Real-Time Systems manuscript No.
(will be inserted by the editor)

Mixed-Criticality Scheduling on Cluster-Based Manycores with Shared Communication and Storage Resources

Georgia Giannopoulou · Nikolay Stoimenov · Pengcheng Huang · Lothar Thiele · Benoît Dupont de Dinechin

Received: date / Accepted: date

Abstract The embedded system industry is facing an increasing pressure for migrating from single-core to multi- and many-core platforms for size, performance and cost purposes. Real-time embedded system design follows this trend by integrating multiple applications with different safety criticality levels into a common platform. Scheduling mixed-criticality applications on today's multi/many-core platforms and providing safe worst-case response time bounds for the real-time applications is challenging given the shared platform resources. For instance, sharing of memory buses introduces delays due to contention, which are non-negligible. Bounding these delays is not trivial, as one needs to model all possible interference scenarios. In this work, we introduce a combined analysis of computing, memory and communication scheduling in a mixed-criticality setting. In particular, we propose: (1) a mixed-criticality scheduling policy for cluster-based many-core systems with *two* shared resource classes, i.e., a shared multi-bank memory within each cluster, and a network-on-chip for inter-cluster communication and access to external memories; (2) a response time analysis for the proposed scheduling policy, which takes into account the interferences from the two classes of shared resources; and (3) a design exploration framework and algorithms for optimizing the resource utilizations under mixed-criticality timing constraints. The considered cluster-based architecture model describes closely state-of-the-art many-core platforms, such as the Kalray MPPA®-256. The applicability of the approach is demonstrated with a real-world avionics application. Also, the scheduling policy is compared against state-of-the-art scheduling policies based on extensive simulations with synthetic task sets.

Keywords mixed criticality scheduling · resource contention · shared memory · NoC · multi-core / many-core systems

Georgia Giannopoulou · Nikolay Stoimenov · Pengcheng Huang · Lothar Thiele
Computer Engineering and Communication Networks Laboratory, ETH Zurich, 8092 Zurich, Switzerland
E-mail: {firstname.lastname}@tik.ee.ethz.ch

Benoît Dupont de Dinechin
Kalray S.A., F38330 Montbonnot Saint Martin, France
E-mail: benoit.dinechin@kalray.eu

1 Introduction

Following the prevalence of multi-core systems in the electronics market, the field of embedded systems has experienced an unprecedented trend towards integrating multiple applications into a single platform. This applies even to real-time embedded systems for safety-critical domains, such as avionics and automotive. Applications in these domains are usually characterized by several criticality levels, known as *Safety Integrity Levels* or *Design Assurance Levels*, which express the required protection against failure.

For the integration of *mixed-criticality* applications into a common platform, many scheduling approaches have been proposed in the research literature. However, most of them do not explicitly address the timing effects of resource sharing. Moreover, existing industrial certification standards require complete timing *isolation* among applications of different criticalities. For this, system designers rely mainly on operating system and hardware-level partitioning mechanisms, e.g., based on the ARINC-653 standard [6]. No existing standards, however, specify how isolation is achieved when several cores share platform resources. Obviously, if several cores access synchronously a memory bus, timing interference among applications of different criticalities cannot be avoided. Also, the resulting blocking delays cannot be always bounded, since the certification authorities for applications of a particular criticality level typically do not possess any information about applications of lower criticality that are co-hosted on the platform. But even if this information is available, the problem of bounding the contention-induced delays is notoriously difficult since micro-architectural features, such as the resource arbitration policy, the access overheads, the memory sub-system organisation, have to be known and accounted for under all interference scenarios.

Several commercial many-core platforms are cluster-based, e.g., the Kalray MPPA®-256 [15] and the STHorm/P2012 [31]. A cluster usually consists of several cores with a local shared memory and a private address space, while clusters are connected by specialized networks-on-chip (NoC). On such architectures, tasks can be delayed when accessing local cluster shared resources not only because of concurrently executing tasks mapped to the same cluster, but also because of data being received/sent from/to other clusters. Typically, the data arriving from other clusters are written to the local cluster memory with the highest priority, thus introducing timing delays to all tasks that try to access the local memory at the same time. Currently, no mixed-criticality scheduling and analysis methods exist that address such interference effects present in modern cluster-based architectures.

This article extends the state-of-the-art by proposing a combined computation, memory, and communication analysis and optimization framework for mixed-criticality systems deployed on cluster-based platforms. In particular, the main contributions of the article can be summarized as follows:

- An architecture abstraction for cluster-based manycores with shared computing (processing cores), storage (local cluster memory, external DDR memory), and communication (NoC) resources is proposed.
- A mixed-criticality multicore scheduling policy is proposed. It follows a flexible time-triggered criticality-monotonic scheme, which enforces isolation between applications of different criticality levels and allows interference on the shared

communication and storage infrastructure only among applications of the same criticality that run in parallel in a cluster.
- A worst-case response time (WCRT) analysis for the flexible time-triggered scheduling policy is proposed. It accounts for the blocking delays not only due to concurrently executed tasks within the same cluster, but also due to NoC data transfers. We assume that NoC flows are statically routed and regulated at the source node [30]. The NoC is modeled and analysed using network and real-time calculus [27,41].
- A heuristic approach for finding an optimized mapping of mixed-criticality task sets to the cores of a cluster with the flexible time-triggered scheduling is proposed. It accounts for the interferences on the shared cluster memory due to both concurrently executed tasks and inter-cluster NoC communication.
- A heuristic approach for finding an optimized partitioning of task data to the memory banks of a cluster is proposed. It also accounts for the interferences on the shared cluster memory banks due to both concurrently executed tasks and inter-cluster NoC communication.
- The two inter-dependent heuristic approaches are efficiently combined which allows to find optimized mappings of tasks to cores and data to memory banks such that the workload distribution is balanced among cores and interference effects are minimized within a cluster.
- We demonstrate the applicability and efficiency of the design optimization approaches as well as the effect of memory sharing and inter-cluster communication on mixed-criticality schedulability using a real-world avionics application. We, also, compare the efficiency of the proposed scheduling policy against state-of-the-art policies based on extensive simulations with synthetic task sets. The proposed policy can outperform the compared policies for harmonic workloads.

The article is organised as follows. Sec. 2 provides an overview of recent publications concerning mixed-criticality scheduling and resource interference analysis. Sec. 3 introduces the considered mixed-criticality task model and the many-core architecture abstraction. Sec. 4 describes the flexible time-triggered scheduling policy (FTTS) for multicore mixed-criticality systems. Sec. 5 presents a method for response time analysis under FTTS, which considers explicitly the delays that each task suffers due to contention on the shared memory path by concurrently executed tasks within a cluster and by incoming NoC traffic. Sec. 6 suggests heuristic optimization approaches for (i) mapping tasks to the cores of a cluster, (ii) mapping data to memory banks, and (iii) an integrated approach for solving these two inter-dependent problems. In Sec. 7 we apply the developed optimization methods to a real-world case study and in Sec. 8 we compare the FTTS with other state-of-the-art mixed-criticality scheduling policies. Sec. 9 concludes the article. Note that to facilitate reading, a summary of the most important notations is included in the Appendix. Also, Fig. 11 presents an overview of the design optimization flow and how the results of the individual sections are integrated into it.

2 Related Work

Mixed-Criticality Scheduling. Scheduling of mixed-criticality applications is a research field attracting increasing attention nowadays. After the original work of Vestal [44], which introduced the currently dominating mixed-criticality task model, several scheduling policies were proposed for both single-core and multi-core systems, e.g., [10,7,9,8,35]. For an up-to-date compilation and review of these policies, we refer the interested reader to the study of Burns and Davis [11].

Among the policies for multicores, we highlight the ones that were designed to target strict global timing isolation among criticality levels, which is a common requirement of certification authorities. Anderson et al. proposed scheduling mixed-criticality tasks on multicores, by adopting different strategies (partitioned EDF, global EDF, cyclic executive) for different criticality levels and utilizing a bandwidth reservation server for timing isolation [5,33]. Tamas-Selicean and Pop presented an optimization method for task partitioning and time-triggered scheduling on multicores [40], complying with the ARINC-653 standard [6], the objective being the minimization of the certification cost. These works along with most existing multicore scheduling policies, however, do not address explicitly the interference when tasks access synchronously shared platform resources and its effect on schedulability. We believe that this can be dangerous since Pellizzoni et al. have shown empirically that traffic on the memory bus in commercial-off-the-shelf systems can increase the response time of a real-time task up to 44% [36]. Also, in Sec. 7, we show that platform parameters, such as the memory or NoC latency or the internal memory organisation, have a significant effect on the schedulability of mixed-criticality task sets.

The scheduling policy of Giannopoulou et al. [20] seems to be one of the first to explicitly consider the effect of resource contention on the memory bus. Based on a flexible time-triggered scheduling scheme and barrier synchronization among cores (FTTS), this policy enables only a statically known set of tasks of the same criticality level to be executed in parallel and interfere on the shared memory bus. The required timing isolation between tasks of different criticality is achieved despite resource sharing, without any need for hardware support. In a subsequent work, Giannopoulou et al. [21] present a design optimization framework for the deployment of mixed-criticality applications under the FTTS policy. The target platforms are multicores with private caches and shared access to a global multi-banked memory. On such platforms, interference occurs on bank level. The optimization regards the static mapping of tasks to computation cores and the mapping of the tasks' private data and communication buffers to memory banks, such that core utilizations are balanced and minimized.

The work in this article is based upon FTTS. The presented results extend and unify the response time analysis and the design optimization methods presented in [20] and [21]. The methods are extended towards cluster-based architectures where a task can experience interference not only from tasks executing concurrently in the same cluster, but also from inter-cluster NoC communication of data being read/written from/to the cluster. Therefore, real-time properties of the NoC flows such as delay and burst characteristics are computed using network calculus, and in-

tegrated into the response time analysis and design optimization methods.

Mixed-Criticality Resource Sharing. Recent works have targeted at bounding/ minimizing the delay that high-criticality tasks suffer due to contention on shared resources, while assuming partitioned task scheduling under traditional (single-criticality) policies, e.g., fixed priority. These methods differ from FTTS in nature, since they accept interference among tasks of different criticality levels as long as it is bounded. FTTS, on the contrary, allows interference on the shared resources among tasks of equal criticality without any form of budgeting. Through dynamic inter-core synchronization, it enables timing isolation among different criticalities, hence the delay imposed to high-criticality tasks by lower-criticality ones is invariably zero.

With regards to shared memory, Yun et al. [50] and Flodin et al. [18] proposed different software-based memory throttling mechanisms (with predefined [50] or dynamically allocated [18] per-core budgets) to explicitly control the inter-core interference. Hardware solutions have also been suggested. Paolieri et al. [34] and Goossens et al. [22] suggested novel memory controller designs for mixed hard real-time and soft real-time systems. These methods require special hardware support as opposed to memory throttling and FTTS. With regards to the NoC infrastructure, Tobuschat et al. [43] implement virtualization and monitoring mechanisms to provide independence among flows of different criticality. Particularly, using back suction [17], they target at maximizing the allocated bandwidth for lower criticality flows, while providing guaranteed service to the higher criticality flows. In our work, we assume real-time guarantees for all NoC flows. However, our work can be also combined with mixed-criticality NoC guarantees, as in the work of Tobuschat et al. [43].

Data Partitioning. In this work, we also address the problem of mapping task data to the banks of a shared memory in order to optimize the memory utilization and minimize the interference among tasks of the same criticality. In the same line, Kim et al. [25] and Mi et al. [32] proposed heuristics for mapping data of different application threads to DRAM banks to reduce the average thread execution times. Contrary to our work, these methods do not provide any real-time guarantees. Liu et al. implemented in [29] a bank partitioning mechanism by modifying the memory management of the operating system to adopt a custom page-coloring algorithm for the data allocation to banks, the objective being throughput maximization. Closer to our objective lies the work of Yun et al. [49], where the authors implemented a DRAM bank-aware memory allocator, using the page-based virtual memory system of the operating system to allocate memory pages of different applications / cores to private banks. The target is performance isolation in real-time systems, since partitioning DRAM banks among cores eliminates bank conflicts. In our work, we assume that bank sharing is inevitable, since tasks share access to buffers for communication purposes. We try to minimize the bank conflicts, though, through a combination of task scheduling and data mapping optimization. Note, that mechanisms like in [29,49] can be used to implement the memory mapping decisions of our design optimization method. Finally, the works of Reineke et al. [39] and Wu et al. [48] rely on DRAM controllers to implement bank privatization schemes, where each core accesses its own banks. Similar to the work of Yun et al. [49], such controllers can ensure performance isolation, however they do not consider data sharing among tasks running on

different cores. Additionally, they are hardware solutions, not applicable to commercial off-the-shelf platforms.

3 System Model

This section defines the task and platform model as well as the requirements that a mixed-criticality scheduling strategy must fulfill for certifiability. The task model and requirements are based on the established mixed-criticality assumptions in literature, but also on an avionics case study we addressed in the context of an industrial collaboration. A description of the avionics application is presented later in Sec. 7. The platform model is inspired mainly by the Kalray MPPA®-256 architecture [16]. An overview of this architecture is provided in Sec. 3.2.

3.1 Mixed-Criticality Task Model

We consider mixed-criticality periodic task sets $\tau = \{\tau_1, \ldots, \tau_n\}$ with criticality levels among 1 (lowest) and L (highest). A task is characterized by a 5-tuple $\tau_i = \{W_i, \chi_i, \mathbf{C}_i, C_{i,deg}, \mathcal{D}ep\}$, where:

- $W_i \in \mathbb{N}^+$ is the task period.
- $\chi_i \in \{1, \ldots, L\}$ is the criticality level.
- $\mathbf{C}_i$ is a size-L vector of execution profiles, where $C_i(\ell) = (e_i^{min}(\ell), e_i^{max}(\ell), \mu_i^{min}(\ell), \mu_i^{max}(\ell))$ represents a lower and an upper bound on the execution time (e_i) and number of memory accesses (μ_i) of τ_i at level of assurance $\ell \leq \chi_i$. Note that execution time e_i denotes the computation or CPU time of τ_i, *without* considering the time spent on fetching data from the memory. Such decoupling of the execution (computation) time and the memory accessing time is feasible on fully timing compositional platforms [47].
- $C_{i,deg}$ is a special execution profile for the cases when τ_i ($\chi_i < L$) runs in *degraded mode*. This profile corresponds to the minimum required functionality of τ_i so that no catastrophic effect occurs in the system. If execution of τ_i can be aborted without catastrophic effects, $C_{i,deg} = (0, 0, 0, 0)$.
- $\mathcal{D}ep(\mathcal{V}, \mathcal{E})$ is a directed acyclic graph representing dependencies among tasks with equal periods. Each node $\tau_i \in \mathcal{V}$ represents a task of τ. A weighted edge $e \in \mathcal{E}$ from τ_i to τ_k implies that within a period the job of τ_i must precede that of τ_k. The weight $w(e)$ denotes the minimum time that must elapse from the completion of τ_i's execution until the activation of τ_j. If $w(e) = 0$, τ_j can be scheduled at earliest right after τ_i. In the following, we refer to $w(e)$ as the *minimum distance constraint*.

For simplicity, we assume that the first job of all tasks is released at time 0 and that the relative deadline D_i of τ_i is equal to its period, i. e. $D_i = W_i$. Furthermore, the worst-case parameters of $C_i(\ell)$ are monotonically increasing for increasing ℓ and the best-case parameters are monotonically decreasing, respectively. Namely, the min./max. interval of execution times and memory accesses in $C_i(\ell)$ is included in the corresponding interval of $C_i(\ell + 1)$. Note that the best-case parameters are only

needed (i) to ensure that the minimum distance constraint of dependent tasks is not violated and (ii) to obtain more accurate results from the response time analysis, as discussed in Sec. 5.

The bounds for the execution times and accesses can be obtained by different tools. For instance, at the lowest level of assurance ($\ell = 1$), the system designer may extract them by profiling and measurement, as in [36]. At higher levels, certification authorities may use static analysis tools with more and more conservative assumptions as the required confidence increases. Note that the execution profile $C_i(\ell)$ for each task τ_i is derived only for $\ell \leq \chi_i$. For $\ell > \chi_i$, there is no valid execution profile since certification at level ℓ ignores all tasks with a lower criticality level. At runtime, if a task with criticality level greater than χ_i requires more resources than initially expected, then τ_i may run in degraded mode with execution profile $C_{i,deg}$.

The motivation behind defining a degraded execution profile, $C_{i,deg}$, is that in safety-critical applications, tasks typically cannot be aborted due to safety reasons. Several prior mixed-criticality scheduling policies in the literature assume, nonetheless, that if a higher criticality task requires at runtime more resources than initially assigned to it (according to some optimistic resource allocation), then all lower criticality tasks can be aborted from that point on, permanently or temporarily (see study [11]). In our work, we assume that each task has a minimal functionality that *must* be executed under all circumstances so that no catastrophic effect occurs in the system. The corresponding execution requirements ($C_{i,deg}$) must be fulfilled by any mixed-criticality scheduling policy.

Finally, the dependency graph $\mathcal{D}ep(\mathcal{V}, \mathcal{E})$ is a common consideration in scheduling to model e.g., data dependencies among tasks. In our work, we introduce the minimum distance constraint for any dependent task pair. This is necessary to model scheduling constraints that stem from inter-cluster communication through a NoC in cluster-based architectures. Such constraints are discussed later in Sec. 3.2 and 6.1.

3.2 Resource-Sharing Platform Model

We consider a cluster $\mathcal{P}$ of m processing cores, $\mathcal{P} = \{p_1, \ldots, p_m\}$. Here, the cores are identical but there are no obstacles to extend our approach to heterogeneous platforms. The mapping of the task set τ to the cores in $\mathcal{P}$ is defined by function $\mathcal{M}_\tau : \tau \rightarrow \mathcal{P}$. Note that $\mathcal{M}_\tau$ is *not* given, but it will be determined by our approach in Sec. 6.

