Resource Management for Mixed-Criticality Systems on Multi-Core Platforms with Focus on Communication

Robin Arbaud, Dávid Juhász, Axel Jantsch
TU Wien, Vienna, Austria
Email: e1633097@student.tuwien.ac.at, david.juhasz@tuwien.ac.at, axel.jantsch@tuwien.ac.at

***Abstract*—The System-on-Chip revolution has not impacted the field of safety-critical systems as widely as the rest of the electronics market. However, it is nowadays acknowledged that sharing resources between applications of different criticality is a key leverage to reduce costs and improve performance. Certification of systems hosting such mixed-criticality application sets requires *sufficient isolation* between criticality levels, i.e. a low-critical task must not cause a fault in a high-critical task, especially a temporal fault, which is likely when several co-running tasks compete for a shared resource. This digest describes implementation schemes which can be included, at both hardware and software levels, in mixed-critical systems to enforce such isolation.**

***Keywords*-Mixed-Criticality; Isolation; Temporal Partitioning; Interference; Communication Protocols; Survey;**

I. INTRODUCTION

A. Why Mixed-Criticality?

Mixed-Criticality results from the application of the global trend in electronics, consisting in integrating more and more components onto a single chip, in the domain of safety-critical applications. Until recently, isolation of tasks required by certification authorities implied that safety-critical tasks were executed on dedicated hardware. Therefore, real-time hardware was often under-utilized due to pessimistic Worst-case Execution Time (WCET) analysis. Since the average execution time is usually much shorter than the WCET, resources had to be reserved much longer than what was actually needed in most cases for the execution of safety-critical tasks.

It is nowadays acknowledged that safety-critical applications should be able to run on the same platform as non-critical applications, complete isolation being deemed too costly with the spread of systems-on-chip. This is regarded as a way to improve resource utilization, by running low or non-critical tasks on slack time made available when a high-critical task terminate before its WCET, which is very common in practice. However, safety-critical and best-effort applications have somewhat conflicting requirements, hence the numerous challenges arising.

This research was partially funded by the European Union's Horizon 2020 Framework Programme for Research and Innovation under grant agreement no 674875 (oCPS Marie Curie Network).

Mixed-criticality is an active research field, as reflected in the review by Burns and Davis [1]. Research can be pigeonholed in two categories: implementation schemes to enable safe and efficient sharing of resources for a part, and scheduling policies maximizing processor utilization while maintaining schedulability at all Criticality Levels (CL) for a second part.

The rest of this digest focuses on the former and is organized as follows. The next two paragraphs provide some explanation about key concepts of Mixed-Criticality Systems (MCS). Section II describes several temporal partitioning mechanisms, making use of mixed-criticality to go beyond static partitioning, thus allowing for efficient resource usage while guaranteeing service latency bounds for critical tasks. Sections III and IV describe communication media arbitration policies aimed at guaranteeing communication latencies for critical traffic, while sacrificing neither resource utilization nor Quality of Service (QoS) for best-effort traffic. Section V gives a couple of examples of how those elements can be integrated into system-wide architectures well-suited for mixed-criticality. Section VI briefly concludes this paper on open problems and topics of mixed-criticality which are not covered in this paper. All mentioned mechanisms are gathered in figure 1.

B. Fundamental concepts

1) Isolation: Certification requires that neither temporal nor logical correctness of a task is jeopardized by another task. While tasks of the same CL can be certified together as a single-criticality subsystem, it is costly and useless to certify the whole system at the higher level of insurance. In fact, as non-critical applications are often developed independently from safety-critical ones, it is well possible that the safety-critical applications must be certified without any knowledge about the behavior of non-critical ones. Therefore, task sets of different criticality must be isolated from each other. Isolation can be achieved in two ways, either spatial or temporal partitioning. In spatial partitioning, applications run on separate hardware. In temporal partitioning, applications run on the same hardware in separate time windows. Full spatial partitioning is deemed too costly to be a viable solution at the system scale, however, it is still

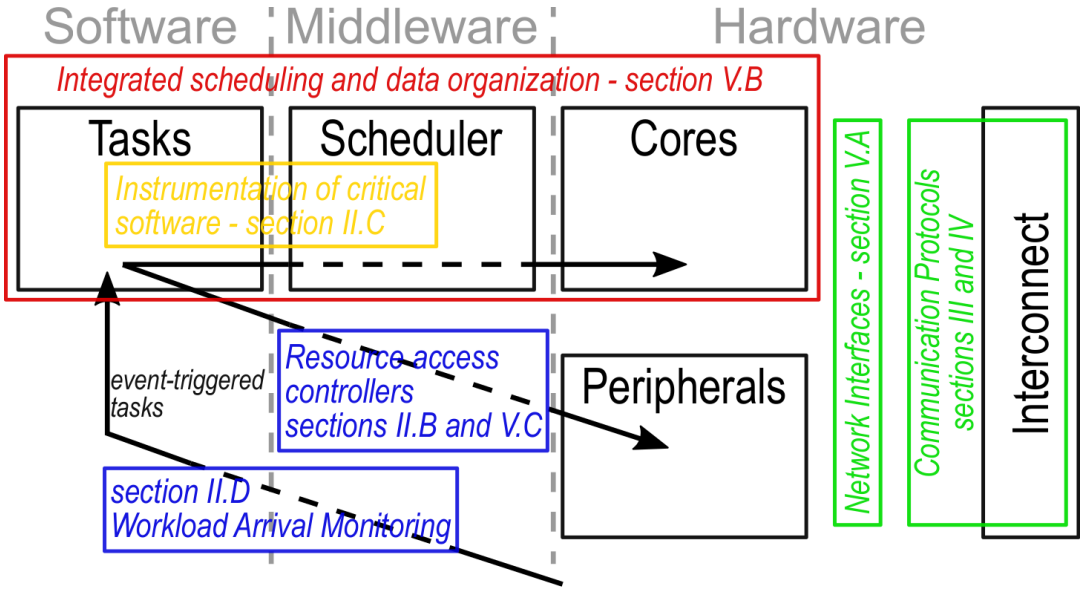

Figure 1. Summary of mechanisms and architectural blocks presented in this paper.

relevant for some elements, like memories, or Networks-on-Chip (NoC) input buffers (see section IV-A). Temporal partitioning enables the sharing of resources, but that is not yet sufficient to achieve good hardware utilization. As already mentioned, WCET are pessimistic bounds which almost never occur in practice, especially for safety-critical applications. Therefore, if a resource is statically reserved for a given application during its entire WCET, this resource will actually not be utilized most of the time. Some dynamic scheme is required to make the partitioning more flexible. Examples of such a scheme are described in section II.

2) Interference: A major challenge in multicore MCS is dealing with interference. Interference occur when an application attempts to access an already in use resource. In that case, either the application must wait for the resource to be available, or the former activity of the resource will be suspended, so that the new application can preempt the resource. Whether preemption is allowed or not, and under which circumstances, depends on the arbitration policy of the resource controller. However, in both cases an application is delayed. Interference is not necessarily a problem, but it must be predictably bounded so that WCET taking it into acount can be derived. Next to that, reducing interference is a key leverage to improve performance of multicore platforms. A list of potential interference sources was established by Kotaba et al. in [3], alongside with some possible solutions to limit them.

The first category of interference sources is interconnects. Predictable arbitration policies are required to derive upper latency bounds taking interference into account. Advanced arbiters make use of criticality and priority to obtain guaranteed latencies while sacrificing neither average performance nor resource utilization (see sections III and IV). However, advanced arbiters need to be implemented in hardware and are often not available on Commercial Off-The-Shelf (COTS) components. An alternate way is to use rate limiters to prevent a master to flood the bus or NoC, like in the IDAMC (see section V-A). Note that those two approaches are complementary and do not exclude each other. The other major sources of interference are memories and especially shared caches. Cutting interference resulting from memory devices is an active research topic, but is outside the scope of this digest focused on communication. Next to those, there are several minor sources of interference, including shared co-processors, interrupt controllers, or I/O devices. Such interference generally has less significant effects, as those resources are relatively often idle in most systems compared to interconnects and memories, and therefore is not as widely studied.

II. TEMPORAL PARTITIONING MECHANISMS

A. Introduction : Full-Scale Mode Switches Limitations

The most basic mixed-criticality specific partitioning mechanism is the so-called criticality mode switch, as described in [2]. Each application in the system is assigned a Criticality Level (CL), and the system can run in different criticality modes. There are as many criticality modes as CL. When the system works normally, it stays in the lowest criticality mode and tasks of all criticality are scheduled. When the system encounters a rare event (e.g. overheat) which degrades its performance, it switches to a higher criticality mode. In criticality mode l, tasks of criticality higher than l are still executed on schedule, but tasks of criticality lower than l are dropped. Depending on the model,

tasks of criticality l are either executed on schedule or on slack time, when no task of higher criticality is running.

