Response Time Analysis of Synchronous Data Flow Programs on a Many-Core Processor

Hamza Rihani
Univ. Grenoble Alpes
CNRS, VERIMAG, F-38000
Grenoble, France

Matthieu Moy
Univ. Grenoble Alpes
CNRS, VERIMAG, F-38000
Grenoble, France

Claire Maiza
Univ. Grenoble Alpes
CNRS, VERIMAG, F-38000
Grenoble, France

Robert I. Davis
University of York, UK
INRIA, France

Sebastian Altmeyer
University of Luxembourg
Luxembourg

ABSTRACT

In this paper we introduce a response time analysis technique for Synchronous Data Flow programs mapped to multiple parallel dependent tasks running on a compute cluster of the Kalray MPPA-256 many-core processor. The analysis we derive computes a set of response times and release dates that respect the constraints in the task dependency graph. We extend the Multicore Response Time Analysis (MRTA) framework by deriving a mathematical model of the multi-level bus arbitration policy used by the MPPA. Further, we refine the analysis to account for the release dates and response times of co-runners, and the use of memory banks. Further improvements to the precision of the analysis were achieved by splitting each task into two sequential phases, with the majority of the memory accesses in the first phase, and a small number of writes in the second phase. Our experimental evaluation focused on an avionics case study. Using measurements from the Kalray MPPA-256 as a basis, we show that the new analysis leads to response times that are a factor of 4.15 smaller for this application, than the default approach of assuming worst-case interference on each memory access.

1. INTRODUCTION

The design, development, and verification of safety-critical real-time embedded systems are subject to specific requirements that follow from guidelines and standards such as DO-178B/C for avionics and ISO26262 for automotive systems. Both the functional and the timing behaviour of such systems is required to be correct. In order to ensure that applications meet their deadlines, predictable upper bounds are required on the execution times of software components. These enable the derivation of sound upper bounds on the worst-case response times (WCRT), from input stimulus to output response, and thus verification of compliance with timing constraints.

With the relatively simple hardware (single processor, no cache, no advanced hardware acceleration features, etc.) used in legacy systems, it was possible to ensure predictability of execution times, to tightly bound worst-case execution times (WCET) either via static analysis or via measurements of all relevant paths, and to compose tight upper bounds on overall response times. Due to an increasing demand for compute performance, combined with Size, Weight, and Power consumption (SWaP) requirements, the emphasis has shifted from ever faster single-core processors, which had reached physical limitations due to issues with heat dissipation, to more complex multi-core and many-core architectures. Ensuring predictable and tightly bounded timing behaviour in such systems is very challenging. This is due to the contention for multiple shared hardware resources between co-running applications on the different processors. Examples include contention for cache, network or memory bus, and other interference effects such as DRAM refreshes. For example, on the Freescale P4080, the latency of a read operation varies from 40 to 600 cycles depending on the total number of cores running competing tasks [14]. Similarly, a 14 times slowdown has been reported [19] due to interference on the L2-cache for tasks running on Intel Core 2 Quad processors. Recent work [22] shows that even with cache partitioning, contention for registers accessed on both cache hits and misses can cause a 21 times slow down due to contention caused by co-runners on the ARM cortex A-15 multi-core architecture.

A few modern architectures are however designed with the requirements of real-time embedded systems, in particular predictability, in mind. In this paper, we study the Kalray MPPA-256 [9] many-core processor (more precisely the second generation, called Bostan). This processor has 256 cores arranged in 16 clusters. The clusters are linked by a 2-D torus dual Network-On-Chip (NoC). Independent applications may run on different clusters. Although inter-cluster interference may be largely eliminated there is still the potential for interference among the tasks running on the same cluster due to accesses to shared resources such as the shared memory [15]. Hardware partitioning and replication (e.g. memory banks) are used to significantly reduce this interference, but do not remove it completely. Details of the architecture are discussed in Section 2.1.

RTNS '16, October 19 - 21, 2016, Brest, France

ISBN 978-1-4503-4787-7/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2997465.2997472

Synchronous Data Flow (SDF) languages such as Lustre [11] and SCADE [4] have been industrialised and are widely used for embedded systems. A certified SDF compiler can produce sequential code which facilitates deterministic behaviour. The challenge is to parallelise this sequential code without loss of determinism. An SDF application can be represented by a set of tasks with dependence relations between them. The tasks produce and consume a statically defined number of tokens. This model is deterministic in terms of communications, i.e., we know the topology and the amount of data sent and received. The parallelism in this case is trivial: tasks can be mapped to available processing elements (cores), respecting the dependence relations, and allowing independent tasks to run in parallel.