Each core in $\mathcal{P}$ has access to a private cache memory (we restrict our interest to data caches, denoted by 'D' in Fig. 1), to a shared RAM memory and to an external DDR memory, which is local to another cluster. The shared cluster memory is organized in several banks. Each bank must have a sequential address space (not interleaved among banks) to limit potential inter-task interference. Under this assumption, two concurrently executed tasks on different cores can perform parallel accesses to the shared memory without delaying each other provided that they access different banks. We assume that each memory bank has a dedicated request arbiter. Also, each core has a private path (bus) to the shared memory. The private paths of the cores are connected to all bank arbiters, as depicted abstractly (for Arb1) in Fig. 1. For the

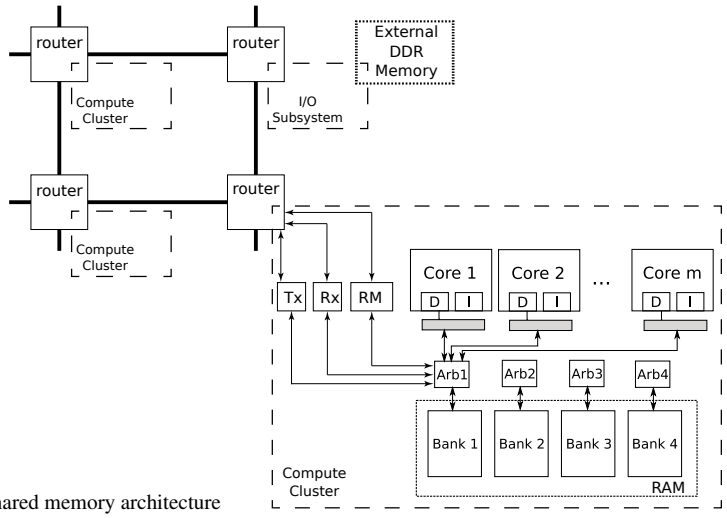


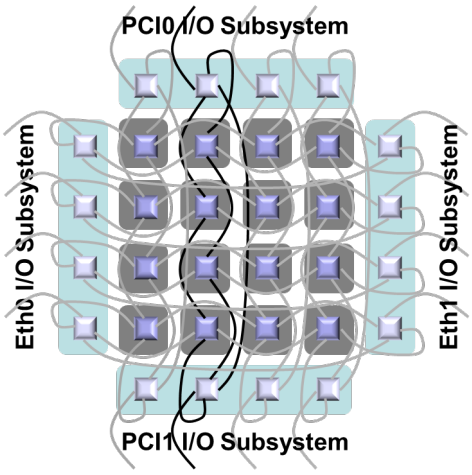
Fig. 1 Shared memory architecture

bank arbitration, we consider the class of round-robin-based policies, potentially with higher priority for some bank masters other than the cores in $\mathcal{P}$ (if such exist), e.g., the Rx interface in Fig. 1. We assume that only one core can access a bank at a time and that once granted, a bank access is completed within a fixed time interval, T_{acc} (same for read/write operations and for all banks). In the meantime, pending requests to the same bank from other cores stall execution on their cores until they are served.

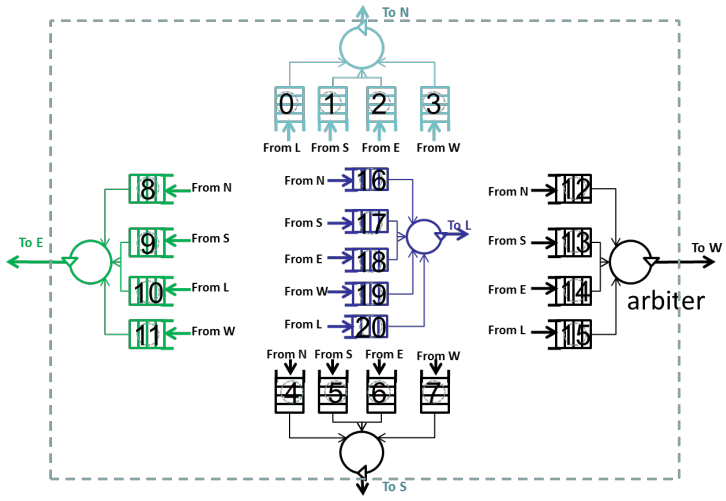
Additionally, the cores in $\mathcal{P}$ have access to external memories of remote clusters, which they access through a Network-on-Chip (NoC). Contention may occur in the NoC on router level every time two flows (virtual channels) need to be routed to the same outgoing link. Again, the assumed arbitration policy is round-robin at packet level.

The discussed abstract architecture model fits very well commercial manycore platforms, such as the Kalray MPPA®-256 [16] and the STHorm/P2012 [31]. In the remainder of the paper we motivate our models and methods based on the former architecture; therefore we look at it into greater detail in the following. Note that our response time analysis in Sec. 5 is valid for hardware platforms without timing anomalies, such as the fully timing compositional architecture which is defined in [47]. On such architectures, a locally worse behavior (e.g., a cache miss instead of a cache hit) cannot lead to a globally better effect for a task (e.g., reduced worst-case response time). The absence of timing anomalies allows the decoupling of execution and communication times during timing analysis. For a more detailed discussion on the property of timing compositionality, the interested reader is referred to [47] and for a rigorous definition of the term in resource-sharing systems to [23], respectively. The MPPA®-256 cores have been shown to be fully timing compositional [16].

Our response time analysis and optimization methods could be also extended to cover other arbitration policies, e.g. the first-ready first-come-first-serve, which seems to be a common policy on COTS multicores [24], on condition that the assumption



(a) MPPA®-256 D-NoC topology



(b) MPPA®-256 D-NoC router model

Fig. 2 MPPA®-256 D-NoC topology and router model

of timing compositionality still holds.

Kalray MPPA® Architecture. The Kalray MPPA®-256 Andey processor integrates 256 processing cores and 32 resource management cores (denoted by 'RM' in Fig. 1 and 3), which are distributed across 16 compute clusters and four I/O sub-systems. Each compute cluster includes 16 processing cores and one resource management core, each with private instruction and data caches. The processing cores and the management core implement the same VLIW architecture. However, the management core is distinguished by its connection to the NoC interfaces. Each I/O subsystem includes four resource management cores that share a single data cache, and

no processing cores. Application code is executed on the compute clusters (processing cores), whereas the I/O sub-systems are dedicated to the management of external DDR memories, Ethernet I/O devices, etc. Each compute cluster and I/O subsystem owns a private address space. The DDR memory is only visible in the address space of the resource management cores of the I/O sub-system. Communication and synchronization among compute clusters and I/O sub-systems is supported by two explicitly routed, parallel networks-on-chip, the data (D-NoC) and the control (C-NoC) network-on-chip. Here, we consider only the D-NoC. This is dedicated to high-bandwidth data transfers and may operate with guaranteed services, thanks to non-blocking routers with flow regulation at the source nodes [30], which is an important feature for the deployment of safety-critical applications.

Fig. 2(a) presents an overview of the MPPA®-256 architecture and the D-NoC topology. Each square in the figure corresponds to a switching node and interface of the D-NoC, for a total of 32 nodes: one per compute cluster (16 internal nodes) and four per I/O sub-system (16 external nodes). The I/O sub-systems are depicted on the four sides. The NoC topology is based on a 2D torus augmented with direct links between I/O sub-systems. Fig. 2(b) depicts the internal structure of a D-NoC router. The MPPA®-256 routers multiplex flows originating from different directions (input ports). Each originating direction (north N, south S, west W, east E, local node L) has its own FIFO queue at the output interface, so flows interfere on a node only if they share a link (output port) to the next node. This interface performs a round-robin arbitration at the packet granularity between the FIFOs that contain data to send on a link. NoC traffic through a router interferes with the memory buses of the underlying I/O sub-system or compute cluster only if the NoC node is a destination for the transfer.

Each of the compute clusters and the I/O sub-systems have a local on-chip memory, which is organized in 16 independent banks with a dedicated access arbiter for each bank. In the compute clusters, this arbiter always grants access to data received (Rx) from the D-NoC. That is, if an access request from D-NoC Rx arrives, it will be immediately served after the current access to the memory bank is completed. The remaining bandwidth is allocated to two groups of bus masters in round-robin fashion. The first group comprises the resource management core, the debug support unit, and a DMA engine dedicated to data transmission (Tx) over the D-NoC. The second group is composed by the processing cores. Inside each group, the allocation policy is also round-robin. In practice, one may abstract the arbitration policy of memory banks as illustrated in Fig. 3, where the debug support unit is omitted for simplicity.

Within a compute cluster, the memory address mapping can be configured either as interleaved or as sequential. In the sequential address configuration, each bank spans 128 KB consecutive addresses. This is the assumed configuration in our work. By using linker scripts, one can statically map private code and data of each processing onto the different memory banks. This guarantees that no interference between processing cores occurs on the memory buses or the arbiters of the memory banks. Such memory mapping optimization to eliminate inter-core interference will be considered in Sec. 6.

When a processing core from a compute cluster requires access to an *external DDR memory*, this is achieved through the I/O sub-systems, since compute clusters

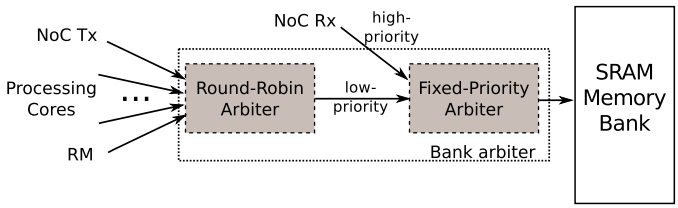


Fig. 3 Memory bank request arbitration in an MPPA®-256 cluster

do not have direct access to the external memories. Each I/O sub-system includes a DDR memory controller, which arbitrates the access among different initiators according to a round-robin policy. The initiators include, among others, the D-NoC interfaces and the resource management cores of the I/O sub-systems.

Communication Protocol between a Compute Cluster and an I/O sub-system. In the remainder of this paper, we consider the processing cores of one MPPA®-256 compute cluster as the core set $\mathcal{P}$, introduced earlier in the abstract model. The cores in $\mathcal{P}$ share access to the SRAM memory banks of the cluster and can transfer data from/to an external DDR memory over the D-NoC. For the data transfer, we consider a specific protocol. This is illustrated in Fig. 4 and described below:

- For each periodic task $\tau_i \in \tau$, which requires data from an external memory, there is a preceding task $\tau_{init,i}$ with the same period and criticality level, which initiates the data transfer. Specifically, $\tau_{init,i}$ communicates with a dedicated listener task which is executed in an I/O sub-system with access to the target DDR. $\tau_{init,i}$ sends a notification to the listener, including all relevant information for the DDR access, e.g., base address in DDR, data length, base address of allocated space in cluster memory. To send the notification, $\tau_{init,i}$ activates the cluster's NoC Tx interface. The transfer is asynchronous, i.e., $\tau_{init,i}$ can complete its execution after sending the notification, without expecting any acknowledgement of its reception. We denote the maximum time required for the transmission of the notification packet(s) over the D-NoC as *worst-case notification time* (WCNT). This time is computed by our response time analysis method is Sec. 5.2.
- Upon reception of the notification, the remote listener **(i)** decodes the request, **(ii)** allocates a D-NoC Tx DMA channel and a flow regulator, **(iii)** sets up the D-NoC Tx DMA engine with the transfer parameters, and **(iv)** initiates the transfer. From there the DMA engine will transmit the data through the D-NoC to the target compute cluster. We denote the maximum required time interval for actions **(i)**-**(iv)** as *worst-case remote set-up time* (WCRST). We assume that the WCRST can be derived based on measurements on the target platform.
- The transmitted packets follow a pre-defined route on the D-NoC before they are written to the local cluster memory by the cluster's NoC Rx interface. We denote the maximum time required for the data transmission over the D-NoC as *worst-case data fetch time* (WCDFT). We show how to compute this time for pre-routed and regulated flows in Sec. 5.2.

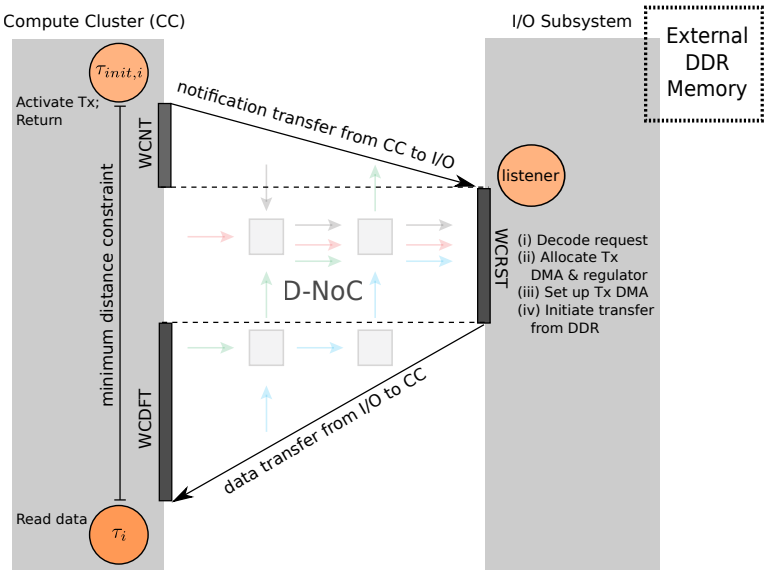


Fig. 4 Communication protocol for reading data from external DDR memory

Note that for the considered protocol, $\tau_{init,i}$ should be scheduled early enough so that the required data are already in the cluster memory when τ_i is activated. This implies that for every pair of $(\tau_{init,i}, \tau_i)$ in τ, where $\tau_{init,i}$ initiates a data transfer from a remote memory for τ_i to use these data, an edge between $\tau_{init,i}$ and τ_i must exist in the dependency graph $\mathcal{D}ep$. The edge is weighted by the minimum distance constraint, which in this case, equals the sum of the worst-case notification time, WCNT, the worst-case remote set-up time, WCRST, and the worst-case data fetch time, WCDFT.

The communication protocol has been described for data transfer between a compute cluster and an I/O sub-system. Note, however, that a similar procedure takes place also for inter-cluster data exchange. Note also that for sending data to a remote memory, a task can directly initiate the transfer through the cluster's NoC Tx interface or by setting up a DMA transfer over the D-NoC. The transfer is assumed asynchronous and no handshaking with the remote cluster is required.

3.3 Mixed-Criticality Scheduling Requirements

Under the above system assumptions, we seek a *correct* scheduling strategy for the mixed-criticality task set τ on $\mathcal{P}$, which will enable *composable* and *incremental certifiability*. We define below the properties of correctness, composable and incremental certifiability, which are crucial for a successful and economical certification process.

Definition 1 A scheduling strategy is *correct* if it schedules a task set τ such that the provided schedule is admissible at all levels of assurance. A schedule of τ is *admissible* at level ℓ if and only if:

- the jobs of each task τ_i, satisfying $\chi_i \geq \ell$, receive enough resources between their release time and deadline to meet their real-time requirements according to execution profile $C_i(\ell)$,
- the jobs of each task τ_i, satisfying $\chi_i < \ell$, receive enough resources between their release time and deadline to meet their real-time requirements according to execution profile $C_{i,deg}$. □

The term *resources*, in this context, refers to both processing time and communication time for accessing the shared memory and NoC.

Definition 2 A scheduling strategy enables *composable certifiability* if all tasks of a criticality level ℓ are temporally isolated from tasks with lower criticality, for all $\ell \in \{1, \ldots, L\}$. Namely, the execution and access activities of a task τ_i must not delay in any way any task with criticality level greater than χ_i. □

The requirement for composability enables different certification authorities to certify task subsets of a particular criticality level ℓ even without any knowledge of the tasks with lower criticality in τ. This is important when several certification authorities need to certify not the whole system, but individual parts of it. Each authority needs information on the scheduling of tasks with higher criticality level than the one considered. Such information can be provided by the responsible authorities for the higher-criticality task subsets.

Definition 3 A scheduling strategy enables *incremental certifiability* if the real-time properties of the tasks at all criticality levels $\ell \in \{1, \ldots, L\}$ are preserved when new tasks are added to the system. □

This property implies that if the schedule of a task set τ is certified as admissible, the certification process will not need to be repeated if new tasks are added later to the system. This is reasonable, since repeating the certification process of already certified tasks if the system is designed incrementally results in excessive costs.

Note that the above notion of correctness (Definition 1) is not new in mixed-criticality scheduling theory. On the other hand, the requirements for composable and incremental certifiability seem to be crucial in safety-critical domains, e.g., avionics, for reducing the effort and cost of certification. Nonetheless, they are usually not considered explicitly in mixed-criticality scheduling literature (see study [11]). Exceptions are the works that are based on temporal partitioning as defined in the ARINC-653 standard [6], such as the approach of Tamas-Selicean and Pop [40], and the works that use servers for performance isolation among applications of different criticality levels, such as the approach of Yun et al. [50].

4 Flexible Time-Triggered Scheduling

This section discusses briefly the Flexible Time-Triggered and Synchronisation-based (FTTS) mixed-criticality scheduling policy for multicores. The reader is referred to [20] for a more detailed presentation. In this section, we assume that an FTTS schedule for a particular task set and platform is *given*. For the given schedule,

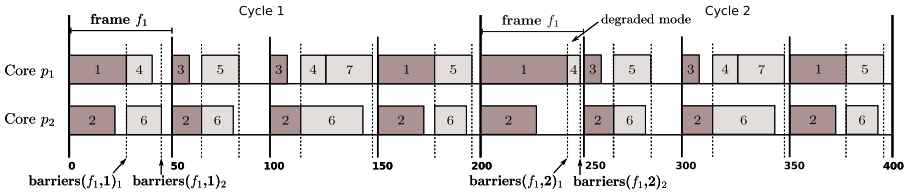


Fig. 5 Global FTTS schedule for 2 cycles (dark annotation: criticality level 2, light: criticality level 1)

we describe the runtime behavior of the scheduler and introduce useful notation. We show how to determine an FTTS schedule (when it is *not given*) later, in Sec. 6.1.

