A Real-Time *virtio*-based Framework for Predictable Inter-VM Communication

Gero Schwäricke* Rohan Tabish† Rodolfo Pellizzoni§ Renato Mancuso‡
Andrea Bastoni* Alexander Zuepke* Marco Caccamo*
* Technical University of Munich, {name.surname}@tum.de
† University of Illinois at Urbana-Champaign, rtabish@illinois.edu
§ University of Waterloo, rpellizz@uwaterloo.edu
‡ Boston University, rmancuso@bu.edu

***Abstract*—Ensuring real-time properties on current heterogeneous multiprocessor systems on a chip is a challenging task. Furthermore, online artificial intelligent applications –which are routinely deployed on such chips– pose increasing pressure on the memory subsystem that becomes a source of unpredictability. Although techniques have been proposed to restore independent access to memory for concurrently executing virtual machines (VM), providing predictable inter-VM communication remains challenging. In this work, we tackle the problem of predictably transferring data between virtual machines and virtualized hardware resources on multiprocessor systems on chips under consideration of memory interference. We design a "broker-based" real-time communication framework for otherwise isolated virtual machines, provide a virtio-based reference implementation on top of the Jailhouse hypervisor, assess its overheads for FreeRTOS virtual machines, and formally analyze its communication flow schedulability under consideration of the implementation overheads. Furthermore, we define a methodology to assess the maximum DRAM memory saturation empirically, evaluate the framework's performance and compare it with the theoretical schedulability.**

I. Introduction

Nowadays, artificial intelligence (AI) algorithms are widely used at *run-time* in complex embedded cyber-physical domains (*e.g.*, automotive, avionics, and industrial automation) that require real-time assurance. Due to the need to reduce size, weight, and power, modern embedded systems on a chip (SoC) that can sustain the heavy data requirements of AI feature not only multicore architectures but also integrate special-purpose accelerators such as FPGAs and GPUs. Examples of such complex heterogeneous multiprocessor systems on a chip (MPSoC) are the Xilinx Ultrascale+ and Versal or the NVIDIA Jetson Xavier [39], [55], [57]. Ensuring the real-time properties of these complex systems is challenging. To make things worse, the need to transfer large amounts of data among cores and accelerators to meet the requirements of AI applications causes the memory hierarchy –shared caches, interconnect, and DRAM– to become a source of unpredictability.

In order to enable a practical real-time analysis of such systems, previous work [31], [33], [58], [59] has focused on *isolating* the access to the memory hierarchy for independent execution contexts (partitions or virtual machines (VM)). A significant advantage of these techniques is that they integrate easily into hypervisors (*e.g.*, [31], [56]) and enable the isolated execution of unmodified VMs. By design, though, the strength of these approaches (*i.e.*, ensuring isolation) is also their major drawback when it comes to establishing communication channels –which, by definition, violate isolation– between VMs.

In this paper, we propose an architecture that enables predictable communication for VMs on heterogeneous MPSoCs under *explicit* consideration of the interference caused by memory communication. We further analyze and evaluate the memory interplay between cores and a DMA engine concurrently accessing the interconnect. DMA engines (widely available on MPSoCs) are specifically designed to perform efficient data copying and thus are the natural accelerator-choice to perform bulk data transfers. Still, it is unclear what their impact is on the timeliness of transfers under memory interference. As basic building blocks, we rely on established mechanisms (cache-coloring, *MemGuard*, hardware-enforced QoS [7], [33], [49], [59]) that have proved to perform well to guarantee isolation, but we: a) extend their scope to isolate independent VM communication *flows* and empirically determine the maximum sustainable DRAM memory bandwidth; b) make them available as a virtio-based, transparent and predictable communication framework; c) provide an analysis of the scheduling of different communication flows under consideration of the maximum sustainable bandwidth and framework overheads, and provide guidelines for system-designers on how to dimension the DMA regulation to both maximize CPU bandwidth and ensure schedulability.

Given the widespread adoption of hypervisors (*e.g.*, [4], [24], [44]) as industry standard to partition complex software systems, our real-time communication framework enables predictable communication between (unmodified) VMs hosted on top of a hypervisor (in our proof-of-concept, *Jailhouse* [4]). The interface between our framework and the unmodified VMs is implemented using the *virtio* standard [1], to ensure the widest portability. Using a standardized virtual device interface at the hypervisor level, the system can be composed of different operating systems, each satisfying the requirements of its applications. By exploiting the flexibility of the virtio approach, our framework provides universal but predictable communication between OSs of different complexity and criticality, for example RTOSs and baremetal algorithms

for hard real-time applications and Linux for soft real-time applications.

Our approach is inspired by the *split driver model* provided by Xen [54] and used in the high-performance domain to multiplex accesses from different VMs to a single hardware device. Under this model, a device (*i.e.*, its hardware resources, including the interfaces –mapped registers, PCI aperture, etc.– to access it) is assigned by the hypervisor configuration to a single VM (normally with special privileges), which then acts as multiplexer (or *broker*) for other VMs that require access to it. The other VMs do not see the physical device but communicate via a virtualized device –emulated at hypervisor level– first with the broker, which in turn schedules the communication flows on the physical device.

Our architecture adopts a similar broker-based approach with the following advantages:

- Asynchronous communication: Once the data to be transferred has been taken in charge by the broker, a VM could continue its execution in parallel to the data transfer (assuming there are no data dependencies).
- Straightforward DMA integration: Since modern MP-SoCs feature at least one DMA, the broker can program the DMA to take care of the copy operations. Since the broker is the only VM entitled to program the DMA, access control to this resource is trivially solved. Furthermore, compared to CPU-based data copies, using DMA engines in a real environment makes the broker's CPU available to perform other computations.
- Centralized scheduling: being in charge of both DMA and shared hardware devices, the broker can centralize scheduling decisions regarding, *e.g.*, priorities of communication flows.

By design, our architecture does not allow direct shared-memory communication between VMs, which we assume to have different criticality. The broker is the only (privileged) VM that mediates communication among different criticality levels. This type of architecture cleanly separates the configuration and resource allocation concerns for partitioned systems (*e.g.*, avionics or automotive ones) and easily integrates onto systems that already support communication concepts such as ARINC's *queuing ports* [5]. Strictly partitioning shared-memory areas and delegating the control over communication to a privileged entity simplifies the certification activities that are required by, *e.g.*, DO-178 C or ISO 26262 standards [27], [46]. In fact, communication among entities with different criticality levels is traditionally subject to careful planning and close scrutiny by certification authorities [47] since this represents an intentional violation of partitioning. Our framework aims to simplify these certification activities by enabling the separation of different independent communication flows.

To summarize, in this paper, we: 1) propose the design of a virtio-based real-time communication framework to enact predictable communication between isolated VMs under consideration of memory interference; 2) contribute a real-time analysis of limited-preemption EDF scheduling of the communication flows under consideration of the measured framework overheads; 3) provide guidelines to system designers on the dimensioning of the system regulation to achieve maximum bandwidth, while preserving the I/O flow schedulability; 4) propose a reference implementation of the framework on the Jailhouse hypervisor [4] using FreeRTOS virtual machines and contribute a methodology to empirically assess the maximum DRAM memory saturation under non-additive memory regulation; and 5) empirically evaluate our solution and compare it against the theoretical analysis.

We introduce the system model in Section II. Section III presents the architecture and technical details on the implementation. Section IV contains the schedulability analysis and summarizes guidelines to dimension the minimum DMA bandwidth. Section V shows how to measure DRAM saturation and our experimental results. We present related work in Section VI, and we conclude in Section VII.

II. System Model

We consider a multi-core platform with m general-purpose application CPUs (or *cores*), identified as CPU_k, $k \in \{1, \ldots, m\}$. Each CPU includes a private level 1 (L1) cache (separated for data and instructions); all the CPUs share a level 2 (L2) cache, which is also the last-level cache (LLC). The caches implement a hardware coherence protocol. The LLC uses a write-back, write-allocate policy for normal cacheable memory. To be as generic as possible, we assume that the LLC line replacement policy is pseudo-random. Memory transactions that miss in LLC cause accesses to DRAM. A cache refill causes a read transaction of size S_l bytes; a dirty line write-back triggers a write transaction of S_l bytes. We assume that the mapping between physical addresses and cache sets is known, *i.e.*, we assume that the cache controller performs no index bits hashing [34].

The main memory subsystem comprises a single DRAM controller in a multi-bank configuration. We assume that bank interleaving is disabled. This way, the bits in the physical addresses (PA) that encode for DRAM banks (bank bits) are more significant than those encoding for DRAM rows (row bits) and columns (column bits).

Our architecture employs a partitioning hypervisor to define a set of isolated partitions. Each partition has a set of statically allocated hardware resources so that the guest OS operating within the partition is only capable of using/accessing a subset of the system resources. We refer to the guest OS and its partitioned hardware resources with the term *virtual machine* (VM). The partitioning hypervisor performs no virtual-CPU scheduling. Although a VM could have multiple CPUs assigned, without loss of generality, in this paper we consider the case that a given VM_k is statically allocated one CPU_k.[1] Furthermore, each VM (and hence CPU) has a statically assigned private slice of DRAM and a private LLC partition using page coloring [31], [33].

We consider sporadic unicast VM-to-VM communication facilitated by a *broker* that is responsible for organizing

[1]The CPU-dependent memory-regulation consideration can be applied independently to each CPU.

