The Road towards Predictable Automotive High-Performance Platforms

Falk Rehm[1], Jörg Seitter[1], Jan-Peter Larsson[3], Selma Saidi[2], Giovanni Stea[4], Raffaele Zippo[5,4], Dirk Ziegenbein[1], Matteo Andreozzi[3], and Arne Hamann[1]

[1]Robert Bosch GmbH, Germany, firstname.lastname@de.bosch.com
[2]Technical University of Dortmund, Germany, firstname.lastname@tu-dortmund.de
[3]Arm, UK, firstname.lastname@arm.com
[4]University of Pisa, Italy, firstname.lastname@unipi.it
[5]University of Florence, Italy, firstname.lastname@unifi.it

***Abstract*—Due to the trends of centralizing the E/E architecture and new computing-intensive applications, high-performance hardware platforms are currently finding their way into automotive systems. However, the Systems-on-Chip (SoCs) currently available on the market have significant weaknesses when it comes to providing predictable performance for time-critical applications. The main reason for this is that these platforms are optimized for average-case performance. This shortcoming represents one major risk in the development of current and future automotive systems. In this paper we describe how high-performance and predictability could (and should) be reconciled in future HW/SW platforms. We believe that this goal can only be reached via a close collaboration among system suppliers, IP providers, semiconductor companies, and OS/hypervisor vendors. Furthermore, academic input will be needed to solve remaining challenges and to further improve initial solutions.**

I. Introduction

There is a clear trend in the automotive domain towards a new paradigm of centralized E/E architectures, where large portions of formerly separated functionalities running on dedicated ECUs are integrated into centralized vehicle integration platforms (VIP) [1]. At the same time, novel computation- and data-intensive algorithms, such as, for instance, predictive maintenance or automated driving (AD) functionalities, are being deployed on these centralized high-performance platforms.

In order to satisfy the tremendous demand of "centralized" computing power, heterogeneous SoCs are being increasingly deployed in automotive systems. These SoCs are µP-based, featuring a variety of integrated specialized accelerators, including GPUs and FPGAs. Examples of this class of SoCs include NXP's S32V vision processor family, or the Tegra series offered by Nvidia.

Compared to traditionally used micro-controllers, these heterogeneous SoCs are highly parallel and feature complex memory systems, composed of multiple levels of on-chip shared SRAM memories (caches or scratch-pads) and off-chip DRAMs. Obviously, the increased complexity of the memory system that is shared between multiple execution engines on the SoC leads to a strong performance correlation between parallel executed applications [2]. While programmatically data is accessed transparently through virtual address spaces, it is physically stored at different (shared) memory locations, with different access latencies that are dynamically influenced by complex access and caching schemes as well as mechanisms for ensuring data coherency and consistency. For instance, in [2] it has been shown that the average (sequential) read access latency can vary by a factor of up to $8x$ on a Nvidia Tegra X1 platform.

When looking especially at mixed-criticality systems with high ASIL levels, it becomes clear that *shared* hardware resources, such as a memory subsystem with caches and DRAM, must be controlled to ensure predictable performance and achieve freedom from interference in space and time as requested by ISO26262. While spatial separation can be controlled, e.g. with a hypervisor and Memory Management Units (MMU/MPU), resource-efficient temporal isolation providing timing predictability while preserving performance is much harder to achieve.

In this paper, we discuss different ideas involving a combination of software mechanisms (Section II) and hardware features (Section III) to address the problem of providing predictable performance on embedded high-performance automotive platforms. To achieve this, we believe that a close collaboration is necessary among system suppliers, IP providers, semiconductor companies, and OS/hypervisor vendors. Moreover, for specific parts and setups, methods from formal performance modeling and analysis can be leveraged in order to understand performance effects and derive sensible system configurations (Section IV). To guarantee performance for complex applications, multiple shared resources of a SoC must be orchestrated and configured adequately (e.g. the interconnect and the memory controller). Individual resources, however, are arbitrated and configured independently of each other, although the overall system performance is strongly influenced by interactions and correlations among multiple resources. This "non-composability" makes system-wide configuration an

978-3-9819263-5-4/DATE21/©2021 EDAA

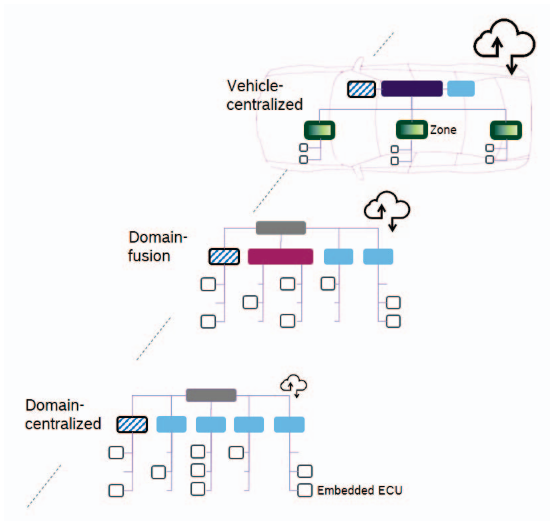

Fig. 1. Three classes of centralized automotive E/E architectures. While domain-centralized and domain-fusion architectures group embedded ECUs according to their function domain, vehicle-centralized architectures combine embedded ECUs according to their mounting position in the vehicle.

extremely challenging and time consuming task. As possible way out, we discuss the idea of admission control borrowed from the networking domain [3] that has the potential to greatly simplify achieving predictable performance in complex SoCs considering a sequence of heterogeneous shared resources accessed in an end-to-end (E2E) fashion (Section V).