Existing work by Puffitsch et al. [18] and Walter and Nebel [23] tackles the challenge of porting and mapping Synchronous Data Flow programs to multi-core architectures by proposing different scheduling techniques. These approaches assume that an upper bound WCET is known for each task, and schedule the tasks accordingly. The scheduling techniques used optimise parameters such as the global execution time, CPU utilisation or energy efficiency. Using WCETs that are independent of co-runner interference can be very pessimistic (as shown in [15]) which may affect the efficiency of the approach. However, we note that accounting for the actual interference from co-runners also depends on the scheduling technique used.

Contributions:

Our approach differs from previous work, in that we consider the influence of scheduling on timing analysis. As a consequence, the scheduling step considers a WCET bound for each task that also accounts for the interference from co-runners.

Our main contribution is an algorithm to compute a static, time-driven, periodic schedule (further detailed in Section 2.3), as commonly used in hard real time systems for maximum predictability. We assume that the mapping of tasks to cores and the execution order is given (either manually or provided by a separate tool), and compute a set of release dates (offsets) and response times for each task. This is an iterative process, with release dates dependent on the response times of preceding tasks, and response times dependent on the set of co-runners, which are in turn dependent on task release dates. The process either converges on a valid, all dependence relations respected, and schedulable configuration or deems the system unschedulable with that task mapping, in which case a different mapping could be tried. The proof of convergence is included in the long version of this paper published as a technical report [20]. It is non-trivial since the usual monotonicity argument does not apply; the sequence of release dates computed at each iteration may not be monotonic.

We target applications running on the Kalray MPPA-256 many-core processor. We identify all the sources of interference for an application running on a compute cluster, and provide a mathematical model for them. The model builds upon the Multi-core Response Time Analysis (MRTA) framework [1]; a generic approach to response time analysis for multi- and many-core systems. Unlike MRTA, we consider a static, time-driven schedule, and hence cannot use the same fixed-point algorithm. Instead, we provide a novel algorithm that uses not only the mapping but also the information about *when* each task is executed to model interference precisely. Finally, we evaluate our approach with a micro-benchmark and apply it to a case study obtained from a realistic avionics application.

Related Work:

In 2014, Lampka et al. [12] proposed an approach based on timed automata and abstract interpretation. The main idea is to analyse a phase-structured task model [17, 21] using Real-Time Calculus to derive the arrival curves for access requests and the availability curves for the shared resources (in this case the shared bus). The authors proposed timed automata models of several bus arbiters (FCFS, Round-Robin, TDMA). Although modelling a more complex arbiter can be feasible in timed automata, it increases the complexity of the model and may lead to an explosion of states during analysis. Architectures such as the Kalray MPPA-256 have several shared resources. This adds to the complexity of the analysis and may significantly affect its scalability.

In 2015, Dasari et al. [7] presented an approach for response time analysis taking into account interference on the bus. The number of accesses were obtained from measurements during the task's execution (Regions of task execution were used for a more fine-grained analysis). The bus itself was modelled by considering the earliest and latest available communication slots for the task under analysis. This representation depends on the arbitration policy of the bus. The authors give mathematical models of the most widely used bus arbiters; however, it is difficult to see how to represent with this approach a less conventional arbitration policy such as that employed in the Kalray MPPA-256.

In 2015, Carle et al. [5] described the implementation of data flow applications on multi-core systems. The authors focus on the optimisation of an off-line schedule taking into account non-functional properties such as release dates and deadlines. This work aims to eliminate the interference on shared resources by providing temporal isolation. Our proposed approach accounts for the interference and hence may be complementary to this previous work.

In 2016, Giannopoulou et al. [10] proposed a response time analysis on the Kalray MPPA 256 considering mixed-criticality scheduling. The main difference with our approach is that [10] considers a "Flexible time-triggered scheduling" model which divides time into frames, and forces a global synchronization barrier between frames. This potentially creates core under-utilization while they wait for the barrier. Our scheduling policy does not require any global barrier. Also, we model the multi-level round-robin arbiter while [10] considers only one level.

Organization: The remainder of the paper is organised as follows. Section 2 describes the system and application models used, and outlines the MRTA framework which we build upon. Section 3 provides response time analysis for synchronous data flow programs running on a compute cluster of the Kalray MPPA-256. This analysis is evaluated in section 4 via a set of micro-benchmarks and a case study application. Section 5 concludes with a summary and discussion of future work.

