Response Time Analysis of Dataflow Applications on a Many-Core Processor with Shared-Memory and Network-on-Chip

Amaury Graillat, Claire Maiza, Matthieu Moy, Pascal Raymond, Benoît Dupont de Dinechin

Response Time Analysis of Dataflow Applications on a Many-Core Processor with Shared-Memory and Network-on-Chip

Amaury Graillat
amaury.graillat@univ-grenoble-alpes.fr
Univ. Grenoble Alpes, CNRS,
Grenoble INP, VERIMAG
38000 Grenoble, France

Claire Maiza
claire.maiza@univ-grenoble-alpes.fr
Univ. Grenoble Alpes, CNRS,
Grenoble INP, VERIMAG
38000 Grenoble, France

Matthieu Moy
matthieu.moy@univ-lyon1.fr
Univ Lyon, EnsL, UCBL, CNRS, Inria,
LIP
F-69342 LYON Cedex 07, France

Pascal Raymond
pascal.raymond@univ-grenoble-alpes.fr
Univ. Grenoble Alpes, CNRS,
Grenoble INP, VERIMAG
38000 Grenoble, France

Benoît Dupont de Dinechin
bddinechin@kalray.eu
Kalray S.A
38330 Montbonnot-Saint-Martin
France

ABSTRACT

We consider hard real-time applications running on many-core processor containing several clusters of cores linked by a Network-on-Chip (NoC). Communications are done via shared memory within a cluster and through the NoC for inter-cluster communication. We adopt the time-triggered paradigm, which is well-suited for hard real-time applications, and we consider data-flow applications, where communications are explicit.

We extend the AER (Acquisition/Execution/Restitution) execution model to account for all delays and interferences linked to communications, including the interference between the NoC interface and the memory. Indeed, for NoC communications, data is first read from the initiator's local memory, then sent over the NoC, and finally written to the local memory of the target cluster. Read and write accesses to transfer data between local memories may interfere with shared-memory communication inside a cluster, and, as far as we know, previous work did not take these interferences into account.

Building on previous work on deterministic network calculus and shared memory interference analysis, our method computes a static, time-triggered schedule for an application mapped on several clusters. This schedule guarantees that deadlines are met, and therefore provides a safe upper bound to the global worst-case response time.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

RTNS'19, Toulouse,

ACM ISBN 000…$00.00
https://doi.org/0.0.0

ACM Reference Format:
Amaury Graillat, Claire Maiza, Matthieu Moy, Pascal Raymond, and Benoît Dupont de Dinechin. 2019. Response Time Analysis of Dataflow Applications on a Many-Core Processor with Shared-Memory and Network-on-Chip . In *rtns.* ACM, New York, NY, USA, 9 pages. https://doi.org/0.0.0

1 INTRODUCTION

Time-critical systems are systems whose response time is part of the specification and for which a too long response time could have dramatic consequences.

Many-core platforms are based on a set of multi-core clusters connected through a Network-on-Chip (NoC). They offer more computational performances than single-core platforms, and offer interesting properties for real-time: partitioning cores into clusters allows spatial isolation, and individual cores can be simple enough to allow a precise worst-case execution time (WCET) analysis. In this paper, we consider Kalray's MPPA2 many-core processor, which is well-suited to hard real-time applications [26]. Its architecture and implementation allow minimizing and bounding interferences, but not fully eliminating them. Hence, a global analysis of applications must be performed in order to ensure that their timing constraints are satisfied.

This paper deals with hard real-time applications, for which deadlines are strict, and provides safe upper bounds on the worst-case response time. We consider precisely a non-preemptive execution model where software is divided into a set of periodic tasks communicating through channels. Each task follows a variant of the Acquisition/Execution/Restitution (AER) execution model [10]: tasks start when their input data is ready in the local memory. The first phase is the execution itself. Then, data is transmitted directly to the target tasks. In other words, we follow a remote-write policy, and do not need the "Acquisition" phase of the AER model. When the target task is located in the same cluster, data is written directly to its input buffer in the local memory; when the target task is located on a different cluster, data is remotely written over the NoC. Such a set of tasks can be extracted automatically from a program written in a synchronous language such as Lustre [17] or its industrial variant Scade [5], or other high-level languages such

as Simulink [20]. Note that this AER execution model is a phased execution model that is also referred to as read/execute/write in the literature [4].

Since we consider hard real-time applications, we optimize the application and analysis for the worst-case. Non-preemptive, static, time-triggered scheduling is therefore a well-suited execution model. Since we consider periodic applications, we need to check the schedulability of one period (or hyper-period in case we use hyper-period expansion), and the rest of the execution will repeat the schedule of this period. For each task, a release date (arrival time) is computed, defined as the offset within a period where the task is allowed to start. Each release date is computed in such a way that the input data for the task are guaranteed to be available at this time. If the data is available earlier, the task still waits for the release date to start. This is not a limitation in a hard real-time context, since starting earlier would not improve the Worst-Case Response Time (WCRT) of the application. On the other hand, a static time-triggered schedule allows a finer analysis of the interference than a more dynamic schedule: the time interval when a task can execute is known statically. Tasks start at their release date, and execute without preemption no longer than their WCRT. This knowledge is exploited in the interference analysis: tasks whose execution intervals do not overlap cannot interfere. In other words, time-triggered schedules allow timing isolation of tasks within a period.

Prior work has been carried out to compute response-time analysis within a cluster [1, 25] or a worst-case traversal time for the NoC [8]. However, there is little research on computing a worst-case response time that considers application mapped to more than one cluster with a complete model of the NoC interference on local memories (see Section5). Considering the interference on memory accesses and the latency due to NoC communications separately is not sufficient: we need to take into account the interference between NoC communications and memory accesses, since a NoC communication implies memory read accesses on the sending side and write accesses on the receiving side.