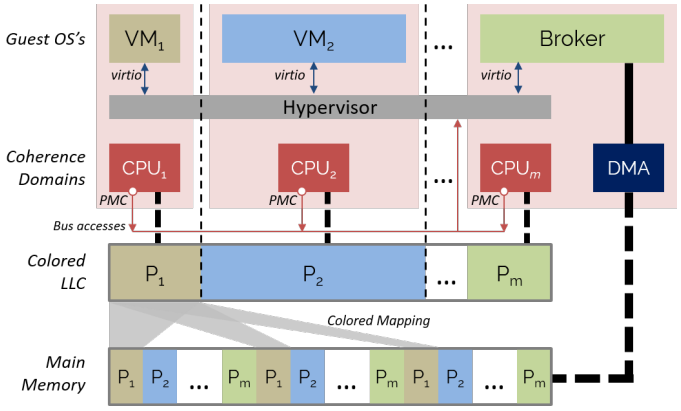


Fig. 1. Proposed system architecture for a system with m processor cores. VM-to-VM communication is mediated by a Broker VM that coordinates DMA transfers. VMs memory is partitioned using cache-coloring.

the data transfers. An overview of the system is displayed in Figure 1. To carry out any memory transfer, the broker exclusively manages a Direct Memory Access (DMA) engine. To achieve more flexibility with respect to configuration and application deployment, in our prototype (Section III), the broker is exclusively allocated to one of the main CPUs of the system, but it could equivalently operate on a specialized CPU with dedicated local scratchpad memory. Such CPUs are commonly available on many MPSoCs (*e.g.*, [57]). In our prototype, the logic of the broker is implemented as bare-metal firmware with minimal memory footprint. We assume that the DRAM traffic generated by the broker's logic is negligible (*i.e.*, the broker's code and data fully fit into the private L1 caches), and we focus on the traffic generated by the broker-controlled DMA engine.[2]

A. Memory Regulation Model

The partitioning hypervisor is responsible for DRAM bandwidth partitioning. Specifically, memory traffic originated by the CPUs as a result of LLC line refills (reads) and write-backs (writes) is regulated with a technique similar to MemGuard [33]. Conversely, the memory bandwidth of the broker's DMA is regulated via hardware QoS extensions, such as the ARM QoS subsystem [7] considered in, *e.g.*, [49].

The generic MemGuard-regulated CPU_k is assigned a quota of Q_k cache refills per regulation period P. Q_k is an absolute number, typically related to architecturally available hardware performance counters (PMC), while P is expressed in seconds. We indicate the upper-bound on the memory bandwidth that can be consumed by CPU_k as $b_k^{\mathbf{c}} = \langle b_k^{\mathbf{c},r}, b_k^{\mathbf{c},w} \rangle$, where $b_k^{\mathbf{c},r}$ (resp., $b_k^{\mathbf{c},w}$) is the read (resp., write) DRAM bandwidth expressed in bytes/second. In the worst-case, any cache refill causes two DRAM transactions – a cache refill plus a write-back. Thus, we make the simplifying assumption that $b_k^{\mathbf{c},r} = b_k^{\mathbf{c},w}$. Given Q_k, it is always possible to compute $b_k^{\mathbf{c},r} = b_k^{\mathbf{c},w} = (Q_k \cdot S_l)/P$.

The DMA used by the broker to perform VM-to-VM data transfers is assigned a QoS level $\mathtt{a}^{\mathbf{b}}$. In this case, the parameter expresses the rate at which read/write transactions are issued by the DMA towards the DRAM. We assume the same level of regulation for both read and write transactions and symmetric communication transfers between source and destination. Therefore, the corresponding data bandwidth achieved by the DMA is $b^{\mathbf{b}} = b^{\mathbf{b},r} = b^{\mathbf{b},w}$.

When using ARM QoS extensions, the $\mathtt{a}^{\mathbf{b}}$ parameter is an integer $\in [1, 2^{12}]$ that determines the transactions/second issuance rate = $(\mathtt{a}^{\mathbf{b}} \cdot f_{clk})/2^{12}$ [49] for the DMA, where f_{clk} is the clock frequency of the interconnect. Thus, one can compute the bandwidth in bytes/second

$$b^{\mathbf{b}} = (S_b \cdot \mathtt{a}^{\mathbf{b}} \cdot f_{clk})/2^{12}, \tag{1}$$

where S_b is the size in bytes of the read/write memory transactions generated by the DMA (block size). The actual bandwidth of the DMA is subject to contention due to memory interference by other bus masters and delays due to the DMA's internal management. We determine the DMA bandwidth limiting under given interference in Section V-A.

We assume that the broker pays a non-negligible overhead O_{dma} to program the DMA engine for a new transfer. As such, it is convenient to program the DMA to transfer (large) chunks of fixed size S_c expressed in bytes. S_c must be a multiple of S_b. The time it takes for the DMA to complete a single transfer of size S_c can then be computed as $O_{dma} + S_c/b^{\mathbf{b}}$.

A key property we rely on is that applications deployed on the CPUs as well as transfers performed by the DMA engine incur negligible contention delay when accessing the main memory as long as the DRAM subsystem always operates below 100% utilization (saturation threshold) [49]. To ensure that this is respected by design, we assume that it is possible to model the impact of bandwidth assignment to cores and the broker's DMA on the DRAM utilization. Specifically, the function $U^{\mathbf{c}}(b_k^{\mathbf{c}})$ is used to compute the worst-case DRAM utilization that results from assigning $b_k^{\mathbf{c}}$ to CPU_k. Similarly, $U^{\mathbf{b}}(b^{\mathbf{b}})$ captures the worst-case DRAM utilization caused by the broker's DMA regulated at $b^{\mathbf{b}}$. Thus, it must hold that:

$$U^{\mathbf{b}}(b^{\mathbf{b}}) + \sum_{k=1}^{m} U^{\mathbf{c}}(b_k^{\mathbf{c}}) \leq 1. \tag{2}$$

B. Communication Infrastructure & Model

We consider a set Γ of n directed, periodic communication *flows* where for each flow, a *sender VM* and a *receiver VM* are uniquely identified. Communication channels between senders and receivers are statically configured and their characteristics, *i.e.*, periodicity and required bandwidth per flow, are known at run-time. Since we focus on inter-VM communication, we assume that the sender and receiver VM are distinct. Each flow $\tau_i \in \Gamma, i \in \{1, \ldots, n\}$ comprises a potentially infinite sequence of packets, and it is defined as a tuple of the form $\langle C_i, D_i, P_i, s_i, r_i \rangle$, where C_i is the data size of its packets, P_i

[2]Note that, although beneficial, a DMA engine is not strictly required by the proposed architecture. In the absence of DMA, the broker VM could directly perform the data transfer to the destination VM. In future work, we plan to extend the analysis to also include this use case.

is its period –or minimal inter-arrival time for sporadic flows– and D_i represents its (arbitrary) relative deadline. Finally, s_i (resp., r_i) represents the index of the sender (resp., receiver) VM and corresponding CPU.

III. Architecture

Figure 1 presents our system architecture. A hypervisor instantiates a specialized VM for communication management (*broker* VM) and up to $m-1$ real-time constrained application VMs (each associated with one core). The hypervisor uses cache coloring [31] to provide dedicated cache partitions to each VM (including the broker VM) and regulates the bandwidth of the VMs using MemGuard [59]. It also configures the DMA engine's bandwidth regulation and memory mapping. The system features an IO-MMU (*e.g.*, [8]) that allows the DMA to use the same address translation as the cores.

A. Virtio Interface

In order to achieve maximum portability across different virtual machines and guest OSs, the communication model as specified in Section II utilizes the widely-used *virtio* specification [1] as the interface for data exchange. The guest operating system can use regular virtio-compliant drivers. The hypervisor exposes a virtualized device (virtio device) to the guest OS and provides a transparent relay of communication requests to the broker VM. Virtio provides several types of virtual devices (*e.g.*, block or network devices).

The virtio specification follows a classic driver model for DMA-capable devices: The driver allocates data buffers and provides them to the (virtual) device in advance for incoming traffic and on-demand for outgoing traffic. Data buffers are organized in a structure called *virtqueue*, which is based on a buffer descriptor table, and two ring buffers for buffer exchange between driver and device. Virtio devices utilize multiple virtqueues for different purposes. A basic device setup uses one RX virtqueue for incoming traffic and one TX virtqueue for outgoing traffic. In addition to the device types, the virtio specification defines their virtqueues, and the format of data exchange with them. Furthermore, different transport options for virtqueues such as MMIO and PCI-based device virtualization are specified.

The event notification between drivers and devices relies on OS/hypervisor-provided primitives. Specifically, a virtio device can notify a driver using interrupts. A driver can notify a device using a synchronous exception by performing an access to a specific address that is mapped with restricted access permissions for the VM.

Our approach uses the interface for socket-type devices, which avoids the overheads of simulating a full communication device (*e.g.*, networking card) by utilizing the inherent ring buffers of the virtio specification for direct packet exchange between the guest and the host (hypervisor). We use one virtqueue for incoming (RX) and one virtqueue for outgoing (TX) data transfers. The socket interface also defines a third type of virtqueue for event data that is not required by our approach and therefore unused in our architecture.

Virtio socket devices transfer data in packets, which consist of a fixed-size header and a data payload of dynamic size. However, since the receiver allocated buffers for incoming traffic in advance, the buffers' (pre-allocated) size can be much smaller than the data payload buffer. This has to be accommodated in the communication infrastructure by splitting the payload and copying it into multiple receiver buffers. As a simplification, we split packets with large payloads into multiple data transfers.

B. Data Flow Scheduling

Our approach employs a trusted communication broker VM for predictable data traffic between otherwise isolated environments. As discussed in Section II, this is achieved by scheduling and rate-limiting the data transfers of statically defined communication flows between pairs of VMs. The broker receives all packet transfer requests and schedules the packets according to EDF using the fixed granularity S_c. The broker then programs the DMA to perform data copies of the same S_c size. The packets are inserted into a queue sorted by increasing absolute deadlines. To reduce the processing overhead on the broker VM, we use a separate queue per guest and let the driver on the sender core insert the packet from hypervisor mode. Due to this optimization, we create a separate cache partition to isolate data that needs to be shared between broker and hypervisor from interference. This partition contains information about the communication channels, the sorted packet queues of all guests, lock objects, and ring buffers for the return of used packet information objects.

Figure 2 shows an example packet transfer from VM_1 to VM_2. When a guest application in VM_1 decides to send a packet ①, its virtio socket driver prepares the packet header and payload in a buffer in the TX virtqueue. VM_1 then ② sends a notification to the virtio socket device. The device, which operates in hypervisor mode, ③ determines the channel of the pending packet and inserts the packet information into the per-VM transfer queue. In detail, the device computes the absolute deadline for the packet as the current time plus the modified relative deadline D'_i for the packet (see Section IV for how D'_i is derived) and inserts the packet into the queue, sorting by the absolute deadline. Then ④ it notifies the broker VM. The broker proceeds to ⑤ schedule all transfers pending from any guest (see also the analysis in Section IV). When a packet is scheduled for transfer, the DMA engine is programmed to ⑥ copy the payload data from the TX buffer of the sender to an empty RX buffer of the receiver. After completion ⑦, the DMA notifies the broker, which in turn ⑧ informs the sender and the receiver VM of the completed transfer. The guest OSs will then ⑨ free the no longer required TX buffer (VM_1) and process the received packet payload (VM_2). Lastly, VM_2 will add a new (or the received) RX buffer to its RX virtqueue to restore the full capacity of the virtqueue for incoming traffic.