The non-preemptive FTTS scheduling policy combines time and event-triggered task activation. A global FTTS schedule repeats over a *scheduling cycle* equal to the hyper-period H of the tasks in τ. The scheduling cycle consists of fixed-size *frames* (set $\mathcal{F}$). Each frame is divided further into L flexible-length *sub-frames*. The beginning of frames and sub-frames is synchronized among all cores in a cluster. The frame lengths can differ, but they are upper bounded by the minimum period in τ. Each sub-frame (except the first of a frame) starts once all tasks of the previous sub-frame complete execution across all cores. Synchronisation is achieved dynamically via a barrier mechanism, for the sake of efficient resource utilization. Each sub-frame contains only tasks of the same criticality level. Note that the sub-frames within a frame are ordered in decreasing order of their criticality and that within a sub-frame, tasks are scheduled sequentially on each core following a predefined order, namely every task is triggered upon completion of the previous one.

An illustration of an FTTS schedule is given in Fig. 5 for seven tasks with hyper-period $H = 200$ ms. Fig. 5 depicts two consecutive scheduling cycles. The solid lines define the frames and the dashed lines the sub-frames, i.e., potential points, where barrier synchronisation is performed. The FTTS schedule has a cycle of $H = 200$ ms and is divided into four frames of equal lengths (50 ms), each with $L = 2$ sub-frames: the first for criticality 2 (high) and the second for criticality 1 (low), respectively. A scheduling cycle includes H/W_i invocations of each task τ_i, i.e., the number of jobs of τ_i that arrive within a hyper-period.

At runtime, the length of each sub-frame varies based on the different execution times and accessing patterns that the concurrently executed tasks exhibit. For example, in Fig. 5, the first sub-frame of f_1 finishes earlier when τ_1, τ_2 run w.r.t. their level-1, i.e., low-criticality profiles (cycle 1) than when at least one task runs w.r.t. its level-2, i.e., high-criticality profile (cycle 2). Despite this dynamic behavior, the sub-frame worst-case lengths can be computed offline for a given FTTS schedule by applying worst-case response time analysis under memory contention (Sec. 5).

Function $barriers : \mathcal{F} \times \{1, \ldots, L\} \rightarrow \mathbb{R}^L$ defines the worst-case length of all sub-frames in a frame, for a particular level of assurance. We denote the worst-case length of the k-th sub-frame of frame f at level ℓ as $barriers(f, \ell)_k$. Note that the k-th sub-frame of f contains tasks of criticality level $(L - k + 1)$. Also, ℓ corresponds to the highest level execution profile that the tasks of f exhibit at runtime. For $\ell > 1$, execution in certain sub-frames of f (with index $k > 1$) may be degraded.

Runtime behavior. Given an admissible FTTS schedule and the *barriers* function, the scheduler manages task execution on each core within each frame $f \in \mathcal{F}$ as follows (init., $\ell_{max} = 1$):

- For the k-th sub-frame, the scheduler triggers sequentially the corresponding jobs. Upon completion of the jobs' execution, it signals the event and waits until the remaining cores reach the barrier.
- Let the elapsed time from the beginning of the k-th sub-frame until the barrier synchronisation be t. Given ℓ_{max}:

$$\ell_{max} = \max \left\{ \underset{\ell \in \{1,\dots,L\}}{\operatorname{argmin}} \left\{ t \leq barriers(f,\ell)_k \right\}, \ell_{max} \right\}, \tag{1}$$

  the scheduler will trigger jobs in all next sub-frames such that tasks with criticality level lower than ℓ_{max} run in degraded mode.
- The two previous steps are repeated for each sub-frame, until the next frame is reached.

Note that the decision on whether a task will run in degraded mode affects only the current frame.

Admissibility. Let an FTTS schedule be constructed such that all H/W_i jobs of each task $\tau_i \in \tau$ are scheduled on the same core within their release times and deadlines and all dependency constraints hold. The FTTS schedule is ℓ-admissible if and only if it fulfills the following condition:

$$\sum_{k=1}^{L} barriers(f,\ell)_k \leq \mathcal{L}_f, \forall f \in \mathcal{F}, \tag{2}$$

where $\mathcal{L}_f$ denotes the length of frame f. If the condition holds for all frames $f \in \mathcal{F}$, all scheduled jobs in S can meet their deadlines at level of assurance ℓ. If it holds for all levels $\ell \in \{1, \cdots, L\}$, it follows that the FTTS schedule is admissible according to Definition 1 of Sec. 3.3. That is, it can be accepted by any certification authority at any level of assurance and the scheduling strategy is correct.

If different certification authorities certify task subsets of different criticality levels, then for composable certifiability (Definition 2), the authorities of lower-criticality subsets need information on the resource allocation for the higher-criticality subsets. For the FTTS scheduling strategy, this information is fully represented by function $barriers$. Therefore, global FTTS enables composable certifiability. Similarly, it enables incremental certifiability (Definition 3), since new tasks can be inserted into their respective criticality level sub-frame if there is sufficient slack time in the frame.

It follows from the above that the computation of function $barriers$ is necessary to evaluate if an FTTS schedule is admissible, but also as an interface among certification authorities and system designers. In the next section, we show how to compute this function step-by-step, considering all possible task interferences for a given FTTS schedule.

5 Response Time Analysis for the Computation of $barriers$

This section describes how to compute function $barriers$ for a given FTTS schedule. For the computation of $barriers$, we need to bound the worst-case length of each sub-frame of the FTTS schedule at every level of assurance $\ell \in \{1, \ldots, L\}$. For this, first we perform worst-case response time (WCRT) analysis for every single task that is scheduled within the sub-frame. Second, based on the results of the first step, we derive the worst-case response time of the sequence of tasks which is executed on every core within the sub-frame (per-core WCRT, CWCRT). Once the last value is computed for all cores in $\mathcal{P}$, the worst-case sub-frame length follows trivially as the maximum among all per-core WCRTs.

The challenge in the above procedure lies in the computation of an upper bound for the response time of a task in a specific sub-frame. Note that for the timing compositional architectures which we consider, such as the MPPA®-256, it is safe to bound the WCRT of a task by the sum of its worst-case execution (CPU) time and the worst-case delay it experiences due to memory accessing and communication [47]. The worst-case execution time of each task is known at different levels of assurance as part of its execution profile $\mathbf{C}$. However, to bound the second WCRT component, one needs to account for the interference on the shared cluster memory, i.e., interference among tasks running in parallel and from the NoC interface, when some of them try to access the same memory bank simultaneously. Therefore, to derive the WCRT of a task in a specific sub-frame, we need to model the possible interference scenarios based on the tasks that are concurrently executed and the NoC traffic patterns and then, to analyse the worst-case delay that such interference can incur to the execution of the task under analysis.

Sec. 5.1 shows how to bound the worst-case delay a task can experience on the shared memory path. For this, we consider as given: the mapping of tasks to the cores of $\mathcal{P}$, $\mathcal{M}_\tau$, the mapping of the tasks' data to memory banks, $\mathcal{M}_{mem}$, the incoming NoC traffic patterns, and the memory access latency, T_{acc}. Sec. 5.2 describes a method for NoC analysis, based on the network and real-time calculus [27,41], which enables us to compute a bound on all NoC incoming traffic patterns at the shared memory of a cluster. This bound is used in the analysis of Sec. 5.1.

5.1 Bounding Delay on Cluster Memory Path

Within an FTTS sub-frame, we identify two sources of delay that a task may experience on the memory path based on the platform model of Sec. 3.2:

I. Blocking on a memory bank arbiter due to contention from other access requesters, specifically any other processing core or the NoC Tx DMA interface. Since contention is resolved among these requesters in a round-robin fashion, the task under analysis will have to wait for its turn in the round-robin cycle to be granted access to the memory bank.
II. Blocking on a memory bank arbiter due to contention from the NoC Rx interface. This requester has higher priority when accessing the memory, so in the worst case, the task under analysis will have to wait for all accesses of the NoC Rx interface to be served before it can gain access to the memory bank.

In the following, we model interference on the shared memory in the form of a *memory interference graph*. Based on this graph, we compute the maximal delay that each task can cause to another when they are executed in parallel and in the presence of incoming traffic from the NoC Rx interface. This way, we are able to estimate the WCRT of each task within the FTTS sub-frame and subsequently, the global worst-case sub-frame length.

5.1.1 Memory Interference among Requesters with Equal Priorities

To model the inter-task interferences due to round-robin-arbitrated memory contention, we introduce a graph representation, called the *memory interference graph* $\mathcal{I}(\mathcal{V}, \mathcal{E})$. We define $\mathcal{V} = \mathcal{V}_\tau \cup \mathcal{V}_{BL} \cup \mathcal{V}_B$, where $\mathcal{V}_\tau$ represents all tasks in τ (running on processing cores and NoC Tx), $\mathcal{V}_{BL}$ represents all memory blocks BL accessed by τ, i.e., the tasks' instructions, data and communication buffers, and $\mathcal{V}_B$ represents all banks B of the shared memory. Each memory block (bank) node is annotated with a corresponding size (capacity) in bytes. $\mathcal{I}$ is composed by two sub-graphs: (i) the bipartite graph $\mathcal{I}_1(\mathcal{V}_T \cup \mathcal{V}_{BL}, \mathcal{E}_1)$, where an edge $e \in \mathcal{E}_1$ from $\tau_i \in \mathcal{V}_T$ to $bl_j \in \mathcal{V}_{BL}$ with weight $w(e)$ implies that task τ_i performs at maximum $w(e)$ accesses to memory block bl_j per execution, and (ii) the bipartite graph $\mathcal{I}_2(\mathcal{V}_{BL} \cup \mathcal{V}_B, \mathcal{E}_2)$, where an edge $e \in \mathcal{E}_2$ from $bl_j \in \mathcal{V}_{BL}$ to $b_k \in \mathcal{V}_B$ denotes the allocation of memory block bl_j in exactly one memory bank b_k. Note that the weighted sum over all outgoing edges of a task τ_i equals the memory access bound of its execution profile at its own criticality level, i.e., $\mu_i^{max}(\chi_i)$. The weights can be, however, refined (reduced) if WCRT analysis is performed for lower levels of assurance, $\ell < \chi_i$. In this case, the weighted sum over the outgoing edged of τ_i equals the (more optimistic) $\mu_i^{max}(\ell)$, which enables tighter WCRT analysis for the specific level of assurance.

Definition 4 Tasks τ_i and τ_j are *interfering* if and only if $\exists k, l, r \in \mathbb{N}^+ : (\tau_i, bl_k) \in \mathcal{E}_1$, $(\tau_j, bl_l) \in \mathcal{E}_1$ and $(bl_k, b_r) \in \mathcal{E}_2$, $(bl_l, b_r) \in \mathcal{E}_2$, i.e., they access blocks in the same bank. □

Fig. 6 presents a possible memory interference graph for a set of five tasks, accessing in total five memory blocks. The memory blocks can be allocated to two banks. Ellipsoid, rectangular and diamond nodes denote tasks, memory blocks, and banks, respectively. Note that for the depicted mapping of memory blocks to banks, tasks τ_1 and τ_2 are interfering, whereas τ_1 and τ_3 or τ_4 or τ_5 are not. Interfering tasks can delay each other when executed in parallel.

In the general problem setting, the mapping of memory blocks to banks, $\mathcal{M}_{mem} : BL \rightarrow B$ ($\mathcal{E}_2$ of $\mathcal{I}$), is not known, but derived by our optimization approach (see Sec. 6). Here, however, for the WCRT analysis we assume that it is fixed. Based on it, we introduce the *mutual delay matrix*, D. D is a two-dimensional matrix ($n \times n$), where $D_{i,j}$ specifies the maximum delay that task τ_i can suffer when executed concurrently with τ_j. $D_{i,j}$ is positive if τ_i and τ_j ($i \neq j$) are **(i)** of the same criticality level, i.e., potentially concurrently executing in an FTTS schedule, and **(ii)** interfering, i.e., accessing memory blocks in at least one common bank.

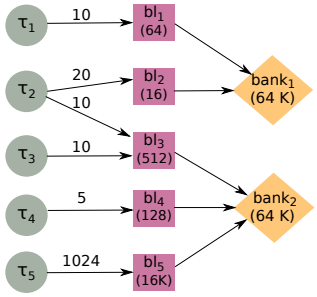


Fig. 6 Memory Interference Graph $\mathcal{I}$ for a dual-bank memory

	τ_1	τ_2	τ_3	τ_4	τ_5
τ_1	0	$10{\cdot}T_{acc}$	0	0	0
τ_2	$10{\cdot}T_{acc}$	0	$10{\cdot}T_{acc}$	0	0
τ_3	0	$10{\cdot}T_{acc}$	0	0	0
τ_4	0	0	0	0	$5{\cdot}T_{acc}$
τ_5	0	0	0	$5{\cdot}T_{acc}$	0

Table 1 Mutual delay matrix D for round-robin arbitration policy for Fig. 6

For the computation of D, we need to consider the bank arbitration policy, which in the case of MPPA®-256 and for the memory requesters that we consider is round-robin. For the round-robin policy, each access request from a task τ_i can be delayed by at most one access from any other concurrently executed task that can read/write from/to the same memory bank which τ_i is targeting. That is because we assume that each core has at most one pending request at a time (Sec. 3). In other words, τ_i can be maximally delayed by a concurrently executed task τ_j for the duration of τ_j's accesses to a shared memory bank, provided that τ_j's accesses are not more that τ_i's accesses to this bank. If τ_i and τ_j share access to more than one memory bank, the sum of potential delays across the banks has to be considered. This yields:

$$D_{i,j} = \sum_{\substack{b,bl:(\tau_i,bl)\in\mathcal{E}_1 \\ \wedge(bl,b)\in\mathcal{E}_2}} \sum_{\substack{bl':(\tau_j,bl')\in\mathcal{E}_1 \\ \wedge(bl',b)\in\mathcal{E}_2}} \min\{w((\tau_i,bl)), w((\tau_j,bl'))\} \cdot T_{acc}. \quad (3)$$

As an example, the mutual delay matrix D is given in Table 1 for the memory interference graph of Fig. 6. We assume that tasks τ_1, τ_2, τ_3 are of criticality level 2, whereas τ_4 and τ_5 of criticality level 1. D represents the worst-case mutual delays when τ_1, τ_2, τ_3 are executed in parallel in the same FTTS sub-frame. Tasks τ_4 and τ_5 are assumed to run in parallel, too, in a different sub-frame compared to τ_1-τ_3.

We use matrix D to compute the worst-case length of the k-th sub-frame of an FTTS frame f at level of assurance ℓ. According to the notation introduced in Sec. 4, the computed length corresponds to $barriers(f,\ell)_k$.

First, we compute the WCRT of every task τ_i executed in the k-th sub-frame of frame f as

$$WCRT_i(f,\ell) = e_i^{max}(\ell) + \mu_i^{max}(\ell) \cdot T_{acc} + d_i(f,\ell), \quad (4)$$

namely, as the sum of its worst-case execution time, the total access time of its memory accesses under no contention, and the worst-case delay it encounters due to contention. This last term, $d_i(f,\ell)$, is defined as:

$$d_i(f,\ell) = \min \left\{ \sum_{\tau_j \in parallel(\tau_i,f)} D_{i,j}, \quad \mu_i^{max}(\ell) \cdot (m-1) \cdot T_{acc} \right\}, \quad (5)$$

where function $parallel : \tau \times \mathcal{F} \rightarrow S_\tau$ defines a set of tasks $S_\tau \subseteq \tau$ that are executed in parallel to a task τ_i (on different cores) in frame f and m is the number of interfering requesters with equal priorities. Note that $\mu_i^{max}(\ell) \cdot (m-1) \cdot T_{acc}$ is a safe upper bound on the delay that a task can suffer due to contention under round-robin arbitration. In Eq. 5, we take the minimum of the two terms to achieve a more accurate estimation. This is useful in cases e.g., where (some of) the parallel executed tasks with τ_i are scheduled sequentially on a single core. It is then possible that not all of them can delay τ_i on the memory path. In such cases, the second bound may be tighter than the one based on matrix D.

Example. For a demonstration of the use of the above equations, let us consider the FTTS schedule of Fig. 5. In the 1st sub-frame of frame f_1, the tasks τ_1 and τ_2 with criticality level $\chi_1 = \chi_2 = 2$ are executed in parallel on $m = 2$ processing cores. Therefore, according to the definition of function $parallel$, it holds that $parallel(\tau_1, f_1) = \{\tau_2\}$, and $parallel(\tau_2, f_1) = \{\tau_1\}$. The accessing behavior of tasks τ_1 and τ_2 is described by the memory interference graph of Fig. 6. The depicted weights on the edges of the memory interference graph, i.e., the number of accesses that each task performs to the respective memory blocks, are derived at level of assurance $\ell = 2$. Based on the graph of Fig. 6, tasks τ_1 and τ_2 can interfere only on $bank_1$ because τ_1 accesses block bl_1 and τ_2 accesses block bl_2, with both blocks being mapped to the same memory bank $bank_1$. Given this, Eq. 3 yields that $D_{1,2} = D_{2,1} = 10 \cdot T_{acc}$. In other words, task τ_1 can delay τ_2 at most 10 times when accessing the shared memory, by issuing interfering access requests to the same bank. The same also holds for the maximal delay that τ_2 can cause to τ_1. These results can be seen in the mutual delay matrix D, which is already given in Table 1. At a next step, by applying Eq. 5, we compute the worst-case delay that task τ_1 can experience due to memory contention in the first frame of the FTTS schedule of Fig. 5, at level of assurance $\ell = 2$:

$$d_1(f_1, 2) = \min\{D_{1,2}, \mu_1^{max}(2) \cdot (2-1) \cdot T_{acc}\} = \min\{10 \cdot T_{acc}, 10 \cdot T_{acc}\}.$$

Similarly for task τ_2,

$$d_2(f_1, 2) = \min\{D_{2,1}, \mu_2^{max}(2) \cdot (2-1) \cdot T_{acc}\} = \min\{10 \cdot T_{acc}, 30 \cdot T_{acc}\}.$$

Hence, $d_1(f_1, 2) = d_2(f_1, 2) = 10 \cdot T_{acc}$. Namely, the WCRT of both tasks is augmented by $10 \cdot T_{acc}$ as a result of the inter-core interference on the shared memory.