Triggering the mode switch is done through monitoring execution times and assigning time requirements which depend on the criticality mode. Tasks WCET are considered to depend on the criticality mode. Tasks deadlines, periods (for periodic tasks) or minimum inter-arrival times (for sporadic tasks), can also be considered to depend on the criticality mode, but this is not the case in all models. In a higher criticality mode, the WCET are longer and the periods are shorter. The deadlines can be either shorter of longer at higher criticality mode, depending on the model. Assuming the system is currently in criticality mode l, if a task of criticality higher than l takes longer than its l-criticality WCET, the system shall switch to a higher-criticality mode, thus dropping low-critical tasks to ensure that high-critical tasks will meet their deadlines.

With such a mode switch, one could imagine certifying l-criticality functions only in the restricted case of a system in criticality mode l. This tackles the need to certify the whole system at the highest level of insurance. However, it does not come without several difficulties. First, different WCET values for safety-critical functions may not necessarily be acceptable, especially considering that the gap between WCET at different CL has to be sufficient to prevent the task, which triggered the mode switch, from missing its deadline. This is not clear when one takes the overhead, induced by a mode switch, into account. Second, when the system features 4 or 5 CL (like in AUTOSAR and ARINC standards, respectively for automotive and avionics fields), dropping tasks of CL 3 to prevent a CL 4 deadline miss is not an acceptable solution. As a matter of fact, low-criticality tasks are still critical, and must guarantee some service in any situation. A possible answer is proposed by Giannopoulou et al. [23] (see section V-B), in the form of a degraded, minimal insurance level that low-criticality tasks must maintain in all criticality modes. Third, the return of the system to a normal execution mode after a criticality mode switch is seldom addressed [1].

Though the mode switch model is widely assumed, especially in the field of mixed-criticality scheduling, it is not suitable for actual implementation of certified systems, or at least not sufficient by itself. Therefore, more advanced mechanisms were proposed, based on stalling or delaying low-criticality tasks rather than dropping them completely. Those mechanisms tolerate interference up to the point where it might cause a high-criticality deadline miss.

B. Memory Access Budgeting for Non-Critical Tasks

This approach was originally proposed by Yun et al. [4], later extended by Flodin et al. [5], and reused by Lampka and Lackorzynski [6]. The principle is to assign a memory access budget to non-critical tasks, to monitor the number of cache misses of those tasks, and to suspend their execution when they exceed their memory access budget. This way, interference resulting from both the memory bus and the memory controller can be bounded. However, response latency is only considered as a whole, without distinction between communication latency and memory read latency.

1) Original approach: The case studied in [4] is restricted to a single critical core for critical tasks, and an arbitrary number of non-critical interfering cores. For each critical task, its WCET in isolation and its worst-case number of cache misses are assumed to be known, as well as the worst-case latency between the issue of a request and its completion. The memory access budget granted to non-critical tasks is periodically replenished to a constant value. The authors propose an algorithm to compute the highest budget which can be granted while ensuring that all critical tasks will meet their deadlines, given the budget replenishment period.

This approach was implemented by the authors on a dual core platform with a Linux kernel. They found their method to yield rather conservative utilization. As a matter of fact, they consider that each memory request from the critical core will take its worst-case time to be responded to, i.e. that every critical memory request will be delayed due to interference from a non-critical one. This pessimistic assumption entails a budget lower than what would be needed for efficient resource utilization. They conclude that their method would probably not scale well w.r.t. an increased number of cores, because the pessimism involved would make the utilization of additional resources low.

2) Extended approach: To improve on the previous mechanism, the authors of [5] abandoned the fixed periodicity of the budget allocator to synchronize budget shifts with arrivals and terminations of critical tasks, what they call hard real-time co-runner specific budgeting. This way, the available budget for non-critical tasks depends on which critical tasks are currently running. Furthermore, if no critical task is running, this method allows slack transfer from critical tasks to non-critical ones. They also extended their method to support any number of critical cores instead of a single one. They assume a given budget for each critical task. The mechanism is based on budget queues, one queue for each critical core. The budget at the head of a queue corresponds to the currently running critical task. If no critical task is running, the budget is shifted to an unlimited one. The control mechanism stalls non-critical cores when one of the queue heads exposes a depleted budget.

The authors have implemented their mechanism on a 6-core platform, using a microkernel and interrupts to transmit the budget updates from critical cores to non-critical ones. They show a significant improvement over the static period mechanism, and claim that the induced overheads are negligible if the hardware platform features suitable interrupts.

A way to compute maximum budgets for this scheme was proposed by Lampka and Lackorzynski in [6]. Assuming a periodic time-triggered scheduling for the critical task set,

they evaluated the worst-case number of memory requests from critical tasks per scheduling time slot. Given the worst-case memory access latency (including communication delays), the minimal number of requests which can always be served during a time slot is easy to calculate. The budget for non-critical cores is then obtained as the minimal number of serviceable requests minus the maximal number of requests from the critical cores. These budgets are safe but pessimistic, especially because the response time of the memory access is always assumed to be the worst-case one, which was already a shortcoming in [4]. Note that the assumption of a static time-triggered scheduling somewhat shifts the scheme back towards the original one, in which budget replenishment were periodic, but the possibility of slack transfer when a critical task terminates early remains an important improvement.

3) Conclusion - Pros and Cons: The main advantages of this mechanism are the low overheads and its implementability on COTS platforms, provided they feature performance counters (for cache miss monitoring) and inter-processor interrupts. A major limitation is its inapplicability to systems with more than 2 CL. Besides, an algorithm for deriving tight budgets compatible with flexible scheduling policies is not yet available, and deriving case-by-case budgets for specific applications remains a difficult problem. The distinction between critical cores and non-critical cores could be dropped if needed, at the cost of higher overheads and simplicity loss.

C. Execution Time Monitoring of Critical Tasks

This method was developed by Kritikakou et al. in [7], [8], [9]. The idea is to monitor the execution times of critical tasks at given observation points, by inserting a routine in the code. When interference from non-critical tasks might cause a high-criticality deadline miss, non-critical tasks are suspended until the critical task termination. The eventuality of a deadline miss is determined by the Remaining WCET (RWCET) at the observation point. The authors distinguish the RWCET in isolation, computed assuming that only critical tasks may run, from the total RWCET, which is the sum of the former plus the worst-case delay due to interference from non-critical tasks.

The authors defined an Extended Control Flow Graph (ECFG) grammar to model and analyze critical tasks, and a low overhead algorithm to compute the RWCET at run-time based on exploiting pre-computed data from static analysis. [7] defines the ECFG grammar and presents a formal proof that a single critical task will always meet its deadline, for any co-running task set, provided that its WCET in isolation is lower than its deadline. [8] extends the mechanism to an arbitrary number of critical tasks and describes a software implementation on a COTS component. [9] improves on the previous scheme, using dynamic setting of the observation points to reduce the run-time monitoring overhead.

1) Safety condition: At a given observation point, the condition under which it is safe to continue normal execution (i.e. tasks of all CL enabled) is called the safety condition. It is valid if the sum of the current execution time, plus the RWCET in isolation, plus the worst-case additional interference delay until next observation point, plus the monitoring and mode switch overheads, is lower than the task deadline.

This condition guarantees that in case of interference peak, there will be enough time remaining by next observation point to switch mode and suspend non-critical tasks, so that the monitored critical task is guaranteed to meet its deadline. Interference from non-critical tasks is only taken into account for the WCET between the current observation points and the next one, since it will be possible to switch mode at the next observation point if needed, to prevent interference afterward.

2) Dynamically computing next observation point: A question to answer when implementing this scheme is the granularity of the observation points. On the one hand, if there are too few of them, the WCET between two points will be consequently larger than the average execution time, which makes the safety condition pessimistic and therefore yields a high number of criticality mode switches. On the other hand, many observation points cause large overheads. In order to mitigate this trade-off, the authors proposed to compute at run-time the furthest observation point by which the safety condition will hold, and to disable intermediate observation points. This way, it is possible to increase the granularity of observation points while limiting the subsequent overheads.

3) Run-time Control Mechanism: At each observation point of each critical task, the safety condition is checked. If the system can run normally until next observation point without any chance that the critical task misses its deadline, nothing happens. Otherwise, a request is send to a centralized controller to suspend non-critical tasks. This controller works in two slightly different ways, depending on whether the observation points are statically or dynamically defined.

With static observation points, the critical task which sent the suspension request will also send a notification to the controller when it terminates. This controller increments a counter by each request it receives, and decrements it when it receives a task termination message. Non-critical tasks execution is enabled only when the counter is zero.

With dynamic observation points, the controller stalls non-critical tasks when requested to do so, and re-enables them only when no critical task is running (idle state). This is because with dynamic observation points, once the non-critical tasks are disabled, all observation points are also disabled. Therefore, only one suspension request can be sent to the controller at a time, and there is no way to know whether it is safe to re-enable non-critical tasks when the critical task which originally issued the suspension request

has terminated, because other critical tasks might need to execute in isolation as well.