II. Performance Predictability in Upcoming Centralized Automotive E/E Architectures

The traditional decentralized automotive E/E architectures are the result of multiple years of evolution of vehicle functionality. By using dedicated hardware for additional and possibly optional functionalities, decentralized architectures enabled and followed the distributed development paradigm between vehicle manufactures and suppliers. Decentralization also facilitated the structural partitioning of the vehicle system into functional domains. On the one hand, this is important for the planning, design and implementation of vehicle functionality in a parallel setup to minimize organizational interfaces. On the other hand, the corresponding functional partitioning (one function - one control unit) limits the functional interfaces and integration effects to the communication networks.

This architectural approach obviously results in a very close link between hardware and software since relocation of functionality is not an architectural driver. While decentralized architectures have carried the industry so far, new architectural drivers have appeared as automotive mega-trends: electrified, autonomous, connected and shared are the keywords that describe future expectations to a vehicle that must be backed by the E/E architecture. Centralized E/E architectures (Figure 1) bring the opportunity of cost and weight savings by reducing the number of control units and promise to reduce complexity in comparison to a distributed E/E architecture. However, the complexity of managing distributed logic with dedicated resources is merely replaced by the complexity of managing centralized logic on a parallel hardware platform with shared resources [1].

In addition, these centralized control units need to host software categories which range from real-time safety-critical embedded software all the way up to "app"-like software without safety requirements that can be updated in the field. In this mixed-criticality setting, it is mandatory to have predictable performance and isolation of applications from each other, with respect to both space and time. This can be achieved by actively managing Quality of Service (QoS) and limiting the contention and interference on shared resources. Unfortunately, the currently available COTS platforms are optimized for high average performance and offer only limited and coarse-grained support for configuring QoS for shared resources such as the interconnect or the DRAM. In order to achieve predictable performance, one has, thus, to resort to software-based methods.

While spatial isolation is well supported, e.g. at the level of POSIX processes, several software measures have been proposed to limit the temporal interference on levels of scheduling, data caching and memory access bandwidth.

Scheduling is concerned with the distribution of CPU resources to applications. In comparison to the well-established priority-based scheduling approaches, reservation-based scheduling approaches show advantages in offering composable QoS guarantees to applications while allowing more flexibility than TDMA-based scheduling [4]. In general, partitioned scheduling, i.e. the pinning of application processes to cores, shows better predictability than global scheduling in multi-core settings as interference effects can be better localized. However, this approach has limitations as well, since in many SoCs the CPU cores are allocated in clusters of multiple cores (usually 2 or 4). These clusters provide shared infrastructure, e.g the L2 cache. So pinning a process on one core of a cluster will still not resolve the interference between cores of the same cluster on the L2 cache, unless that cache is partitioned. Extreme isolation mechanisms such as a "stop-the-world" approach, where the execution of ASIL-D safety application on a single CPU core will stall all other cores in the system in order to generate a single-core equivalent scenario, are not adequate due to their performance penalty.

The previously mentioned issue of interference through caching can be addressed with cache coloring (e.g., [5]), exploiting the fact that (depending on the organization of the cache) certain address ranges will map to the same cache line. By choosing the mapping of virtual memory pages to physical pages with this in mind, performance-optimal memory allocation as well as cache partitioning can be achieved. However, this comes at the price of a factual smaller cache for each partition and additionally fine-grained page-mapping that can

cause side-effects in terms of page-table walks. Cache coloring can be supported by software on operating system or hypervisor level. Also, cache partitioning is directly supported by novel HW mechanisms such as Arm DynamIQ, see Section III-A.

In order to address interference topics outside of a CPU cluster, e.g. the access to DRAM, performance counters integrated in the SoC can be used to actively limit the number of requests and reserve memory bandwidth at the level of cores, hypervisor partitions or single applications, using software-based mechanisms such as Memguard [6]. This is an effective mechanism to limit interference. However, the more fine-grained the objects to be isolated get, the higher the overhead becomes. This overhead could be reduced if the SoC exposed more information, e.g., the source of a particular request, or implemented less coarse-grain resource partitioning mechanisms than those available in current SoCs (where QoS mechanisms are available at the cluster level, if at all) directly in HW, see Section III-B.

All these concepts are sophisticated approaches with their individual drawbacks, such that their stand-alone configuration is already quite intricate for an industrial practitioner in real-world application scenarios. However, there are additional interactions among these mechanisms. If you, e.g., use cache coloring to reserve cache for real-time critical applications in order to prevent cache thrashing by non-real-time applications, you effectively reduce the cache size for all applications. This could in turn lead to more DRAM traffic which will increase the DRAM interference also towards the real-time applications. Finding an optimal configuration for these interacting mechanisms is highly dependent on the characteristics of applications and the HW platform. Thus, automated profiling as well as sophisticated configuration tools are required. Considering updates in the field at operation time, it is absolutely crucial that there is as little human intervention required in this as possible.

In addition to these quantitative dependencies among these resources, the different resources (e.g., interconnect and memory) need also to be available at the same time in order to avoid interference due to resource contention. An approach to solve this is the E2E admission control presented in Section V.

III. Resource contention avoidance mechanisms in high-performance Arm-based systems

Hardware can do more to help software reduce contention in shared resources by providing mechanisms that enable tighter observability and controllability of the behaviour of individual workloads than current software-based approaches are able to achieve.

Such hardware mechanisms can be broadly classified into three groups:

1) Identification mechanisms that allow software to label traffic flows in the system, for example labelling traffic belonging to a particular workload or virtual machine;

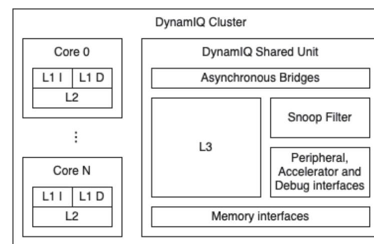

Fig. 2. Logical composition of a DynamIQ cluster, adapted from [7]

2) Monitoring mechanisms that allow software to observe the behaviour of traffic flows within the shared resources they pass through, for example in terms of their cache occupancy or bandwidth utilisation;
3) Control mechanisms that allow software to configure differential treatment of flows in shared resources, for example by restricting cache occupation or influencing arbitration policies in networks-on-chip (NoCs).