We model NoC transmissions by inserting additional endpoint tasks such that a global response-time analysis may be applied. This response-time analysis takes the interference on shared resources (shared bus and memory) into account, including both interference between software tasks and interference between software tasks and the NoC transmission (TX) and reception (RX) engines.

Contribution. The contribution of this paper is a global model to compute the task release date for applications with both shared-memory and NoC transfers. We propose a new execution model where NoC communications are implemented with multiple tasks, each one having its own release date. We exploit hard-real time software properties and hardware mechanisms to guarantee the absence of interference on some tasks, and to avoid circular dependencies in the timing analysis. We consider a Remote Direct Memory Access (RDMA) NoC where a core can write to a remote memory address through the NoC. Furthermore, we integrate the worst-case NoC transmission delay in this model to satisfy data dependencies and tighten bounds on the memory interferences from the NoC.

Section 2 presents the context of this work: the hard-real time software properties, the Kalray MPPA2 processor, a method to compute the task release dates and a method to compute the WCTT on the NoC. Section 3 explains the sources of interferences we focus on, and presents our new WCRT analysis for handling the NoC interferences. Section 4 gives the results of the experiments on the Kalray MPPA2 processor. Related work is discussed in Section 5.

2 CONTEXT

2.1 Hard Real-Time Application Model

We consider reactive systems, operating in interaction with their environment, which generally involves the physical world. They are used in domains of application like control engineering in industry (power plants) or transportation (avionics, train, automotive). These systems are hard-real time, meaning that each reaction must occur before a strict, fixed deadline.

Such applications may be developed directly in low-level languages (e.g. C), but more often, they are developed using model-based design tools, like Scade Suite (avionics) or Simulink (automotive, industrial). In this case the task's code is generated automatically and the compilation process may give useful guarantees on the application execution. Code generation tools targeting sequential execution is available and deployed industrially, see KCG [5]. The area of code generators targeting multi-core is more recent: an extension of KCG, the Multicore Code Generator (MCG) to generate parallel programs was proposed in [7, 22]. MCG deals with the target-independent code generation, and requires a complementary integration tool to deal with the specificities of the target architecture like memory access interference and NoC communications. A research prototype of code generator from Lustre programs targeting Kalray's MPPA2 processor is proposed in [14]. We base the present work on this code generator; the same principles could apply to an integration tool for MCG.

We consider applications as sets of tasks generated from high-level languages and implement them on many-core platforms; their characteristics influence our solution:

- They are hard real-time, thus only the worst-case execution/reaction time matters: enhancing the average or best case is irrelevant.
- The communications between the tasks are statically known: each communication is oriented and described by a "channel" information that includes the type of the data, the producing and the consuming task. When the application comes from a high level data-flow design (Scade, Simulink), channel information is readily available.
- Each functional task is a purely computational piece of code, that generally follows the AER (Acquisition/Execution/Restitution) policy. This is the case when the code is automatically generated from Scade or Simulink. This property is interesting since it is a starting point for implementing a synchronization mechanism: a task requires all its inputs before producing any output.

At this point in the code generation flow, the application is therefore a tasks graph, with dependencies between tasks. Tasks follow a variant of the AER model where the Acquisition phase is not needed: tasks write directly to the input buffer of the receiving

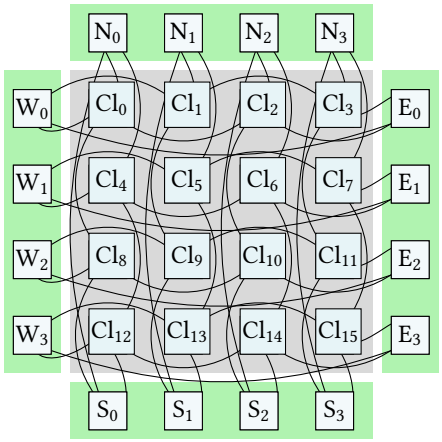


Figure 1: Overall view of the MPPA processor. Cl_i squares are clusters (containing 16 cores each), and $\{N,S,E,W\}_i$ (North/-South/East/West) squares are NoC routers part of the I/O clusters.

task. Data dependencies are annotated in the task graph with the amount of data transmitted. Classical WCET analysis techniques (e.g. OTAWA [3]) are used to compute each task's WCET (not accounting for inter-tasks interferences).

2.2 The Kalray MPPA2 Processor

The MPPA2 is a many-core processor composed of 16 compute clusters and 4 I/O clusters, pictured in Figure 1. Each compute cluster is made of 16 cores and a shared memory. The shared memory is composed of 16 banks with independent arbiters. For a more comprehensive description, see [9].

Clocks of the clusters are mesochronous, i.e. they have the same frequency but may have different phase. Each cluster contains a Debug Support Unit (DSU), containing a cycle counter register. This counter is initialized synchronously on the whole processor and can be used to synchronize all compute cores [15], making the MPPA a suitable platform for time-triggered processor-wide scheduling.

Fig. 2 shows a closer view of the path taken by a communication between two clusters. Data is transmitted from the memory of a source cluster to the memory of another cluster. This is the communication we focus on in this paper.

(1) Core 1 reads the data in the memory and copies this data to the buffer of a TX (transmission) engine. We call `copy_noc(tx_engine, buffer, size)` the piece of code executing on the core that takes care of this copy. In a cluster, accesses to TX engine buffers from the different cores are arbitrated on a single bus. Consequently, concurrent requests can be delayed due to interferences. Data is retained in the buffer.
(2) After the data is fully copied to the buffer of the TX engine, the communication is initiated and the packet is sent to the NoC. This is triggered by a core, that emits the end of transmission (EOT). The code executing on the core is a non-blocking procedure that ends the transmission on the sender side and initiates the transmission from the TX buffer to the NoC. We call it `EOT_noc(tx_engine)`.
(3) Then, the packet traverses the NoC. The worst-case latency from the initiation to the destination cluster is called WCTT. Packets enter the destination cluster through the NoC local link.
(4) The RX engine is responsible for copying the received data to the shared memory (5). Concurrent writes to the same memory bank from the NoC RX and from the cores are arbitrated with a fixed-priority policy. Writes from the NoC have the highest priority. Cores have lower priority and are arbitrated with each other in round robin. There is one arbiter per memory bank: this allows spatial isolation since an access to one bank does not interfere with accesses to other banks. A typical scenario is that writes from the NoC RX can interfere with accesses done during the Execute phase of a task running before the receiving task, on the same core hence accessing the same memory bank.