4) Conclusion - Pros and Cons: Except the controller which gathers suspension requests, which is very lightweight, this method consists only in instrumentation of the high-critical code. This makes it easy to implement on COTS platforms, but also entails rather large overheads. Besides, it only supports dual-criticality systems. Its main advantage is that monitoring the actual execution time induces less conservative utilization than monitoring resource access requests like in the previous scheme. The worst-case interference is assumed only between two observation points at a time, and if a critical task has already started to execute smoothly, the actual execution time of its already executed part, which is likely much shorter than the WCET, will be used to determine the need for mode switching.

D. Workload Arrival Monitoring

This method is specifically tailored for event-triggered systems. It was originally developed for hard real-time systems in numerous papers including [10], then extended to MCS by Neukirchner et al. [11], [12]. The idea is to monitor the activation of tasks, in order to detect when an incoming task might cause a deadline miss of an already running task or might not meet its own deadline (those two cases are not necessarily discriminable). If the incoming task is of low criticality, it can be dropped or delayed, and if it is of high-criticality, already running low-criticality tasks can be temporarily stalled. The reaction to take is not developed by the authors. This method is primarily intended to monitor the activity of CPU, but can in principle be used with diverse kind of resources.

1) System and Event Models: This model was described by Wandeler et al. [10]. Tasks are activated by events; each of those events has a given type, which can be mapped to the WCET of the requested task. The metric which serves as reference is the Workload Arrival Function (WAF). It denotes the requested workload, in terms of WCET, occurring per time interval. On each task activation, the monitor computes the workload based on the newly incoming event and the events associated to tasks which may still be running. An exception can then be triggered if the latest incoming request might cause the WAF to exceed the serviceable workload. The authors of [11] show how to make the monitoring overhead constant w.r.t. the number of events in terms of computation time and of memory space, using a safe approximation of the exact WAF.

2) Group Monitoring Scheme: In the hard real-time version of the method, all task activations are individually monitored. This enforces interference bounds between all tasks without consideration of criticality levels, whereas it is only necessary to do so between tasks of different CL. Besides, monitoring the activations of tasks independently from each other prevents from using any information about correlation between tasks, therefore the assumed scenario is always the worst-case interference. The method described in [11] uses monitoring of task group activations. By grouping together tasks of the same criticality, the redundant isolation between those is dropped. By grouping together tasks correlated with each other, it is possible to take correlation into account to make the estimation of the worst-case requested workload more accurate and less pessimistic. A group of tasks is regarded as one virtual task, which fires upon firing of any task the group contains. What differs between activations resulting from one task or another is the ensuing event type. There are as many event types as different WCET of tasks within the group.

The authors evaluated the benefits of their method against individual tasks monitoring. They showed that group monitoring yields a significantly lower rate of triggered exceptions, and this effect increase with the number of tasks per group. This is caused by the possibility given to some tasks to exceed their WAF budget as long as other tasks in the same group can compensate for it by not using their whole budget. The second effect is the reduction of the worst-case interference when tasks of a group are correlated. Correlation indicates some kind of activation pattern, which can be used to lower the WAF estimate (compared to individual task analysis) without compromising the ability of the system to execute those tasks.

An example of computing WAF upper bounds from an event stream model is proposed in [10]. The same method could be applied to both high- and low-critical tasks, but in order to reduce the analysis effort on the latter, the authors propose to use as maximum allowed WAF value the highest WAF observed in test or in simulation. This way, the highest measured low-critical WAF becomes the actual, accurate, worst-case WAF, as any overrun of this WAF bound would cause tasks to be dropped or delayed.

3) Combining Several Monitors: So far, only one monitor was considered. This is extended in [12] to several monitors for different groups of tasks. Rather than assigning independent WAF to each monitor, this method is based on guarantee interface tuples. Those are vectors containing WAF upper bounds for each monitor, so that timeliness can be guaranteed. It is likely that several tuples exists which are safe, as more activations in a task group can be compensated by fewer in another. Interface tuples can be obtained by computing highest WAF bound for a monitor (i.e. interface tuple element) while other elements are kept constant and recursively doing so for all elements in the tuple. Varying the order in which individual WAF bounds are maximized, a pareto set of interface tuples is obtained.

Each monitor has an active interface tuple, which corresponds to the one with lowest WAF for this monitor, and which has not been invalidated. When a monitor notices a WAF overrun comparing to its active interface tuple, this tuple gets invalidated for every monitor in the system, and all

monitors whose active tuple was the invalidated one switch to another tuple. As long as each monitor has at least one active tuple, all tasks will meet their deadlines. When a monitor has no tuple left to switch to, newly incoming tasks through this monitor might cause a deadline miss, but tasks issued through another monitor might still be executed safely if this monitor still has an active tuple. Interface tuples can be safely set as valid anew when the system is idle.

4) Conclusion - Pros and Cons: This method is intermediate in terms of pessimism between the two previous ones. No information about the actual execution time is used contrary to the execution time monitoring scheme, so only worst-case data is used. However, correlation between tasks activations within groups and sets of pareto interface tuples between monitors enables more accurate assumptions about interference profiles than a global budgeting. Besides, even if event-triggered tasks are often supported, they are usually treated as periodic tasks with their period equal to their minimal inter-arrival time, which leads to overallocation of resources when the event doesn't occur that often, and generally causes a mode switch if the event occurs too soon. This tailored method captures the actual properties of event-triggered tasks.

III. MIXED-CRITICALITY BUS ARBITERS

Communication protocols for mixed-criticality must abide by some specific requirements, and therefore some specific metrics should be used to evaluate their performance. As discussed by Napier et al. [13], the lack of well established performance metrics and test cases led to difficult comparison of different protocols. If some metrics are often used (e.g. maximal measured latency for non-critical message delivery under various critical loads), the test cases are not necessarily comparable. Besides, the schemes are often evaluated based on the sole metric they were designed to improve, without consideration of the potential decreases they might induce in other performance metrics. The key point in all protocols is a guaranteed worst-case latency for critical traffic. This being given, Napier et al. identified 3 important properties. The first one is resource utilization. This is linked to the ability to derive sufficiently tight worst-case latencies for critical traffic, i.e. not over-allocating resources. This can also be analyzed in terms of schedulability, i.e. the amount of (critical) task sets which can be scheduled on fixed hardware resource, as this depends directly on tightening latency bounds. The second one is Quality of Service (QoS) for non-critical traffic. This consists in reducing the average latency of non-critical traffic. The third one is scalability w.r.t. number of bus masters or NoC nodes. It is not independent from the two others, as, especially for NoC, the gain achieved by a protocol over another might strongly depend on the NoC size and load.

A summary of all described communication protocols is given in table I. Most of them feature only 2 CL, nevertheless it is not necessarily a big shortcoming. Even if more CL are required at the system level, it can be sufficient that the interconnection media allow for two QoS classes, namely guaranteed latency traffic for high-criticality tasks and best-effort traffic for non and low-criticality tasks.

A. TDMA-based Bus Arbiters

1) Limitations of the TDMA protocol: The simplest fully predictable bus arbitration scheme is the so-called Time-Division Multiple Access (TDMA) protocol. With a TDMA arbiter, each bus master is assigned time slots, and data transfer is allowed during those time slots only. The assigned time slots sequence is statically defined and repeats itself periodically. The two major limitations of TDMA are its lack of flexibility, and that it often yields low bus utilization, since a slot will be unused if the master to which it is assigned has no data to transmit. Optimization of the assignation sequence to minimize the execution time of a given communication task set has been addressed by Rosén et al. [14]. Their minimization algorithm consists of two nested optimization loops. The inner one is aimed at finding the slots order which minimizes the total worst-case latencies of a given communication task set (that is the time needed to transmit a given set of messages, with specified sizes, arrival time and dependencies), given a slot size for each master. The outer loop minimizes the same cost function, but modifies the slots sizes. The authors also considered the special case of equal slot sizes for all masters. Since the optimization assumes a given communication task set, the result is only optimal for the static schedule of computation tasks issuing that particular communication task set. Besides, no notion of criticality is used. However, this can still be useful for assigning slots to high-criticality tasks within more complex protocols built on top of TDMA, like the one described in the next section.

2) Extending the TDMA protocol: This arbiter was proposed by Cilku et al. [15]. The CPU are virtualized through a hypervisor, and virtual CPU are separated into critical and non-critical ones. The arbitration is done between virtual CPU rather than between tasks. This allows to reduce overheads compared to a task-based arbitration, without loss of flexibility. The arbiter is layered. The first layer is TDMA and is used for arbitration between critical cores. The second layer is Round Robin (RR) and is used for non-critical cores. The Round Robin policy grants the bus control to each master in turn. The master releases his control only when it has no data left to transmit, and the control is then passed on to the next master. This scheme yields maximal bus utilization, but is not predictable, especially if the behavior of some non-critical masters is not accurately known, which is common in practice. Besides, nothing prevents a master from monopolizing the bus, causing starvation of others.

