPERT 2015* (2015), 25.

[41] Antonio Paolillo, Paul Rodriguez, Vladimir Svoboda, Olivier Desenfans, Joël Goossens, Ben Rodriguez, Sylvain Girbal, Madeleine Faugere, and Philippe Bonnot. 2017. Porting a Safety-Critical Industrial Application on a Mixed-Criticality Enabled Real-Time Operating System. In *Proc. 5th Workshop on Mixed Criticality Systems (WMC), RTSS*. 1–6.

[42] I Ripoll, M Masmano, V Brocal, S Peiro, P Balbastre, and A Crespo. 2010. Configuration and Scheduling Tools for TSP Systems Based on XtratuM. *Real-time Linux Workshop* (2010), 7.

[43] RTCA/DO-178C. 2012. Software Considerations in Airborne Systems and Equipment Certification. (2012).

[44] J. M. Rushby. 1982. Proof of Separability A Verification Technique for a Class of Security Kernels. In *International Symposium on Programming*, G. Goos, J. Hartmanis, W. Brauer, P. Brinch Hansen, D. Gries, C. Moler, G. Seegmüller, J. Stoer, N. Wirth, Mariangiola Dezani-Ciancaglini, and Ugo Montanari (Eds.). Vol. 137. Springer Berlin Heidelberg, Berlin, Heidelberg, 352–367.

[45] Lui Sha, John P. Lehoczky, and Ragunathan Rajkumar. 1986. Solutions for Some Practical Problems in Prioritized Preemptive Scheduling. In *RTSS*.

[46] Siemens. 2018-11-20T11:46:27Z. Jailhouse: Linux-Based Partitioning Hypervisor. https://github.com/siemens/jailhouse. (Nov. 2018-11-20T11:46:27Z).

[47] Soham Sinha, Richard West, and Ramesh Peri. PAStime: Progress-Aware Scheduling for Time-Critical Computing.

[48] Udo Steinberg and Bernhard Kauer. 2010. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. ACM, New York, NY, USA, 209–222.

[49] H. Su, P. Deng, D. Zhu, and Q. Zhu. August 2016. Fixed-Priority Dual-Rate Mixed-Criticality Systems: Schedulability Analysis and Performance Optimization. In *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 59–68.

[50] H. Su, N. Guan, and D. Zhu. August 2014. Service Guarantee Exploration for Mixed-Criticality Systems. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*. 1–10.

[51] H. Su and D. Zhu. March 2013. An Elastic Mixed-Criticality Task Model and Its Scheduling Algorithm. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. 147–152.

[52] Manohar Vanga, Andrea Bastoni, Henrik Theiling, and Björn B. Brandenburg. 2017. Supporting Low-Latency, Low-Criticality Tasks in a Certified Mixed-Criticality OS. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems (RTNS '17)*. ACM, New York, NY, USA, 227–236.

[53] S. Vestal. December 2007. Preemptive Scheduling of Multi-Criticality Systems with Varying Degrees of Execution Time Assurance. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. 239–243.

[54] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. May 2008. The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.* 7, 3 (May May 2008), 36:1–36:53.

[55] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. 2011. RT-Xen: Towards Real-Time Hypervisor Scheduling in Xen. In *Proceedings of the Ninth ACM International Conference on Embedded Software (EM-SOFT '11)*. ACM, New York, NY, USA, 39–48.

[56] Ying Ye, Zhuoqun Cheng, Soham Sinha, and Richard West. 2018-01-24. vLibOS: Babysitting OS Evolution with a Virtualized Library OS. *arXiv:1801.07880 [cs]* (Jan. 2018-01-24). arXiv:cs/1801.07880