We analyze in details the possible interferences at each stage later in Section 3.1.

2.3 Multi-Core Interference Analysis (MIA)

We distinguish the WCET of a task in isolation, i.e. when there are no memory interferences, and the Worst Case Response Time (WCRT) of the task which accounts for interferences (there are no task preemptions to consider).

Memory interferences have an effect on the execution time of the tasks. Conversely, the execution time of each task may impact the execution time of the others, since it changes the set of tasks concurrently accessing the memory. Consequently, the whole application has to be analyzed globally and information about the release date and response time of each task is required to analyze the interferences. A time-triggered execution ensures that tasks execute between the statically known release date and deadline.

Rihani *et al.* [25] introduce an algorithm computing the release dates and WCRT of the tasks. The algorithm takes as input a task graph (tasks, dependencies between tasks, and optionally minimal release dates for some tasks). For each task, it needs the WCET in isolation and the worst-case memory demand (number of accesses to each memory bank). The mapping and execution order of tasks on each core is known. Mapping and scheduling are out of the scope of our work; an external tool is used as a black-box. Based upon this information, MIA computes a static time-triggered schedule, i.e. a release date and a WCRT, accounting for interference for each task.

The algorithm starts from the WCET in isolation of the tasks and iteratively adds the delays due to the interferences reconsidering each time the set of interfering tasks. It relies on the memory demand of each task and a model of the memory arbiter. This algorithm has been implemented in the MIA tool[1]. Our contribution is to integrate a NoC model in release dates and WCRT analysis performed by MIA. This allows analyzing applications mapped to

[1] http://www-verimag.imag.fr/Multi-core-interference-Analysis.html

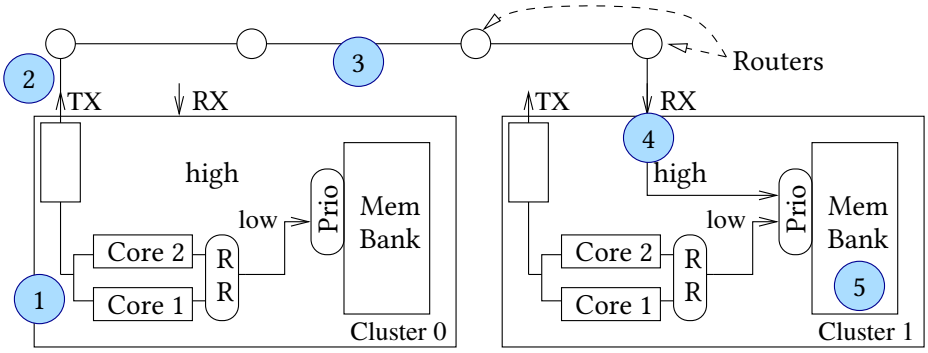


Figure 2: NoC communication path from a core (1) to a remote memory bank (5). In each cluster, only one bank and one TX engine are represented. (4) is the memory bank arbiter.

multiple clusters, which rely on both shared-memory and NoC transmission.

2.4 NoC Worst-Case Traversal Time Bounds

When the traffic at the NoC input is known, the estimation of a WCTT bound can be simplified and the absence of buffer overflow can be checked. Several methods exist to bound latency of packets through the NoC.

Deterministic Network Calculus (DNC) is a theory dedicated to the performance analysis of computer networks such as ATM and Internet. More recently, it has been used for Avionics Full Duplex Switched Ethernet (AFDX) networks certification [12, 16]. DNC relies on an upper-bound on the input traffic called arrival curve. This arrival curve can be ensured by bandwidth limiter. Then, the methods gives service curve of the network's elements. Operation on these curves allows computing an upper-bound on the latency for each flow and an upper-bound of the buffer level.

Recursive Calculus [11] (RC) is a method to bound the WCTT of best effort traffic. Ayed *et al.* [2] applied the RC theory to handle both the MPPA2 and the Tilera TILE64 many-core processor.

In this work, we use the method from [6] which is based on the DNC framework, applied to the MPPA's NoC. It provides latency bounds from source to destination memory.

3 HANDLING NOC INTERFERENCES IN WCRT ANALYSIS

3.1 Possible Sources of Interference

In this section, we enumerate and focus on the sources of interferences along the transmission path detailed in Figure 2. For each source of interference, we explain how it will be handled in our analysis.

The first source of interference is encountered already at point (1) that is the transmission from a core to the NoC. In the MPPA2 architecture, the cluster's TX engines are accessed by the cores through a single bus. As a consequence, it may introduce interferences and delays between tasks of the cluster accessing the TX engines if accesses are done concurrently. Instead of analyzing these interferences, our solution is to avoid them: in our execution model we forbid `copy_noc` and `EOT_noc` from being executed at the same time in a cluster by adding artificial edges in the task graph. This way, we can guarantee that there is no concurrent access to the NoC TX, hence no interference between these accesses. This method to avoid interference is similar to the work of Melani *et al.* [21], where tasks are scheduled in a way that forbids concurrent accesses of write tasks to avoid contention. We add artificial edges to the dependency graph to enforce a total order and no concurrence between tasks accessing the NoC TX, i.e. steps (1)+(2), before running the scheduling algorithm.