In this scheme, the RR arbiter can be seen as a bus master, which is assigned time slots in the TDMA sequence. Critical

virtual cores have their own TDMA slots, while non-critical virtual cores have none and thus can only access the bus through the RR arbiter. The slots assigned to non-critical cores are all consecutive, and followed by an empty slot during which none of the masters is allowed to transmit. This makes sure that even in case of an error in a non-critical task, the bus will be ready to be used by the critical core as soon as it is granted control.

The authors tested their arbiter on a Field Programmable Gate Array (FPGA) board with a synthetic task executed concurrently on 4 cores accessing a shared memory through a bus. They measured the computation time needed to complete the task on each core, for three different bus arbiters. They claim that their dual-layer arbiter yields only a 20% increase in computation time compared to bare RR arbitration for running the task on non-critical cores, and divides the computation time by more than 3 compared to pure TDMA. However, they do not mention the way they assigned TDMA slots to different cores, while this was shown to be of utmost importance by Rosén et al. [14]. Besides, the synthetic task set they used is the same on all cores, critical and non-critical ones, and critical and non-critical hardware are also identical. It would be interesting to evaluate this scheme on a more heterogeneous task set.

B. Criticality and Requirements Aware Bus Arbiter

This arbiter, abridged CArb, was proposed by Hassan and Patel in [16].

1) System Model: The target system is a multicore platform, with one task per core (this condition serves the sole purpose of discarding any side complexities due to the scheduling algorithm, as the focus is set on the bus arbiter). An arbitrary number of criticality levels is supported. Tasks are grouped into criticality classes based on their CL. The system features as many criticality modes as CL. In criticality mode l, tasks of criticality $c \leq l$ are executed on slack time and prioritized by descending criticality, and tasks of criticality $c > l$ are executed on schedule. This is true for both computation and communication resources.

2) Weighted Harmonic Round Robin Arbiter: CArb is built on an improved RR arbiter, called weighted harmonic RR. This differs from the basic RR arbiter in two ways. First, starvation is prevented by granting the bus control for one request only. If a master has several requests to transmit, it must wait until it is granted control again. The window in which the master has control of the bus is called a slot, which is actually very similar to TDMA slot, but defined in the domain of transmission requests rather than time. Second, the slots are not equally distributed between masters. Some masters get more slots than others. The order in which slots are allocated to masters is designed to enforce evenly distributed slots. For example, if a master has two slots per period, the number of slots between its first and second slots of a given period is the same as between its second slot of this period and its first slot of the next period.

3) CArb: CArb is a two-level arbiter, of which both levels are based on weighted harmonic RR. The first level consists in inter-class arbitration, that is arbitration is first performed between criticality classes. Once a criticality class has been elected, a second intra-class arbitration occurs between all the tasks comprised in this criticality class. The scheduling scheme needs 3 sets of parameters: the class weights, i.e. the number of class slots allocated to each criticality class per period; the window sizes, i.e. the number of task slots in a class slot (this can be set independently for each class, but must be the same in all slots of a given class); the task weights, i.e. the number of task slots allocated to each task per period. It is worth noting that the period of the inter-class arbiter is not the same as the one of the intra-class arbiter, and furthermore that different criticality classes can feature different values of their intra-class arbiters periods.

Finding values for those parameters is addressed by the authors in the form of a proposed optimization problem formulation, which can be solved using traditional optimization algorithms. They do not seek to optimize the transmission time of a given message set, but rather the bus utilization, expressed by the length of the arbiter period, i.e. the number of slots in the sequence which is periodically repeated.

4) Making CArb Dynamic: If the system runs on criticality mode l, and a task of criticality $c > l$ exceeds its l-criticality WCET, the basic scheme for MCS would be to switch the full system to criticality mode $l + 1$. The authors propose to perform a mode switch at the arbiter level only rather than a full-scale mode switch. The idea is based on the decomposition of the total WCET as the sum of the WCET in isolation plus the worst-case interference delay. When the WCET in isolation is exceeded by a small amount, the arbiter reassigns the slots of criticality lower than c to higher criticality tasks, and the tasks of lower criticality can access the bus on slack time only. When the WCET increase is too large to be compensated solely by reducing interference from the bus, mechanisms like the ones described in section II can be applied and a full-scale mode switch be performed if necessary. Since the arbiter slot sequences for each criticality mode are statically defined, the reduction of the worst-case response time of the bus that can be achieved is known, so setting the limit at which an arbiter mode switch is not sufficient is easy. The only additional need for this scheme compared to the previously described static CArb is a scheduling table for each criticality mode of the arbiter instead of a global one.

It is further possible to reassign only part of the time slots, and not all of them, based on how large the WCET overrun is. The cost is an additional scheduling table for each intermediate WCET overrun limit. This improves the QoS of low-criticality tasks, but does not yield any improvements w.r.t. avoiding full-scale mode switches.

With such an arbiter mode switch, the overheads are considerably lower than with a full-scale mode switch, and what is more, the low-criticality tasks are not dropped, even if they are no longer guaranteed to meet their deadlines. If execution times return to lower values, the arbiter will also return to its normal scheduling table at the beginning of the next period. The authors tested their arbiter in simulation, based on an avionics use case and synthetic task sets. They showed the effectiveness of their arbiter at enforcing worst-case latencies and at postponing full-scale mode switches.

IV. MIXED-CRITICALITY NoC PROTOCOLS

There are two main categories of NoC protocols : Wormhole and Store-And-Forward (SAF). The former is more widely used, as it requires smaller buffers than SAF. Two mixed-criticality wormhole-based protocols were proposed by Burns, Harbin and Indrusiak [17], [18] on the one hand and Tobuschat and Ernst [19] on the other hand. A protocol combining wormhole for low-criticality messages and SAF for high-criticality ones was proposed by Dridi et al in [20]. An orthogonal approach based on adaptive routing was developed by Kostrzewa et al. [21].

A. Wormhole Protocol for Mixed-Criticality (WPMC)

WPMC was originally proposed by Burns et al. [17], and then improved into WPMC-FLOOD by Indrusiak et al. [18].

1) Generic Wormhole NoC Protocols: In a wormhole NoC, messages are divided into fixed-size flits. They include a header flit, which contains the number of flits in the message and its destination. Each router has a number of input buffers to store received flits, and those buffers are associated to Virtual Channels (VC). All flits of a message shall use the same VC, and the VC index shall be encoded in each flit. The routing policy is deterministic, so that all flits of a message follow the same path. The router knows to which message an incoming flit belongs based on the VC which transmitted the message and the size of the message which was encoded in the header flit. A priority preemptive arbitration is often used for the output stage, based on the VC indexes, e.g. a VC with smaller index has higher priority. If preemption is available at message level only, it is no longer needed to encode the VC index in each flit, but a flit-level preemption is essential for MCS, if non-critical message are not ensured to be sufficiently short. Within a VC, simple arbitration such as FIFO (as in WPMC) or RR (as in the protocol defined in [19]) can be used between messages.

A flow control is furthermore needed, to ensure that no data can be transmitted through a VC when the corresponding buffer at the input of the downstream router is full. For WPMC, the authors consider a credit-based mechanism [17]. For the protocol proposed in [19], the authors assume no back pressure, i.e. buffers of sufficient length to handle any potential load.

2) Introducing Criticality Levels - WPMC: Criticality is introduced in two ways. First, a criticality mode is implemented at the link level. That is, each link can be independently set to a criticality mode. When a link is in high-criticality mode, it can only transmit high-criticality messages. At startup, all links are in low-criticality mode. When a router receives a flit from a link in high-criticality mode, it switches all its output links to high-criticality mode. This way, high-criticality mode propagates through the network alongside with high-criticality flits. Besides, the network interfaces are responsible to check whether the incoming messages are within their timing specifications or not. If some message is not, the network interface can drop the message or trigger a mode switch at all the output links of the local router. Mode switches are always initiated by a network interface.

This takes us to the second way criticality is introduced. Messages (and VCs) are divided into high-criticality and low-criticality ones. Three parameters are considered criticality-dependent, and thus have different values depending on the criticality mode of the NoC: the period or minimum inter-arrival time, the maximum network latency without interference, and the maximum additional delay due to interference. The period is shorter at higher CL, the latencies are larger at higher CL. When a network interface notices a high-criticality message that is not within its low-criticality specifications, it triggers the mode switch, which will then be propagated by the message. When a low-criticality message is detected outside its low-criticality specifications, it is dropped, but no criticality mode switch occurs.

The authors considered only 2 CL, but there is no limitation to the number of supported CL in theory. In practice, transmitting criticality mode requires dedicated control wires, so supporting 4 CL would be possible with one additional line (2 link state lines), and so on. Supporting more CL would therefore induce area overheads.