Once the WCRT of all tasks in the k-th sub-frame of frame f are computed at all levels of assurance $\ell \in \{1, \ldots, L\}$, we derive the WCRT of the task sequence on each core p, $CWCRT_{p,k}(f, \ell)$, by summing up the WCRTs of the tasks that are mapped on p in the particular sub-frame. For an illustration of the notation used, please refer to Fig. 7. It follows trivially that:

$$barriers(f, \ell)_k = \max_{1 \leq p \leq m} CWCRT_{p,k}(f, \ell) \tag{6}$$

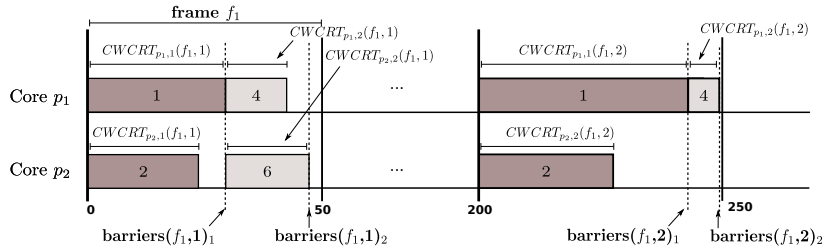


Fig. 7 Computation of $barriers(f_1, \ell)_k$ for $\ell = \{1, 2\}$ and $k = \{1, 2\}$ for the FTTS schedule of Fig. 5

5.1.2 Memory Interference from Requesters with Higher Priority

When introducing the memory interference graph in Sec. 5.1.1, we implicitly assumed that all memory blocks (tasks' instructions, private data and communication buffers) fit into the memory banks of the shared cluster memory so that the mapping of blocks to banks can be decided offline and no remote access to other compute clusters or the I/O sub-systems is required. However, in realistic applications, such as the flight management system which is used for evaluation in Sec. 7, there may be tasks which need access to databases or generally, complex data structures that do not fit in the shared memory of a compute cluster. We assume that these data structures are stored in the external DDR memories and parts of them (e.g., some database entries), which can fit into the cluster memory together with the data of the remaining tasks, are fetched whenever required.

One possible implementation of a remote data fetch protocol has been described for the MPPA®-256 platform and similar manycore architectures in Sec. 3.2. Here, we focus on its last step, namely the actual transfer of data packets from the I/O subsystem to the cluster over the NoC. During the data transfer, the NoC Rx interface tries to write to the shared cluster memory every time a new packet arrives at the cluster. Therefore, any task in the cluster attempting to access the memory bank, where the remote data are stored, will experience blocking due to the higher priority of the NoC Rx interface (see Fig. 3). In the worst case, it will have to wait for the whole remote transfer to complete before it is granted access to the memory bank.

To model the interference from the NoC Rx interface, **first**, we extend the memory interference graph, as shown in Fig. 8. The task with the bold outline represents the DMA transfer from the I/O sub-system (resp. another compute cluster) to the compute cluster. There can be arbitrarily many tasks representing DMA transfers. The weight δ of the newly added edge from `Rx` to the target memory block can be derived as the minimum between (i) the total number of fetched packets and (ii) the maximum number of packets that can be fetched over the NoC in the time interval of one frame. To compute δ for a particular data flow, we need information about the flow regulation, the flow route, and the NoC configuration. We show how to use this information to derive δ in Sec. 5.2 (Eq. 13).

Second, we update the mutual delay matrix D by setting entries $D_{j,j}$ to $\delta \cdot T_{acc}$ for all tasks τ_j that are interfering with `Rx`, as shown in Table 2. This applies to all tasks independently of their criticality level or whether they run in parallel to `Rx`, and it

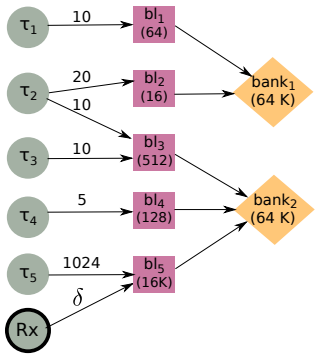


Fig. 8 Memory Interference Graph $\mathcal{I}$ for a dual-bank memory with higher-priority interference from the NoC Rx requester

	τ_1	τ_2	τ_3	τ_4	τ_5
τ_1	0	$10{\cdot}T_{acc}$	0	0	0
τ_2	$10{\cdot}T_{acc}$	$\boldsymbol{\delta \cdot T_{acc}}$	$10{\cdot}T_{acc}$	0	0
τ_3	0	$10{\cdot}T_{acc}$	$\boldsymbol{\delta \cdot T_{acc}}$	0	0
τ_4	0	0	0	$\boldsymbol{\delta \cdot T_{acc}}$	$5{\cdot}T_{acc}$
τ_5	0	0	0	$5{\cdot}T_{acc}$	$\boldsymbol{\delta \cdot T_{acc}}$

Table 2 Mutual delay matrix D for round-robin arbitration with higher priority for Rx for Fig. 8

expresses that an interfering task τ_j can be delayed by δ higher-priority requests any time it executes. The update helps during memory mapping optimization to distribute the memory block(s) to which Rx writes in disjoint banks compared to all remaining blocks.

Third, we update the computed in Sec. 5.1.1 per-core WCRTs, $CWCRT_{p,k}(f,\ell)$ for all $p \in \mathcal{P}$. Recall that $CWCRT_{p,k}(f,\ell)$ in a given FTTS schedule denotes the sum of WCRTs of the tasks that are mapped on core p in the k-th sub-frame of frame f, given the task execution profiles at level of assurance ℓ. In the following, we consider the communication protocol between a compute cluster and an I/O subsystem, as described in Sec. 3.2. For every task $\tau_i \in \tau$, which uses remote data from the DDR, and its preceding task $\tau_{init,i}$, which initiates the transfer from the I/O sub-system, let f_{prec} and f_{succ} be the FTTS frames, where $\tau_{init,i}$ and τ_i are scheduled, respectively. Since tasks τ_i and $\tau_{init,i}$ have the same criticality level, χ_i, it follows that they are scheduled in the k-th sub-frame of frames f_{prec} and f_{succ}, respectively, where $k = L - \chi_i + 1$. For instance, in a dual-criticality system ($L = 2$), if τ_i and $\tau_{init,i}$ have criticality level $\chi_i = 2$, they will be scheduled in the 1st sub-frame of their corresponding FTTS frames. For the update of the per-core WCRTs $CWCRT_{p,k}(f,\ell)$, we distinguish two cases, depending on whether frames f_{prec} and f_{succ} are equal or not. Particularly, for every core $p \in \mathcal{P}$:

- If $f_{prec} = f_{succ}$ *and* in the k-th sub-frame of f_{prec} there are tasks on p scheduled between $\tau_{init,i}$ and τ_i, which are interfering with Rx, then $CWCRT_{p,k}(f_{prec},\ell)$ is increased by $\delta{\cdot}T_{acc}$, for all $\ell \in \{1,\ldots,L\}$. In other words, if there is at least one task executing between $\tau_{init,i}$ and τ_i, which can access a common memory bank as Rx, this (these) task(s) can be delayed by the higher priority NoC Rx data transfer by up to $\delta \cdot T_{acc}$. This is accounted for by increasing the $CWCRT_{p,k}(f_{prec},\ell)$ accordingly.
- If $f_{prec} \neq f_{succ}$, then for each sub-frame k' from the k-th sub-frame of f_{prec} up to and including the k-th sub-frame of f_{succ}: if there are tasks on p other than $\tau_{init,i}$, τ_i, which are interfering with Rx, then $CWCRT_{p,k'}(f',\ell)$ for the including FTTS frame f': $f_{prec} \leq f' \leq f_{succ}$ is increased by $\delta \cdot T_{acc}$, for all $\ell \in \{1,\ldots,L\}$.

The intuition is similar as in the previous case. If any sub-frame between the one, where $\tau_{init,i}$ is scheduled and the one, where τ_i is scheduled, includes tasks that are accessing a common bank as Rx, then this (these) task(s) can be delayed by the higher priority NoC Rx data transfer by up to $\delta \cdot T_{acc}$. Note, however, that in every frame f': $f_{prec} \leq f' \leq f_{succ}$, such an increase to $CWCRT_{p,k'}(f', \ell)$ happens only once, for the first sub-frame k' that includes interfering tasks with Rx. This is done for tighter analysis[1].

Example. As an illustration of the per-core WCRT updates of the third step above, consider again the FTTS schedule of Fig. 5. Suppose that τ_4 initiates a remote data transfer for τ_5, which reads the fetched data. Both tasks have criticality level $\chi_4 = 1$. For the first instance of the dependent tasks, $f_{prec} = f_1$ (frame where τ_4 is scheduled), $f_{succ} = f_2$ (frame where τ_5 is scheduled), and $k = 2$ (corresponding sub-frame within the above frames). Since $f_{prec} \neq f_{succ}$, for the per-core WCRT updates we have to consider all FTTS sub-frames starting from the 2nd sub-frame of f_1 up to and including the 2nd subframe of f_2. In this case, there are 3 such sub-frames to be considered. We assume that the memory accessing behavior of tasks τ_1 to τ_5 and the DMA transfer Rx are modelled by the memory interference graph of Fig. 8. Tasks τ_6 and τ_7 of the FTTS schedule are *not* modelled in this graph because they perform *no* accesses to the shared memory. Given these assumptions, it follows that $CWCRT_{p_1,2}(f_1, \ell)$ and $CWCRT_{p_2,2}(f_1, \ell)$ for the 2nd sub-frame of f_1 and $\ell = \{1, 2\}$ remain unchanged. This is because on core p_1 no other task than τ_4 is scheduled, so the DMA transfer cannot cause any delay in this sub-frame on this core. Also, on core p_2, although there is a scheduled task, τ_6, this is not interfering with Rx. So, again, the DMA transfer cannot delay its execution. In contrast, $CWCRT_{p_1,1}(f_2, \ell)$ and $CWCRT_{p_2,1}(f_2, \ell)$ for the 1st sub-frame of f_2 and $\ell = \{1, 2\}$ are increased by $\delta \cdot T_{acc}$. This is because the scheduled tasks τ_3 (on core p_1) and τ_2 (on core p_2) are interfering with Rx (accessing the same memory bank $bank_2$). Finally, $CWCRT_{p_1,2}(f_2, \ell)$, $CWCRT_{p_2,2}(f_2, \ell)$ for the 2nd sub-frame of f_2 remain unchanged, since no task other than τ_5 is scheduled on p_1 and the unique task on p_2, τ_6, is not interfering with Rx.

Finally, after the updates discussed in the above three steps are completed, the computation of function $barriers$ follows from Eq. 6 for the updated $CWCRT_{p,k}(f, \ell)$ values.

5.1.3 Tighter Response Time Analysis

In Sec. 5.1.1 and 5.1.2, we derived closed-form expressions for the worst-case sub-frame lengths of the FTTS schedule (function $barriers$). Although several sources of pessimism were avoided, there may be still cases where the computed bounds are not tight. E.g., if the memory accesses of the tasks follow certain patterns (dedicated access phases, non-overlapping in time) such that even if two tasks are executed in

[1] Given the definition of δ, Rx cannot perform more than δ high-priority memory accesses within one single frame. Therefore, it is too pessimistic to increase $CWCRT_{p,k'}(f', \ell)$ for several sub-frames k' of the same frame f'. This would lead to a potential increase of $barriers(f', \ell)_L$ by multiples of $\delta \cdot T_{acc}$.

parallel, they cannot interfere on the memory path, then the given bounds do not reflect this knowledge. If more accurate response time analysis is required, the method of [19], which uses a model of the system with timed automata [4] and model checking to derive the tasks' WCRT, is suggested. The system model in [19] specifies shared-memory multicores, however it can be easily extended to model also the incoming traffic from the NoC, as computed in Sec. 5.2 (Eq. 11).

Since a model checker explores exhaustively all feasible resource interference scenarios, the above method is highly complex. Therefore, during design optimization (Sec. 6), where we need to compute function $barriers$ for often thousands of potential FTTS schedules, it is preferred to use the WCRT bounds as derived earlier. We can then apply the method of [19] to the optimized FTTS solution to refine the computation of $barriers$. Also, if no admissible FTTS schedule can be found during optimization, the same method can be applied to the best encountered solutions, as the more accurate computation of $barriers$ may reveal admissible schedules.

5.2 Bounding Delay for Data Transfers over NoC

This section shows how to characterize the incoming NoC traffic at a shared cluster memory. Based on the incoming traffic model, we compute upper bounds on (i) the delay for transferring a given amount of packets over a NoC and (ii) the number of accesses that the NoC Rx interface performs to the shared cluster memory in a given time interval. The first result (Eq. 12) is used for defining the minimum distance constraint between a task initiating a remote data transfer, $\tau_{init,i}$, and a task using the fetched data, τ_i, in the remote fetch protocol of Sec. 3.2. The second result (Eq. 13) is used as a parameter of the memory interference graph, representing the maximal interference from the NoC Rx interface, as discussed in Sec. 5.1.2.

We consider an explicitly routed NoC with wormhole switching and assume that each traffic stream uses a dedicated predetermined virtual channel throughout the NoC which is in line with previous approaches, see [51]. Each NoC node is a router and also a flow source / sink. Routers contain only FIFO queues, with one set of queues per outgoing link, with round-robin arbitration. Routers are work-conserving, i.e., no idling if data is ready to be transmitted.

All flows are (σ, ρ) regulated at the sources [30], and the parameters of the regulators are selected such that performance guarantees are provided for all flows, and no FIFO queue in a router can overflow, therefore, stalls due to backpressure flow control are not present. However, we assume that stalls due to switch contention are present, i.e., when packets from different input ports or virtual channels compete for the same output port. Such assumptions simplify the presented NoC analysis, however, if necessary, backpressure stalls can be easily integrated by using existing results, see [13, 42, 51].

Since we consider hard real-time guarantees on the NoC, we have to show that each network packet in a certain flow needs to be delivered to its destination within a fixed deadline. For the analysis, we use the theory of Network and Real-time calculus [14, 27, 41]. It is a theory of deterministic queuing systems for communication networks and scheduling of real-time systems. Network calculus has been applied in

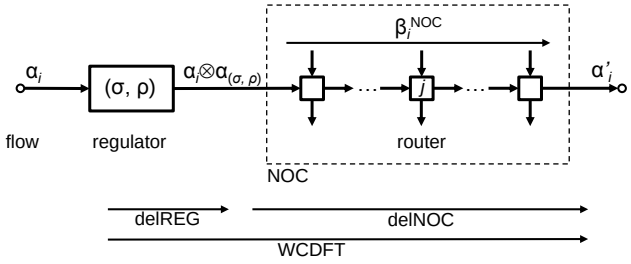


Fig. 9 Modeling the packet flow i through a (σ, ρ)-regulator and a sequence of routers j.

recent works and its effectiveness has been validated for the analysis of NoC-based systems, see [37,38,51].

The theory analyzes the flow of packet streams through a network of processing and communication resources in order to compute worst-case backlogs, end-to-end delays, and throughput. The overall modelling approach is shown in Fig. 9.

A General Packet Stream Model. Packet streams are abstracted by the function $\alpha(\Delta)$, i. e. an upper arrival curve which provides an upper bound on the number of packets in *any* time interval of length $\Delta \in \mathbb{R}$ where $\alpha(\Delta) = 0$ for all $\Delta \leq 0$. Arrival curves substantially generalize conventional stream models such as sporadic, periodic or periodic with jitter. Note, a packet here is defined as a fixed-length basic unit of network traffic. Variable-length packets can be viewed as a sequence of fixed-length packets.

A General Resource Model. The availability of processing or communication resources is described by the function $\beta(\Delta)$, a lower service curve which provides a lower bound on the available service in *any* time interval of length $\Delta \in \mathbb{R}$ where ($\beta(\Delta) = 0$ for all $\Delta \leq 0$). The service is expressed in an appropriate workload unit compatible to that of the arrival curve, e. g. , packets.

Packet stalls at routers can happen when packets from different input ports or virtual channels compete for the same output port. We assume a round-robin arbiter for every router j where each flow i is given a fixed slot of size s_i, e. g. , proportional to the maximum packet size for this flow. The packet size is defined as the number of (fixed-length) packets of a flow. The accumulated sizes of all flows going through a router j can be expressed as $s^j = \sum_{i \text{ flows through } j} s_i$. If the lower service curve that the router can provide is denoted as β^j, then the lower service curve under round-robin arbitration for a flow with slot size s_i is [51]:

$$\beta_i^j(\Delta) = \frac{s_i}{s^j}\beta^j(\Delta - (s^j - s_i)), \tag{7}$$

which expresses the fact that in the worst-case a packet may always have to wait for $s^j - s_i$ time units before its slot becomes available.

A Resource Model for a Network. When a packet flow traverses a system of multiple interconnected components, one needs to consider the service curve provided by the system as a whole, i.e., the system service curve is a concatenation of the individual service curves [27,42]. For example, the concatenation of two routers 1 and 2 with lower service curves β_i^1 and β_i^2 for a flow i can be obtained as:

$$\beta_i^{1,2}(\Delta) = \beta_i^1 \otimes \beta_i^2(\Delta),$$

where $\otimes$ is the min-plus algebra convolution operator that is defined as:

$$(f \otimes g)(\Delta) = \inf_{0 \leq \lambda \leq \Delta} \{f(\Delta - \lambda) + g(\lambda)\}.$$

As a result, the cumulative service β_i^{NOC} for a flow i is the convolution of all individual router service curves on the path of the flow through the network:

$$\beta_i^{\text{NOC}}(\Delta) = (\bigotimes_{i \text{ flows through } j} \beta_i^j)(\Delta). \tag{8}$$

Flow Regulator. A flow regulator with a (σ, ρ) shaping curve delays packets of an input flow such that the output flow has the upper arrival curve $\alpha_{(\sigma,\rho)}(\Delta) = (\rho \cdot \Delta + \sigma)$ for all $\Delta > 0$ independent of the timing characteristics of the input flow, and it outputs packets as soon as possible without violating the upper bound $\alpha_{(\sigma,\rho)}$.