In the following paragraphs, we provide an overview of two Arm technologies that implement hardware-based mechanisms. We illustrate how the DynamIQ Shared Unit (DSU) and Memory System Partitioning and Monitoring (MPAM) architecture extension can help reduce resource contention.

A. The DynamIQ Shared Unit

DynamIQ is a compute cluster technology that allows compatible cores to be integrated into a heterogeneous or homogeneous cluster alongside a DynamIQ Shared Unit (DSU). The DSU is a subsystem that includes an optional shared L3 cache, control logic, and external interfaces [7]. Processors that can form DynamIQ clusters include Cortex-A78 [8], Cortex-A76AE [9] and Cortex-A65 [10]. Figure 2 shows a DynamIQ cluster comprised of multiple cores integrated alongside a DSU.

The L3 cache in the DSU is shared between all processors in the cluster, and the DSU supports hardware-based cache partitioning to mitigate the risk of contention of data flows in the cache. [7]

1) Scheme ID: The identification mechanism in the DSU cache partitioning scheme is based on software-configurable *scheme IDs*. Scheme IDs are comprised of 3 bits, allowing software agents to be assigned into one of 8 scheme ID groups. Scheme IDs can be set by privileged system software such as operating systems or hypervisors. Hypervisors can delegate a subset of scheme IDs to virtualised guest operating systems and restrict their use of other scheme IDs by configuring mask and override registers that replace part of the scheme ID set by the guest operating system with equivalent override bits controlled by the hypervisor.

2) L3 partitioning: The L3 cache in a DSU is 12- or 16-way set-associative and is logically split into 4 partition groups of 3 or 4 ways each. Each group can be configured in one of two ways:

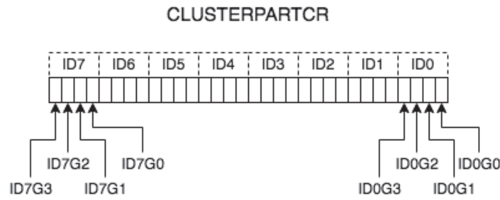

Fig. 3. Assignment of partition groups to schemeIDs in the DynamIQ Shared Unit L3 Cluster Partition Control Register

- Private to a scheme ID. This prevents allocations by other scheme IDs into the group
- Unassigned. This allows the ways within the group to be allocated by any scheme ID

Partitioning is configured by writing into a 32-bit register, where each register bit corresponds to a combination of scheme ID and partition group. Setting a bit in this register indicates that a group is private to the corresponding scheme ID, and a group for which none of the bits have been set is deemed to be unassigned. Figure 3 shows the mapping of partition groups to scheme IDs within the register.

The DSU partitioning mechanism can provide isolation between up to 4 traffic flows. As an example of a possible configuration, consider a system with a hypervisor running two virtual machines (VMs). The VMs run, respectively, a Real-Time Operating System (RTOS) with two real-time workloads, and a general-purpose operating system (GPOS). The hypervisor assigns itself the schemeID 7 (`0b111`), the GPOS VM the scheme ID 0 (`0b000`), and the RTOS VM the two schemeIDs 2 (`0b010`) and 3 (`0b011`). Assignment of schemeIDs within the real-time VM is delegated to the RTOS using an override mask of `0b110` and an override value of `0b01x`. The GPOS VM can be prevented from unilaterally changing its schemeID by setting an override mask of `0b111`.

The L3 cache is then partitioned between the hypervisor, GPOS VM and RTOS VM by writing the value `0x80004201` into the partition configuration register. This sets partition group 3 to be private to schemeID 7 (the hypervisor), partition group 2 to be private to schemeID 0 (the GPOS VM) and partition groups 1 and 0 to be private to schemeIDs 2 and 3 (the RTOS VM).

B. MPAM

The Armv8.4-A Memory System Resource Partitioning and Monitoring (MPAM) extension is an architectural approach to resource contention avoidance. MPAM provides workload identification of memory traffic throughout the system, as well as standard monitoring and control interfaces for observation of workload performance and apportioning of system resources like cache capacity and memory bandwidth. MPAM identifiers can be attached to memory system requests from CPUs [11] or to device traffic going through a System Memory Management Unit (SMMU) [12].

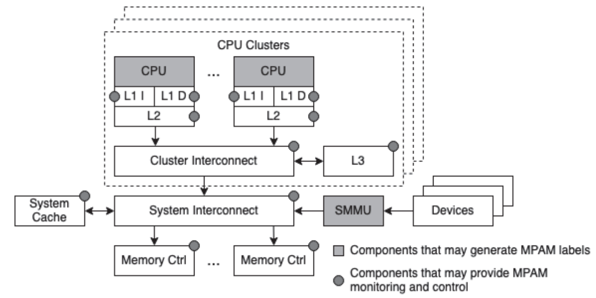

Fig. 4. Example of an MPAM system, with components highlighted according to their MPAM capability

Figure 4 shows a system diagram of an example MPAM system with CPUs and SMMU generating identification labels and other providing monitoring and control functionality based on MPAM labels.