3) WPMC-FLOOD Improvements: The authors proposed two improvements of their protocol in [18]. The router for this protocol is presented in figure 2. First, to avoid dropping low-criticality messages in case of criticality mode change, they allow low-criticality messages to be transmitted over a link in high-criticality mode if this link cannot be used to transmit any high-criticality message. This can happen for two reasons. First, there are no critical flits in the upstream router buffers. Second, the downstream router buffers are full, thus preventing the high-criticality flits from being transmitted. Since high and low criticality messages are transmitted over different VCs, they use different buffers, so there could be space available for transmitting low-criticality messages. This makes the mode switch equivalent to changing the priorities of a VC, so that high-criticality VCs have always priority over low-criticality ones. When the system works normally, it can be beneficial to grant high

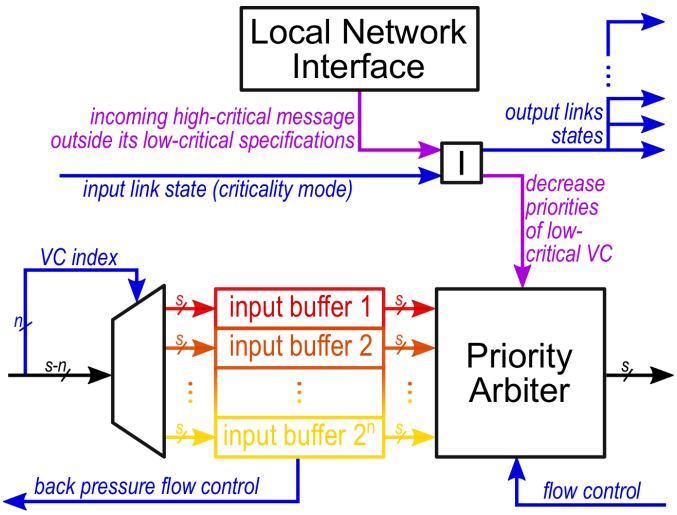

Figure 2. Router for WPMC-Flood protocol. Only one input and one output link are presented. s is the flit size.

priority to low-criticality VCs to improve the QoS of best-effort traffic.

Second, they propose to modify the way the high-criticality mode propagates through the network. Instead of transmitting it alongside high-criticality messages, the mode switch propagates to all neighboring routers, thus forcing the whole network to switch mode. They demonstrated an increase in schedulability in simulations with randomly generated task sets compared to the basic WPMC. However, this might not scale too well w.r.t. the NoC size, as it depends strongly on how long it takes in the worst case for the whole network to switch mode, which is directly proportional to the NoC size. Besides, though they showed an increase of the QoS for low-criticality tasks compared to WPMC, they do not compare this new mode switch procedure with the mode switch propagation procedure used in WPMC while enabling low-criticality messages to use high-criticality links on slack time.

B. Wormhole Protocol with Blocking Counter

Tobuschat and Ernst [19] proposed a protocol with criticality on the VC/message level only. They consider guaranteed latency critical traffic and best-effort non-critical traffic. VCs used by critical messages have lower priorities than VC used by non-critical messages. This optimizes the QoS of best-effort traffic. To enforce critical messages to be delivered in time, the header flits of critical messages contain a Blocking Counter (BC). This counter is initialized by the network interface at the highest number of times the message can be stalled in the network while keeping to its time requirements. Each time the message cannot access the output link of the router in which it waits, its BC is decremented.

The BC is only stored in the header flit of the message, therefore the message must not be split. Thus, a best-effort VC can be preempted at flit level, but a guaranteed latency VC can only be preempted at message level.

When a buffer exposes a message whose BC is zero, the authors propose three alternative actions. First, the corresponding VC becomes critical and gets priority over all best-effort VCs and normal (non critical) guaranteed latency VCs. This demands low hardware overhead, but allows for some more stalling of the critical message if there are flits from another message in the same VC/buffer. In the second possibility, the critical message can be redirected to a specific queue, thus bypassing the arbiter. A third possibility is to introduce some priorities depending on the BC of each message.

It is here assumed that no back pressure will ever occur, i.e. that the input buffers of all routers are large enough to handle any potential load. Including back pressure into this scheme would be possible but complex. This is because once a message BC has reached zero, it must not be delayed due to back pressure. If the first of the three aforementioned actions is taken, a mechanism anticipating messages switching into critical mode and making sure that buffers of the downstream routers will not get full at the wrong instant would be very complex to implement. The authors argue that a rate constraining mechanism (e.g. the one described in section II-D) at the network interface coupled with sufficiently large input buffers is simpler and can make the assumption of no back pressure safe. Therefore, they assume a given upper arrival curve for each type of message, that is an upper bound to the number of messages of a certain type which can be issued per time interval. In the second proposed reaction, this is a lesser problem, as no back pressure should ever occur on a VC reserved for traffic that cannot be delayed any more.

The authors demonstrate how to formally compute upper limits to the transmission latencies of all messages in a given message set. The message set comprises the size, the criticality class, the timing requirements (minimum and maximum arrival curves, deadline) and the initial BC value of each message. Non-critical messages have neither a deadline nor a BC value.

Furthermore, a method is proposed to compute the maximal initial BC value which enforces that all critical messages are delivered before their deadline. First, the BC of all critical messages are set to zero, then the worst-case delivery time of each message is computed, and the results are compared to the critical messages deadlines. If a message is delivered earlier than needed, its BC is increased and all latencies are computed again. These steps are repeated until a message latency is found which exceeds the message deadline. They also simulated their protocol, implementing the first of the 3 possibilities described above and using a RR arbitration within messages of a same VC. They showed improved QoS of non-critical traffic compared to a standard

Table I
SUMMARY OF COMMUNICATION PROTOCOLS

Designation	Type	Nr. of CL	Main feature(s)	Ref.
TDMA-based arbiter	Bus	2	Complete temporal isolation between CL (down to TDMA) → easy timing analysis	[15]
CArb	Bus	n	2 arbitration layers, between criticality classes and within them; arbiter-level mode switch	[16]
WPMC	NoC	2+	Network-level mode switch; check for traffic timing compliance in the network interface	[17], [18]
Wormhole with BC	NoC	2	Priority given to non-critical traffic as long as possible to improve its QoS	[19]
DAS	NoC	2	Prioritized SAF transmission of critical message aimed at tightening WC bounds	[20]
Adaptive Routing	NoC	2	Spatial (instead of temporal) partitioning triggered upon critical messages issues	[21]

wormhole scheme with higher priorities for critical traffic, and upper bounded latency for critical traffic.

C. Double Arbiter and Switching (DAS)

This protocol was proposed by Dridi et al. in [20]. It aims at a less pessimistic WCET analysis compared to wormhole protocols like the ones described above. DAS features $N + 1$ VCs, where N is the maximum number of high-criticality flows which can request a link at the same time. The N first VCs are dedicated to high-criticality traffic, the remaining one is for low-criticality traffic. A wormhole protocol without priorities is used for non-critical traffic. Low-criticality traffic has lowest priority and can be preempted at flit level by any high-criticality message. High-criticality messages are transmitted following a Store And Forward (SAF) protocol, requiring the buffering of complete messages in the routers. Additionally, high-criticality traffic can only be preempted at message level. Arbitration between high-criticality VCs is done according to a RR policy. The use of SAF for high-criticality traffic is justified because high-criticality messages are expected to be short, therefore the induced buffering overheads are tolerable. The authors claim that the use of SAF allows less pessimistic WCET analysis. However, they do not provide any formal proof of this assertion. They simulated their scheme at transaction level and compared it against a standard wormhole protocol and observed significantly smaller latencies for high-criticality traffic.

D. Adaptive Routing of Non-Critical Messages

This scheme was proposed by Kostrzewa et al. [21]. While the former protocols use temporal partitioning to isolate critical from non-critical traffic, this approach uses spatial partitioning. Upon critical message issue, all links this message will go through are reserved, and non-critical traffic is re-routed to alternate, longer paths. Once the critical message has been delivered, links are again accessible for non-critical traffic.

There are two kinds of routing policies: destination-based routing and source routing. With source routing, the full path of a message is specified by its sender. With destination-based routing, only the destination is specified by the sender and the path is computed by the routers along the way. Destination-based routing is more widely used, as specifying a full path for each message induces some overhead. It is used in all previously described protocols. However, implementing dynamic destination-based routing would require complex router design, hence the use of source routing here.

A control layer is responsible for multicasting the NoC state (i.e. which links are currently reserved for critical traffic) to all best-effort senders which may use a reserved link. When a critical message is sent, its sender also notifies the control layer and indicates the path of the critical message. When another sender specifies the path for a best-effort message, if the shortest path goes through a link which is currently reserved for critical traffic, the sender must specify a detoured path which does not interfere with any critical path. This increases the latency for best-effort traffic, but avoids stalling it, and improves resource utilization by redirecting the load to links where congestion cannot cause a deadline miss. However, interference is not completely suppressed, as some non-critical messages may have been issued before the critical one, which will use some links along the critical path. This must be taken into account in the static analysis, but this interference is bounded by the sum of the control layer response time plus the best-effort message transmission time. Besides, critical traffic could be prioritized over best-effort traffic, however this is not addressed in this paper [21], which considers RR arbitration between all messages in the routers.