Interference between flows transmitted on the NoC (3) are dealt with by the DNC analysis. DNC is a very general approach, and can deal with interference with other task models including non-periodic traffic. It does not need to make assumption about the execution model of other sources of traffic on the same chip. It only relies on configuration of hardware traffic limiters in the TX engines of each cluster. Since data being sent come from the TX's input buffer, the TX engine only interferes with other flows in the NoC during this stage, but no interference happen with other elements in the sending cluster.

The next source of interference is at point (4): the bus arbiter to reach a memory bank is shared by the cluster and the receiving task. At this point we partially avoid interferences by limiting the cores that are impacted by the reception of a NoC communication. The architecture allows attributing one memory bank per core. The data and code corresponding to a task are attributed to the bank of the core executing this task. NoC reception buffers are located in the memory bank of the destination task. Packet receptions trigger write access to the memory bank of the buffer. This write access may only interfere with the memory accesses of the destination task and other write phases of tasks on the same core. This limited interference scope avoids interfering with all tasks executed on all cores.

Furthermore, as in [15], the task execution is time-triggered and the implementation relies on a global clock synchronization protocol.

In summary, the only interference related to NoC communications happen when the receiving task is writing, to the memory bank of the destination task.

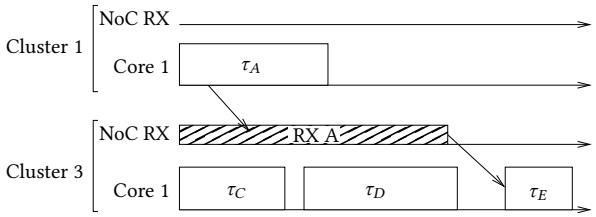


Figure 3: NoC Transmission from τ_A to τ_E on different clusters. RX A represents the possible interference from NoC to the memory of τ_E.

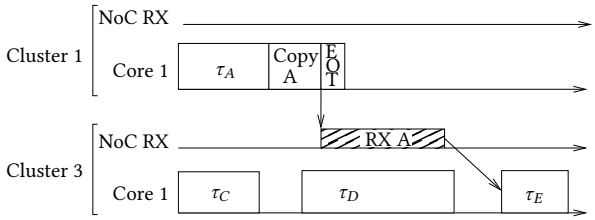


Figure 4: NoC Transmission from τ_A to τ_E on different clusters. The NoC communication is represented with two tasks: Copy and EOT.

3.2 Motivational Example and Naive Approach

To illustrate our approach, consider the data-flow application depicted in Fig. 3. It is mapped on several clusters. Core 1 of Cluster 3 executes tasks C, D and E. Task E depends on outputs of task A. Task A runs on Core 1 of Cluster 1 and sends its output through the NoC as soon as possible. The transmission is represented with the hatched rectangles. The WCET and the memory demand are known for each task. The WCTT and the size of the transmission are known.

Our aim is to compute the WCRT and release date r for each task taking into account both memory interferences between the cores and from the NoC. In other words, we compute the first date when the inputs of the tasks are guaranteed to be available and the task can start. For instance, in Fig. 3, τ_E starts after τ_D has finished and the data from τ_A has been received. Data transmitted through the NoC are written to the memory bank of τ_E, hence possibly interfering the other tasks of the core. Our purpose is to compute the first and the last date when the transmission may write to the destination bank. The first and the last date are respectively the beginning and the end of the hatched rectangle of Fig. 3.

τ_A executes three procedures in sequence: the computation, `copy_noc` and `EOT_noc`. The best-case execution time of the tasks and the best-case traversal time of the NoC transmission are not known. As a consequence, the first date when the NoC transmission may write to the memory is 0.

The last date when the NoC transmission writes to the remote memory is the last date when the sending task has finished, plus the WCTT. Consequently, the NoC transmission is represented with a task RX_A starting at the same time as τ_A of duration $WCRT(\tau_A) + WCTT$.

The RX task appears in the model used for the analysis but does not correspond to any physical task in the implementation. It is an over approximation of the interval where the NoC reception may occur.

This method is quite simple but leads to a pessimistic computation since the duration of RX_A is longer than required. In fact, RX_A starts after the call to the `EOT_noc` procedure, hence at a date strictly greater than 0. Since the RX accesses have the highest priority in the shared-memory arbiter, an overapproximation of the execution interval of RX has important consequences: all memory accesses by concurrent tasks could be delayed by the number of accesses done by the RX task. It is therefore important to minimize the duration of RX_A to limit the number of tasks that may interfere with RX and therefore minimize its impact on the analysis.

This "naive" approach does not guarantee the mutual exclusion of accesses to the TX engine. However, in our experiments the computed schedules are such that no concurrent access occur, hence the comparison with our proposed solution (which does ensure mutual exclusion) is fair.

3.3 Proposed Solution

As stated previously, a NoC transmission is composed of a copy of the data to the TX engine's buffer (`copy_noc`) and the `EOT_noc` procedure to initiate the transfer. The packet is guaranteed to stay in the buffer until the communication is initiated with `EOT_noc`.

Our proposed method, illustrated in Fig. 4, improves the naive solution by splitting τ_A into a computation task, and two extra tasks to implement the communication: the copy task and the EOT task, each one having its own release date. Compared to the AER phased model, we split the Restitution (R) phase into a phase to copy data (`copy_noc`) and one to initiate the NoC (`EOT_noc`). The release dates of these tasks are computed the same way as for other tasks by MIA. Note that in this model, `EOT_noc` has its own release date, which does not depend on the best case execution time of precedent tasks. The NoC transmission cannot start before this date. Without knowledge about the NoC's best case transmission time, the first date when the transmission can write to the remote memory can be set to the start of `EOT_noc` (i.e. the end date of `copy_noc`). Then, the duration of RX_A is $WCTT + WCRT(\texttt{EOT\_noc})$. This improvement makes the analysis less pessimistic than the naive approach since the duration of the RX task no longer includes the task computation.

To split the Restitution phase, we modified the code generator so that communication between nodes implying the NoC generate two tasks for `copy_noc` and `EOT_noc`. Extra care must be taken to avoid cyclic dependencies in the analysis, as explained in the next section.