1) Identification: Identification in MPAM is based on two types of identifiers:

- Partition Identifiers (PARTID) that identify the partition that generated a particular request for the purpose of monitoring and control
- Performance Monitoring Group (PMG) identifiers that identify agents within a partition for the purpose of monitoring

For example, an operating system can assign a PARTID to a workload to control its usage of a shared cache. The workload may be comprised of multiple operating system processes or execution threads, each of them possibly assigned an individual PMG within the PARTID. This allows a control policy to be applied to the entire workload, while monitoring can be performed at the granularity of individual processes or threads.

2) PARTID spaces: PARTIDs exist in one of four spaces:

- Physical non-secure PARTIDs for non-virtualised non-secure software
- Virtual non-secure PARTIDs for virtualised non-secure software
- Physical secure PARTIDs for non-virtualised secure software
- Virtual secure PARTIDs for virtualised secure software

The security space of a PARTID is determined by the TrustZone security state of the agent that made the request. The security space is encoded in an additional MPAM_NS bit alongside the PARTID and PMG. Restricting the ability of non-secure software to set control policies that apply to secure software mitigates the risk of side-channel information leaks between the secure and non-secure world.

The PARTIDs that memory requests are labelled with are termed physical PARTIDs (pPARTIDs). MPAM also provides for virtual PARTIDs (vPARTIDs) in order to allow hypervisors to delegate a subset of pPARTIDs to a guest operating system. Each guest OS can then manage its own contiguous vPARTID

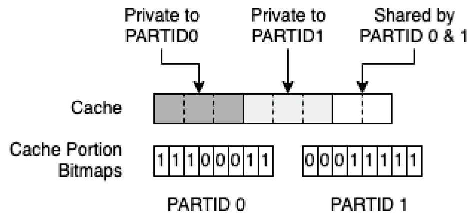

Fig. 5. Example assignment of cache portions to partitions using MPAM cache-portion partition bitmaps

space, and vPARTIDs are automatically translated back into pPARTIDs using mapping system registers [11] or translation tables [12] under hypervisor control.

3) Monitoring interfaces: MPAM provides two standard monitoring interfaces, both of which are optional:

- Cache-storage usage monitors that report the cache utilisation for a given PARTID and PMG
- Memory-bandwidth usage monitors that report the number of bytes transferred for a given PARTID and PMG

Up to 2^{16} monitors of each type can be implemented by each memory system resource. Monitors can be configured to filter requests by type, for example read or write, and by a choice of PARTID and PMG or PARTID only. MPAM monitors can optionally support capture registers that hold the monitor value after a capture event, allowing the values in multiple registers at a given point in time to be frozen and then read out sequentially. Capture events can be external to a resource, for example driven by a timer interrupt, or generated locally by writing into a capture register.

4) Control interfaces: MPAM provides 6 types of standard control interfaces, all of which are optional:

- Cache-portion partitioning
- Cache maximum-capacity partitioning
- Memory-bandwidth portion partitioning
- Memory-bandwidth minimum and maximum partitioning
- Memory-bandwidth proportional-stride partitioning
- Priority partitioning

Cache-portion partitioning subdivides a cache resource into a number of portions of equal and fixed size, up to a maximum of 2^{15} portions. The ability of a partition to allocate into a portion P_n is determined by bit B_n in a memory-mapped cache-portion bitmap register. This allows flexibility in portion assignment: a portion can be shared by a group of partitions, be private to a single partition, or remain open for allocation by any partition. Figure 5 illustrates an example of an apportioning of a cache with 8 portions between two PARTIDs, with two private cache partitions and one shared.

Cache maximum-capacity partitioning limits the ability of a partition to occupy more than a configurable fraction of the cache capacity. Cache maximum-capacity partitioning can be combined with cache-portion partitioning, for example to restrict the ability of a single partition to occupy all of the capacity of cache portions that have been made available to multiple partitions.

Memory-bandwidth portion partitioning subdivides memory bandwidth into a number of portions (quanta), up to a maximum of 2^{12} portions. The ability of a partition to use a bandwidth quantum Q_n is determined by bit B_n in a memory-mapped memory-bandwidth portion bitmap register.

Memory-bandwidth minimum and maximum partitioning allow setting of a minimum guaranteed and maximum permitted memory bandwidth that is applied to a partition in the presence of contention.

Memory-bandwidth proportional-stride partitioning is based on a configurable stride for each partition, permitting a partition to consume bandwidth in proportion to its own stride relative to the strides of other partitions that are competing for bandwidth.

Priority partitioning provides a way for resources to expose partition-based configuration of internal arbitration policies. These can be used by system software for fine-grained control over scheduling and arbitration policies in the memory system.

C. Summary

The hardware mechanisms provided by the DSU and by implementations based on the MPAM architecture offer improvements in efficiency and efficacy over software-based resource contention avoidance approaches like cache colouring. By decoupling partitioning from memory management code, hardware-based cache partitioning imposes fewer restrictions on memory allocation and permits better utilisation of the cache and downstream memory resources. In addition to cache partitioning, MPAM provides several types of control interfaces that can help limit memory bandwidth contention, for example in networks-on-chip or memory controllers.

IV. Supporting system design with formal performance analysis

Systems meant for mission-critical environments, such as automotive, aeronautical, robotic etc., must be designed so that they meet pre-specified QoS requirements. In other words, it is not sufficient that they are found to meet QoS requirements via ex-post performance analysis, which is usually performed via simulation, hence having limited coverage. They must instead meet those requirements by design, ex-ante. This requires in turn formal methods to be able to infer QoS guarantees from a behavioral description of the system. In particular, worst-case, deterministic bounds are the type of guarantees that best lend themselves to ex-ante formal certification, since they do not rely on assumptions on possible statistical patterns of a system's input, which might in turn not be verified in an operational environment. This is also important for design considerations: if you have a formal method to characterize the worst-case behavior of a system, you can use it during its design phase (e.g., to tune its parameters) so as to obtain a desired behavior.