Since computing the shortest free path is demanding in terms of computation time, the authors use statically defined sets of paths. Critical traffic has only one path (not necessarily the shortest one) to achieve predictability, and best-effort traffic can have an arbitrary number of paths, sorted by their lengths. At run time, the sender only has to select the shortest path of which no link is currently reserved for critical traffic.

The senders are also responsible for sending notifications when the transmission of the critical message is over. This can be done either based on a timeout mechanism, that is the message is considered delivered a fixed amount of time after its transmission began, or when the last flit of the message has been transmitted. In the latter case some blocking might occur if a non-critical message is issued downstream of the critical message sender, depending on the latency of the control signal transmission.

Assuming that upper and lower latency bounds for the

transmission of a message along a given path are available, the authors show how to calculate the available slack for each critical message, and the maximum transmission time overhead needed to re-route best-effort traffic. If the available slack is larger than the overhead, then the scheme will ensure that all critical messages are delivered in time. If the slack is not sufficient, then the path used by the critical message can be permanently reserved for critical traffic, thus eliminating the need for detouring non-critical traffic.

The authors tested their scheme in simulation. They showed significant gains in best-effort traffic average latency compared to a standard wormhole protocol. They also showed that the overheads in terms of transmission time increase proportionally with the number of critical senders and the number of critical messages they issue, which is reasonable.

V. MIXED-CRITICALITY ARCHITECTURES

A. Integrated Dependable Architecture for Many-Cores

This architecture, proposed by Motruk et al. [22], is oriented toward fault tolerance and fault containment. It is based on a NoC suited for mixed-criticality, interconnecting tiles. Tiles can contain any component or group of components. One can imagine basic processing tiles, with one core, a timer and an interrupt controller, memory tiles with a DRAM bank and its controller, I/O tiles, or much more complex tiles, e.g clusters of cores with several local memory banks like proposed in [23] (see section V-B). There is no limitation to the complexity of a tile. The mixed-criticality is not explicitly considered, except at the NoC level. Since the content of tiles is left unspecified, some mixed-criticality relevant problems are not addressed, e.g. arbitration between incoming requests on a memory tile. The resource server presented in section V-C is a possible addition. This architecture focuses on the NoC, the network interface, and a centralized System Controller (SC).

1) Network Interface: The network interface is presented in figure 3. It is divided in two distinct parts, which are parallelized. The first one is the Address Translation Table (ATT), which enables the virtualization of resources physically on distant tiles but which need to be accessed from the local tile. Those distant resources are assigned a virtual address, valid on the local tile only. The ATT maps this virtual address to the data needed to actually access the corresponding resource: a VC index, a route to the destination tile (source routing is used here, but destination-based routing could be used as well), and the address of the resource on the destination tile. Note that the VC used does not depend on the actual destination, but on its virtual address. A same destination may have several distinct virtual addresses on the local tile, to enable data transmission over different VCs, and hence transmission of traffic of different CL.

The second part of the network interface is the run-time monitoring mechanism. This works in a very similar way to the first version of the budgeting mechanism described in section II-B. On the one hand, there is a counter for each entry in the ATT, which is incremented each time a request is issued to the corresponding virtual address, and an access budget to this virtual address. On the other hand, there is a time interval counter which resets all counters when it reaches its maximum value, provided no budget overrun happened during the last time interval. Note that if a single budget was exceeded, none of the budgets on the tile will be replenished.

When a budget overrun occurs, 4 reactions can be taken. The tile can be stalled until the end of the current time interval. It can be reset at the end of the interval. It can be disabled until the whole system is reset. A message can be sent to the SC. The authors suggest that the latter should always be done, at least as information to the SC. Those actions do not exclude each other, i.e. any combination of them can be set.

2) Centralized System Controller: The SC has several important roles to ensure dependability, beside being able to issue messages and requests like any other tile. First, it specifies which reactions should be taken in case of budget overrun for each tile (the authors did not consider individually specified reactions for each virtual address of each tile). This can be changed dynamically, depending on the overrun messages received by the SC, the current NoC load, or other available information. Second, the SC can externally force any of the possible automated reactions of the network interface, i.e. it can reset, stall or disable tiles, at any time. Third, and most important, the SC is responsible for dynamically configuring the ATT. To demonstrate how useful this can be, let the two following examples be reviewed.

First example, if the SC detected a tile as faulty and disabled it, this tile cannot be accessed from other tiles any more. To prevent other tiles trying to access it from being impacted by the fault, the virtual addresses leading to the faulty tile can be remapped to another tile, identical to the former one but correctly functioning. This achieves fault tolerance through active redundancy. Second example, if a faulty task running on a tile attempts to flood the NoC, but the tile cannot be stalled right now because of other tasks running on it, the corresponding ATT line can be disabled, to prevent faulty messages from flooding the NoC, and to allow other tasks on the same tile to access the NoC (since request counters are only reset when no budget overrun occurs on the whole tile). This achieves fault containment between the tile and the NoC, and to a certain extent between applications running on the same tile.

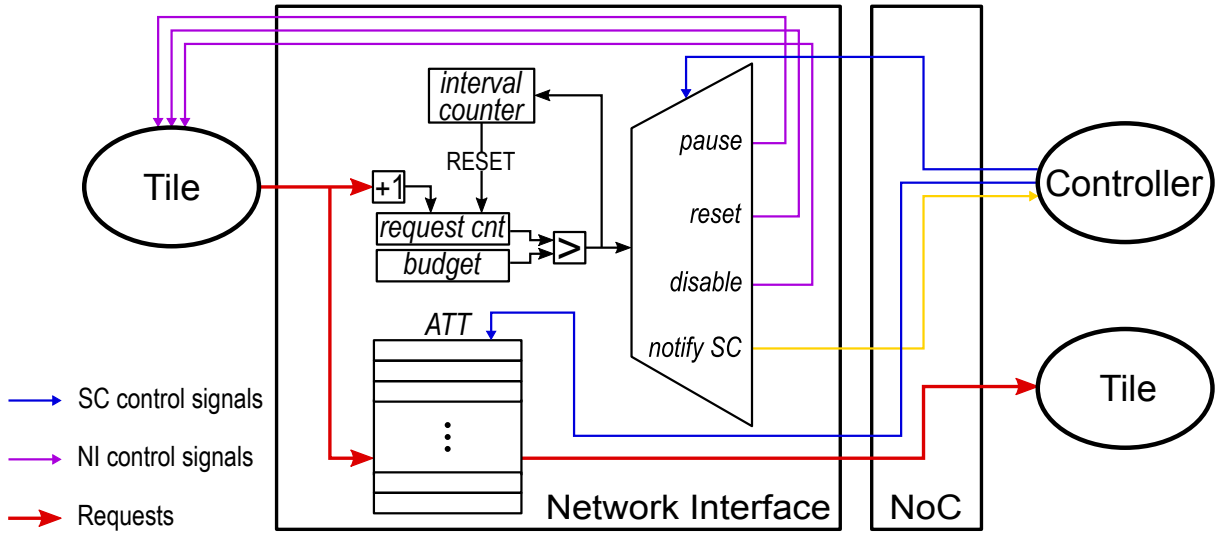

Figure 3. IDAMC network interface

B. Cluster-Based Many-Cores Architecture

This architecture was presented by Giannopoulou et al. [23]. Compared to the previously introduced IDAMC, it also uses a NoC and largely independent tiles. However, it does not feature a centralized controller, rather the control is distributed over all tiles. Two types of tiles are specified, respectively called computation clusters and I/O subsystems.

1) Hardware Architecture: The hardware architecture corresponds to the Kalray MPPA 256 platform. It includes 16 identical computation clusters and 4 I/O subsystems, interconnected via a NoC. Each computation cluster integrates 16 processing cores with caches, 16 SRAM banks of 128 kB, each with its own memory controller, a resource management core, and 3 NoC interfaces for transmission, reception, and resource management respectively. All memory controllers are directly connected to all cores and NoC interfaces. Thus, they are *de facto* responsible for arbitration between all cores and NoC interfaces. The NoC receiver interface has always priority. When the input buffer of the NoC receiver is empty, RR arbitration is performed between other components. Each I/O subsystem is connected to 4 NoC nodes to improve the available throughput.