Delay Bounds. A packet stream constrained by an upper arrival curve α_i is first regulated by a (σ, ρ) regulator and then traverses a network that offers a cumulative lower service curve β_i^{NOC} of the routers on the packet path. As is well known from the network and real-time calculus, the maximum packet delay is related to the maximal horizontal distance between functions (see Fig. 10) which is defined as:

$$\text{del}(f, g) = \sup_{\lambda \geq 0} \{\inf\{\tau \geq 0 : \ f(\lambda) \leq g(\lambda + \tau)\}\}.$$

Now, the worst-case delay at the regulator delREG experienced by a packet from a flow i constrained by an arrival curve α_i and regulated by a (σ, ρ) regulator can be computed as follows [45]:

$$\text{delREG} = \text{del}(\alpha_i, \alpha_{(\sigma,\rho)}). \tag{9}$$

The output of the regulator is constrained by $(\alpha_i \otimes \alpha_{(\sigma,\rho)})(\Delta)$ and therefore, the worst-case packet delay delNOC for flow i within the NOC that has a cumulative lower service β_i^{NOC} can be determined as:

$$\text{delNOC} = \text{del}(\alpha_i \otimes \alpha_{(\sigma,\rho)}, \beta_i^{\text{NOC}}). \tag{10}$$

An example of worst-case delay computation is shown in Fig. 10. We consider a single router which serves a single flow constrained by an upper arrival curve $\alpha_{(\sigma,\rho)}$. The NOC consists of a single router that provides to the flow a lower rate-latency service curve $\beta_{r,T}(\Delta) = r(\Delta - T)$ for $\Delta > T$, and $\beta_{(r,T)}(\Delta) = 0$ otherwise. The arrival curve implies that the source can send at most σ packets at once, but not more

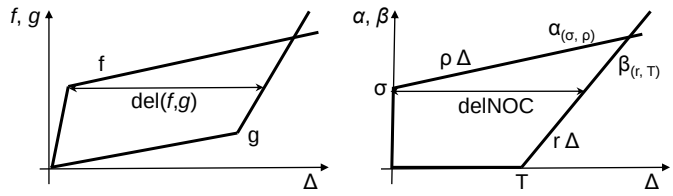


Fig. 10 Delay bound defined as the maximum horizontal distance illustrated for an arrival curve of a (σ, ρ) regulated flow and a single router providing a rate-latency service curve $\beta^l_{r,T}$

than ρ packets/cycle in the long run, while the service curve implies a pipeline delay of T for a packet to traverse the router, and an average service rate of r packets/cycle. As shown in Fig. 10, the worst-case delay bound corresponds to the maximum horizontal distance between the upper output arrival curve of the flow regulator and the lower service curve of the NOC.

Output Flow Bounds. When a packet stream constrained by arrival curve α_i is regulated by a (σ, ρ) regulator and traverses a network that offers a cumulative lower service curve β_i^{NOC}, the processed output flow is bounded by α_i' computed as follows [46]:

$$\alpha_i'(\Delta) = ((\alpha_i \otimes \alpha_{(\sigma,\rho)}) \oslash \beta_i^{\text{NOC}})(\Delta), \tag{11}$$

where $\oslash$ is the min-plus algebra deconvolution operator that is defined as:

$$(f \oslash g)(\Delta) = \sup_{\lambda \geq 0}\{f(\Delta + \lambda) - g(\lambda)\}.$$

Data Transfer Delay. Finally, we compute an upper bound on the total delay for transferring a buffer of data using multiple packets, e. g. , the maximum delay for transferring 4KB of data from external memory over a NoC. The availability of data can be modeled as an upper arrival curve which has the form of a step function: $\alpha_i(\Delta) = B$ for $\Delta > 0$, where B is the total amount of data measured as the number of packets. Then an upper bound on the total delay can be computed as a sum of the delays computed with equations (9) and (10), where the first one gives a bound on the delay experienced by all of the data at the regulator, i.e., the bound on the delay for the regulator to transmit all of the data, while the second one gives a bound on the delay for transferring the last packet of the data through the NoC .

In other words a bound on the total delay for transferring a buffer of B packets regulated by a (σ, ρ) regulator over a network NOC providing a cumulative lower service curve of β_i^{NOC} can be computed as:

$$\text{WCDFT} = \text{delREG} + \text{delNOC}, \tag{12}$$

where $\alpha_i(\Delta) = B$ for $\Delta > 0$, delREG and delNOC are defined in (9) and (10), respectively, and WCDFT denotes the worst-case data fetch time, as originally defined in the remote fetch protocol of Sec. 3.2.

Note that the above model is valid under the assumption that during the data transfer, the regulator does not stall because there are no packets available for transmission, e. g. , in the case of external memory data fetch, the memory controller should be able to put packets fast enough in the buffer of the regulator. Moreover, the regulator should not experience stalls due to backpressure.

Bound on NoC Rx Memory Accesses within an FTTS Frame. For representing interference from the NoC Rx interface within a compute cluster, in Sec. 5.1.2 we introduced the value δ, which bounds the number of memory accesses that the NoC Rx can perform within a time frame f. For the time interval of the frame f, denoted as $\mathcal{L}_f$, δ can be not greater than $\alpha'(\mathcal{L}_f)$ (Eq. 11), but also not greater than the total number of packets in the transmitted buffer, B. Therefore,

$$\delta = \min\left\{\alpha'(\mathcal{L}_f), B\right\}. \tag{13}$$

The D-NoC in the Kalray Platform. The models and methods described above are compatible with the manycore MPPA®-256 platform (Sec. 3.2). In the following, we give a short summary of flow regulation for the MPPA® NoC, which is based on (σ, ρ) regulators. Precisely, in the MPPA®-256 processor, each connection is regulated at the source node by a packet shaper and a traffic limiter in tandem. This regulator can be configured via two parameters, both defined in units of 32-bit flits: (i) a *window length* (T_w), which is set globally for the NoC node, and (ii) the *bandwidth quota* (N_{max}), which is set separately for each regulator. At each cycle, the regulator compares the length of a packet scheduled for injection plus the number of flits sent within the previous T_w cycles to N_{max}. If not greater, the packet is injected at the rate of one flit per cycle.

The (σ, ρ) parameters can be set at the source node through T_w and N_{max} (all measured in units of 32-bit flits, including header flits). We link these parameters with the (σ, ρ) model by observing that $\rho = N_{max}/T_w$. This corresponds to the fact that no regulator may let more than N_{max} flits pass over any duration T_w. On the other hand, the regulator is allowed to emit continuously until having sent N_{max} flits within exactly N_{max} cycles. This defines a point on the $\rho+\sigma$ linear time function and by regression, the value of the function at time $t = 0$ (corresponding to σ) is found to be $\sigma = N_{max}(1 - N_{max}/T_w)$. Note that $\sigma \geq 0$. For a more detailed presentation of the MPPA® NoC flow regulation, the interested readers are referred to [16].

The MPPA® NoC routers multiplex flows originating from different directions. Each originating direction has its own FIFO queue at the output interface, so flows interfere on a node only if they share a link to the next node. This interface performs a round-robin arbitration at the packet granularity between the FIFOs that contain data to send on a link. The NoC routers have thus been designed for simplicity while inducing a minimal amount of perturbations on (σ, ρ) flows. An additional benefit of this router design is that eliminating backpressure from every single queue through (σ, ρ) flow regulation at the source effectively prevents deadlocks. It is thus not necessary to resort to specific routing techniques such as turn-models. Selecting (σ, ρ) parameters for all flows is treated as an optimization step during design time, which however is outside of the scope of this article.

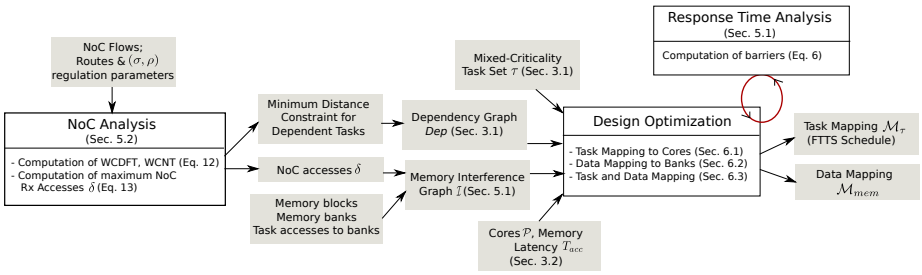


Fig. 11 Design Flow

6 Design Optimization

Sec. 4 presented the runtime behavior of the FTTS scheduler and Sec. 5 the response time analysis for tasks that are scheduled under FTTS and can experience blocking delays on the shared memory path of a cluster, either by other tasks executing concurrently in the cluster or by the incoming traffic from the NoC. For both parts, we assumed a given FTTS schedule, with known task mapping to cores and data mapping to memory banks. In this section, we discuss the problem of actually finding an FTTS schedule while optimizing resource utilization in our system.

The problem can be formulated as follows. *Given*: (i) a periodic mixed-criticality task set τ, (ii) a cluster consisting of processing cores $\mathcal{P}$ with access to a banked memory, (iii) the memory interference graph $\mathcal{I}$ with undefined edge set $\mathcal{E}_2$, (iv) the memory access latency T_{acc}; *Determine*: the mapping $\mathcal{M}_\tau : \tau \rightarrow \mathcal{P}$ of tasks to processing cores and the mapping $\mathcal{M}_{mem} : BL \rightarrow B$ ($\mathcal{E}_2$ of $\mathcal{I}$) of memory blocks to banks, *such that*:

- all tasks meet their mixed-criticality real-time requirements at all levels of assurance,
- the workload is balanced among the cores,
- the minimum distance constraints of the dependency graph $\mathcal{D}ep$ hold, and
- the memory bank capacities are not surpassed.

The mapping $\mathcal{M}_\tau$ defines both the spatial partitioning of tasks among the cores in $\mathcal{P}$ as well as the timing partitioning into frames and the execution order on each individual core. These three aspects (spatial, timing partitioning, relative execution order) determine fully an FTTS schedule for task set τ.

For each of the two considered optimization problems ($\mathcal{M}_\tau$ in Sec. 6.1 and $\mathcal{M}_{mem}$ in Sec. 6.2), we assume an existing solution to the other one. Finally, we show how to solve both optimization problems in an integrated manner (Sec. 6.3). To facilitate reading, please refer to Fig. 11, which depicts the inputs and outputs of the optimization procedure, as well as the flow of analyses (NoC analysis, response time analysis) and information (task set, platform model, etc.) which enable us to determine some of the inputs, e.g., the memory interference graph, and to evaluate the visited solutions during optimization.

6.1 Task Mapping $\mathcal{M}_\tau$ Optimization

The problem of optimal task mapping on multiple cores is known to be NP-hard, resembling the combinatorial bin-packing problem. A possible approach to optimizing $\mathcal{M}_\tau$ for FTTS was suggested in [20] and is summarized below.

The approach implements a heuristic method based on simulated annealing [26]. Initially, it determines a random task mapping solution, resp. FTTS schedule for the given task set τ. Specifically, it selects the FTTS cycle as the hyper-period H of tasks in τ and also, the FTTS frame lengths depending on the task periods (the greatest common divisor of the periods is used unless otherwise specified by the system designer). For every task $\tau_i \in \tau$, it selects arbitrarily a core on which the task will be mapped. Then, it computes the number of jobs that are released by task τ_i within a hyper-period H and the range of FTTS frames, to which each job can be scheduled, such that it is executed between its release time and absolute deadline. For every job of τ_i, it selects arbitrarily an FTTS frame from the allowed range. This procedure is repeated for all tasks in τ. The constraints that must be respected during the generation of the FTTS schedule are:

- All jobs of the same task are scheduled on the same core.
- For every dependency $\tau_i \rightarrow \tau_j$ in the dependency graph $\mathcal{D}ep$, the jobs of the two tasks are scheduled on the same core, with a job of τ_j being scheduled in the same or a later frame than the corresponding job of τ_i within their common period. If they are scheduled in the same frame, the job of τ_j must succeed that of τ_i. In any case, the sum of the best-case execution times of the jobs that are scheduled in between τ_i and τ_j and the lengths of the intermediate frame(s) (if any exist between the two jobs) must be no lower than their minimum distance constraint, i.e., the weight of the corresponding edge in $\mathcal{D}ep$. Note that this is an extension to the original method of [20].

Note that this is the procedure that a system designer would, also, follow to generate a random FTTS schedule for a given task set τ. It provides no guarantee on the schedule admissibility.

Once a random initial mapping solution is determined, the optimizer applies simulated annealing [26] to explore the design space for task mapping. Particularly, new solutions are found by applying two possible variations with given probabilities: (i) re-mapping all jobs of a randomly selected task (and its dependent tasks) to a different core or (ii) re-allocating one job of a randomly selected task to a different FTTS sub-frame or to a different position within the same sub-frame. Design space exploration is restricted to solutions that satisfy the dependency constraints in $\mathcal{D}ep$. The exploration terminates when it converges to a solution or a computational budget is exhausted.

A task mapping solution is considered optimal if all jobs meet their deadlines at all levels of assurance, i.e., the schedule is admissible, and the worst-case sub-frame lengths are minimized, implying a balanced workload distribution. Based on these requirements, we define the cost function of the optimization problem as:

$$Cost(S) = \begin{cases} c_1 = \max_{f \in \mathcal{F}} \left\{ \max_{\ell \in \{1,\ldots,L\}} late(f,\ell) \right\} & \text{if } c_1 > 0 \\ c_2 = \|barriers\|_3 & \text{if } c_1 \leq 0, \end{cases} \tag{14}$$

where $late(f, \ell)$ expresses the difference between the worst-case completion time of the last sub-frame of f and the length of f:

$$late(f, \ell) = \sum_{i=1}^{L} barriers(f, \ell)_i - \mathcal{L}_f. \quad (15)$$

If $late(f, \ell) > 0$, the tasks in f cannot complete execution by the end of the frame for their ℓ-level execution profiles. Therefore, with this cost function, we initially guide design space exploration towards finding an admissible solution. When such a solution is found, cost c_1 becomes negative or 0. Then, c_2, i.e., the 3-norm of all sub-frame lengths, $\forall f \in \mathcal{F}, \forall \ell \in \{1, \ldots, L\}$, is used to minimize the worst-case lengths of all sub-frames. The 3-norm of a vector x with n elements (here, positive real numbers) is defined as $||x||_3 := \left(\sum_{i=1}^{n} |x_i|^3\right)^{1/3}$. We selected the particular value to map (represent) the vector with the $barriers$ values for all $f \in \mathcal{F}$ and $\ell \in \{1, \ldots, L\}$, as we empirically found this to be the best among other considered norms, such as the average, the maximum, the sum or the Euclidean norm. Namely, the selected norm provides a trade-off between reducing the worst-case sub-frame lengths (to ensure schedulability) and enabling progress in the optimization via improving the average-case lengths. However, alternative norms, such as the ones mentioned above can be also used in our cost function (14).

Note that during exploration, the $barriers$ function is computed for each visited solution, as discussed in Sec. 5. For the WCRT analysis, the memory mapping $\mathcal{M}_{mem}$ is assumed to be known.

The task mapping optimization method can be easily extended to account for fixed task preemption points, mapping constraints, solution ranking, among others. Please refer to [20] for a more detailed discussion.

6.2 Memory Mapping $\mathcal{M}_{mem}$ Optimization

The goal of memory mapping optimization is to determine a static allocation of the tasks' instructions, private data and communication buffers (memory blocks) BL to banks B of the shared memory ($\mathcal{E}_2$ of memory interference graph $\mathcal{I}$), so that the timing interferences of tasks when accessing the memory are minimized. Also, the total size of the allocated memory blocks in a bank should not surpass the bank capacity. This constraint holds e.g., for the memory mapping in Fig. 6.

For this problem, we adopt a heuristic method based on simulated annealing, similar to the task mapping optimization. The method is presented in the form of pseudocode in Listing 1. It receives as inputs an initial temperature T_0, a temperature decreasing factor $a \in (0, 1)$, the maximum number of consecutive variations with no cost improvement that can be checked for a particular temperature $Fail_{max}$, a stopping criterion in terms of the final temperature T_{final}, and a stopping criterion in terms of search time (computational budget) $time_{max}$. It returns the best encountered solution(s) in the given time.

The algorithm starts with an arbitrary solution S, satisfying the bank capacity constraints. If **GenerateInitialSolution** can provide no such solution, exploration is aborted. Otherwise, design space exploration is performed by examining random

Algorithm 1 Modified Simulated Annealing for Memory Mapping $\mathcal{M}_{mem}$

Input: $T_0, a, Fail_{max}, T_{final}, time_{max}$
Output: $\bar{S}_{best}$
1: $S \leftarrow$ **GenerateInitialSolution()**
2: **if** $S == \emptyset$ **then**
3: **return** $null$
4: **end if**
5: $\bar{S}_{best} \leftarrow \{S\}, S_{cur\_best} \leftarrow S, Cost_{min} \leftarrow$ **Cost**(S)
6: $T \leftarrow T_0$
7: $FailCount \leftarrow 0$
8: $time \leftarrow$ **StartTimer()**
9: **while** $time < time_{max}$ and $T > T_{final}$ **do**
10: $S' \leftarrow$ **Variate**(S)
11: **if** $e^{-(\textbf{Cost}(S')-\textbf{Cost}(S))/T} \geq$ **Random**(0,1) **then**
12: $S \leftarrow S'$
13: **end if**
14: **UpdateBestSolutions**(S')
15: **if Cost**(S') $< Cost_{min}$ **then**
16: $S_{cur\_best} \leftarrow S'$
17: $Cost_{min} \leftarrow$ **Cost**(S')
18: $FailCount \leftarrow 0$
19: **else**
20: $FailCount \leftarrow FailCount + 1$
21: **end if**
22: **if** $FailCount == Fail_{max}$ **then**
23: $T \leftarrow a \cdot T$
24: $S \leftarrow S_{cur\_best}$
25: $FailCount \leftarrow 0$
26: **end if**
27: **end while**

variations of the memory mapping. Particularly, **Variate** selects non-deterministically a memory block and remaps it to a different memory bank such that no bank capacity constraint is violated. The new solution S' is accepted if $e^{-\big(Cost(S')-Cost(S)\big)/T}$ is no lesser than a randomly selected real value in (0,1). The cost of S' is, also, compared to the minimum observed cost, $Cost_{min}$. If it is lower than $Cost_{min}$, the new solution and its cost are stored, even if transition to S' was not admitted. The temperature T of the simulated annealing procedure is reduced geometrically with factor a. Reduction takes place every time a sequence of $Fail_{max}$ consecutive solutions are checked, none of which improves $Cost_{min}$. After temperature reduction, exploration continues from the so-far best found solution (S_{cur_best}). Design Space exploration terminates when the lowest temperature T_{final} is reached or the computational budget $time_{max}$ is exhausted.