As far as QoS is concerned, the most important bounds are on the backlog, which allows system builders to dimension buffer space at the elements so as to avoid losses, and on the delay, which allows them to compute component-wise or E2E guarantees on the response time of an application.

Network Calculus (NC, [13]) is a theory for computing such bounds, which has been originally devised for QoS in the Internet (laying the foundations for the IntServ and DiffServ architectures), and has later found applications in several domains, including avionic networks, time-sensitive networks, industrial Ethernet. In the embedded and real-time systems domain, a variant of network calculus, called real-time calculus, is often used. In network calculus, the worst-case service offered to a flow by a component is modeled as a function of time, called service curve. By comparing a service curve with an arrival curve, that bounds from the above the traffic generated by that flow over time, bounds on the backlog and delay can be computed. NC allows composable E2E analysis: one can determine an E2E service guarantee by composing per-node service curves. Being able to characterize systems as service curves is the fundamental (and certainly non-trivial) step that allows, on the one hand, to design systems that meet pre-specified worst-case performance guarantees, and, on the other hand, to setup service negotiation frameworks (e.g., admission control, route computation, resource reservation etc.) between an application and the underlying system.

A. Worst-case analysis of a FR-FCFS DRAM controller

In this section, it is shown how to compute worst-case delay (WCD) guarantees in a DRAM controller. The DRAM is a shared resource, where contention among different masters may affect performance and jeopardize deadlines. A First-Ready, First-Come-First-Served (FR-FCFS) DRAM controller is chosen as an example to show how to compute a service curve for incoming read requests. First, an upper bound on the WCD [14] is computed. Furthermore, it is shown that this upper bound is not overly pessimistic by computing a lower bound whose gap to the upper bound is null to negligible in practical cases. It is worth noting that:

- The method described below can be applied to any memory technology (e.g., DDR3, DDR4, LPDDR4, etc.), by changing the values of the timing parameters;
- deriving both bounds is computationally inexpensive (milliseconds at most).

A DRAM module is used by multiple devices, that send their read and write requests to a controller. The latter arbitrates requests and schedules DRAM commands. The system is pictured in Fig. 6

In order to capture the worst-case behavior, some assumptions need to be made, concerning all the mechanisms that have been devised over the years to improve the average case: therefore it is assumed that no short-circuit between reads and writes occur (i.e., read requests whose response is already in a

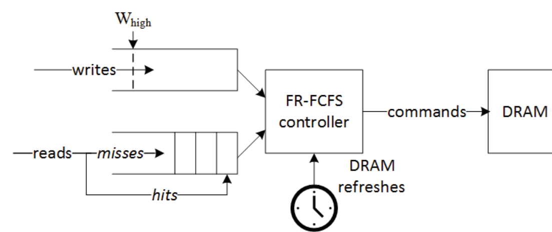

Fig. 6. Model of the FR-FCFS DRAM controller

pending write, hence could be answered without going to the DRAM); it is also assumed that all requests target the same bank, hence the controller must serve them sequentially. The focus is on read requests (instead of writes) since the former are on the critical path for the master requesting them, whereas the latter are not, and can be deferred. A FR-FCFS controller maintains a separate queue for reads and writes, and alternates between serving one queue or the other, also scheduling refresh commands periodically. One must distinguish “row miss” and “row hits” read requests (hereafter misses and hits for short). The former pay a higher time overhead, since a “row open” command has to be issued before being able to serve these requests. For this reason, while misses are scheduled FCFS in the read queue, hits are promoted to the front of the read queue. The above prioritization is limited to a maximum of N_{cap} (this is necessary to avoid starvation of misses). Moreover, switching between the read and write queues incurs a time overhead, hence frequent switching should be avoided. The FR-FCFS controller serves writes in a batch, according to a watermark policy, pictured in Fig. 7. The relevant parameters are the high and low watermark thresholds, W_{high},W_{low}, and the write batch length N_{wd}. When in read mode, the controller switches to serving writes when either of the following conditions holds:

1) The read queue is empty, and there are at least W_{low} write requests in queue;
2) There are at least W_{high} write requests in queue.

When in write mode, the controller switches to serving reads when either of the following conditions holds:

1) The read queue is empty, and write queue is below $max(W_{low} - N_{wd}, 0)$;
2) The read queue is not empty, and N_{wd} writes have been served.

In a worst-case scenario the read queue is never empty, hence both conditions 1 can be neglected without loss of generality. The only relevant parameters are W_{high} and N_{wd}. Last, refresh operations are needed to avoid loss of data at the DRAM. It is assumed that they are scheduled when a refresh timer expires, after the completion of the ongoing read or write request.

The aim is to bound the delay that a read miss experiences, as a function of its position in the read queue. Call t_N the time at which a read miss entering the read queue at the Nth position

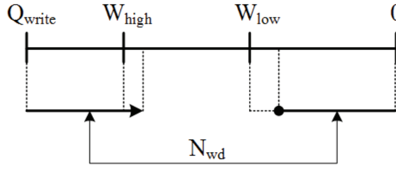

Fig. 7. Watermark policy for read/write switching.