2) Task Model: Tasks are considered to be periodic. A task execution profile contains the worst-case and best-case execution time without considering time spent for memory accesses, and also the worst-case and best-case number of memory accesses of the task. Each task features as many execution profiles as its criticality level (as a task of CL l must be scheduled at all CL under l), plus one degraded execution profile for the case when the system has switched to a CL higher than l. This enables to specify a minimal service for low-critical tasks rather than dropping them completely, thus improving on most basic mixed-criticality models. Should the task be dropped completely, its degraded execution profile is set to $(0, 0, 0, 0)$. Next to the periods, deadlines, CL and execution profiles of all its tasks, a task set model also includes a dependency graph. This is a directed acyclic graph with weighted edges. Nodes represent tasks, and an edge from a task τ_i to a task τ_j indicates that τ_i should terminate before the execution of τ_j starts, because of data dependencies. Additionally, the weight on the edge from τ_i to τ_j indicates the minimum time interval between the termination of τ_i and the execution of τ_j.

3) Task Scheduling: The scheduling is performed at the cluster level, that is a task set is scheduled to execute on a given cluster. The proposed scheduling policy (presented in [23]) divides the scheduling period into frames and sub-frames. Within a given sub-frame, only tasks of the same CL can execute. This prevents tasks of different criticality to interfere with each other on the cluster. Besides, when a task needs to read data from a distant resource, be it off-chip memory, any peripheral linked to an I/O subsystem, or another computation cluster, an additional task is created, which sends the request to get these data through the NoC. The dependency graph features an edge between the task which requests the data and the task which needs it, and the weight of this edge corresponds to the worst-case response time of the resource, plus worst-case latency of both transmissions through the NoC. This way, a task will never be delayed due to interference from other clusters. Interference is not suppressed, as it must be taken into account in the NoC and resource response time, but it is kept under control. So, interference can only cause additional delay within a cluster, and only between tasks of the same criticality. This is sufficient for certifying mixed-criticality integration provided that task sets of various criticality levels meet their corresponding certification requirements,

but not necessarily for efficient resource usage and good schedulability.

4) Optimization problem: Optimizing resource allocation is in this case a two-fold problem. First, tasks must be scheduled onto cores, and second, data must be stored onto memory banks. The task mapping problem solution should minimize the worst-case sub-frame length. This has to do with load balancing. The data mapping problem solution should minimize the worst-case interference. Those two aspects are highly interdependent, as interference will arise when two tasks are scheduled on the same sub-frame and need to access the same memory bank, and as interference is a major contributor to WCET of tasks, which are used for scheduling. The authors propose two methods, based on simulated annealing, to solve this optimization problem. They found that the most efficient method depend on the particularities of each case. The first possibility is to compute a Pareto set of possibly optimal memory mappings assuming worst-case task mapping, then to solve the task mapping problem for each points in this pareto set. The final result is the solution for which the given task set WCET is the shortest. The second possibility is to solve both problems iteratively, one after the other, until convergence.

C. Resource Server

The application of resource servers to support mixed-criticality was proposed by Brandenburg in [24]. Resource servers prevent direct access to shared resources. Their purpose is to control the accesses to a given peripheral, which can ensure some properties about the resource accessibility, provided the inter-process communication (IPC) protocol which manages accesses to the server is suitably designed. Typical examples of those properties are fault containment and worst-case response time. Resource servers coupled to IPC protocols which take criticality into account are therefore particularly suitable architectural blocks for MCS. A generic IPC protocol for resource server access, specifically tailored for MCS, and aimed at being implemented within a microkernel, was described in [24]. It is described at the transaction level, which means communication and some hardware details are abstracted. Nevertheless, communication must be taken into account for the static analysis, either within the WCET of the operating system calls composing the server implementation or within the response time of the resource itself. As the described protocol makes use of some scheduling properties to ensure worst-case latencies, it does require two assumptions about the scheduling policy to hold, but remains very widely applicable. Figure 4 shows the system model and the detailed server structure.

1) System Model and Assumptions on Scheduling: Cores are grouped into clusters. Each cluster is equipped with its own top-level scheduler. Tasks are grouped into reservations, which are statically assigned to a given cluster. Each reservation features a budget and a priority. The top-level scheduler assigns as many reservations as available cores in the cluster, more precisely the reservations of highest priority with non-depleted budget. The two assumptions made on the scheduling policy are the following. One, whenever a reservation is active, it spends its budget at a fixed, unique speed. A reservation which has a pending request is always regarded as active, even if it is currently not running. Two, a reservation priority can only change when its budget gets either depleted or replenished. Provided those two assumptions hold, this resource server can be used with any scheduling policy. The system model itself is sufficiently generic to include a wide range of cases.

2) Server Structure: The proposed server is built upon several queues, which gather requests from all clusters. Two of them are shared FIFO queues and can be accessed by any cluster. There are two more queues per cluster, which are reserved for requests coming from a given cluster.

The first shared queue is called *background queue*, and gathers non-critical requests. Requests from this queue can only be served when the second shared queue, called *global queue*, is empty. The global queue gathers critical requests, but at most one per cluster at a time. This implements starvation-free RR arbitration between clusters. When a cluster issues a request before the previous one was served, this additional request cannot enter the global queue, thus it is stored in a queue reserved for the cluster.

There are two cluster-specific queues for that purpose. The first one, called *head queue* is FIFO and its size is equal to the number of cores in the cluster minus one. As long as there are any spots free in the head queue, requests are appended at the tail of this queue. The second sub-queue, called *tail queue* is priority-based and its size is left unspecified. When the cluster's head queue is full, requests are inserted in the tail queue based on the priority of the reservation which issued the request. Critical requests pass from the tail queue to the head queue, and from the head queue to the global queue when spots get free. The size of the head queue should be minimized to make an efficient use of priorities, but the proposed static analysis makes the given size a minimum: with a shorter head queue latency bounds cannot be ensured, as a request could be blocked in the tail queue for ever.

3) Methods for Server Implementation: The author splits the software needed to implement its scheme at the microkernel level in four methods. Those methods enable communication between the high-level software and the server through calls to the operating system. The first method is *invoke*, and is called by a task to instantiate a request. The second one is *reply*, and is called upon request completion by the server itself to send back the output of the serviced request (if any) to the processor. This method is also responsible for checking if there are any pending requests to service next, and if so, to update request queues according to their respective policies. The third one is *abort*,

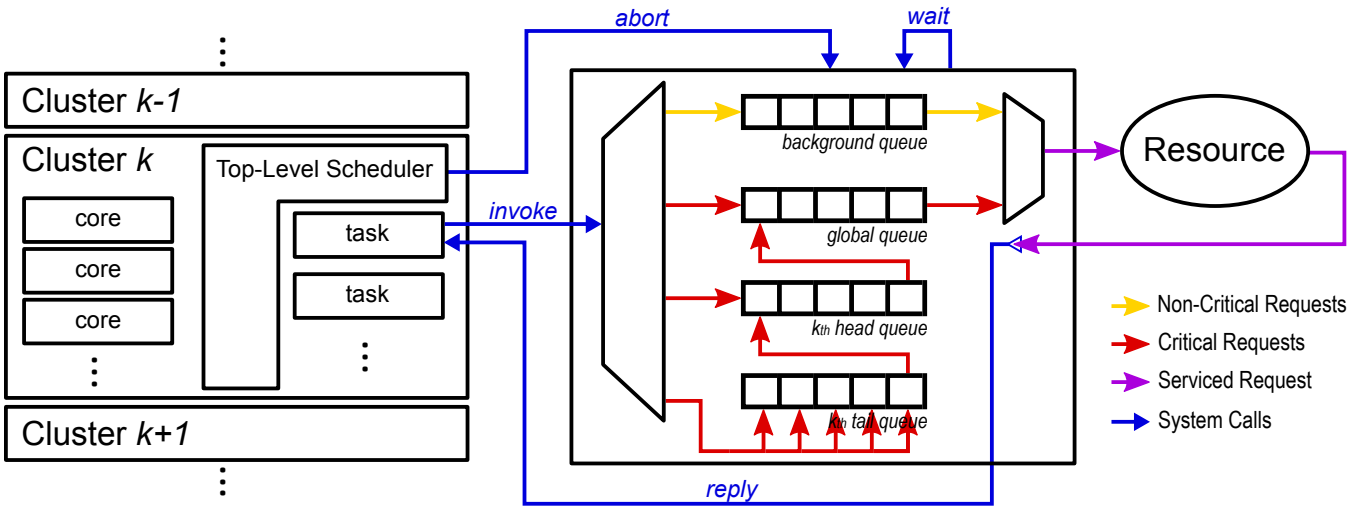

Figure 4. Resource Server Architecture. The left side is the high-level software, the right side is the low-level server. Layers communicate through OS calls.

and is called by the top-level scheduler when a task having issued a request is aborted to remove this request from the queue. The last one is *wait* and is called when all queues are empty to wake up the server upon arrival of the next request.

The author provides in [24] a formal analysis to demonstrate that the worst-case latency is effectively bounded, and simulation results indicate the same. The upper bound derived is quasi proportional to the number of cores within the cluster which issued the request and also to the total number of clusters which can simultaneously access the resource. This highlights the strong dependence between the worst-case latency and the scheduling parameters.