3.4 Handling Communications Cyclic Dependencies

WCRT and cycle problem. This section presents the problem of cycle that can occur when the method is applied to a program where two clusters are sending data to each other through the NoC. The issue is that the duration of the RX tasks depends on the duration of the EOT, and when the EOT task performs memory accesses, its duration depends on the memory access from the NoC.

For example, in Fig. 5, Cluster 1 and Cluster 3 are communicating in both ways. The WCRT of RX_1 is $\text{WCTT}_{1\to3}$ + $\text{WCRT}(\text{EOT}_1)$.

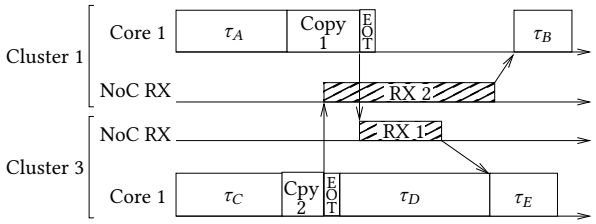


Figure 5: Cycle problem where two clusters are communicating in both ways.

Since RX is arbitrated with the highest priority, we do not need to consider other SMEM accesses interfering with the reception of data. However, WCRT(EOT_1) depends on memory interferences, hence it depends on WCRT(RX_2) and the release date of RX_2, may or may not allow the analysis to conclude that RX_2 and EOT_1 have no overlap hence no interference. Similarly, the WCRT of RX_2 depends itself on the WCRT of RX_1 and its release date. In other words, the duration of an RX task depends on the duration of the corresponding EOT task.

The difference with the usual task graphs analyzed by the MIA tool is that task RX_1 does not depend on EOT_1, but its WCRT depends on WCRT(EOT_1). This kind of dependency is not supported by existing interference analysis.

Cycle Breaking Method. This method relies on an implementation of the TX and EOT tasks ensuring that EOT does not perform any data memory access. Furthermore, we guarantee that there is no concurrent task accessing the TX engine bus. This implementation ensures WCRT(EOT) = WCET(EOT) since there is no interference.

The duration of RXs is computed using the WCET of EOT tasks and the WCTT. Since we consider a response time for EOT which does not depend on interferences, only one instance of the release date computation tool is required even if there are cycles.

EOT is just a single store instruction writing in a hardware register. It does not perform memory access other than the fetch, and since the MPPA architecture has different arbiters for memory banks and TX, this register access cannot interfere with accesses to memory banks. Nevertheless, the fetch for this instruction can lead to an instruction cache miss. The implementation must ensure that this miss occurs in the `copy_noc` function instead of the `EOT_noc` function. There are several methods: either a `__builtin_prefetch` can be called to fetch the instruction into the cache at the end of the `copy_noc` task, or the code alignment can be chosen to make sure that the instruction cache line containing EOT has already been loaded when it executes. We use the latter in the experiments.

4 EXPERIMENTS

In this section, we evaluate our method concerning two aspects: the gain obtained compared to the naive method, and the efficiency of the guaranteed time-triggered execution.

4.1 Evaluation of the timing analysis model

In this section we show how much our method improves the naive version introduced in Section 3.2. For this purpose we implemented

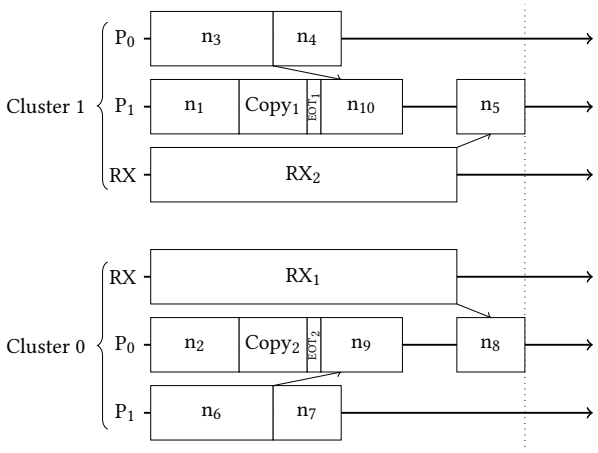


Figure 6: Schedule using the naive approach on an application mapped to two clusters.

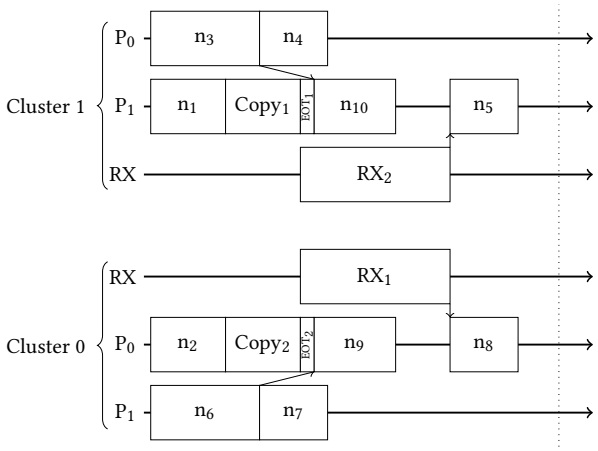


Figure 7: Final schedule using the improved approach on an application mapped to two clusters.

the naive method without our method to break the cycle: we over-approximate the duration of the calculation+Copy+EOT tasks instead. This duration is used to compute the duration of the RX task. Then we apply the release date computation procedure from Rihani *et al.* [25]. The estimated response time of EOT is then injected in the next iteration.

For this experiment, we consider an application composed of 10 tasks mapped on two clusters, using two cores per cluster. The WCTT is 120 cycles and each NoC transfer is 20 words long. For the improved version, the EOT task has a WCET of 20 cycles. $Copy_1$ and $Copy_2$ have a WCET of 100 cycles and 30 words of memory demand. Each task $n_1 ... n_{10}$ has WCET of 100 cycles and a memory demand of 30 words of 32-bits. For the naive version, we consider tasks including all stages: (n_1+$Copy_1$+EOT_1) and (n_2+$Copy_2$+EOT_2) have a WCET of 220 cycles and 60 words of memory demand.