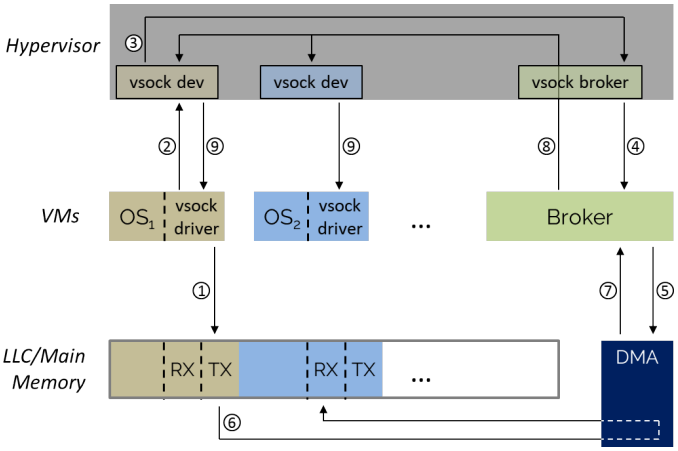


Fig. 2. Communication flow in the virtio backend for an example packet transfer from VM_1 to VM_2.

C. Implementation

We have validated our architecture by implementing it on a Xilinx ZynqMP ZCU102 embedded MPSoC [57]. The platform features four ARM Cortex-A53 cores sharing a last level L2 cache of 1 MiB size. Each core has 32 KiB of private L1 instruction and data caches. The cores provide cache-refill performance counters needed for MemGuard. Our implementation employs the coherent DMA provided by the ZCU102 and limits its transfer rate using the interconnect's hardware QoS regulators [7].

We use the partitioning hypervisor Jailhouse [4]. By courtesy of earlier works [31], [49] that made their implementations publicly available, the Jailhouse version used in this work integrates cache coloring, MemGuard, and QoS bandwidth regulation. We implemented the virtio PCI transport layer based on Jailhouse's virtualized PCI root controller. The virtio PCI transport layer enables automatic discovery of our virtio socket devices by the OS. The socket device is implemented in the hypervisor and integrates with the broker VM for data exchange and signaling.

The communication broker is implemented as a baremetal application with minimal footprint, and its code and data sections fit entirely in the core's L1 caches. For all code and data of the broker as well as all shared data between hypervisor and broker, we use a dedicated cache partition in the L2 cache to reduce cache interference with the other VMs. Packet notifications are implemented using chip-wide broadcast events (`SEV/WFE`) to eliminate IPI delays and IRQ processing overheads. The overhead for notification ④ is therefore included as a small constant overhead in the queue insertion ③ and the queue searching ⑤.

We have integrated our architecture with FreeRTOS [22] guest instances[3] where we implemented a virtio socket driver and the PCI transport layer. We note that Linux provides a standard virtio socket driver. However, since the Linux kernel enforces extra data copies for secure communication between kernel and user-space, we believe the Linux implementation might introduce considerable overheads. We plan to integrate the Linux driver and evaluate its performance in future work.

Our implementation of the memory bandwidth regulation currently does not take the (comparatively few) memory accesses performed by the hypervisor and by the address translation units into account. Furthermore, the hypervisor address space is not colored, and memory accesses by the hypervisor can result in the eviction of cache lines.

[3]The implementation is available at https://github.com/gschwaer/rt-virtio.

IV. Analysis

In this section, we discuss how to assess schedulability for a flow set Γ assuming a DMA bandwidth $b^{\mathbf{b}}$. Following the architecture described in Section III, we assume that a packet of flow τ_i arrives when the guest application sends it and completes when the receiver VM is notified of the transfer; hence, we say that τ_i is schedulable if the difference between completion and arrival time for any packet of τ_i is no larger than its relative deadline D_i.

We show how Γ can be transformed into a set of sporadic tasks Γ' whose feasibility can be tested using standard analysis techniques for uniprocessor scheduling, such that Γ' being schedulable implies that Γ is also schedulable. Specifically, we will transform each flow τ_i into a limited preemption [9], [12], sporadic task $\tau'_i = \langle C'_i(b^{\mathbf{b}}), D'_i, P'_i, J'_i, q'_i(b^{\mathbf{b}})\rangle$, where $C'_i(b^{\mathbf{b}})$ is the execution time of the task, P'_i its minimum interarrival time, D'_i its relative deadline, J'_i its release jitter, and $q'_i(b^{\mathbf{b}})$ the maximum time that the task executes in non-preemptive mode.

Let $a_{i,j}$ denote the arrival time of the j-th packet of flow τ_i; by definition in our model, it holds that $a_{i,j+1} \geq a_{i,j} + P_i$. As discussed in Section III-B, the absolute deadline of the packet is computed by the virtio device. The arrival time $a'_{i,j}$ of the j-th job of the transformed task τ'_i corresponds to the instant at which the absolute deadline is computed. Let $O_s^{\min}, O_s^{\max}$ denote the minimum and maximum time, respectively, between the guest application sending the packet and the virtio device computing its deadline. This implies that for any packet/job, $a'_{i,j} \geq a_{i,j} + O_s^{\min}$ and $a'_{i,j} \leq a_{i,j} + O_s^{\max}$; and from $a_{i,j+1} \geq a_{i,j} + P_i$ we derive: $a'_{i,j+1} - O_s^{\min} \geq a'_{i,j} - O_s^{\max} + P_i$, which means that τ'_i is still sporadic but with a reduced period $P'_i = P_i + O_s^{\min} - O_s^{\max}$. As we will show in Section V-B, in practice the difference $O_s^{\max} - O_s^{\min}$ is small (less than one microsecond in our implementation).

Note that, even if the absolute deadline of the packet is computed at time $a'_{i,j}$, the packet does not immediately become eligible for scheduling at the broker; this is because the device needs to insert the packet in the corresponding queue and inform the broker. We model such behavior by introducing a release jitter J'_i for the transformed task, where J'_i represents the maximum time required for queue insertion and notification.

As discussed in Section III-A, each packet of τ_i is broken into a set of $\lceil C_i/S_c \rceil$ transfers: specifically, $\lceil C_i/S_c \rceil - 1$ transfers of length S_c, and one of length $C_i^{\text{last}} = C_i - \left(\lceil C_i/S_c \rceil - 1\right) \cdot S_c$. Each of the former has a transfer time $O_{dma} + S_c/b^{\mathbf{b}}$; note that O_{dma} includes the time required to

make a scheduling decision, the time to program the DMA engine before the transfer, the time for processing the DMA notification once the transfer is completed, and the time it takes to trigger a notification of a partial transfer to the receiver VM. The last transfer suffers an additional overhead $O_{pckt,i}$ to remove the packet from its sender's queue, resulting in a transfer time $O_{dma} + C_i^{\text{last}}/b^{\mathbf{b}} + O_{pckt,i}$. We show how to determine the values of such overheads in details in Section V-B. Furthermore, the broker cannot make any scheduling decision while a transfer is ongoing. We thus model the scheduling of data transfers by associating each task τ_i' with an execution time:

$$\begin{aligned} C_i'(b^{\mathbf{b}}) &= (\lceil C_i/S_c \rceil - 1) \cdot (O_{dma} + S_c/b^{\mathbf{b}}) + \\ &\quad O_{dma} + C_i^{\text{last}}/b^{\mathbf{b}} + O_{pckt,i} = \\ &= \lceil C_i/S_c \rceil \cdot O_{dma} + C_i/b^{\mathbf{b}} + O_{pckt,i}, \end{aligned} \tag{3}$$

obtained by summing the transfer times of all packets, and a non-preemptive time equal to the maximum transfer time of any packet:

$$q_i'(b^{\mathbf{b}}) = O_{dma} + \begin{cases} C_i^{\text{last}}/b^{\mathbf{b}} + O_{pckt,i} & \text{if } C_i \leq S_c; \\ \max\left(S_c/b^{\mathbf{b}}, C_i^{\text{last}}/b^{\mathbf{b}} + O_{pckt,i}\right) & \text{if } C_i > S_c. \end{cases} \tag{4}$$

Finally, let O_r to denote the time required by the receiver VM to obtain such notification after it is triggered by the broker. When scheduling flow τ_i, our implementation uses a modified relative deadline $D_i' = D_i - O_s^{\max} - O_r$. Hence, if Γ' is schedulable, any packet of τ_i will complete transfer no later than $a_{i,j}' + D_i - O_s^{\max} - O_r$; since $a_{i,j}' \leq a_{i,j} + O_s^{\max}$ and it takes O_r to notify the receiver VM after the last data transfer of the packet, this means that the packet will complete no later than $a_{i,j} + D_i$, *i.e.*, τ_i and indeed Γ is schedulable according to our model.[4]

We next summarize the schedulability analysis for sporadic, limited-preemption tasks under EDF in [9], [12]. Note that [9], [12] do not consider release jitter, but such term can be integrated in the analysis following related work (*e.g.*, see [50]). We use $U_i'(b^{\mathbf{b}}) = C_i'(b^{\mathbf{b}})/P_i'$ and $U'(b^{\mathbf{b}}) = \sum_{i=1}^{n} U_i'(b^{\mathbf{b}})$ for the utilizations of task τ_i' and task set Γ', respectively.