4) Introducing more than two Criticality Levels: This part is suggested as an improvement in [24], but neither a formal proof of correctness nor simulation results are provided.

This system integrates well with a full-scale criticality mode switch, as when low-criticality tasks are dropped down to such a mode switch, the top-level scheduler may call the abort method to abandon requests issued by low-criticality tasks. Next to that, a mode switch is also possible at the server level. Two parameters can be considered criticality-dependent, the worst-case service latency of the resource (without considering the server delays and overheads), and the number of clusters which might concurrently compete for the resource. The server can easily detect whenever one of those parameters exceeds its low-criticality value, and react accordingly by displacing low-criticality requests to the head of the background queue.

VI. CONCLUSION

This digest describes mechanisms to implement systems supporting the integration of mixed-criticality task sets. It is limited to considerations on shared resource access, and especially communication media. However, this is only part of the challenges of mixed-criticality integration. As stated in the introduction, there are 2 main branches of the mixed-criticality research, namely mechanisms to enable safe and efficient sharing of resources on the one hand and mixed-criticality scheduling policies on the other hand. Combination of those two domains with each other has just begun to be addressed, and most mixed-criticality scheduling researches assume the basic mode switch model described in section II-A. [23] is the only paper to our knowledge deeply integrating scheduling with a specific hardware architecture.

Without speaking of mixed-criticality scheduling, which is a very active research field, some research on implementation mechanisms were not covered in this paper. Extensive research has been conducted on memory elements: controllers, memory architectures, cache organizations and cache replacement policies supporting MCS. See [1] for an extensive review. Static analysis has also been overlooked in this paper, however getting worst-case response times and latency bounds in the context of complex multicore systems is a real challenge. Some research points have not yet been addressed in detail, especially interrupts management and data sharing between tasks of different criticality. Both those elements can jeopardize the correctness (temporal resp. logical) of a critical task, unless all interrupt routines are certified at the highest level of insurance, and all data used by a task of criticality l can only be accessed by tasks of the same criticality. Those constraints are yet too restrictive to permit actual co-running of applications of different criticality. In a nutshell, mixed-criticality integration is a promising paradigm to increase resource utilization and efficiency of safety-critical systems, but there is still much work to do on this topic.

REFERENCES

[1] A. Burns and R. Davis, "Mixed-Criticality Systems - A Review", 10th edition, University of York, UK, Jan. 2018.

[2] R. Ernst and M. di Natale, "Mixed-Criticality Systems - A History of Misconceptions ?", in *IEEE Design and Test, vol. 33 issue 5*, Oct. 2016. doi: 10.1109/MDAT.2016.2594790

[3] O. Kotaba, J. Nowotsch, M. Paulitsch, S. M. Petters, H. Theiling, "Multicore in Real-Time Systems - Temporal Isolation Challenges due to Shared Resources", CISTER technical report, 2014.

[4] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, L. Sha, "Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality", in *24th Euromicro Conf. on Real-Time Systems*, 2012 IEEE. doi: 10.1109/ECRTS.2012.32

[5] J. Flodin, K. Lampka, W. Yi, "Dynamic Budgeting for settling DRAM Contention of Co-Running Hard and Soft Real-Time Tasks", in *Proc. of the 9th IEEE Int. Symp. on Industrial Embedded Systems*, 2014 IEEE. doi: 10.1109/SIES.2014.6871199

[6] K. Lampka and A. Lackorzynski, "Resolving Contention for Networks-on-Chips: Combining Time-Triggered Application Scheduling with Dynamic Budgeting of Memory Bus Use", in *Lecture Notes in Computer Science, vol. 9629, pp. 137-152*, 2016 Springer. doi: 10.1007/978-3-319-31559-1_12

[7] A. Kritikakou, C. Pagetti, O. Baldellon, M. Roy, C. Rochange, "Run-time Control to Increase Task Parallelism in Mixed-Critical Systems", in *26th Euromicro Conf. on Real-Time Systems*, 2014 IEEE. doi: 10.1109/ECRTS.2014.14

[8] A. Kritikakou C. Pagetti, C. Rochange, M. Faugére, S. Girbal, D. Gracia Pérez, "Distributed Run-time WCET Controller for concurrent critical tasks in Mixed-Critical Systems" in *Proc. of the 22nd Int. Conf. on Real-Time Networks and Systems*, 2014 ACM. doi: 10.1145/2659787.2659799

[9] A. Kritikakou, T. Marty, M. Roy, "DYNASCORE: DYNAmic Software COntroller to Increase REsource Utilization in Mixed-Critical Systems", in *ACM Trans. on Design Automation of Electronic Systems, vol. 23 issue 2*, Dec. 2017. doi: 10.1145/3110222

[10] E. Wandeler, A. Maxiaguine, L. Thiele, "Quantitative Characterization of Event Streams in Analysis of Hard Real-Time Applications", in *Real-Time Systems, vol. 29, pp. 205-225*, 2005 Springer. doi: 10.1007/s11241-005-6885-x

[11] M. Neukirchner, P. Axer, T. Michaels, R. Ernst, "Monitoring of Workload Arrival Functions for Mixed-Criticality systems", in *34th Real-Time Systems Symp.*, 2013 IEEE. doi: 10.1109/RTSS.2013.17

[12] M. Neukirchner, S. Quinton, R. Ernst, K. Lampka, "Multi-mode Monitoring for Mixed-Criticality Real-Time Systems", in *2013 Int. Conf. on Hardware/Software Co-Design and System Synthesis*, 2013 IEEE. doi: 10.1109/CODES-ISSS.2013.6659021

[13] K. Napier, O. Horst, C. Prehofer, "Comparably Evaluating Communication Performance within Mixed-Criticality Systems", in *WMC 4th Int. Workshop on Mixed Criticality Systems*, Nov 2016, Porto, Portugal. hal-01417283

[14] J. Rosén, A. Andrei, P. Eles, Z. Peng, "Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip", in *28th Real-Time Systems Symp.*, 2007 IEEE. doi: 10.1109/RTSS.2007.24

[15] B. Cilku, A. Crespo, P. Pushner, J. Coronel, S. Peiro, "A TDMA-based Arbitration Scheme for Mixed-Criticality Multicore Platforms", in *2015 Int. Conf. on Event-based Control, Communication, and Signal Processing*, 2015 IEEE. doi: 10.1109/EBCCSP.2015.7300671

[16] M. Hassan and H. Patel, "Criticality and Requirements Aware Bus Arbitration for Multicore Mixed-Criticality Systems", in *2016 Real-Time and Embedded Technology and Applications Symp.*, 2016 IEEE. doi: 10.1109/RTAS.2016.7461327

[17] A. Burns, J. Harbin, L. S. Indrusiak, "A Wormhole NoC Protocol for Mixed-Criticality Systems", in *2014 Real-Time Systems Symp.*, 2014 IEEE. doi: 10.1109/RTSS.2014.13

[18] L. S. Indrusiak, J. Harbin, A. Burns, "Average and Worst-Case Latency improvements in Mixed-Criticality Wormhole Networks-on-Chips", in *27th Euromicro Conf. on Real-Time Systems*, 2015 IEEE. doi: 10.1109/ECRTS.2015.12

[19] S. Tobuschat and R. Ernst, "Efficient Latency Guarantees for Mixed-Criticality Networks-on-Chips", in *2017 Real-Time and Embedded Technology and Applications Symp.*, 2017 IEEE. doi: 10.1109/RTAS.2017.31

[20] M. Dridi, S. Rubini, M. Lallali, M. J. S. Flórez, F. Singhoff, J.-P. Diguet, "DAS: A efficient NoC Router for Mixed-Criticality Real-Time Systems", in *2017 Int. Conf. on Computer Design*, 2017 IEEE. doi: 10.1109/ICCD.2017.42

[21] A. Kostrzewa, S. Tobuschat, L. Ecco, R. Ernst, "Adaptive Load Distribution in Mixed-Critical Networks-on-Chip", in *22th Asia and South Pacific Design Automation Conf.*, 2017 IEEE. doi: 10.1109/ASPDAC.2017.7858411

[22] B. Motruk, J. Diemer, R. Buchty, R. Ernst, M. Berekovic, "IDAMC: A Many-Core Platform with Run-Time Monitoring for Mixed-Criticality", in *14th Int. Symp. on High-Assurance Systems Engineering*, 2012 IEEE. doi: 10.1109/HASE.2012.19

[23] G. Giannopoulou, N. Stoimenov, P. Huang, L. Thiele, B. Dupont de Dinechin, "Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources", in *Real-Time Systems, vol. 52*, 2016 Springer. doi: 10.1007/s11241-015-9227-y

[24] B. B. Brandenburg, "A Synchronous IPC Protocol for Predictable Access to Shared Resources in Mixed-Criticality Systems", in *2014 Real-Time Systems Symp.*, 2014 IEEE. doi: 10.1109/RTSS.2014.37