In the case of the naive approach, the cyclic dependency requires to re-launch MIA with an updated duration for the RX tasks until convergence (2 iterations are needed for this example).

The naive and improved schedules are depicted respectively in Fig. 6 and Fig. 7. A detailed comparison is given in Table 1. To ease comparison, we show WCRT of Copy and EOT tasks which are normally included in n_1 and n_2 in the naive implementation, but n_1 + Copy_1 + EOT_1 is considered as a single task in the analysis (and likewise for n_2 + Copy_2 + EOT_2). The **reduced WCRT's** of the RX tasks and of n_3 lead to a **reduced global WCRT** and the one of task n_3, and an **earlier release** date for 6 tasks. In the naive version, RX_1 interferes with n_2 + Copy_2 + EOT_1 and n_9 which run on the core associated with the target memory bank, and with n_6, which writes to the target memory bank. Since RX_1 writes 20 words with a high priority, it delays tasks that may interfere by 20 cycles. In the proposed version, RX_1 is guaranteed to start later, and does not interact with n_2, $textCopy_2$ nor n_6, which are therefore 20 cycles faster.

This globally shows that our model is good when the NoC communication leads to memory interference.

Task	Release date		WCRT	
	Naive	Improved	Naive	Improved
Application			**550**	**540**
Copy_1	**130**	**110**	**100**	**110**
Copy_2	**130**	**110**	**100**	**110**
EOT_1	**230**	**220**	20	20
EOT_2	**230**	**220**	20	20
RX_1	**0**	**220**	**450**	**220**
RX_2	**0**	**220**	**450**	**220**
N_1	0	0	**130**	**110**
N_2	0	0	**130**	**110**
N_3	0	0	**180**	**160**
N_4	**180**	**160**	100	100
N_5	**450**	**440**	100	100
N_6	0	0	**180**	**160**
N_7	**180**	**160**	100	100
N_8	**450**	**440**	100	100
N_9	**250**	**240**	120	120
N_{10}	**250**	**240**	120	120

Table 1: Comparison of WCRT and release dates with both methods (in bold where there is a difference).

4.2 Evaluation of our guaranteed Time-triggered execution

This experiment is performed on two applications: the ROSACE case study and a synthetic benchmark running on 64 cores. Our improved method with cycle breaking is used to implement them.

We evaluate the Worst-case response time of our Time-Triggered (*WCRT TT*) solution, and compare it to 2 different implementations: (1) a *single-core* implementation, where all tasks are executed sequentially, gives a basic execution time to measure the speedup due to parallelization; (2) a best effort implementation, where each task starts as soon as possible, gives the Event Triggered execution time (*measured ET*); this *Measured ET* does not offer any real-time guaranties, but gives a rough (and unreachable) upper-bound to the speedup due to parallelization.

To simplify the experimental setup, the WCET in isolation is evaluated using measurements on the Kalray MPPA2 running at 400 Mhz. Stronger guaranties would be obtained using a WCET analysis tool like OTAWA [3]. This would not change the interference analysis step, hence would not change the conclusions of this experiment.

Note that, in these experiments, there is no computation in parallel with the NoC communications due to the structure of these benchmarks.

4.2.1 ROSACE Case Study. ROSACE [23] is a case-study whose structure is inspired by true avionic control applications. Only altitude is controlled, hence it does not require heavy computation. It is composed of a controller and an aircraft simulator. We implement the whole application on the MPPA2. The simulator and the controller run on two different clusters communicating through the NoC. The solution relies on 2 cores of the environment cluster and 5 cores of the controller cluster. Five floating-point numbers are transmitted from the environment to the controller and 2 floating-point numbers are transmitted from the controller to the environment. Our implementation relies on hyper-period expansion similarly to [15] and [25].

In order to experiment the influence of computation load on the execution time, we created several versions of the application with different computation loads in each computing task of the controller. The "pure-ROSACE" version, the "ROSACE+100" where 100 cycles are added in each computing task of the controller, the "ROSACE+200" where artificial computations of 200 cycles are added to each node. We measure the latency of the computation as a duration between the beginning of the period and the reception by the environment of the data computed by the controller.

	ROSACE	ROSACE+100	ROSACE+200
Single-Core	4.01μs (x1)	20.13μs (x1)	36.13μs (x1)
Measured ET	3.73μs (x1.08)	9.66μs (x2.08)	15.66μs (x2.31)
WCRT TT	5.49μs (x0.73)	13.44μs (x1.50)	21.24μs (x1.70)

Table 2: ROSACE case study. Speedup of the parallel executions on the MPPA2.

Table 2 shows the timing obtained for each implementation. ***WCRT TT* offers good speedups** for ROSACE+100 and ROSACE+200. The difference with the speedups obtained with the best-effort execution (*Measured ET*) gives an idea of the cost of timing guarantees.

The performance is reduced for pure-ROSACE since, the cost of computation is negligible compared to the one of NoC communication. Note that the performance loss is due to the time-triggered nature of the model, but not to the way we combine NoC and memory interference analyses.

4.2.2 Synthetic Benchmark on 64 Cores. We designed a synthetic benchmark running on 64 cores. The purpose is to apply our method on a highly parallel application with important data transfers and observe the time-triggered implementation. The application runs

on 16 clusters and in each cluster, the program executes 4 tasks in parallel on 4 cores.

Fig. 8 shows NoC communications involved in the application. Clusters are sorted so that clusters with direct links are represented adjacent to form a grid, but their number correspond to the physical layout. I/O clusters and their routers are not represented. The dispatch operation (Fig. 8a) follows a tree of nodes while the gather operation is composed of point to point communication to cluster 0.