For an interval of length t, the demand bound function $DBF_i(t, b^{\mathbf{b}})$ of task τ_i' is the maximum cumulative execution requirement of all jobs of τ_i' that have both their release times and absolute deadlines within the interval. When including release jitter and arbitrary deadlines, this is computed as:

$$DBF_i(t, b^{\mathbf{b}}) = \max\left(0, 1 + \left\lfloor \frac{t - (D_i' - J_i')}{P_i'} \right\rfloor\right) \cdot C_i'(b^{\mathbf{b}}). \tag{5}$$

[9], [12] show that if Γ' is not schedulable, then at least one of the following conditions must hold: **(1)** $\exists\ t \geq 0$ such that:

$$\sum_{i=1\ldots n} DBF_i(t, b^{\mathbf{b}}) > t; \tag{6}$$

[4]Note that the reverse is not true, *i.e.*, it is possible for Γ to be schedulable and Γ' to be unschedulable. Hence, the schedulability condition in the Theorem 1 is necessary and sufficient for Γ' but only sufficient for Γ.

(2) $\exists\ t, t'$ with $t \geq t' > 0$ and a job of a task τ_j' with absolute deadline strictly greater than t such that the job executes non-preemptively between time 0 and t' and:

$$\sum_{i=1\ldots n, i\neq j} DBF_i(t, b^{\mathbf{b}}) > t - t'. \tag{7}$$

Note that by definition $t' \leq q_j'(b^{\mathbf{b}})$, and hence $t - t' \geq t - q_j'(b^{\mathbf{b}})$. Therefore, Equation 7 implies:

$$q_j'(b^{\mathbf{b}}) + \sum_{i=1\ldots n, i\neq j} DBF_i(t, b^{\mathbf{b}}) > t. \tag{8}$$

Also note that since the job of τ_j' executes starting at 0 and has absolute deadline after t, it must hold $D_j' > t$. Assume $D_j' > t \geq D_j' - J_j'$; then by definition $DBF_j(t, b^{\mathbf{b}}) \geq C_j'(b^{\mathbf{b}}) \geq q_j'(b^{\mathbf{b}})$. This implies:

$$\sum_{i=1\ldots n} DBF_i(t, b^{\mathbf{b}}) \geq q_j'(b^{\mathbf{b}}) + \sum_{i=1\ldots n, i\neq j} DBF_i(t, b^{\mathbf{b}}) > t, \tag{9}$$

meaning that Condition (1) is also violated at time t. For this reason, it suffices to check Condition (2) for $t < D_j' - J_j'$, for which $DBF_j(t, b^{\mathbf{b}}) = 0$ and thus $\sum_{i=1\ldots n, i\neq j} DBF_i(t, b^{\mathbf{b}}) = \sum_{i=1\ldots n} DBF_i(t, b^{\mathbf{b}})$.

A schedulability criterion can then be constructed by negating Condition (1) and (2). In the following theorem, we express it in a compact way by maximizing the value of $q_j'(b^{\mathbf{b}})$ in Equation 8 over all tasks that meet the condition $D_j' - J_j' > t$.

Theorem 1: A sporadic limited-preemption task set Γ' is schedulable under EDF if and only if:

$$\forall\ t \geq 0 : Q(t, b^{\mathbf{b}}) + \sum_{i=1\ldots n} DBF_i(t, b^{\mathbf{b}}) \leq t, \tag{10}$$

where:

$$Q(t, b^{\mathbf{b}}) = \max 0 \cup \{q_j'(b^{\mathbf{b}}) \mid \forall j \in \{1\ldots n\}, D_j' - J_j' > t\}. \tag{11}$$

Note that Theorem 1 does not provide a schedulability test, because we cannot test an infinite number of values of t. However, it is simple to see that it is sufficient to test those values of t for which the demand bound function for some task changes, which comprise the following set:

$$\mathcal{D} = \{k \cdot P_i' + D_i' - J_i' \mid \forall i \in \{1\ldots n\}, k \in \mathbb{N}\}. \tag{12}$$

Furthermore, following [10], [11], [23], it can be shown that if Conditions (1), (2) are violated, then it must hold $U'(b^{\mathbf{b}}) > 1$ or $t < T^*(b^{\mathbf{b}})$, with:

$$T^*(b^{\mathbf{b}}) = \begin{cases} H & \text{if } U'(b^{\mathbf{b}}) = 1; \\ \min\left[H, \max\left(\max_{i=1}^{n}\{D_i' - J_i'\}, \right.\right. & \\ \left.\left. \frac{1}{1-U'(b^{\mathbf{b}})} \cdot \sum_{i=1}^{n} U_i'(b^{\mathbf{b}}) \cdot \left(P_i' - (D_i' - J_i')\right)\right)\right] & \text{if } U'(b^{\mathbf{b}}) < 1, \end{cases} \tag{13}$$

Algorithm 1: Compute minimum DMA bandwidth for which Γ' is schedulable

input: Transformed task set Γ'
output: Minimum bandwidth $b^{\mathbf{b}}_{\min}$; or **FAILURE** if the task set cannot be scheduled
if $U'(b^{\mathbf{b}}) \leq 1$ *cannot be satisfied* **then**
  return FAILURE
Compute $b^{\mathbf{b}}_{\min,0}$ (Eq. 16)
$b^{\mathbf{b}}_{\min} \leftarrow b^{\mathbf{b}}_{\min,0}$
for $t_k \in \mathcal{D}$ *in increasing order* **do**
  if $t_k \geq T^*(b^{\mathbf{b}}_{\min})$ **then**
    return $b^{\mathbf{b}}_{\min}$
  if $Q(t_k, b^{\mathbf{b}}) + \sum_{i=1}^{n} DBF_i(t_k, b^{\mathbf{b}}) \leq t_k$ *cannot be satisfied* **then**
    return FAILURE
  Compute $b^{\mathbf{b}}_{\min,k}$ (Eq. 18 for case $Q(t_k, b^{\mathbf{b}}) = 0$)
  $b^{\mathbf{b}}_{\min} \leftarrow \max(b^{\mathbf{b}}_{\min}, b^{\mathbf{b}}_{\min,k})$

where $H = \text{lcm}(P'_1, \ldots, P'_n)$ is the hyperperiod. Therefore, instead of Equation 10 the schedulability test can use the equivalent condition:

$$U'(b^{\mathbf{b}}) \leq 1 \wedge \quad (14)$$
$$\forall\, t \in \mathcal{D}, t < T^*(b^{\mathbf{b}}) : Q(t, b^{\mathbf{b}}) + \sum_{i=1\ldots n} DBF_i(t, b^{\mathbf{b}}) \leq t.$$

Note that the test involves performing a $O(n)$ computation for each time point in $\mathcal{D}$ until $T^*(b^{\mathbf{b}})$. As discussed in [9], [12], if $U'(b^{\mathbf{b}})$ is upper bounded by a constant $c < 1$, then the number of points to be tested, and thus the complexity of the test, is pseudo-polynomial.

A. Minimum DMA Bandwidth

Equation 14 allows us to check the schedulability of the transformed task set Γ', and therefore also of the original flow set Γ, assuming that the bandwidth $b^{\mathbf{b}}$ available to the DMA is given. However, in general the system designer might be more interested in specifying a set of flows, and then determining the minimum value of $b^{\mathbf{b}}$ under which Γ' is schedulable; minimizing $b^{\mathbf{b}}$ maximizes the remaining memory bandwidth that can be assigned to the m cores according to Equation 2. A naive way of computing such minimum bandwidth $b^{\mathbf{b}}_{\min}$ would be to use binary search over the values of $b^{\mathbf{b}}$ that satisfy Equation 14. However, in the remaining of this section we show that we can directly compute $b^{\mathbf{b}}_{\min}$ with the same computational complexity of running the test in Equation 14 –that is, by performing a $O(n)$ computation for each tested point in $\mathcal{D}$. For simplicity of exposition, let us index the time instants in $\mathcal{D}$ as $t_1, t_2, \ldots, t_k, \ldots$ in increasing order. We also use $\eta_i(t) = \max\left(0, 1 + \left\lfloor \frac{t-(D'_i-J'_i)}{P'_i} \right\rfloor\right)$ to denote the number of jobs included in the demand bound function for τ'_i at t.

Note that the condition $U'(b^{\mathbf{b}}) \leq 1$ can be rewritten to:

$$\sum_{i=1}^{n} \Big(\lceil C_i/S_c \rceil \cdot O_{dma} + C_i/b^{\mathbf{b}} + O_{pckt,i}\Big)/P'_i = \quad (15)$$
$$\sum_{i=1}^{n} \Big(\lceil C_i/S_c \rceil \cdot O_{dma} + O_{pckt,i}\Big)/P'_i + \frac{\sum_{i=1}^{n} C_i/P'_i}{b^{\mathbf{b}}} \leq 1.$$

Then if $\sum_{i=1}^{n} \Big(\lceil C_i/S_c \rceil \cdot O_{dma} + O_{pckt,i}\Big)/P'_i \geq 1$, the condition cannot be satisfied, and the task set is unschedulable. Otherwise, we obtain:

$$b^{\mathbf{b}} \geq \frac{\sum_{i=1}^{n} C_i/P'_i}{1 - \sum_{i=1}^{n} \Big(\lceil C_i/S_c \rceil \cdot O_{dma} + O_{pckt,i}\Big)/P'_i}. \quad (16)$$

Let $b^{\mathbf{b}}_{\min,0}$ denote the minimum value of $b^{\mathbf{b}}$ that satisfies Equation 16 (that is, the right hand size of the equation). We use $b^{\mathbf{b}}_{\min,k}$ to denote the minimum value of $b^{\mathbf{b}}$ that satisfies Equation 10 for $t = t_k$; we can obtain $b^{\mathbf{b}}_{\min,k}$ by rewriting Equation 10 in a way similar to Equation 15. However, because it can either hold $Q(t, b^{\mathbf{b}}) = 0$ or $Q(t, b^{\mathbf{b}}) = O_{dma} + S_c/b^{\mathbf{b}}$ or $Q(t, b^{\mathbf{b}}) = O_{dma} + C^{\text{last}}_j/b^{\mathbf{b}} + O_{pckt,j}$ for some task τ'_j, we have to consider three cases. If $\nexists \tau'_j : D'_j - J'_j > t$, then $Q(t, b^{\mathbf{b}}) = 0$ and we obtain:

$$\sum_{i=1\ldots n} DBF_i(t, b^{\mathbf{b}}) = \quad (17)$$
$$\sum_{i=1}^{n} \eta_i(t) \cdot \Big(\lceil C_i/S_c \rceil \cdot O_{dma} + O_{pckt,i}\Big) +$$
$$\frac{\sum_{i=1}^{n} \eta_i(t) \cdot C_i}{b^{\mathbf{b}}} \leq t.$$

Again, if $\sum_{i=1}^{n} \eta_i(t) \cdot \Big(\lceil C_i/S_c \rceil \cdot O_{dma} + O_{pckt,i}\Big) \geq t$, the condition cannot be satisfied, and the task set is unschedulable. Otherwise we obtain:

$$b^{\mathbf{b}}_{\min,k} = \frac{\sum_{i=1}^{n} \eta_i(t) \cdot C_i}{t - \sum_{i=1}^{n} \eta_i(t) \cdot \Big(\lceil C_i/S_c \rceil \cdot O_{dma} + O_{pckt,i}\Big)}. \quad (18)$$

Similar equations can be derived if there exists a task τ'_j with $D'_j - J'_j > t$, in which case we need to consider both the case of $Q(t, b^{\mathbf{b}}) = O_{dma} + C^{\text{last}}_j/b^{\mathbf{b}} + O_{pckt,j}$, where τ'_j is the task with the largest value of C^{last}_j among those that satisfy $D'_j - J'_j > t$, as well as the case $Q(t, b^{\mathbf{b}}) = O_{dma} + S_c/b^{\mathbf{b}}$ (if for any such task it holds $C_i > S_c$). In this case, $b^{\mathbf{b}}_{\min,k}$ is taken as the maximum between the cases that apply.