is scheduled. The curve that joins points (t_N, N) is a service curve for this system, hence can be used in a compositional analysis to obtain E2E performance metrics. The alert reader can easily see that the rate at which writes arrive at the system will impact time t_N. If that rate is too high, a periodic pattern of one read miss followed by a batch of N_{wd} writes will soon establish, interrupted only by occasional refreshes. Based on the above pattern, one could clearly establish a crude, pessimistic upper bound with pen and paper. However, this will not be representative of working conditions: masters tend not to exhibit patterns of unlimited writes; physical rate limiters (e.g., token buckets) are often employed at the entrance of a shared network, to avoid congestion; the interconnection network has a finite capacity, hence acts as an implicit rate limiter for memory requests anyway. Thus, knowledge of the write arrival rate at the controller in the computation of the upper and lower bound is integrated. This allows to compute tighter bounds, at the price of complicating the analysis. A general – and enforceable – model for limited arrival rates in NC is the token-bucket shaper, with arbitrary but known parameters burst and rate. The burst parameter b (the vertical offset) models the fact that concurrent requests may arrive near-simultaneously. This happens, e.g., because of different masters sending requests that arrive at the DRAM controller back-to-back, even though each individual master is rate-limited. The rate parameter r (the slope of the line) is the aggregate average rate of the masters that are using the DRAM. The fact that a process $R(t)$ is upper bounded by a token-bucket shaper with a shaping curve $\alpha(\tau) = b + r\tau$, $\tau > 0$, implies that $\forall\tau R(t+\tau) \leq \alpha(\tau) + R(t)$. In other words, the only legitimate processes are those that never intersect the shaping curve. Besides being a useful model for an aggregate traffic process, a token-bucket shaper can be practically implemented in hardware. At a high level, the algorithm consists of the following steps:

1) Compute the time T_N it takes to serve N read misses.
2) Add the time T_H that it takes to schedule N_{cap} read hits back-to-back. This is because the time that it takes to serve a batch of hits is convex with their number, hence scheduling them back-to-back generates the largest delay. Note that this may lead to an unrealistic schedule (hence, an upper bound on the WCD), since there is no guarantee that a gap large enough may exist between two write batches to schedule N_{cap} read hits. Call $T = T_N + T_H$.

TABLE I
DRAM TIMING PARAMETERS (NS)

	DDR3_1600
tCK	1.25
tBurst	5
tRCD	13.75
tCL	13.75
tRP	13.75
tRAS	35
tRRD	6
tXAW	30
tRFC	260
tWR	15
tWTR	7.5
tRTP	7.5
tRTW	2.5
tCS	2.5
tREFI	7800
tXP	6
tXS	270

TABLE II
UPPER AND LOWER BOUNDS ON THE WCD (NS)

Write rate	Lower bound	Upper bound
4 Gbps	1971.711	1977.542
5 Gbps	2957.983	2963.814
6 Gbps	3934.259	3950.086
7 Gbps	5886.811	6908.902

3) Compute the largest number of write batches that can be scheduled within T, and add their time overhead to T;
4) Compute the largest number of refreshes that can be scheduled within T, and add their overhead to T.

Steps 3 and 4 – which only involve trivial algebra – must be iterated until T converges to a stable value. This is because every time that T is increased, new write batches or refreshes may be included, that had not been considered at the previous step. Convergence is reached within few iterations. Once T has converged, assume that the read miss under study (the Nth) is at the end of the schedule, and mark (T, N) as a point in the service curve.

Note that, if the above algorithm computes a feasible schedule, delay T is the WCD (since it is both an upper bound and a lower bound on the WCD itself). Otherwise, it can be complemented via a lower bound that benchmarks it. As a lower bound, one computed using steps 1, 3, and 4 above is used, and scheduling N_{cap} hits as soon as possible, possibly partitioning them among several batches. A bound on the maximum difference between the lower and upper bound can be computed, which is $O(N_{cap})$. The two bounds are in fact quite near.

Table II reports the lower and upper bounds computed assuming a DDR3 DRAM, with parameters taken from a DDR3-1600 4 Gbit datasheet, also reported in Table I. Controller parameters are $W_{high} = 55$, $N_{wd} = 16$, and $N_{cap} = 16$. The write arrival rate varies between 4 and 7 Gbps, assuming a burst of 8.

The results clearly show that the bounding algorithms are

very effective, except when the write rate is very high (last line). Through them, one can compute a service curve for the DRAM technology being used, that can be composed with other guarantees (e.g., a WCD on the transit of the interconnection network) to compute E2E guarantees a priori. Moreover, one can design controllers with appropriate parameter values (e.g., W_{high}, N_{wd}, N_{cap}), so as to meet pre-specified guarantees. The result of the last line shows that the bounding algorithms have room for improvement.

V. Admission Control for Guaranteeing E2E QoS in MPSoCs

Heterogeneous Multi-Processor Systems-on-Chip (MPSoCs) feature a large number of tightly-coupled shared resources. In order to conduct memory accesses, an application must generally acquire several shared (interconnect and memory) resources with independent arbiters and often provided by different vendors. Each shared resource may be further divided into sub-resources (i.e., managed by sub-arbiters). For instance, many modern MPSoCs are equipped with Networks-on-Chips (NoCs) featuring wormhole-switching and multi-stage arbitration (e.g. iSLIP). DRAMs feature as well complex internal hierarchical structure. They are composed of multiple modules which are further structured in a number of banks used to store data. Each bank contains a matrix-like structure where data is located along with a row buffer. The matrix-like structure is not visible to the memory controller and all data exchanges are performed through the corresponding row buffer.

Conventional network and memory resources do not take into account interference between different threads/applications when making scheduling decisions and resources are not reserved in advance. Each router conducts its arbitration locally, i.e. packets are switched as soon as they arrive and ongoing transmissions compete for link bandwidth and buffer space, independently of other routers. Memory accesses are translated by the memory controller into internal DRAM commands used to access data and read/write from row buffers. COTS memory controllers are optimized for the average-case performance and because of this they rely on the open-row policy. A FR-FCFS scheduling policy, as described in the previous section, is often used to prioritize memory requests accessing the same memory region (i.e. the same row) over other requests, to maximize row-hit rate, and thereby performance.