As shown in Fig. 8c, in each cluster a `split` task receives the inputs from the NoC. The results are centralized by a `join` task and sent through the NoC by specific tasks (`NoC_Copy` and `NoC_EOT`). Tasks `t1`, `t2`, `t3` and `t4` are computation tasks.

The packets are 17-flit long, i.e. 17 words of 32 bits. In average the WCTT are 98 cycles for the flows of Fig. 8a and 1227 cycles for the flows of Fig. 8b. The difference is due to the highest number of concurrent flows of the gather operation compared to the dispatch operation.

	Execution time	Speedup
Single-core	358 ms	x1
Measured ET	31 ms	x11.46
WCRT TT	51 ms	x7.08

Table 3: Synthetic benchmark running on 64 cores.

Table 3 shows the speedup obtained by each method. *WCRT TT* has a **good speedup** and is 38% slower than the non-guaranteed best-effort *Measured ET* method. Since *WCRT TT* offers **strong real-time guarantees**, this result is good.

5 RELATED WORK

Skalistis *et al.* [27] present a method similar to Rihani *et al.* [25] to compute time-triggered schedules taking into account interferences. This method considers shared-memory communications and communication through the NoC. However, the architecture is different: they consider a Direct Memory Access (DMA) engine which is programmed by the core with an address and a size and this DMA copies data from the memory to the remote memory through the NoC. A NoC communication is modeled with 3 tasks: an initialization task to configure the DMA, a transfer task which performs both local memory accesses and remote memory accesses and the finalization task which ends when the DMA has finished. The duration of the transfer task corresponds to the WCTT plus the copy of the data. This is close to the model of Tendulkar *et al.* [28] where the transfer is in two phases and represented with an initialization task and a transfer task. At the opposite, in our model the core is responsible for copying the data from the memory to the NoC transmission buffer. Consequently, the duration of our reception task (RX) does not include the duration of the copy from the local memory.

If we focus only on the NoC model and aside from the hardware differences, our solution allows a finer analysis since we can decouple the copy from the NoC transmission. In other terms, their approach would lead to intermediate results between our naive and our improved method. Furthermore, our method relies on the MIA tool for interference analysis, which has been shown to lead to more precise results [24].

Note that there is extensive work on timing verification techniques for scheduling of the NoC, or considering that tasks on each core execute out of local memory with the only interaction with packet flows being through a consideration of release jitter (surveyed in [18, 19]). This complementary work could be combined to what we present in this paper. As said before, our paper is also orthogonal to work focusing on schedling and memory or task mapping. For instance, work by Giannopoulou et al. [13] uses network calculus to bound the worst-case traversal time over the NoC and introduces endpoint tasks. They focus on the problem of memory bank assignment to minimize interferences. As future work, it would be interesting to combine their work to what we present in this paper.

6 CONCLUSION AND FUTURE WORK

In this paper we present a method to compute a global, time-triggered schedule on a many-core platform. We focus on NoC communications between tasks running on different clusters, and their impact on memory access interferences. Our solution is integrated into an existing method focused on shared-memory communication in a single cluster. To fit the existing model, we model the NoC communication using virtual tasks. To avoid pessimism due to uncertainty on the possible execution time of the NoC reception, we split the replication phase in two tasks, and identify precisely the instant where the NoC transmission starts. To avoid cyclic dependency, we use the hardware mechanisms provided by the MPPA to avoid any memory access, hence any potential interference, during the transmission phases.

We introduce an implementation and modeling method that leads to tight response-time and an efficient scheduling (earlier release date) by splitting network access from task computation. We applied the method to a use-case inspired from avionic application and a synthetic benchmark to evaluate the time-triggered execution. The time-triggered implementation offers real-time guarantees and a parallelization speedup close to the best-effort execution.

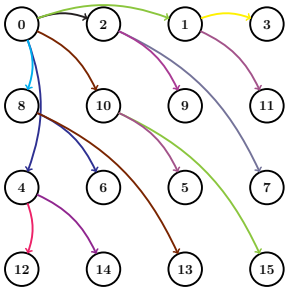
We describe a problem due to cycles in the communications through the NoC. In the paper, we solved this problem by breaking the cycle using hardware mechanisms. As future work, this problem could be solved by a fixed-point computation, as used in section 4.1 for the naive approach. This would make the approach applicable to architecture where our hardware mechanisms does not exist. However, to be generalized, it would be necessary to prove that the fixed-point iteration terminates.

REFERENCES

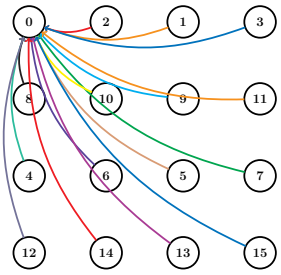
[1] Sebastian Altmeyer, Robert I Davis, Leandro Indrusiak, Claire Maiza, Vincent Nelis, and Jan Reineke. 2015. A generic and compositional framework for multi-core response time analysis. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*. ACM, 129–138.

[2] Hamdi Ayed, Jérôme Ermont, Jean-luc Scharbarg, and Christian Fraboul. 2016. Towards a unified approach for worst-case analysis of Tilera-like and Kalray-like NoC architectures. In *Factory Communication Systems (WFCS), 2016 IEEE World Conference on*. IEEE, 1–4.

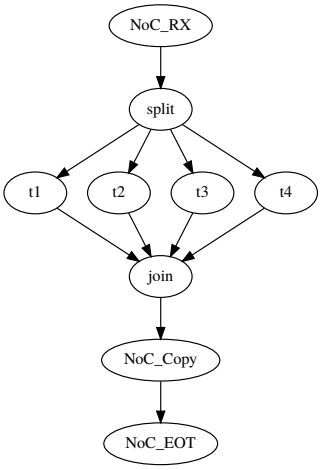
[3] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. 2010. OTAWA: an open toolbox for adaptive WCET analysis. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer, 35–46.