Finally, Algorithm 1 shows how to compute $b^{\mathbf{b}}_{\min}$. We initially set $b^{\mathbf{b}}_{\min} = b^{\mathbf{b}}_{\min,0}$. Then, we iterate over the points in $\mathcal{D}$. At each step, we update $b^{\mathbf{b}}_{\min}$ as the maximum between its previous value, and the newly computed $b^{\mathbf{b}}_{\min,k}$. If at any step the schedulability condition cannot be met no matter the value of $b^{\mathbf{b}}$, the algorithm fails. Otherwise, the algorithm terminates once it reaches a t_k greater than or equal to $T^*(b^{\mathbf{b}}_{\min})$, as this guarantees that the condition is met for all subsequent values of t under the computed DMA bandwidth $b^{\mathbf{b}}_{\min}$.

V. Experimental Evaluation

This section presents the empirical methodology we adopt to measure the maximum DRAM bandwidths under CPU and DMA regulation while still satisfying Equation 2. We validate the architecture proposed in Section III and present our overhead-aware schedulability experiments.

A. Measuring DRAM Saturation

We study the DRAM saturation point from the perspective of the CPU and the DMA. Following a similar methodology as [49], we first establish a relationship between the worst DRAM stress generated by CPUs alone and then under various MemGuard budget values. We then study the amount of bandwidth extracted by the DMA from the memory subsystem and its relationship to the corresponding QoS regulation values. Finally, we investigate the combined contributions of MemGuard and QoS regulation levels.

1) MemGuard Regulation and DRAM Utilization Using CPU: In order to generate worst-case stress for the DRAM controller from the CPU subsystem, we use our publicly available *USTRESS* benchmark [49]. *USTRESS* uses write access patterns (which produces worse stress patterns than reads [26]) that miss in the L2 cache and access a closed DRAM row (row miss) on the same bank. *USTRESS* can therefore maximize the DRAM controller utilization. In fact, the bandwidth reported by *USTRESS* progressively decreases when the benchmark is (synchronously) executed on a progressively increasing number of cores.

We adapted the DRAM geometry of the original benchmark to the DRAM geometry of our Xilinx ZynqMP ZCU102. On the ZCU102, through documentation and experimentation, we found the DRAM organization to be three column shift bits, two bits for the bank, two bits for the bank group, and fifteen bits for the row. The combined bandwidth of the cores when synchronously executing *USTRESS* remains stable at approximately 960 MB/s, which is thus the maximum bandwidth that can be extracted from the DRAM memory subsystem at maximum utilization.

Next, we study the relationship between the maximum bandwidth of the memory subsystem and the MemGuard regulation values. As described in Section II, MemGuard works by assigning each CPU_k a regulation budget of Q_k cache refills per regulation period P. Q_k is measured in terms of L2 cache refills assessed via performance counters. In our setup, MemGuard regulation period and budget values are enforced by the Jailhouse hypervisor. Smaller P values allow for a finer-grained CPU regulation at the expense of increased overheads: an interrupt is generated and processed by the hypervisor at every regulation period in order to police the CPUs' budgets. Prohibitively small P values would cause MemGuard to misbehave due to excessive overheads. We, therefore, conducted experiments to select the smallest possible *viable* value of P for our setup (which represents a worst-case scenario for MemGuard regulation strategy). Furthermore, since we rely on the granularity of L2 cache-refill performance counters for Q_k, we also evaluated the minimum

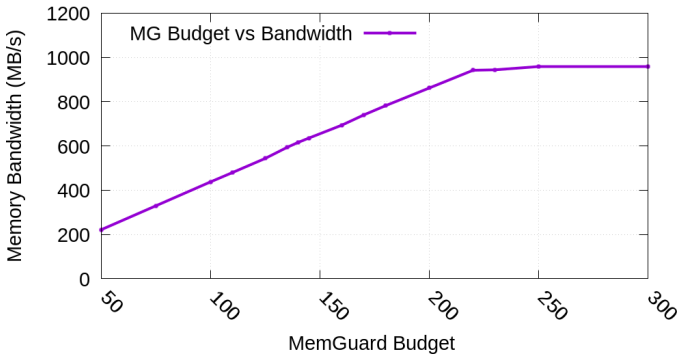


Fig. 3. *USTRESS* bandwidth with increasing Q_k and P = 30 μs.

Q_k that can be feasibly enforced for a CPU_k. Our empirical results suggest that, with our setup, any value $P < 30$ μs and any value $Q_k < 50$ (at $P = 30$ μs) would cause MemGuard to misbehave.[5]

In order to establish the relationship between maximum memory bandwidth of the memory subsystem and MemGuard regulation, we, therefore, set $P = 30$ μs and progressively increase Q_k starting from 50 while accessing the memory subsystem using *USTRESS*. The results are presented in Figure 3, which shows a linear trend until the bandwidth reaches 960 MB/s at $Q_k = 225$. This corresponds to the maximum bandwidth the CPUs can extract from the DRAM at maximum utilization.

2) QoS Regulation and DRAM Utilization Using DMA: The ZCU102 has two DMA engines: the *ADMA* is cache-coherent, whereas the *GDMA* is not. Both DMAs can be QoS-regulated [57]. Since we assume cache-coherent DMA transfers (see Section II), we used the ADMA engine in our experiments. We configured the DMA burst length to its maximum, *i.e.*, 16, resulting in a 128 B transfer per transaction. Once again, all the transactions always target the same bank, but the sequential access pattern will produce row hits. Without QoS regulation and any interference, the bandwidth achievable from the DMA depends on the transfer chunk size S_c. For $S_c \in \{4, 8, 16, 32, 64\}$ KiB, we measured a maximum throughput of approximately 2000, 2440, 2800, 3080, and 3190 MB/s, respectively.

The QoS supports different regulation modes [7] controlled by a set of read (`ar_r`, `ar_b`, `ar_p`) and write parameters (`aw_r`, `aw_b`, `aw_p`). By setting `ar_b` = `aw_b` = 1, the parameters `ar_r` and `aw_r` are sufficient to enforce a strict regulation at the specified `ar_r` (`aw_r`) level. In our experiments, we always use `ar_r` = `aw_r` corresponding to the QoS level $\mathtt{a}^{\mathbf{b}}$ described in Section II.

Figure 4 reports the DMA memory bandwidth for increasing QoS regulation values $\mathtt{a}^{\mathbf{b}}$ = `ar_r` = `aw_r` and different chunk sizes S_c when accessing the same DRAM bank. The figure also shows the maximum theoretical bandwidth computed using Equation 1, with the maximum transaction size

[5]Stressing the system using *USTRESS* with $P < 30$ μs caused consistent discrepancies between measured and nominal bandwidth.

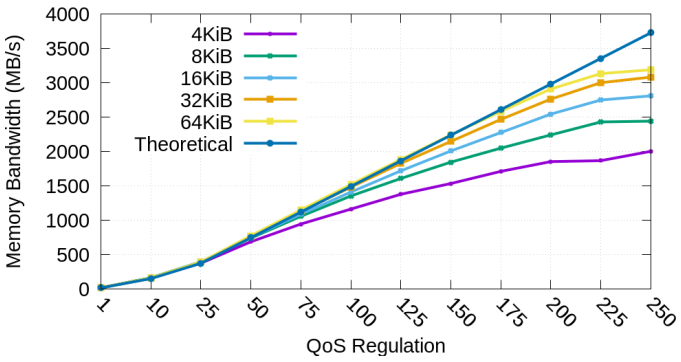


Fig. 4. DMA bandwidth for different sizes of S_c and QoS regulation values.

S_b = 128 B and the DRAM frequency f_{clk} = 0.5 GHz. In the least regulated cases $\texttt{a}^\texttt{b} = 250$, the bandwidth differs significantly depending on the chunk size. Thus larger S_c chunk sizes are preferred when only minimal QoS regulation is applied.

3) Combining MemGuard and QoS Regulation: The approach proposed in [49] to *independently* compute the MemGuard regulation for the cores and QoS level for the DMA –such that Equation 2 is satisfied– is not directly applicable to our setup. On the ZCU102, we are unable to prioritize DMA traffic over CPU traffic statically. When the CPUs are active before a regulation event happens, they can temporarily force the DMA below its assigned bandwidth due to the bursty nature of MemGuard regulation. The lost bandwidth is not recovered due to the strict QoS regulation ($\texttt{ar\_b} = \texttt{aw\_b} = 1$).

Therefore, we empirically determine the different DMA and CPU contributions: We consider three active CPUs and one DMA. The CPUs are regulated via MemGuard at $P = 30\mu s$ and perform memory transactions using *USTRESS*. Starting with a per-CPU MemGuard budget $Q_k = 50$ and proceeding step-wise towards higher Q_k values, we generate decreasing DMA traffic by controlling the QoS. For each Q_k level, we determine the maximum $\texttt{a}^\texttt{b}$ such that the CPU bandwidth is unaffected by the presence of concurrent DMA transfers. Once such Q_k and $\texttt{a}^\texttt{b}$ levels are known, the DMA bandwidth corresponding to this $\texttt{a}^\texttt{b}$ can be determined via Figure 4.[6] Each combination (Q_k, $\texttt{a}^\texttt{b}$) identifies a point that satisfies Equation 2 and results in the maximum utilization of the DRAM.