The granularity of application requests is therefore often different from the shared resources' granularity of arbitration. While applications issue data transmissions (cache lines or DMA), routers arbitrate data flits and packets, and memory controllers schedule internal DRAM commands. An application data transmission is decomposed into a number of flits or packets and internal DRAM commands. This results in a complex spectrum of direct and indirect interference between data streams, which may jeopardize predictability and endanger system safety.

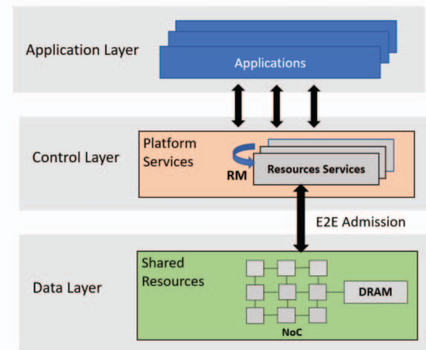

Fig. 8. A logical view of E2E admission control considering different resources services (i.e. regulation rates) configured by the resource manager (RM) for shared resources.

Approaches like traffic shaping and memory throttling using rate control are well-known methods to support QoS. However, applying rate control in an E2E fashion in the presence of multiple heterogeneous resources operating at different granularity requires fine-grained synchronization and proper configuration of individual shapers to meet the E2E QoS requirements dictated by a given application. This is even more complex if dynamic adaptation is considered, to comply with changes in the state of the system such as the number of applications or changes in their requirements. Hence, there is a need for abstractions to map QoS requirements from applications to resources, and to orchestrate the configuration of regulation parameters for provided resources services.

Admission control can be used as an alternative method to provide applications with a global resource arbitration. It allows one to decouple the data layer, where transmission is performed, from the control layer, which is responsible for allocation and arbitration of available resources, see Fig 8. The idea of admission control is not new, it is often used in the IT domain in combination with Software-Defined Networking (SDN) to implement routing processes that are more dynamic and efficient than physical ones implemented in network switches [3]. It has been used in different existing works to provide real-time capabilities and facilitate adaptation and reconfiguration [15].

In [16], admission control was applied to provide real-time guarantees for (mixed) critical communication and memory traffic in MPSoC. The proposed approach provides an overlay network built on top of existing NoC architectures. Whenever an applications is granted admission, E2E access allocation of a sequence of shared network and memory resources is achieved. This control layer has a global view of current traffic in the network and can dynamically adapt the rate control at which running applications can access shared resources to the state of the system [17].

In order to support admission control, standard NoC archi-

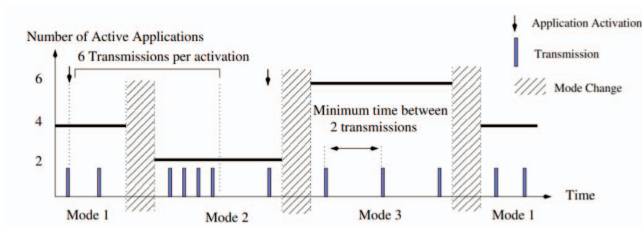

Fig. 9. Adaptive resource services defined by the RM as traffic injection rates according to the system mode [17].

tectures are extended by introducing local supervisors, called clients, at each node. The role of clients is to prevent non-authorized accesses, adjust the access rates to the NoC for each application, release the NoC resources (inform the RM whenever an application terminates), and prevent unbounded NoC accesses. Clients can be implemented fully in software or as an independent hardware module controlling accesses before they arrive to the network interface, to allow the integration with existing commercially available components.

At each source node, a monitor regulates the rate at which the source can inject traffic in the NoC. This regulation is performed dynamically (at run-time) according to the system load i.e. the number of simultaneously active applications using a special scheduling unit called Resource Manager (RM). The RM possesses knowledge of the global state of the NoC (i.e., which application is active) and which resources are occupied. Using this information, the RM may decrease or increase the injection rates for a particular node, as depicted in Fig 9, depending on the current system mode. Each mode is defined by the number of currently active applications, and determines the minimum time separating every two transmissions issued from the same application. The mechanism is capable of enforcing symmetric guarantees, where transmission rates decrease uniformly for all applications when the number of senders running in parallel (system mode) increases. Non-symmetric guarantees where transmission rates depend not only on the current mode but also on the application's importance can also be enforced. The non-symmetric mode can be used in a mixed-criticality system to maintain the critical application guarantees while reducing best effort traffic (e.g., by allowing higher rates for critical applications).

Dynamic rate regulation is performed using a protocol-based access layer implemented within the existing NoC architecture. The protocol consists of four control messages: activation (actMsg), termination (terMsg), stop (stopMsg) and configuration (confMsg). The RM must be informed about the activation and termination of each application. Therefore, whenever an application is activated and trying to conduct the first transmission its request is trapped by the client. It remains blocked until acknowledged by the RM with a confMsg. Later, the corresponding client sends an activation message to the RM. Similarly, when a client detects the termination of an application it issues a terMsg message to the RM. The activation and termination messages are processed by the RM in order of arrival. Each of them initiates the transition of the system to a different mode. Before changing the rates, the RM sends to the clients supervising active applications a stop message (stopMsg) to block all accesses to the NoC from the corresponding node. Clients then wait for the confMsg communicating the current system mode. After receiving the confMsg, clients adjust the rate and unblock transmissions. Note that a trade-off analysis is required at design time to determine the overhead of the synchronization protocol and the frequency at which mode changes can be performed to support dynamics.