(a) Dispatch from Cluster 0



(b) Gather to Cluster 0



(c) In each cluster, 4 cores execute tasks in parallel.

Figure 8: Overview of the synthetic benchmark involving 16 clusters.

[4] Mathias Becker, Dakshina Dasari, Borislav Nicolic, Benny Åkesson, Vincent Nélis, and Thomas Nolte. 2016. Contention-Free Execution of Automotive Applications on a Clustered Many-Core Platform. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*. 14–24.

[5] Gérard Berry. 2007. SCADE: Synchronous design and validation of embedded control software. In *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*. Springer, 19–33.

[6] Marc Boyer, Benoît Dupont de Dinechin, Amaury Graillat, and Lionel Havet. 2018. Computing Routes and Delay Bounds for the Network-on-Chip of the Kalray MPPA2 Processor. In *ERTS 2018-9th European Congress on Embedded Real Time Software and Systems*.

[7] Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet. 2018. Scade 6: from a Kahn Semantics to a Kahn Implementation for Multicore. In *Forum on specification & Design Languages (FDL)*. Munich, Germany. https://hal.archives-ouvertes.fr/hal-01960410.

[8] Benoît Dupont de Dinechin and Amaury Graillat. 2017. Network-on-chip service guarantees on the kalray mppa-256 bostan processor. In *Proceedings of the 2nd International Workshop on Advanced Interconnect Solutions and Technologies for Emerging Computing Systems*. ACM, 35–40.

[9] Benoît Dupont de Dinechin, Duco van Amstel, Marc Poulhiès, and Guillaume Lager. 2014. Time-critical Computing on a Single-chip Massively Parallel Processor. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE '14)*. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, Article 97, 6 pages. http://dl.acm.org/citation.cfm?id=2616606.2616725

[10] Guy Durrieu, Madeleine Faugere, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and Wolfgang Puffitsch. 2014. Predictable Flight Management System Implementation on a Multicore Processor. In *ERTS 2014-7th European Congress on Embedded Real Time Software and Systems*.

[11] Thomas Ferrandiz, Fabrice Frances, and Christian Fraboul. 2012. A sensitivity analysis of two worst-case delay computation methods for spacewire networks. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*. IEEE, 47–56.

[12] Fabien Geyer and Georg Carle. 2016. Network engineering for real-time networks: comparison of automotive and aeronautic industries approaches. *IEEE Communications Magazine* 54, 2 (2016), 106–112.

[13] Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, Lothar Thiele, and Benoît Dupont de Dinechin. 2016. Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources. *Real-Time Systems* 52, 4 (2016), 399–449.

[14] Amaury Graillat, Matthieu Moy, Pascal Raymond, and Benoît Dupont de Dinechin. 2018. Parallel code generation of synchronous programs for a many-core architecture. In *DATE*. IEEE, 1139–1142.

[15] Amaury Graillat, Matthieu Moy, Pascal Raymond, and Benoît Dupont De Dinechin. 2018. Parallel Code Generation of Synchronous Programs for a Many-core Architecture. In *DATE 2018 - Design, Automation and Test in Europe*. Dresden, Germany. https://hal.inria.fr/hal-01667594

[16] Jérôme Grieu. 2004. *Analyse et évaluation de techniques de commutation Ethernet pour l'interconnexion des systèmes avioniques*. Ph.D. Dissertation. Institut National Polytechnique de Toulouse.

[17] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous dataflow programming language Lustre. *Proc. IEEE* 79, 9 (Sept. 1991), 1305–1320.

[18] S. Hesham, J. Rettkowski, D. Goehringer, and M. A. Abd El Ghany. 2017. Survey on Real-Time Networks-on-Chip. *IEEE Transactions on Parallel and Distributed Systems* 28, 5 (May 2017), 1500–1517. https://doi.org/10.1109/TPDS.2016.2623619

[19] A.E. Kiasari, A. Jantsch, and Z. Lu. 2013. Mathematical Formalisms for Performance Evaluation of Networks-on-chip. *ACM Comput. Surv.* 45, 3, Article 38 (July 2013), 41 pages. https://doi.org/10.1145/2480741.2480755

[20] The Mathworks. [n. d.]. *Simulink: User's Guide*.

[21] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo. 2015. Memory-Processor Co-Scheduling in Fixed Priority Systems. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems (RTNS)*. 87–96.

[22] Bruno Pagano, Cédric Pasteur, Günther Siegel, and R Knížek. 2018. A model based safety critical flow for the AURIX multi-core platform. *Embedded Real-Time Software and Systems (ERTS'18)* (2018).

[23] Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. 2014. The ROSACE case study: From simulink specification to multi/many-core execution. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. IEEE, 309–318.

[24] Hamza Rihani. 2017. *Many-Core timing Analysis of Real-Time Systems*. Ph.D. Dissertation. Univ. Grenoble Alpes.

[25] Hamza Rihani, Matthieu Moy, Claire Maiza, Robert I Davis, and Sebastian Altmeyer. 2016. Response Time Analysis of Synchronous Data Flow Programs on a Many-Core Processor. In *RTNS'16*. ACM, 67–76.

[26] Selma Saidi, Rolf Ernst, Sascha Uhrig, Henrik Theiling, and Benoît Dupont de Dinechin. 2015. The shift to multicores in real-time and safety-critical systems. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*. IEEE Press, 220–229.

[27] Stefanos Skalistis and Alena Simalatsar. 2016. Worst-Case Execution Time Analysis for Many-Core Architectures with NoC. In *Proceedings of the 14th International Conference on Formal Modelling and Analysis of Timed Systems*, Springer (Ed.). 211–227.

[28] Pranav Tendulkar, Peter Poplavko, Ioannis Galanommatis, and Oded Maler. 2014. Many-core scheduling of data parallel applications using SMT solvers. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*. IEEE, 615–622.