Table I reports the different (Q_k, $\texttt{a}^\texttt{b}$) value points. For example, when $Q_k = 150$, the CPUs are heavily regulated, and their bandwidth (220 MB/s) does not decrease even when the DMA is not regulated ($\texttt{a}^\texttt{b} = 0$ means that the QoS regulation is disabled). On the contrary, for $Q_k = 210$, any $\texttt{a}^\texttt{b} > 3$ would cause the CPUs bandwidth (308 MB/s) to decrease. Starting from $Q_k \geq 225$, the CPUs are interfering with each other even without any DMA-generated traffic.

[6]Given the observed delta between theoretical and experimental QoS to bandwidth conversion, Figure 4 should be preferred to determine the effective DMA bandwidth.

B. Implementation Overheads

We evaluate our implementation described in Section III-C by measuring critical overheads and delays as well as VM-to-VM event signaling (see Table II). We insert non-invasive timestamping into our implementation. The timestamps are read out during planned idle intervals of the system and used to calculate the overheads. As a timing source, we use the CPU's system timer. It is synchronized across all cores, which permits the measurement of cross-core notification delays (Δ_{BR}). The measurements shown in Table II can be directly mapped to actions marked in Figure 2. Our test system employs two communicating VMs and four channels for two bidirectional links between the VMs. The implementation of the communication broker compiles to 4243 B of instructions and 8904 B of data. Dynamically allocated memory never exceeds 12288 B. Code and data fit into L1 caches for this configuration, which validates our assumption that the broker should not impact the memory bandwidth calculations. The allocated 128 KiB L2 cache partition is sufficient to host multiple communicating guests and channel configurations. In our configuration 3264 B are used. Per additional channel or VM we require additional 56 B or 1152 B, respectively. Hypercall and IRQ-related experiments were run 1000 times. Packet processing-related measurements are based on 4000 sent packets.

The derived overheads are used to bound the terms $O_s^{\min}, O_s^{\max}, O_{dma}, O_{pckt,i}, J_i', O_r$, as used in the analysis in Section IV. The virtio device assigns a timestamp after entering hypervisor mode and passing the PCI transport layer, which yields (Δ_{HC} is round trip time):

$$O_s^{\min} = \Delta_{HC}^{\min}/2 + \Delta_{PT}^{\min}. \tag{19}$$

$$O_s^{\max} = \Delta_{HC}^{\max}/2 + \Delta_{PT}^{\max}. \tag{20}$$

The DMA overhead O_{dma} includes all overheads at the broker for steps ⑤ to ⑧ in Figure 2, except the the DMA transfer time itself and the time $O_{pckt,i}$ to remove the packet from its sender's queue. Based on Table II, this results in:

$$O_{dma} = m \cdot \Delta_{PS}^{\max} + \Delta_{PD}^{\max} + \Delta_{HC}^{\max} + \Delta_{IP}^{\max} + \Delta_{FT}^{\max}, \tag{21}$$

where the overhead Δ_{PS} of finding the packet with the earliest deadline is paid m times because there are m per-VM queues. Because of the ordered per-VM queues, the broker only has to access the first element in each queue. This operation is implemented lock-free. When the DMA finishes the transfer, it will interrupt the broker, which results in an IRQ injection done by the hypervisor Δ_{HC}, and the processing of the DMA interrupt by the broker Δ_{IP}. At last, the broker notifies the receiver and the sender by sending an IPI to each from hypervisor mode Δ_{FT}.

The IPI delay Δ_{BR} is the time between the hypervisor sending the IPI and the receiver finishing the IRQ processing. The DMA overhead O_{dma} considers the full hypervisor entry and exit in Δ_{FT}. However, the hypervisor exit is happening in parallel to the IPI delay in Δ_{BR}. To correct for the duplicate

TABLE I
DMA QOS REGULATION VALUES TO PROTECT CORES AT FIXED MEMGUARD BUDGET.

Total MemGuard Budget $\sum Q_k$	*Per Core Assigned MemGuard Budget* Q_k	*USTRESS Bandwidth (MB/s) 1 CPU only*	*USTRESS Bandwidth (MB/s) with 3 CPUs and DMA*	*DMA Bandwidth (MB/s) @ QoS* $\mathtt{a}^{\mathtt{b}}$ *s.t. BW of CPUs unaffected*
150	50	220	220	1074 @ QoS 0 (disabled)
165	55	244	244	649 @ QoS 60
180	60	264	264	485 @ QoS 40
195	65	286	286	148 @ QoS 10
210	70	308	308	47 @ QoS 3
225	75	330	318	No valid QoS possible

TABLE II
OVERHEADS AND DELAYS IN OUR IMPLEMENTATION (IN NANOSECONDS) AND MAPPING TO COMMUNICATION FLOWS IN FIGURE 2.

	Symbol	Actions Fig. 2	MIN	AVG	MAX	MED
Hypervisor entry & exit (Sender OS to Hyp.)	Δ_{HC}	②	939	941	949	939
PCI transport layer (Hyp.)	Δ_{PT}	③	111	295	757	292
Packet parsing (Hyp.)	Δ_{PP}	③	181	433	1161	424
Queue locking (Hyp.)	Δ_{QL}	③	71	98	142	101
Queue insertion (Hyp.)	Δ_{QI}	③	50	74	101	70
Queue find location for insertion (per packet) (Hyp.)	Δ_{QA}	③	24	26	30	26
Queue remove (Broker)	Δ_{QR}	⑧	20	33	40	30
Find packet with earliest deadline (per VM) (Broker)	Δ_{PS}	⑤	20	32	71	30
Process packet and program DMA (Broker)	Δ_{PD}	⑤	373	406	949	393
Finalize transfer incl. IPI (Broker & Hyp.)	Δ_{FT}	⑧	2222	2350	2646	2353
DMA IRQ processing (Broker)	Δ_{IP}	⑦	797	811	828	808
OS notification IPI + IRQ processing by receiver (Broker to Receiver OS)	Δ_{BR}	⑧+⑨	1370	1403	1460	1400

accounting of the hypervisor exit, we upper bound the receiver notification delay as:

$$O_r = \Delta_{BR}^{\max} - \frac{\Delta_{HC}^{max}}{2}. \tag{22}$$

Note that we use the maximum for Δ_{HC} because we offset Δ_{HC}^{max} used in O_{dma}.

The remaining terms $O_{pckt,i}, J_i'$ involve adding and removing packets from priority queues; hence, it is necessary to discuss the queue implementation. We use a linked list implementation for the priority queues that requires n comparisons for a queue containing n elements. The time for insertion of a packet by VM_k without locking can be bounded using the maximum amount of packets sent by VM_k that can be pending at any time. Under the assumption that a packet is schedulable, it cannot be pending for longer than D_i. Hence, given the set Γ_k of flows that have VM_k as their sender, the maximum amount of pending packets sent by VM_k is $\sum_i \lceil D_i/P_i \rceil$ for $\tau_i \in \Gamma_k$ and $\Gamma_k \subseteq \Gamma$. Thus, queue insertion is bounded by:

$$O_{QI,k} = \Delta_{QI}^{max} + \sum_{\tau_i \in \Gamma_k} \lceil D_i/P_i \rceil \cdot \Delta_{QA}^{max}. \tag{23}$$

Additionally, the priority queues are protected by a lock, so the sender and the broker can be blocked for a short amount of time. The lock is based on the Jailhouse implementation of a spinning ticket lock, such that the blocking time with two contenders can be bounded to the maximum duration the contender spends holding the lock. Blocking due to the broker is given by:

$$B_B = \Delta_{QL}^{max} + \Delta_{QR}^{max}. \tag{24}$$

Blocking due to the sender is given by:

$$B_{S,k} = \Delta_{QL}^{max} + O_{QI,k}^{max}. \tag{25}$$

The time to remove a packet $\tau_i \in \Gamma_k$ from the queue of sender VM_k is then:

$$O_{pckt,i} = B_{S,k} + B_B. \tag{26}$$

Finally, we consider the release jitter. Our implementation uses event signaling and lock-free access to the first queue entry (the packet with the earliest deadline) such that the maximum time between the assignment of the deadline on VM_k and the first instant where the broker can consider the packet $\tau_i \in \Gamma_k$ in a scheduling decision is given by:

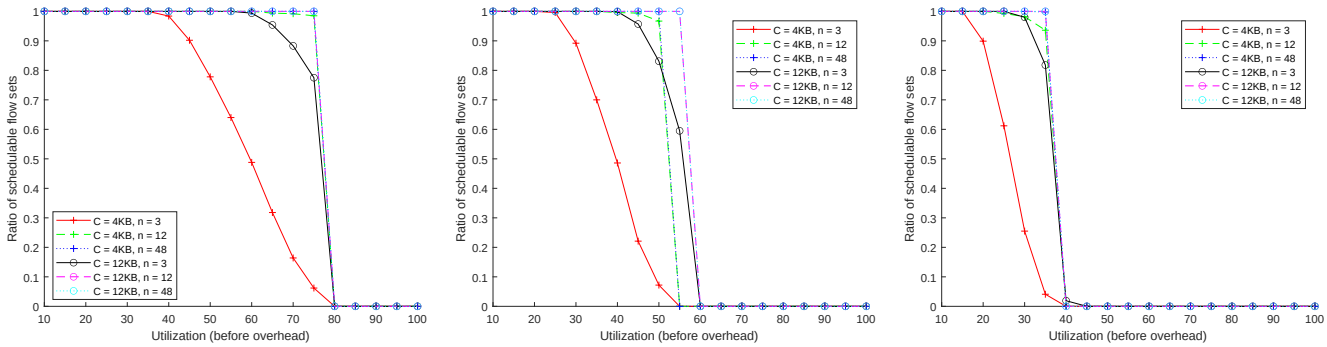
$$J_i' = (\Delta_{PP}^{max} + B_{S,k} + B_B) \cdot |\Gamma_k| : \tag{27}$$

for each packet insertion, the virtio device has to parse the packet header and update the queue, for which it can be blocked due to the broker modifying the queue. After the insertion, the packet is immediately visible by the broker. The multiplication term results from the capability of the virtio driver to prepare and send more than one TX buffer with one send operation (*i.e.*, at most $|\Gamma_k|$). Additionally, a prior transfer might have just started, so the packet can only be considered after the completion of the prior transfer.