Memory mapping affects the WCRT of a task τ_i by defining which of the tasks that can be executed in parallel with it are also interfering with it. The less interfering tasks, the lower the delay τ_i experiences when accessing the shared memory. Therefore, to evaluate a memory mapping solution we select a cost function which reflects the increase in task WCRT due to interference on the shared memory banks. The cost function is based on the mutual delay matrix D, which was introduced in Sec. 5.1.1, and has two alternative definitions.

One alternative to solve the optimization problem, is to compute (part of) the *Pareto set* of memory mapping solutions with minimal interference between any two tasks of the same criticality level. The intuition behind this approach is that we try to minimize simultaneously all elements of the mutual delay matrix D, namely all blocking delays that a task can cause to any other task (of the same criticality). This problem can be seen as a multi-objective optimization problem, with the n^2 elements of matrix D as individual cost functions. For this set of objectives, we compute the Pareto set of memory mapping solutions. Algorithm 1 maintains such a set $\bar{S}_{best}$ of non-dominated solutions. In particular, a newly visited mapping solution S' with matrix D' is inserted to the set $\bar{S}_{best}$ if it has a lower value for at least one element of D' than the corresponding element of any solution in $\bar{S}_{best}$. If a solution $S \in \bar{S}_{best}$ is dominated by S', i.e., S' has lower or equal values for all elements of D, then S is removed from the set. This update is performed by **UpdateBestSolutions**.

Another alternative is to define the scalar cost function D_{avg} as the *average* over all elements of matrix D, i.e., the average delay tasks of the same criticality level cause to each other when interfering on shared banks. Then we can find the best solution in terms of D_{avg}. In this case, $\bar{S}_{best}$ contains only one solution characterized by the minimum encountered D_{avg}.

6.3 Integrated Task and Memory Mapping Optimization

The problems of optimizing $\mathcal{M}_\tau$ and $\mathcal{M}_{mem}$ are inter-dependent. Namely, design space exploration for the optimization of the task mapping requires information on the memory mapping for computing function $barriers$. Similarly, matrix D, which defines the cost of a memory mapping solution, can be refined for a particular task mapping, depending on the tasks that can be executed in parallel. In the following, we outline two alternative approaches towards an integrated optimization solution.

I. **Task mapping optimization for each memory mapping in Pareto set.** As discussed previously, one can compute using Algorithm 1 part of the Pareto set $\bar{S}_{best}$ of memory mapping solutions that minimize the interference between any two tasks of the same criticality level. These solutions consider that all tasks of the same criticality level are potentially executed in parallel (*worst-case task mapping*). The next step is to solve the task mapping optimization problem for each memory mapping in the set $\bar{S}_{best}$. Finally, the combination of solutions which minimizes $||barriers||_3$ is selected.

II. **Iterative task and memory mapping optimization.** Since the complexity of computing the Pareto set solutions for the memory mapping optimization problem can be prohibitive, one can select an iterative solution to the two problems. Then, for each visited solution during design space exploration for $\mathcal{M}_\tau$, a memory mapping optimization is also performed to find the solution with minimized cost $||barriers||_3$. Here, we use the cost function D_{avg}.

It cannot be said that one method clearly outperforms the other in terms of efficiency. However, depending on the sizes of the search spaces of the task mapping and memory mapping optimization problems, one algorithm can perform faster than the other [21].

7 A Case Study

To evaluate the proposed design optimization approaches, we use an industrial implementation of a flight management system. This application has been a major use-case of the European Certainty project [2]. The purpose for the evaluation is first, to show applicability of our optimization methods and demonstrate the results of the response time analysis for the optimized task and memory mapping solution. Second, we investigate the effect of various platform parameters, such as the memory access latency and the number of memory banks, or design choices, such as the selection of routes and (σ,ρ) parameters for the NoC flows, on the schedulability of the application under FTTS. The optimization framework has been implemented in Java and the evaluation was performed on a laptop with a 4-core Intel i7 CPU at 2.7 GHz and 8 GB of RAM. The source code of the experiments is available for downloading at [1].

Flight Management System. The flight management system (FMS) from the avionics domain is responsible for functionalities, such as the localization of an aircraft, the computation of the flightplan that guides the auto-pilot, the detection of the nearest airport, etc. We look into a subset of the application, consisting of 14 periodic tasks for sensor reading, localization, and computation of the nearest airport. Seven are characterized by safety level DAL-B (we map it to criticality level 2, i.e., high) and seven by safely level DAL-C (we map it to criticality level 1, i.e., low) based on the DO-178B standard for certification of airborne systems [3]. The periods of the tasks vary among 200 ms, 1 sec, and 5 sec, as shown in Table 3. Based on these, we select the cycle and the frame length of the FTTS as $H = 5$ sec and $\mathcal{L}_f = 200$ ms, respectively. The worst-case execution times of the tasks were derived through measurements on a real system or for few tasks, for which the code was not available (e.g., $\tau_{init,13}$), based on conservative estimations. A discussion on how the worst-case execution time parameters of the FMS tasks can be derived can be found in the technical report [12] of the Certainty project [2]. For the level-2 profiles $C_i(2)$ of tasks τ_i with $\chi_i = 2$, we augment the worst observed execution times by a factor of 5. Similarly, for the memory accesses, we consider conservative bounds based on the known memory footprints for the tasks and derive the $C_i(2)$ parameters by multiplying these bounds by 5. Factor 5 is selected arbitrarily to augment the worst-case task parameters and thus, increase the safety margins. Such augmentation of the worst-case parameters seems a common industrial practice for safety-critical applications. The best-case execution time and access parameters are taken equal to 0 due to lack of more accurate information. Last, the degraded profiles $C_{i,deg}$ of tasks τ_i with $\chi_i = 1$ correspond to no execution, i.e., $C_{i,deg} = (0, 0, 0, 0)$. The task periods, criticality levels, level-1 and level-2 worst-case execution times and memory accesses, as well as the memory blocks that they access and the maximum number of accesses to each block at the tasks' criticality levels are shown in Table 3.

To model the memory accessing behavior of the tasks according to Sec. 5.1, we define a memory interference graph $\mathcal{I}$ with the following memory blocks: one block per task with size equal to the size of its data as measured on the deployed system, and one block per communication buffer with known size, too. This yields in total 27 memory blocks.

Purpose	Task	CL	Period (ms)	Level-1 (Level-2) Exec. Time (ms)	Level-1 (Level-2) Memory Accesses	Accessed Mem. Blocks (Max. Accesses)
Sensor data acquisition	τ_1	2	200	11 (55)	213 (1065)	b_1 (100), b_2 (425), b_4 (70), b_6 (190), b_8 (190), b_{10} (90)
	τ_2	1	200	20 (0)	117 (0)	b_3 (10), b_4 (107)
	τ_3	1	200	18 (0)	129 (0)	b_5 (10), b_6 (119)
	τ_4	1	200	18 (0)	129 (0)	b_7 (10), b_8 (119)
	τ_5	1	200	20 (0)	129 (0)	b_9 (10), b_{10} (119)
Localization	τ_6	2	200	7 (35)	145 (725)	b_2 (425), b_{11} (100), b_{12} (100), b_{13} (90), b_4 (10)
	τ_7	2	1000	6 (30)	56 (280)	b_{13} (90), b_{14} (100), b_{15} (90)
	τ_8	2	5000	6 (30)	57 (285)	b_{15} (17), b_{16} (100), b_{17} (90), b_{19} (78)
	τ_9	2	1000	6 (30)	57 (285)	b_{17} (90), b_{18} (100), b_{21} (78), b_{22} (17)
	τ_{10}	1	200	20 (0)	130 (0)	b_{12} (120), b_{24} (10)
	τ_{11}	1	1000	20 (0)	113 (0)	b_{19} (103), b_{25} (10)
	τ_{12}	1	200	20 (0)	113 (0)	b_{21} (103), b_{26} (10)
Nearest Airport	τ_{13}	2	1000	48 (192)	1384 (6920)	b_{17} (90), b_{20} (100), b_{23} (1610), b_{27} (5120)
	$\tau_{init,13}$	2	1000	2 (10)	18 (90)	b_{17} (90) Triggers Rx to b_{27} (403)

Table 3 Flight Management System

The FMS requires access to a navigation database with a memory footprint of several tens of MB. Particularly, task τ_{13}, which is responsible for the computation of the nearest airport, needs read-access to certain entries (up to 4 KB of data). In the following, we assume that the database is maintained in an external DDR memory and the required data are fetched to the local memory of a cluster, where the FMS application is executed. The data transfer is initiated by the preceding task $\tau_{init,13}$, which has the same criticality level and period as τ_{13}. The memory block corresponding to the database data is bl_{27}. We add a high-priority task $\texttt{Rx}_{13}$ to $\mathcal{I}$ to indicate the remote data transfer. $\texttt{Rx}_{13}$ is connected to the memory block b_{27} via an edge with weight δ.

Note that the FMS contains no task dependencies other than that between the task requesting the database entries for the computation of the nearest airport, $\tau_{init,13}$, and the task that performs the computation, τ_{13}. In the following, we show how to compute the minimum distance constraint between $\tau_{init,13}$ and τ_{13} in the dependency graph $\mathcal{D}ep$ as well as the weight δ for the edge from the node $\texttt{Rx}_{13}$ to bl_{27} in the memory interference graph $\mathcal{I}$.

NoC Flow Routing and Regulation. Task τ_{13} reads upon each activation 4 KB of data from the database. The data are transferred from the remote cluster with access to the DDR over the D-NoC in packets of 4 Bytes. Namely, a transfer of 1024 packets must be executed between any two successive executions of τ_{13}. We assume that this flow is (σ, ρ) regulated at the I/O sub-system, with $\sigma = 10$ packets and $\rho = 2000$ packets/sec. The (σ, ρ) parameters are selected arbitrarily here, such that they are reasonable for the required amount of transferred data and they allow the transfer over the NoC to be completed within a period of task τ_{13}. The flow routing is fixed and passes through *two* D-NoC routers. On the first router, the flow can encounter interference from *one* more flow on the output link. Respectively, on the second router, the flow interferes with *three* more flows on the output direction. The clock frequency

on the chip is 400 MHz. The routers forward the packets over the D-NoC links at a rate of 1 packet/cycle, equiv. 400,000,000 packets/sec. Given the above assumptions and by applying Eq. 12 and 13 of Sec. 5.2, we derive the worst-case data fetch time, WCDFT = 511.4 ms, and the maximum number of packets fetched during 200 ms (duration of an FTTS frame), $\delta = 403$, respectively.

Based on the remote fetch protocol of Sec. 3.2, to specify the minimum distance constraint between task $\tau_{init,13}$ and τ_{13}, we need to know besides WCDFT, the worst-case notification time, WCNT, for the transfer of the notification from $\tau_{init,13}$ to the remote listener task in the I/O sub-system as well as the worst-case remote set-up time for the data transfer, WCRST. WCNT can be derived in a similar way as WCDFT. Assuming that $\tau_{init,13}$ sends only one packet (4 Bytes) to the remote cluster, following the exact same route as the flow from the I/O sub-system to the cluster, it follows from Eq. 12 that WCNT = 0.4 ms. Regarding WCRST, a conservative bound is assumed to be given, WCRST = 25 ms (here, arbitrary selection). Summarizing on the above results, the dependency between $\tau_{init,13}$ and τ_{13} in the dependency graph $\mathcal{D}ep$ is weighted with the minimum distance constraint: w =(0.4 + 25 + 511.4) ms = 536.8 ms.

Platform Parameters. For the deployment of the FMS, we consider a target platform resembling a cluster of the MPPA®-256 platform. In particular, $\mathcal{P}$ includes 8 processing cores with shared access to 8 memory banks of 128 KB each. We consider round-robin arbitration on the memory arbiters with higher priority for the NoC Rx interface, according to the description in Sec. 3.2. Once a memory access is granted, the fixed memory latency is $T_{acc} = 55$ ns. The memory latency bound has been empirically estimated on the MPPA®-256 platform, using benchmark applications. Note, however, that it is not necessarily a safe bound for the MPPA®-256 memory controller.

7.1 Design Optimization and Response Time Analysis

With the first experiment we intend to evaluate the applicability and efficiency of the optimization framework, which was developed in Sec. 6, w.r.t. the deployment of the FMS on a compute cluster. The scheduling policy in the cluster is FTTS, with a cycle of $H = 5000$ ms, consisting of 25 frames with length 200 ms each, based on the periods of our task set. Each FTTS frame is divided into two sub-frames, since the FMS has $L = 2$ criticality levels. We configure the simulated-annealing algorithm (Listing 1 of Sec. 6) for both task and memory mapping optimization with parameters: $a = 0.8$, $Fail_{max} = 100$, T_0 based on the cost of 100 random solutions, $T_{final} = 0.1$, $time_{max} = 30$ min. For the task mapping $\mathcal{M}_\tau$ optimization, the probabilities of selecting a sub-frame or core variation are 0.85 and 0.15, respectively. The memory mapping optimization $\mathcal{M}_{mem}$ uses as objective function the average value of the delay matrix, D_{max} and is performed for each visited task mapping solution during design space exploration, i.e., according to the integrated solution II of Sec. 6.3. The overall optimization goal is to maximize the slack time at the end of the frames (equiv. minimize $||barriers||_3$), which indicates a maximal exploitation of computation parallelism and memory accessing parallelism.

Frame	Core	Sub-frame 1	Sub-frame 2	Frame	Core	Sub-frame 1	Sub-frame 2
f_1 [0,200]	p_1	$\tau_{init,13}$	$\tau_{10}, \tau_2, \tau_3$	f_2 [200,400]	p_1	-	$\tau_{10}, \tau_2, \tau_3$
	p_2	τ_6, τ_1	$\tau_4, \tau_{12}, \tau_5$		p_2	τ_6, τ_1	$\tau_4, \tau_{12}, \tau_5$
f_3 [400,600]	p_1	-	$\tau_{10}, \tau_2, \tau_3$	f_4 [600,800]	p_1	τ_{13}	$\tau_{10}, \tau_2, \tau_3$
	p_2	τ_6, τ_1	$\tau_4, \tau_{12}, \tau_5, \tau_{11}$		p_2	τ_6, τ_1	$\tau_4, \tau_{12}, \tau_5$
f_5 [800,1000]	p_1	τ_7, τ_9	$\tau_{10}, \tau_2, \tau_3$	f_6 [1000,1200]	p_1	$\tau_{init,13}$	$\tau_{10}, \tau_2, \tau_3$
	p_2	τ_6, τ_1	$\tau_4, \tau_5, \tau_{12}$		p_2	τ_6, τ_1	$\tau_4, \tau_5, \tau_{12}$
f_7 [1200,1400]	p_1	τ_7	$\tau_2, \tau_{10}, \tau_3$	f_8 [1400,1600]	p_1	τ_9	$\tau_{10}, \tau_2, \tau_3$
	p_2	τ_6, τ_1	$\tau_4, \tau_{12}, \tau_5$		p_2	τ_6, τ_1	$\tau_4, \tau_5, \tau_{11}, \tau_{12}$
f_9 [1600,1800]	p_1	-	$\tau_{10}, \tau_3, \tau_2$	f_{10} [1800,2000]	p_1	τ_{13}	$\tau_{10}, \tau_2, \tau_3$
	p_2	τ_6, τ_1	$\tau_4, \tau_{12}, \tau_5$		p_2	τ_6, τ_1	$\tau_4, \tau_{12}, \tau_5$
f_{11} [2000,2200]	p_1	$\tau_{init,13}$	$\tau_{10}, \tau_2, \tau_3$	f_{12} [2200,2400]	p_1	-	$\tau_{10}, \tau_2, \tau_3$
	p_2	τ_6, τ_1	$\tau_{12}, \tau_5, \tau_9$		p_2	τ_6, τ_1	$\tau_4, \tau_{11}, \tau_{12}, \tau_5$
f_{13} [2400,2600]	p_1	τ_7	$\tau_{10}, \tau_2, \tau_3$	f_{14} [2600,2800]	p_1	τ_{13}	$\tau_{10}, \tau_2, \tau_3$
	p_2	τ_6, τ_1	$\tau_{12}, \tau_4, \tau_5$		p_2	τ_6, τ_1	$\tau_4, \tau_{12}, \tau_5$
f_{15} [2800,3000]	p_1	τ_9	$\tau_{10}, \tau_2, \tau_3$	f_{16} [3000,3200]	p_1	-	$\tau_{10}, \tau_2, \tau_3$
	p_2	τ_1, τ_6	$\tau_4, \tau_{12}, \tau_5$		p_2	τ_6, τ_1	$\tau_4, \tau_{12}, \tau_5$
f_{17} [3200,3400]	p_1	$\tau_{init,13}, \tau_9$	$\tau_{10}, \tau_2, \tau_3$	f_{18} [3400,3600]	p_1	τ_7, τ_8	$\tau_{10}, \tau_2, \tau_3$
	p_2	τ_6, τ_1	$\tau_4, \tau_{12}, \tau_5, \tau_{11}$		p_2	τ_6, τ_1	$\tau_{12}, \tau_4, \tau_5$
f_{19} [3600,3800]	p_1	-	$\tau_{10}, \tau_2, \tau_3$	f_{20} [3800,4000]	p_1	τ_{13}	$\tau_{10}, \tau_2, \tau_3$
	p_2	τ_6, τ_1	$\tau_4, \tau_5, \tau_{12}$		p_2	τ_6, τ_1	$\tau_4, \tau_{12}, \tau_5$
f_{21} [4000,4200]	p_1	-	$\tau_{10}, \tau_2, \tau_3$	f_{22} [4200,4400]	p_1	$\tau_{init,13}, \tau_9$	$\tau_{10}, \tau_2, \tau_3$
	p_2	τ_6, τ_1	$\tau_4, \tau_{12}, \tau_5$		p_2	τ_6, τ_1	$\tau_{13}, \tau_4, \tau_5, \tau_{11}$
f_{23} [4400,4600]	p_1	-	$\tau_{10}, \tau_2, \tau_3$	f_{24} [4600,4800]	p_1	τ_7	$\tau_{10}, \tau_2, \tau_3$
	p_2	τ_6, τ_1	$\tau_4, \tau_{12}, \tau_5$		p_2	τ_6, τ_1	$\tau_5, \tau_{12}, \tau_4$
f_{25} [4800,5000]	p_1	τ_{13}	$\tau_{10}, \tau_2, \tau_3$				
	p_2	τ_6, τ_1	$\tau_4, \tau_{12}, \tau_5$				

Table 4 Optimized task mapping $\mathcal{M}_\tau$ for FMS on a 2-core, 2-bank subset of a compute cluster

We consider all possible configurations with one to eight processing cores and one to eight memory banks for the deployment of the FMS. The optimized task and memory mapping solution, which yields the minimum value for the objective function $||barriers||_3$, is found for the configuration with *two* processing cores and *two* memory banks. Selecting more memory banks or more cores is not beneficial since it does not lead to a solution of lower cost, namely a solution with more slack time at the end of the frames. This is important information for a system designer, who tries not only to design a safe system, but also to allocate the minimal amount of resources.