Providing E2E guarantees across computation and communication resources often requires complex analysis approaches, such as compositional performance analysis [18], [19] for the worst-case E2E timing behavior. By decoupling the data layer where transmission is performed from the control layer responsible for allocation and arbitration of available resources, data transfers are established and scheduled at a higher logical level before applications acquire access to physical shared resources. Arbitration between multiple applications is then shifted from individual (sub) resources to a centralized control unit which has a global view of the system (i.e. both applications and resources). This allows to simplify analytical timing analysis models used to bound interference effects and compute timing guarantees on the the E2E latency of individual transmissions. Bounding the timing effects of shared resources requires a careful analysis of requests arrival (that determine interference) at every resource and its corresponding scheduling/arbitration policy. With admission control, interference analysis can account for applications requests arrival at the centralized control unit instead of individual flits/packets/commands arrival at every (sub) resource. This, in turn, reduces the complexity of coupling different resources timing analysis which usually leads to pessimistic formal guarantees or decreased performance and utilization.

VI. Conclusion

In this paper we discussed current efforts in the automotive industry to use high-performance hardware platforms for mixed-criticality and time-critical applications in high-integration scenarios. We argued that, due to shortcomings of available platforms in the market, software mechanisms are currently the only way to retroactively equip them with required predictable performance. Furthermore, we presented upcoming Arm technologies, namely DynamIQ and MPAM, that when used in future IPs will greatly contribute to overcome the explained disadvantages of purely software-based measures. Due to these developments, we are confident that a close cooperation among system suppliers, IP providers, semiconductor companies, and OS/hypervisor vendors will enable future

automotive HW/SW platforms that combine high-performance with predictability.

In addition, research is needed to further enhance the currently envisioned initial solutions. While it is desirable to have formal analyses for configuring system components and give formal guarantees at design-time, the lack of open specifications and the complexity of industrial-grade components often lead to overly pessimistic analytic bounds which prevent the wide-spread use of formal analysis. We showcased that for individual components, such as a FR-FCFS DRAM controller, it is in principle possible to derive tight performance bounds. However, as interacting heterogeneous components are considered, E2E formal analysis is highly complex and hardly feasible. Approaches such as admission control mechanisms can allow to simplify the system view on tightly-coupled shared hardware resources and simplify formal performance analysis.

References

[1] S. Saidi, S. Steinhorst, A. Hamann, D. Ziegenbein, and M. Wolf. Special session: Future automotive systems design: Research challenges and opportunities. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–7, 2018.

[2] R. Cavicchioli, N. Capodieci, and M. Bertogna. Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms. In *22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–10, 2017.

[3] J. Leguay, L. Maggi, M. Draief, S. Paris, and S. Chouvardas. Admission control with online algorithms in SDN. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, pages 718–721, 2016.

[4] A. Hamann, S. Saidi, D. Ginthör, C. Wietfeld, and D. Ziegenbein. Building End-to-End IoT Applications with QoS Guarantees. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020.

[5] Y. Ye, R. West, Z. Cheng, and Y. Li. COLORIS: A dynamic cache partitioning system using page coloring. In *23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 381–392, 2014.

[6] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013.

[7] *Arm DynamIQ Shared Unit Technical Reference Manual*, October 2019. Version r4p1. https://developer.arm.com/documentation/100453/0401.

[8] *Arm Cortex-A78 Core Technical Reference Manual*, May 2020. Version r1p1. https://developer.arm.com/documentation/101430/0101.

[9] *Arm Cortex-A76AE Core Technical Reference Manual*, October 2018. Version r0p0. https://developer.arm.com/documentation/101392/0000.

[10] *Arm Cortex-A65 Core Technical Reference Manual*, February 2019. Version r1p1. https://developer.arm.com/documentation/100439/0101.

[11] *Arm Architecture Reference Manual Supplement Memory System Resource Partitioning and Monitoring (MPAM) for Armv8-A*, July 2020. Version B.b. https://developer.arm.com/documentation/ddi0598/bb/.

[12] *Arm System Memory Management Unit Architecture Specification, SMMU architecture version 3*, August 2020. Version D.a. https://developer.arm.com/documentation/ihi0070/da.

[13] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, volume 2050 of *Lecture Notes in Computer Science*. Springer, 2001.

[14] Matteo Andreozzi, Frances Conboy, Giovanni Stea, and Raffaele Zippo. Heterogeneous systems modelling with adaptive traffic profiles and its application to worst-case analysis of a DRAM controller. In *44th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2020, Madrid, Spain, July 13-17, 2020*, pages 79–86. IEEE, 2020.

[15] Guy Durrieu, Gerhard Fohler, Gautam Gala, Sylvain Girbal, Daniel Gracia Pérez, Eric Noulard, Claire Pagetti, and Simara Pérez. DREAMS about reconfiguration and adaptation in avionics. In *ERTS 2016*, Toulouse, France, January 2016.

[16] A. Kostrzewa, S. Saidi, L. Ecco, and R. Ernst. Dynamic admission control for real-time networks-on-chips. In *21st Asia and South Pacific Design Automation Conference, ASP-DAC 2016*, pages 719–724. IEEE, 2016.

[17] Adam Kostrzewa, Sebastian Tobuschat, Rolf Ernst, and Selma Saidi. Safe and dynamic traffic rate control for networks-on-chips. In *Tenth IEEE/ACM International Symposium on Networks-on-Chip, NOCS 2016, Nara, Japan, August 31 - September 2, 2016*, pages 1–8. IEEE, 2016.

[18] Robin Hofmann, Leonie Ahrendts, and Rolf Ernst. CPA: compositional performance analysis. In Soonhoi Ha and Jürgen Teich, editors, *Handbook of Hardware/Software Codesign*, pages 721–751. Springer, 2017.

[19] Arvind Easwaran and Insup Lee. Compositional schedulability analysis for cyber-physical systems. *SIGBED Rev.*, 5(1):6, 2008.