C. Schedulability Experiments

To assess the impact of the measured overheads, we generated synthetic flow sets and tested their schedulability based on the analysis in Section IV. A flow set is generated as follows. We first assign a maximum allowable DMA bandwidth $b^{\mathtt{b}}$ in MB/s, a number of VMs v, flows n, a packet size C, and a desired system utilization (prior to overhead inflation) $U \in [0, 1]$. We then randomly and uniformly generate a utilization U_i for each flow [14], such that $U = \sum_{i=1}^{n} U_i$, and compute the inter-arrival time P_i based on $U_i = (C/b^{\mathtt{b}})/P_i$. Each flow is randomly assigned to a sender among the v VMs.

Figure 5 shows the obtained results of the ratio of schedulable flow sets for a system with $v = 4$ VMs where we



(a) DMA BW = 150 MB/s (b) DMA BW = 500 MB/s (c) DMA BW = 1000 MB/s

Fig. 5. Ratio of schedulable flow sets: 4 VMs, flows = $\{3, 12, 48\}$, packet size = $\{4, 12\}$ KiB, and DMA bandwidth = $\{150, 500, 1000\}$ MB/s.

vary the DMA bandwidth in $\{1000, 500, 150\}$, the number of flows in $\{3, 12, 48\}$, and the packet size in $\{4, 12\}$ KiB. For each utilization point, we generated and tested 1000 flow sets. Note that schedulability significantly improves as the DMA bandwidth $b^{\mathbf{b}}$ decreases. This is because the utilization is computed based on the transfer time $C/b^{\mathbf{b}}$; for lower values of $b^{\mathbf{b}}$, the transfer time becomes larger while the overheads in Table II remain constant. Thus their impact on the schedulability of the flow sets is reduced. Also, note that the system performs better for a larger number of tasks and packet size: as those parameters increase, the average period of each task at a given utilization also increases, which lessens the blocking effect due to the non-preemptive transfer of a data chunk of size $S_c = 4$ KiB.

We also measured the time required to run the schedulability analysis for 48 flows and a packet size of 12 KiB. We implemented the analysis using (single-threaded) Matlab code executed on an AMD Ryzen 5 3600 and obtained a median of 0.34ms, mean of 4.9ms, 95th percentile of 8.0ms, and maximum of 25.82s. This shows that the time required to perform the analysis is small for most cases, even with a large number of flows. Note that increasing the number of VMs would not affect the complexity of the analysis.

D. Comparison

In Table III, we show a comparison of our implementation and the schedulability analysis. We used binary search to find the smallest period for which a single packet of a given size can be transferred when the DMA is limited to a given bandwidth, where the deadline is set equal to the period. We ran the same scenario in our implementation for 1,000 iterations to obtain the maximum latency. We selected four scenarios from the middle of the possible QoS ranges (QoS 40 and QoS 10, see Table I) to select viable combinations of utilization and periods to run the experimental scenarios. As Table III shows, the measurement of FreeRTOS confirms the analytical limits.

TABLE III
COMPARISON OF IMPLEMENTATION (LATENCY) AND ANALYSIS (PERIOD) FOR A UNIDIRECTIONAL FLOW WITH IMPLICIT DEADLINES.

Experiment		FreeRTOS		
DMA (MB/s)	**Size (KiB)**	**Latency (ns)**	**Period (ns)**	**Diff (%)**
148	4	34,535	36,933	-6.5
148	12	99,818	103,171	-3.3
485	4	14,747	17,703	-16.7
485	12	42,777	45,480	-5.94

VI. RELATED WORK

This work analyzes the problem of predictably copying data among VMs and virtualized hardware resources on MPSoCs under consideration of maximum memory bandwidth and memory interference. The issues of real-time data communication among different VMs have been tackled from different angles with software- and hardware-based solutions.

a) Software-based solutions: Cache partitioning (*e.g.*, *MemGuard*) [33] and bank partitioning [59] are well-known techniques to mitigate and eliminate memory interference among independent partitions on MPSoCs. Kloda *et al.* [31] proposed a deterministic approach to carefully control the architecture-specific address bits used at the cache and DRAM level. In their work, I/O devices are statically assigned to only one VM. Sohal *et al.* [49] leveraged the *Quality of Service* [7] infrastructure offered by some ARM boards to create a regulating framework for system-wide bandwidth management and control. Their work does not explicitly consider I/O nor data transfer between VMs, but we adopt the same QoS techniques to control DMA bandwidth in our architecture. Tabish *et al.* [52] proposed a communication architecture for strictly partitioned multicore processors. Their work considers interference but only at the DRAM-bank level and does not explicitly address virtualized environments. Several works [20], [25], [32], [36], [53] have addressed the problem of managing I/O in the context of the Quest-V separation kernel. Although real-time bounds on the I/O communication are provided, the effect of memory contention and interference at interconnect level is not explicitly considered. The MC^2 project [19], [30] implemented inter-core communication using a shared DRAM bank. Their approach does not explicitly consider virtualization and the use of a DMA engine to perform data transfer. A recent work from Casini *et al.* [17] proposed a

hypervisor-based architecture that shares similarities with the architecture provided by this work. Similar to this work, their work formally analyzed the end-to-end latencies of I/O data transfers under consideration of virtualization overheads. However, the problems of memory-related interference, the use of DMA to perform data transfers, and the real-time scheduling of independent I/O flows are not considered by their work. In addition, contrary to [17], this work uses the virtio API [1], thus enabling unmodified execution of virtualized OSs. Several real-time resource access protocols exist to manage memory regions shared among VMs of different criticality (we refer to [15] for a recent review). Compared to our architecture, approaches based on shared memory are much more difficult to configure, as they require a global knowledge on the activation of tasks within different VMs (knowledge that might not even be available to the system integrator). Furthermore, these approaches require difficult to produce certification artefacts to document the interplay and mitigation strategies for the partition interference channels [46] that they create. Pellizzoni *et al.* [41]–[43] proposed WCET analytical bounds that explicitly consider I/O, but consider neither memory interference nor virtualized environments. The predictable execution model [40] and the scratchpad-centric OS [51] propose a three-phase execution model to address the predictability of execution and I/O phases. Several works [2], [3], [45] have tackled the related problems of latency and scheduling in network communication, and in both the real-time and high-performance computing areas, works exist that experimentally evaluate data transfer techniques in virtualized environments (*e.g.*, [6], [21], [48]). In [16], [18], the memory interference on several NVIDIA-based SoCs has been experimentally characterized.

In order to manage I/O complexity, several approaches – although without any formal guarantee– are adopted in the industrial world. For example, in the avionics domain, the ARINC [5] standard mandates *queuing ports* to manage inter-partition communication, and the Xen Hypervisor uses a split-driver and ring buffer model [54] to multiplex I/O requests coming from different VMs.

b) Hardware-based solutions: Compared to software-only solutions, prototyping on hardware and/or FPGA is more complex and time-intensive. Unsurprisingly, therefore fewer hardware-based solutions exist. Jiang *et al.* [28], [29] proposed the Virtualized Complicated Device Controller and MCS-IOV hardware extensions to enable predictable virtualization of I/O. Betti *et al.* [13] implemented FPGA hardware extensions to manage I/O data transfers on COTS systems. IOMPU and MPIOV [37], [38] are solutions that improve the management of PCI-based hardware devices. SR-IOV [35] devices can isolate different communication (network) flows. Contrary to these works, our solution does not require extra hardware.

VII. Conclusion and Future Work

In this paper, we have proposed and implemented a framework to predictably transfer data between otherwise isolated VMs and virtualized hardware resources on heterogeneous MPSoCs under consideration of memory contention. The framework leverages standard mechanisms (*e.g.*, cache-partitioning and QoS regulation) to control memory interference and is based on the observation that DRAM utilization is additive when working below the saturation point. Thanks to its virtio-based design, the framework can be directly used by unmodified guest OSs. While in this paper we have evaluated a lightweight implementation based on FreeRTOS, we plan to investigate Linux virtio drivers in the future.

Our analysis of the communication flows supported by the framework enables designers to bound the maximum bandwidth assigned to cores and the DMA, and to meet the deadline constraints of the flows while avoiding over-utilization of the DRAM memory controller, thus causing unpredictable latencies. Our evaluation has shown the matching of the analysis and a low-overhead implementation realized on top of the Jailhouse hypervisor with a baremetal broker VM and FreeRTOS guests.

Our framework targets (certifiable) systems with different VM criticalities. As such, the architecture does not adopt a zero-copy approach, but it requires *explicit* copy operations between VMs. However, each copy operation –and its implicit interference– is controlled by a trusted broker component. Given its lightweight implementation, the broker only minimally extends the trusted codebase (and thus a certification effort). On the other hand, this architecture devotes a core (even if not necessarily an application core on an MPSoC) to the broker functionality, and it requires to separate the hypervisor-broker communication using cache partitioning.

There are multiple directions that we would like to investigate as future work. The current architecture benefits from the use of a DMA engine to perform the data transfer. However it would be interesting to investigate the size-dependent trade-off between DMA programming and using the broker's CPU to perform the data copy actively. Additionally, other data transfer architectures are possible. For example, the broker VM can become superfluous by shifting the burden of performing the data copy onto the sender or receiver VM. Such architectures would require a different analytical model than the one we currently adopted, making a comparison between approaches even more challenging. Finally, extending the model to consider multicast communication would be an additional challenge.

Acknowledgments

This work has been partially supported by the NSERC, CMC Microsystems, and the National Science Foundation (NSF) under grant numbers CNS 1932529, CNS 1815891. Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research. We want to thank Zubair Waheed (an undergraduate at the University of Waterloo) for performing an initial performance estimation on the ZCU102 platform.

REFERENCES

[1] OASIS Committee Specification 01. Virtual I/O Device (VIRTIO) Version 1.1. https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html, April 2019.