For the configuration with two cores and two memory banks, the optimization framework returns an admissible FTTS schedule after evaluating 4919 task and memory mapping combinations and converging to one within 4.3 minutes. The optimized task and memory mapping are shown in Table 4 and 5, respectively. Note that the dependency between tasks $\tau_{init,13}$ and τ_{13} is respected, and that they are scheduled in different frames on the same core such that the minimum distance constraint (536.8 ms) is not violated. For the optimized solution, function $barriers$ can be computed, based on the memory interference graph $\mathcal{I}$ and the memory latency T_{acc}, as described in Sec. 5.1. The values of $barriers$, i.e., the worst-case sub-frame lengths for every FTTS sub-frame and level of assurance, as computed by our optimization framework, are shown in Table 6. Note that for every FTTS frame, the sum of barriers for its two sub-frames, under both levels of assurance, is not greater than the size of the frames, i.e., 200 ms. This shows that the admissibility condition of Eq. 2 is valid, which yields the FTTS schedule admissible.

Bank	Mapped memory blocks
1	b_1 to b_{15}, b_{24}
2	b_{16} to b_{23}, b_{25} to b_{27}

Table 5 Optimized memory mapping $\mathcal{M}_{mem}$ for FMS on a 2-core, 2-bank subset of a compute cluster

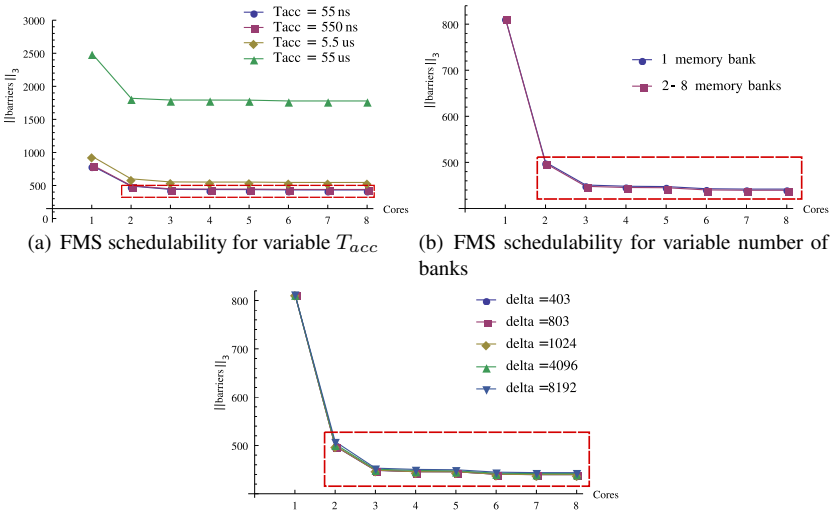
Frame f	Level-1 $barriers(f,1)$		Level-2 $barriers(f,2)$		Frame	Level-1 $barriers(f,1)$		Level-2 $barriers(f,2)$	
	Subframe1	Subframe2	Subframe1	Subframe2		Subframe1	Subframe2	Subframe1	Subframe2
f_1	18	58.1	90.1	0	f_2	18	58.1	90.1	0
f_3	18	78.1	90.1	0	f_4	48.1	57.9	192	0
f_5	18	58.1	90.1	0	f_6	18	58.1	90.1	0
f_7	18	58.1	90.1	0	f_8	18	78.1	90.1	0
f_9	18	58.1	90.1	0	f_{10}	48.1	57.9	192	0
f_{11}	18	58.1	90.1	0	f_{12}	18	78.1	90.1	0
f_{13}	18	58.1	90.1	0	f_{14}	48.1	57.9	192	0
f_{15}	18	58.1	90.1	0	f_{16}	18	58.1	90.1	0
f_{17}	18	78.1	90.1	0	f_{18}	18	58.1	90.1	0
f_{19}	18	58.1	90.1	0	f_{20}	48.1	57.9	192	0
f_{21}	18	58.1	90.1	0	f_{22}	18	78.1	90.1	0
f_{23}	18	58.1	90.1	0	f_{24}	48.1	57.9	192	0
f_{25}	18	58.1	90.1	0					

Table 6 Computation of $barriers$ for $\mathcal{M}_{mem}$ (Table 4), $\mathcal{M}_{mem}$ (Table 5), $T_{acc} = 55$ns, memory interference graph $\mathcal{I}$

7.2 Effect of Platform Parameters and Design Choices on FTTS Schedulability

With the second experiment we intend to evaluate the sensitivity of our optimization approach when certain parameters, e.g., the number of memory banks, the memory access latency T_{acc}, the incoming traffic from the NoC, the number of available processing cores vary. We evaluate schedulability of the FMS application for the alternative configurations (combinations of the above parameters), based on the cost $||barriers||_3$ of the optimized task and memory mapping solution in each case. For the definition of metric $||barriers||_3$, see the discussion on the cost function (14) in Sec. 6.1. The lower the cost of the optimized solution, the higher the probability that an admissible FTTS schedule for the considered configuration exists. In all following scenarios, the optimizer converges to a task and memory mapping solution in less than 7 minutes. For the simulated-annealing algorithm, we use the same parameters as in the previous section.

First, we evaluate the effect of the *memory access latency* T_{acc} on the FMS schedulability. We assume that the value of T_{acc} varies within {55 ns, 550 ns, 5.5 us, 55 us}. We perform design space exploration after fixing the number of memory banks to two. Fig. 12(a) shows how the schedulability metric, $||barriers||_3$, changes for the FMS as the number of available cores increases from one to eight, for different T_{acc} values. For each combination of T_{acc} and number of cores, the depicted point in Fig. 12(a) corresponds to the best found solution by our optimization framework. The value on the y-axis represents the 3-norm $||barriers||_3$ for the optimized task and memory mapping solution. The points within the dashed rectangle correspond to schedulable implementations, namely to combinations of T_{acc} and number of cores m, for which the optimized mapping solution is admissible according to Definition 1



(a) FMS schedulability for variable T_{acc}

(b) FMS schedulability for variable number of banks

(c) FMS schedulability for variable number of NoC Rx memory accesses in an FTTS frame δ

Fig. 12 Effect of platform and design parameters on FMS schedulability under the FTTS policy.

of Sec. 3.3. The points that do not fall into this rectangle correspond to implementations, for which the optimizer could not converge to any admissible solution within the given time budget of 30 minutes. We observe that, like in the previous section, the FMS schedulability under FTTS increases or remains stable as the number of cores increases. This is partly explained by the low task set utilization of the FMS. Two processing cores suffice to find an admissible FTTS schedule, whereas more cores are not beneficiary. Moreover, the effect of T_{acc} in schedulability is significant. For $T_{acc} \in \{5.5\text{ us}, 55\text{ us}\}$ no admissible schedule can be found even when all cores of the cluster are utilized. This is an important indicator that in shared-memory multi-core and many-core platforms, the increase of cores must be followed by a simultaneous increase in the memory bandwidth (reduction of T_{acc}) or a reduction of the memory contention, for achieving essential gain.

Second, we evaluate the effect of *data partitioning* on the FMS schedulability. We fix T_{acc} to 550 ns and perform design space exploration for all cluster configurations with one to eight memory banks and one to eight cores. Fig. 12(b) shows the schedulability metric (cost $||barriers||_3$ of the optimized task and memory mapping solution), as the number of cores increases, for the configurations with one bank (blue line) or more than one banks (magenta line). Deploying more than two memory banks does not improve the optimized solutions. This can be partly explained by the low memory utilization of the FMS (fraction of cluster memory required for task data and communication buffers). Also, note that the cost of the solutions for one bank are only marginally worse than those for several banks. By carefully examining the optimized solutions, we conclude that in cases where no flexibility exists w.r.t. memory mapping, the optimizer tends to select the task mapping by maximally distributing

the tasks across the FTTS frames (not letting empty frames), such that a minimal set of tasks are executed in parallel in the same frame and hence, interfere on the memory bank. The periods of the FMS tasks and the considered dimensioning of the FTTS schedule (25 frames over $H = 5$sec) help in this direction, since several tasks have a high degree of freedom in the range of frames to which they can be mapped. We conclude that for the FMS, the combined task and memory mapping optimization performs efficiently, in the sense that the optimizer exploits maximally the flexibility in solving one problem (task mapping), when the flexibility of the second problem (memory mapping) is limited.

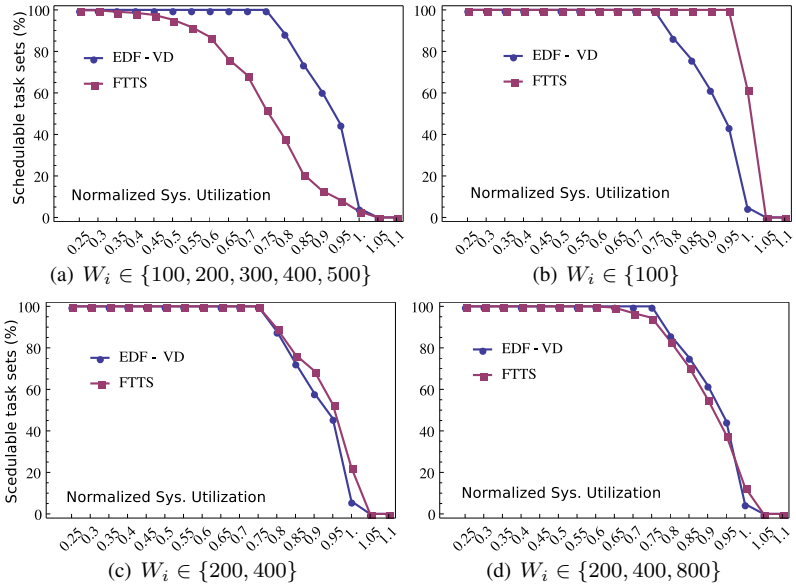
Finally, we evaluate the effect of the *incoming NoC traffic* at the cluster memory on the FMS schedulability. We assume that by selecting different regulation parameters and/or NoC routes for the data flow that is requested by task $\tau_{init,13}$, we can affect the maximum number of NoC Rx accesses to the local memory, δ, such that it varies within $\{403,803,1024,4096,8192\}$. We fix T_{acc} to 550 ns and the number of memory banks to 2. The FMS schedulability metric (cost $||barriers||_3$ of the optimized task and memory mapping solution) for increasing number of cores and for the different δ values is shown in Fig. 12(c). Again, schedulability is not severely affected by increased incoming NoC traffic. This is achieved in that the optimizer isolates the memory block corresponding to the fetched database entries (b_{27}), so that no or very few FMS tasks are interfering with the higher-priority $\texttt{Rx}_{13}$ requester, thus exploiting the memory accessing parallelism that the two memory banks enable. This way, the WCRT of the FMS tasks becomes immune to changes of δ. This observation justifies the benefits of the combined task and memory mapping optimization, where the interference of the tasks on shared platform resources is explicitly considered.

8 Comparison of FTTS to Existing Mixed-Criticality Scheduling Policies

In this section, we evaluate the efficiency of the FTTS policy in finding admissible schedules against state-of-the-art scheduling policies that have been proposed for mixed-criticality systems. For the following discussion, we distinguish previous mixed-criticality approaches into two categories: (i) scheduling policies that do not consider sharing of platform resources, such as the memory and NoC, and (ii) scheduling policies that target at eliminating or bounding the inter-core interference on shared platform resources. The FTTS policy falls into the second category. In the next sub-sections, we present quantitative or qualitative comparisons of FTTS to representative policies of the two categories.

8.1 FTTS vs Resource-Agnostic Scheduling Policies

The benefit of using FTTS in the presence of shared platform resources, e.g., memory banks and networks-on-chip, has been discussed and evaluated in Sec. 7. Here, we evaluate the limitations posed by the (flexible) time-triggered implementation of FTTS and their impact on schedulability. Hence, we compare FTTS to more dynamic, state-of-the-art MC scheduling strategies, particularly the EDF-VD algorithm



(a) $W_i \in \{100, 200, 300, 400, 500\}$ (b) $W_i \in \{100\}$

(c) $W_i \in \{200, 400\}$ (d) $W_i \in \{200, 400, 800\}$

Fig. 13 Schedulable task sets (%) vs. normalized system utilization for FTTS and EDF-VD ($m = 1$), $U_L = 0.05, U_L = 0.75, Z_L = 1, Z_L = 8, P = 0.3$, 1000 task sets per utilization point

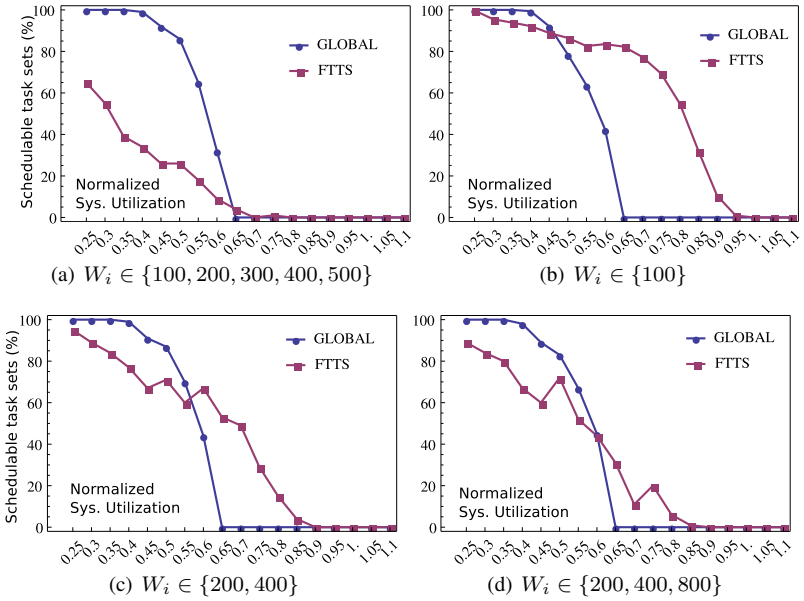
for single-core [7] and its variant GLOBAL for multicore systems [28]. Since these algorithms do not consider resource sharing, comparison is based upon synthetic task sets that require no accesses.

For task set generation we use the algorithm of [28] (TaskGen, Fig. 4 in [28]) for 2 criticality levels. Per-task utilization U_i is selected uniformly from $[U_L, U_H] = [0.05, 0.75]$ and the ratio Z_i of the level-2 utilization to level-1 utilization is selected uniformly from $[Z_L, Z_H]$ =[1,8]. The probability that a task τ_i has $\chi_i = 2$ is set to $P = 0.3$. Period W_i is randomly selected from the set $\{100, 200, 300, 400, 500\}$. Because FTTS cannot handle dynamic preemption, if the assigned execution time of a task is larger than the maximum frame length of the FTTS scheduling cycle, the task is split into sub-tasks, each "fitting" within a FTTS frame.

Fig. 13(a)-13(d) (FTTS vs. EDF-VD) and Fig. 14(a)-14(d) (FTTS vs. GLOBAL) show the fraction of task sets that are deemed schedulable by the considered algorithms as a function of the ratio U_{sys}/m (normalized system utilization). U_{sys} is defined in [28] as follows:

$$U_{sys} = \max\left(U_{\mathrm{LO}}^{\mathrm{LO}}(\tau) + U_{\mathrm{HI}}^{\mathrm{LO}}(\tau), U_{\mathrm{HI}}^{\mathrm{HI}}(\tau)\right), \tag{16}$$

where $U_{\mathrm{x}}^{y}(\tau)$ represents the total utilization of the tasks with criticality level x for their $y-$level execution profiles (LO≡1, HI≡2). Note that the normalized utilization increases from 0.25 to 1.10 in steps of 0.05. For each utilization point in the graphs, 100 or 1000 randomly generated task sets (as annotated in respective figure) are considered. To check schedulability of each randomly generated task set for **FTTS**, we use the optimization framework of Sec. 6.1 and check condition (2) for the optimized solution. For the design space exploration, we use the same configuration for the simulated annealing algorithm (Listing 1 of Sec. 6) as in Sec. 7 and a time budget of 10



(a) $W_i \in \{100, 200, 300, 400, 500\}$ (b) $W_i \in \{100\}$

(c) $W_i \in \{200, 400\}$ (d) $W_i \in \{200, 400, 800\}$

Fig. 14 Schedulable task sets (%) vs. normalized system utilization for FTTS and GLOBAL ($m = 4$), $U_L = 0.05, U_L = 0.75, Z_L = 1, Z_L = 8, P = 0.3$, 100 task sets per utilization point

minutes. In all cases, however, the optimizer converged to a solution in less than 5 minutes. Additionally, to check schedulability of each randomly generated task set for **EDF-VD** and **GLOBAL**, we check the sufficient conditions from [7,28]. These conditions are given below.

– For EDF-VD on single cores [7]:

$$U_{\mathrm{HI}}^{\mathrm{HI}}(\tau) + U_{\mathrm{LO}}^{\mathrm{LO}}(\tau) \cdot \frac{U_{\mathrm{HI}}^{\mathrm{LO}}(\tau)}{1 - U_{\mathrm{LO}}^{\mathrm{LO}}(\tau)} \leq 1. \tag{17}$$

– For GLOBAL on multicores with m cores [28]:

$$U_{\mathrm{LO}}^{\mathrm{LO}}(\tau) + \min\left(U_{\mathrm{HI}}^{\mathrm{HI}}(\tau), \frac{U_{\mathrm{HI}}^{\mathrm{LO}}(\tau)}{1 - 2 \cdot U_{\mathrm{HI}}^{\mathrm{HI}}(\tau)/(m+1)}\right) \leq \frac{m+1}{2}, \tag{18}$$

On single-core systems, FTTS faces two limitations compared to EDF-VD, i.e., the fixed preemption points and the time-triggered frames. EDF-VD is more flexible with scheduling task jobs as they arrive and can preempt them any time. The results of Fig. 13(a) show that as the utilization increases, EDF-VD can schedule 0 up to 52.9% ($U_{sys} = 0.85$) more MC task sets than FTTS (on average, 17.9% higher schedulability than FTTS). The impact of the FTTS limitations on schedulability becomes even clearer if we repeat the experiment such that these limitations are avoided. This happens when all tasks have the same period ($W_i = 100$), hence the FTTS cycle consists only of 1 frame. The corresponding results in Fig. 13(b) exhibit now reverse trends, with FTTS being able to schedule up to 57.2% ($U_{sys} = 1.0$) more task sets than EDF-VD (on average, 10.5% higher schedulability than EDF-VD). In fact, if we

consider safety-critical applications with harmonic task periods, such as the FMS of Sec. 7, the performance of FTTS is comparable to that of EDF-VD. This can be seen in Fig. 13(c) and 13(d), where the task periods for the generated task sets are selected uniformly from sets $\{200, 400\}$ (2 periods) and $\{200, 400, 800\}$ (3 periods), respectively. In the case of 2 harmonic periods, FTTS can schedule up to 16.2% (on average 2.2%) more tasks sets than EDF-VD. In the case of 3 periods, FTTS can schedule up to 8.1% more task sets ($U_{sys} = 1$), but on average across all utilization points, it schedules 1.2% less task sets than EDF-VD. The comparable performance of the two policies in terms of schedulability for equal or harmonic task periods is a significant outcome, given that FTTS was designed targeting at timing isolation rather than efficiency.

On multicores, we expect GLOBAL to perform more efficiently than FTTS not only because of the previously discussed advantages, but also because it enables task migration. Namely, several jobs of the same task can be scheduled on different cores and a preempted job can be resumed on a different core. The results of Fig. 14(a) show that the effectiveness of GLOBAL in finding admissible schedules for the generated task sets is up to 65% higher ($U_{sys} = 0.40$) than for FTTS. Recall, however, that the increased efficiency comes at the cost of ignoring the timing effects of shared resources which are not negligible especially in the presence of task migrations. If the limitations of FTTS are avoided as before, the results (Fig. 13(b)) are again reversed. Then, FTTS schedules up to 82.3% ($U_{sys} = 0.65$) more task sets than GLOBAL(on average, 20.8% higher schedulability than GLOBAL). Fig. 14(c) and 14(d) show the schedulability vs. utilization trends when the task periods are selected from the harmonic sets $\{200, 400\}$ and $\{200, 400, 800\}$, respectively. In the case of 2 harmonic periods, can schedule up to 53% (on average 3.8%) more tasks sets than GLOBAL. In the case of 3 periods, it can schedule up to 31% more task sets, but on average across all utilization points, it schedules 3.6% less task sets than GLOBAL. Therefore, schedulability is again comparable between the two policies when the task periods are harmonic.

It follows that FTTS, despite its imposed limitations for achieving timing isolation, e.g., the lack of dynamic preemption, the static partitioning of tasks among cores, and the fixed-length frames, can actually compete with state-of-the-art scheduling algorithms, which were designed with efficiency in mind. In other words, FTTS is not only a policy that enables global timing isolation for certifiability, but also a competent solution for efficient (processing, memory, communication) resource utilization in mixed-criticality environments.

8.2 FTTS vs Resource-Aware Scheduling Policies

The category of resource-aware scheduling includes policies, such as [50, 18, 34, 22], which were described in Sec. 2 in the context of mixed-criticality resource sharing. Currently, a direct comparison of FTTS to the approaches of these works is not applicable for the following reasons:

– The memory controllers suggested in [34, 22] are custom hardware solutions for mixed hard real-time and soft real-time systems. In our work, the proposed mixed-

safety-criticality scheduling policy and response time analysis are developed for a specific cluster-based platform model, motivated by the Kalray MPPA®-256 architecture. Namely, the considered system model is fundamentally different between [34, 22] and our work.
– The OS-supported memory throttling mechanisms proposed in [50, 18] do not consider the existence of a NoC on the platform and the interference on the shared memory by incoming traffic from the NoC, as commonly existing in cluster-based platforms. Unless these methods are extended to account for the NoC traffic, a comparison to the FTTS in terms of schedulability (based on the analysis presented in this paper) is not meaningful.

To the best of our knowledge, there is no previous work combining mixed-criticality scheduling with analysis of inter-core interference on *both* shared memory and NoC resources. This is the reason why a quantitative comparison between FTTS and existing policies of the resource-aware scheduling category cannot be provided.

9 Conclusion

This article extends the state-of-the-art for mixed-criticality systems by presenting a unified analysis approach for computing, memory, and communication scheduling. It targets modern cluster-based manycore architectures with two shared resource classes: a shared multi-bank memory within each cluster, and a network-on-chip (NoC) for inter-cluster communication and access to external memories. To model such architectures and the communication flows through the NoC, we extend and concretize the system model that was introduced in previous work [20], having the Kalray MPPA®-256 architecture as reference. Additionally, we introduce a protocol for inter-cluster communication with formally provable timing properties. For the scheduling of mixed-criticality applications on cluster-based architectures, we propose a mixed-criticality scheduling policy (FTTS), which enforces global timing isolation between applications of different criticalities in order to provide certifiability properties. This is achieved by allowing only applications of equal criticality to be executed in parallel and hence, interfere on the shared memory and communication infrastructure. Response time analysis for this policy, which was introduced in [20, 21] for systems with shared memory, is substantially extended to (i) model interference on the shared memory of a cluster by concurrently executing tasks in the cluster, but also by incoming traffic from the NoC; (ii) bound safely and tightly the end-to-end delays for data transfers through the NoC; (iii) model the incoming traffic from the NoC in the form of arrival curves from the real-time calculus; (iv) integrate the results of the extended memory interference analysis and the novel NoC analysis. Moreover, design exploration methods are presented targeting at the optimization of resource utilization within a cluster at the levels of computing (core utilization), memory (exploitation of internal memory structure for data partitioning), and communication (management of incoming traffic from a NoC). The applicability and efficiency of the optimization approach are demonstrated for an industrial implementation of a flight management system. Finally, the proposed scheduling policy is compared

quantitatively (in terms of schedulability) and qualitatively (when a direct comparison was not applicable) to state-of-the-art policies for mixed-criticality systems. The quantitative comparison shows that it performs better in terms of schedulability for harmonic workloads.

As future work, we would like to step from the system-level design optimization to the actual deployment of a mixed-criticality application, such as the flight management system, on a commercial many-core platform. The Kalray MPPA®-256 is a potential target platform since it is a concrete example of our abstract architecture model and its runtime environment provides support for intra-cluster barrier synchronization, explicit data partitioning among memory banks, and NoC flow regulation, all key features for the low-overhead implementation of the FTTS scheduler and the validity of the analysis methods.

Acknowledgements The authors would like to thank the anonymous reviewers for their valuable feedback. This work has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) project CERTAINTY under grant agreement number 288175.

References

1. The dol-critical framework for mixed-criticality applications on multicores. `https://www.tik.ee.ethz.ch/~certainty/download.html`.
2. European commission's 7th framework programme: Certification of real-time applications designed for mixed criticality (certainty). `www.certainty-project.eu`.
3. RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification, 1992.
4. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183 – 235, 1994.
5. J. Anderson, S. Baruah, and B. Brandenburg. Multicore operating-system support for mixed criticality. In *Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, 2009.
6. ARINC. ARINC 653-1 avionics application software standard interface. Technical report, 2003.
7. S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. Van der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *ECRTS*, pages 145–154, 2012.
8. S. Baruah, B. Chattopadhyay, H. Li, and I. Shin. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems*, 50(1):142–177, 2014.
9. S. Baruah and G. Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *RTSS*, pages 3–12, 2011.
10. S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *RTAS*, pages 13–22, 2010.
11. A. Burns and R. Davis. Mixed criticality systems: A review. `http://www-users.cs.york.ac.uk/burns/review.pdf`.
12. Certainty. D8.3 - validation results. Technical report, 2014.
13. C.-S. Chang. *Performance Guarantees in Communication Networks*. Springer, 2000.
14. R. L. Cruz. A calculus for network delay. i. network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.
15. B. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. de Massas, F. Jacquet, S. Jones, N. Chaisemartin, F. Riss, and T. Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *HPEC*, pages 1–6, 2013.
16. B. de Dinechin, D. van Amstel, M. Poulhies, and G. Lager. Time-critical computing on a single-chip massively parallel processor. In *DATE*, pages 1–6, 2014.
17. J. Diemer and R. Ernst. Back suction: Service guarantees for latency-sensitive on-chip networks. In *NOCS*, pages 155–162, 2010.
18. J. Flodin, K. Lampka, and W. Yi. Dynamic budgeting for settling dram contention of co-running hard and soft real-time tasks. In *SIES*, pages 151–159, 2014.

19. G. Giannopoulou, K. Lampka, N. Stoimenov, and L. Thiele. Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems. In *EMSOFT*, pages 63–72, 2012.
20. G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *EMSOFT*, pages 17:1–17:15, 2013.
21. G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Mapping mixed-criticality applications on multi-core architectures. In *DATE*, pages 1–6, 2014.
22. S. Goossens, B. Akesson, and K. Goossens. Conservative open-page policy for mixed time-criticality memory controllers. In *DATE*, pages 525–530, 2013.
23. S. Hahn, J. Reineke, and R. Wilhelm. Towards compositionality in execution time analysis-definition and challenges. In *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 2013.
24. H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *RTAS*, pages 145–154, 2014.
25. Y. Kim, J. Lee, A. Shrivastava, and Y. Paek. Operation and data mapping for cgras with multi-bank memory. In *LCTES*, pages 17–26, 2010.
26. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.
27. J.-Y. Le Boudec and P. Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*, volume 2050. Springer, 2001.
28. H. Li and S. Baruah. Global mixed-criticality scheduling on multiprocessors. In *ECRTS*, pages 166–175, 2012.
29. L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *PACT*, pages 367–376, 2012.
30. Z. Lu, M. Millberg, A. Jantsch, A. Bruce, P. van der Wolf, and H. T. Flow regulation for on-chip communication. In *DATE*, pages 578–581, 2009.
31. D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit. Platform 2012, a many-core computing accelerator for embedded socs: Performance evaluation of visual analytics applications. In *DAC*, pages 1137–1142, 2012.
32. W. Mi, X. Feng, J. Xue, and Y. Jia. Software-hardware cooperative dram bank partitioning for chip multiprocessors. In *Network and Parallel Computing*, volume 6289 of *LNCS*, pages 329–343. 2010.
33. M. Mollison, J. Erickson, J. Anderson, S. Baruah, J. Scoredos, et al. Mixed-criticality real-time scheduling for multicore systems. In *ICCIT*, pages 1864–1871, 2010.
34. M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware support for wcet analysis of hard real-time multicore systems. In *ISCA*, pages 57–68, 2009.
35. R. Pathan. Schedulability analysis of mixed-criticality systems on multiprocessors. In *ECRTS*, pages 309–320, 2012.
36. R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha. Coscheduling of cpu and i/o transactions in cots-based embedded systems. In *RTSS*, pages 221–231, 2008.
37. Y. Qian, Z. Lu, and W. Dou. Analysis of communication delay bounds for network on chips.
38. Y. Qian, Z. Lu, and W. Dou. Analysis of worst-case delay bounds for on-chip packet-switching networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(5):802–815, May 2010.
39. J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. Pret dram controller: bank privatization for predictability and temporal isolation. In *CODES+ISSS*, pages 99–108, 2011.
40. D. Tamas-Selicean and P. Pop. Design optimization of mixed-criticality real-time applications on cost-constrained partitioned architectures. In *RTSS*, pages 24–33, 2011.
41. L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *ISCAS*, pages 101–104, 2000.
42. L. Thiele and N. Stoimenov. Modular performance analysis of cyclic dataflow graphs. In *EMSOFT*, pages 127–136, 2009.
43. S. Tobuschat, P. Axer, R. Ernst, and J. Diemer. Idamc: A noc for mixed criticality systems. In *RTCSA*, pages 149–156, 2013.
44. S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *RTSS*, pages 239–243, 2007.
45. E. Wandeler, A. Maxiaguine, and L. Thiele. Performance analysis of greedy shapers in real-time systems. In *DATE*, pages 444–449, Munich, Germany, 2006.

46. E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System architecture evaluation using modular performance analysis - a case study. *International Journal on Software Tools for Technology Transfer*, 8(6):649 – 667, 2006.
47. R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966 –978, 2009.
48. Z. P. Wu, Y. Krish, and R. Pellizzoni. Worst case analysis of dram latency in multi-requestor systems. In *RTSS*, pages 372–383, 2013.
49. H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *RTAS*, pages 155–166, 2014.
50. H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *ECRTS*, pages 299–308, 2012.
51. J. Zhan, N. Stoimenov, J. Ouyang, L. Thiele, V. Narayanan, and Y. Xie. Designing energy-efficient noc for real-time embedded systems through slack optimization. In *DAC*, pages 1–6, 2013.

Appendix: Notation Summary

Notation	Meaning	Source
System Model - Sec. 3		
τ	Task set	Fixed
L	Number of criticality levels	Fixed
W_i, χ_i	Period and criticality level of task τ_i	Fixed
$C_i(\ell)$	Execution profile with lower and upper bounds on execution time (e_i) and number of memory accesses (μ_i) of τ_i at level $\ell \leq \chi_i$	Fixed
$C_{i,deg}$	Execution profile of τ_i at level of assurance $\ell > \chi_i$	Fixed
$\mathcal{D}ep$	Dependency Graph	Fixed The edges (dependencies) are known, but their weight (minimum distance constraint) is determined from NoC analysis (Sec. 5.2)
$\mathcal{P}$	Set of processing cores on a target cluster	Fixed
T_{acc}	Memory access latency	Fixed
$\mathcal{M}_\tau$	Mapping of tasks of τ to cores of $\mathcal{P}$	Optimized in Sec. 6
Remote Fetch Protocol - Sec. 3.2		
$\tau_{init,i} \rightarrow \tau_i$	Initiating task for remote data fetch, task using fetched data	Fixed
WCNT	Worst-case time for transfer of notification from $\tau_{init,i}$ to listener in remote cluster	Computed in Sec. 5.2 (Eq. 12)
WCRST	Worst-case time for set-up of DMA transfer in remote cluster	Based on measurements
WCDFT	Worst-case time for complete transfer of data from remote cluster	Computed in Sec. 5.2 (Eq. 12)
Flexible Time-Triggered and Synchronization based Scheduling - Sec. 4		
H	FTTS cycle	Computed as hyper-period of τ
$\mathcal{F}$	Set of FTTS frames	Computed after selecting frame lengths
$\mathcal{L}_f$	Length of FTTS frame f	Selected manually
$barriers(f,l)_k$	Worst-case length of k-th sub-frame in frame f at level ℓ	Computed for a given $\mathcal{M}_\tau, \mathcal{M}_{mem}$ in Sec. 5.1 (Eq. 6)
Response Time Analysis - Sec. 5		
$\mathcal{I}$	Memory interference graph	Consisting of $\mathcal{E}_1, \mathcal{E}_2$
$\mathcal{E}_1$	Mapping of tasks to memory blocks	Fixed
$\mathcal{E}_2$	Mapping of memory blocks to memory banks	Optimized in Sec. 6
$\mathcal{M}_{mem}$	Equivalent to $\mathcal{E}_2$	Optimized in Sec. 6
D	Mutual delay matrix	Computed in Sec. 5.1.1 and 5.1.2
`Rx`	Special node in $\mathcal{I}$ indicating a high-priority NoC Rx access	Added to $\mathcal{I}$ in Sec. 5.1.2
δ	Weight of edge between `Rx` and accessed memory block(s)	Computed in Sec. 5.2 (Eq. 13)
$WCRT_i(f,l)$	Worst-case response time of τ_i in frame f at level ℓ	Computed in Sec. 5.1.1 (Eq. 4)
$CWCRT_{p,k}(f,l)$	Worst-case response time of tasks executing on core p in the k-th sub-frame of frame f at level ℓ	Computed in Sec. 5.1.1, updated in Sec. 5.1.2
Design Optimization - Sec. 6		
$\|barriers\|_3$	3rd norm of $barriers$ for all $f \in \mathcal{F}, \ell \in \{1,\ldots,L\}$	Computed for each candidate $\mathcal{M}_\tau$ (Eq. 6)
T_0	Initial temperature	Parameter of SA algorithm
a	Temperature decreasing factor	Parameter of SA algorithm
T_{final}	Final temperature	Parameter of SA algorithm
$time_{max}$	Time budget	Parameter of SA algorithm