[2] Laure Abdallah, Mathieu Jan, Jérôme Ermont, and Christian Fraboul. Reducing the Contention Experienced by Real-Time Core-to-I/O Flows over a Tilera-Like Network on Chip. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, page 86–96, 2016.

[3] Saeed Abedi, Neeraj Gandhi, Henri Maxime Demoulin, Yang Li, Yang Wu, and Linh Thi Xuan Phan. RTNF: Predictable Latency for Network Function Virtualization. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, page 368–379, 2019.

[4] Siemens AG. Jailhouse hypervisor. https://github.com/siemens Accessed: 2021-02-08.

[5] Airlines Electronic Engineering Commitee. ARINC Specification 653 P1-5, 2019.

[6] Gabriele Ara, Luca Abeni, Tommaso Cucinotta, and Carlo Vitucci. On the use of kernel bypass mechanisms for high-performance inter-container communications. In Michèle Weiland, Guido Juckeland, Sadaf R. Alam, and Heike Jagode, editors, *High Performance Computing - ISC High Performance 2019 International Workshops, Frankfurt, Germany, June 16-20, 2019, Revised Selected Papers*, volume 11887 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2019. doi:10.1007/978-3-030-34356-9_1.

[7] ARM. ARM CoreLink QoS-400 Network Interconnect Advanced Quality of Service. https://developer.arm.com/documentation/dsu0026/latest Accessed: 2021-02-08.

[8] ARM. ARM System Memory Management Unit Architecture Specification - SMMU architecture version 2.0. https://developer.arm.com/documentation/ihi0062/latest Accessed: 2021-02-08.

[9] Sanjoy Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, page 137–144, 2005.

[10] Sanjoy K. Baruah, Rodney R. Howell, and Louis E. Rosier. Feasibility Problems for Recurring Tasks on One Processor. *Theor. Comput. Sci.*, 118(1):3–20, September 1993.

[11] Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor. In *In Proceedings of the 11th Real-Time Systems Symposium*, page 182–190. IEEE Computer Society Press, 1990.

[12] Marko Bertogna and Sanjoy Baruah. Limited Preemption EDF Scheduling of Sporadic Task Systems. *Industrial Informatics, IEEE Transactions on*, 6:579–591, 12 2010.

[13] Emiliano Betti, Stanley Bak, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Real-Time I/O Management System with COTS Peripherals. *IEEE Transactions on Computers*, 62(1):45–58, 2013.

[14] Enrico Bini and Giorgio C. Buttazzo. Measuring the Performance of Schedulability Tests. *Real-Time Syst.*, 30(1–2):129–154, May 2005.

[15] Björn B. Brandenburg. Multiprocessor Real-Time Locking Protocols: A Systematic Review, 2019. arXiv:1909.09600.

[16] Nicola Capodieci, Roberto Cavicchioli, Ignacio Sañudo Olmedo, Marco Solieri, and Marko Bertogna. Contending memory in heterogeneous SoCs: Evolution in NVIDIA Tegra embedded platforms. In *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, page 1–10, 2020.

[17] Daniel Casini, Alessandro Biondi, Giorgiomaria Cicero, and Giorgio Buttazzo. Latency Analysis of I/O Virtualization Techniques in Hypervisor-Based Real-Time Systems. In *27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2021)*, 2021.

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

[19] Micaiah Chisholm, Namhoon Kim, Bryan C Ward, Nathan Otterness, James H Anderson, and F Donelson Smith. Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 57–68. IEEE, 2016.

[20] Matthew Danish, Ye Li, and Richard West. Virtual-CPU Scheduling in the Quest Operating System. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, page 169–179, 2011.

[21] N. El-Sayed, A. Mukkara, P. Tsai, H. Kasture, X. Ma, and D. Sanchez. KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 104–117, 2018.

[22] FreeRTOS. FreeRTOS Real-time operating system for microcontrollers. https://www.freertos.org/.

[23] Laurent George, Nicolas Rivierre, and Marco Spuri. Preemptive and Non-Preemptive Real-Time UniProcessor Scheduling. *INRIA*, RR-2966, 1996.

[24] Bosch GmbH. ETAS RTA Hypervisor. https://www.etas.com/en/products/rta-vrte.php Accessed: 2021-02-08.

[25] Ahmad Golchin, Soham Sinha, and Richard West. Boomerang: Real-Time I/O Meets Legacy Systems. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, page 390–402, 2020.

[26] Mohamed Hassan. Reduced latency dram for multi-core safety-critical real-time systems. *Real-Time Systems*, pages 1–36, 2019.

[27] International Standardization Organization. ISO 26262:2018(E) Road vehicles — Functional safety, 2018.

[28] Zhe Jiang and Neil Audsley. VCDC: The Virtualized Complicated Device Controller. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, page 5:1–5:21, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[29] Zhe Jiang, Xiaotian Dai, Pan Dong, Ran Wei, Dawei Yang, Neil Audsley, and Nan Guan. Towards an Analysable, Scalable, Energy-Efficient I/O Virtualization for Mixed-Criticality Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, page 1–1, 2021.

[30] N. Kim, S. Tang, N. Otterness, J. Anderson, F. D. Smith, and D. Porter. Supporting I/O and IPC via Fine-Grained OS Isolation for Mixed-Criticality Real-Time Task. *Real-Time Systems*, 56(4):349–390, 2020.

[31] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna. Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, page 1–14, 2019.

[32] Ye Li, Richard West, Zhuoqun Cheng, and Eric Missimer. Predictable Communication and Migration in the Quest-V Separation Kernel. In *2014 IEEE Real-Time Systems Symposium*, page 272–283, 2014.

[33] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, page 45–54, 2013.

[34] Clémentine Maurice, Nicolas Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *2015 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, volume 9404, page 48–65, 2015.

[35] Microsoft. Introduction to Single Root I/O Virtualization. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/single-root-i-o-virtualization–sr-iov- Accessed: 2021-02-10.

[36] Eric Missimer, Katherine Missimer, and Richard West. Mixed-Criticality Scheduling with I/O. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, page 120–130, 2016.

[37] Daniel Muench, Michael Paulitsch, and Andreas Herkersdorf. IOMPU: Spatial Separation for Hardware-Based I/O Virtualization for Mixed-Criticality Embedded Real-Time Systems Using Non-transparent Bridges. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, page 1037–1044, 2015.

[38] Daniel Münch, Michael Paulitsch, Oliver Hanka, and Andreas Herkersdorf. MPIOV: Scaling hardware-based I/O virtualization for mixed-criticality embedded real-time systems using non transparent bridges to (Multi-Core) multi-processor systems. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, page 579–584, 2015.

[39] NVIDIA. NVIDIA Jetson AGX Xavier. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/ Accessed: 2021-02-08.

[40] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A Predictable Execution Model for COTS-Based Embedded Systems. In *2011 17th IEEE Real-*

Time and Embedded Technology and Applications Symposium, page 269–279, 2011.

[41] Rodolfo Pellizzoni, Bach D. Bui, Marco Caccamo, and Lui Sha. Coscheduling of CPU and I/O Transactions in COTS-Based Embedded Systems. In *2008 Real-Time Systems Symposium*, page 221–231, 2008.

[42] Rodolfo Pellizzoni and Marco Caccamo. Toward the Predictable Integration of Real-Time COTS Based Systems. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, page 73–82, 2007.

[43] Rodolfo Pellizzoni and Marco Caccamo. Impact of Peripheral-Processor Interference on WCET Analysis of Real-Time Embedded Systems. *IEEE Transactions on Computers*, 59(3):400–415, 2010.

[44] The Linux Foundation Projects. ACRN hypervisor. https://projectacrn.org Accessed: 2021-02-08.

[45] Tao Qian, Frank Mueller, and Yufeng Xin. Hybrid EDF Packet Scheduling for Real-Time Distributed Systems. In *2015 27th Euromicro Conference on Real-Time Systems*, page 37–46, 2015.

[46] RTCA Inc. RTCA/DO-178C Software Consideration in Airborne Systems and Equipment Certification, December 2011.

[47] RTCA Inc. Supporting Information for DO-178C and DO-278A, December 2011.

[48] Ignacio Sañudo, Roberto Cavicchioli, Nicola Capodieci, Paolo Valente, and Marko Bertogna. A Survey on Shared Disk I/O Management in Virtualized Environments under Real Time Constraints. *SIGBED Rev.*, 15(1):57–63, March 2018.

[49] Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. E-WarP: A System-wide Framework for Memory Bandwidth Profiling and Management. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, 2020.

[50] Marco Spuri. Analysis of Deadline Scheduled Real-Time Systems. *INRIA*, RR-2772, 1996.

[51] Rohan Tabish, Renato Mancuso, Saud Wasly, Rodolfo Pellizzoni, and Marco Caccamo. A real-time scratchpad-centric OS with predictable inter/intra-core communication for multi-core embedded systems. *Real-Time Systems*, 55, 10 2019.

[52] Rohan Tabish, Jen-Yang Wen, Rodolfo Pellizzoni, Renato Mancuso, Heechul Yun, Marco Caccamo, and Lui Sha. SCE-Comm: A Real-Time Inter-Core Communication Framework for Strictly Partitioned Multicore Processors. In *2020 9th Mediterranean Conference on Embedded Computing (MECO)*, page 1–6. IEEE, 2020.

[53] Richard West, Ye Li, Eric Missimer, and Matthew Danish. A Virtualized Separation Kernel for Mixed-Criticality Systems. *ACM Trans. Comput. Syst.*, 34(3), June 2016.

[54] Xen. Xen Split Driver Model. https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview Accessed: 2021-02-08.

[55] Xilinx. Xilinx Versal. https://www.xilinx.com/products/silicon-devices/acap/versal.html Accessed: 2021-02-08.

[56] Xilinx. Xilinx Xen Support with Cache-Coloring. https://github.com/Xilinx/xen/commits/xilinx/release-2020.2. Accessed: 2021-02-08.

[57] Xilinx. ZCU 102 MPSoC TRM. https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf.

[58] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, page 155–166, 2014.

[59] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory Bandwidth Management for Efficient Performance Isolation in Multi-Core Platforms. *IEEE Transactions on Computers*, 65(2):562–576, 2016.