2. SYSTEM AND APPLICATION MODEL

This section presents the system and application models considered, thus defining the context for our work. We

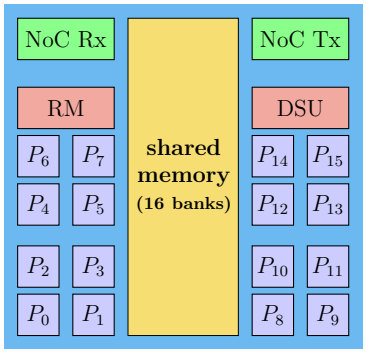

Figure 1: Compute cluster architecture for the Kalray MPPA-256

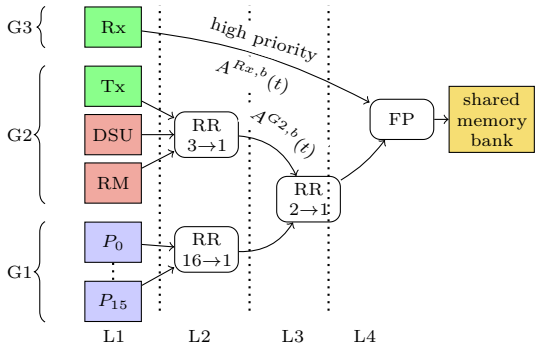

Figure 2: Request arbitration to a shared memory bank (RR: Round-Robin. FP: Fixed Priority)

describe the hardware architecture and its relevant characteristics, give an outline of the MRTA framework which we build upon, and describe the application model that we later analyse.

2.1 Kalray MPPA-256 Architecture

The Kalray MPPA-256 is a many-core processor [9]. It is composed of 16 tiles (called compute clusters) of $16 + 1$ cores. The processor is connected to the external environment through 2 I/O quad-core clusters. Inter-cluster communication is achieved via a 2D-torus dual Network-On-Chip (NoC) for data and control. In this paper, we are interested in applications running on a compute cluster and the interference due to intra-cluster communications.

Compute Cluster: Figure 1 illustrates the architecture of a single compute cluster. It has 16 cores plus 1 Resource Manager (RM). The compute cluster connects to the NoC via two DMA (Direct Memory Access) interfaces; one for receiving (Rx) and one for transmitting (Tx). The cluster also has a Debug Support Unit (DSU).

The cores have an in-order Very Long Instruction Word (VLIW) pipeline and separate 8 KByte 2-way set-associative private caches with 64 Byte lines for instructions and data. The data cache has a write buffer with 8 fully associative 64-bit entries. There is no cache coherency mechanism between the cores. Each core has its own real-time clock. Clocks in the same cluster are synchronous.

Shared Memory: In order to provide spatial isolation, the memory is partitioned into 16 banks. Each memory bank is accessed via a separate bus arbiter which significantly reduces the amount of interference compared to the alternative of a single arbiter. There are two possible configurations for the memory banks: *interleaved mode* where sequential memory addresses move from one bank to another, and *blocked mode* where each block of 128 KB consecutive memory addresses are contained in a memory bank. In this paper, we assume that blocked mode is selected, since it gives more control over the bus interference. This is because with blocked mode, cores that access different memory banks go through different arbiters hence they do not interfere with each other. We use a fixed association between cores and memory banks. More precisely, in our application model, each task has a local memory buffer, and the buffers of all tasks running on the same core are mapped to the same memory bank. As a result, `read` accesses are private but `write` requests may access another core's memory bank.

Bus Arbitration: Figure 2 illustrates the specific multi-level policy used to arbitrate accesses to the shared memory. We distinguish three groups which are arbitrated over three levels:

- $G1 = \{i \in [0, 15] : P_i\}$: access requests from the 16 cores are initially subject to round-robin arbitration.
- $G2 = \{Tx, DSU, RM\}$: access requests from the Resource Manager (RM), Debug Support Unit (DSU) and Tx requests to the NoC are initially subject to round-robin arbitration.
- $G3 = \{Rx\}$: Rx requests from the NoC.

At level $L1$, requests issued by data and instruction caches local to a core are processed by a local round-robin arbiter. At level L2, there is round-robin arbitration within each of the groups $G1$ and $G2$. This is followed by round-robin arbitration between these two groups at level $L3$. Finally, $G3$ is included in the last level of the arbitration ($L4$), which uses a non-preemptive fixed priority (FP) arbiter and gives the highest priority to access requests coming from $G3$.

To summarise, an access request from a task running on a core crosses three levels of round-robin arbitration and a level of fixed priority arbitration to reach the shared memory.

2.2 Multicore Response Time Analysis

In this subsection, we outline the generic framework for Multi-core Response Time Analysis (MRTA) introduced by Altmeyer et al. [1], which we subsequently build upon.

Given a set of n sporadic tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$, where each task τ_i has a period or a minimum inter-arrival time T_i and a deadline D_i and is statically assigned to a core, the MRTA framework computes the response time of each task taking into account the total interference at the different levels of the hardware that could occur during the task's response time. By convention, we use P_x to mean the core that the task under analysis is mapped to, and P_y to indicate some other core. The subset of tasks mapped to a core P_y is denoted by Γ_y.

In the MRTA framework, tasks are represented by a set of traces, each of which consists of an ordered list of instructions, where each instruction carries information about the memory locations accessed (if any). A set of exhaustive traces (i.e. for different paths) can be used to give a sound over-approximation of the memory demand and the processor demand of a task by taking the maximum

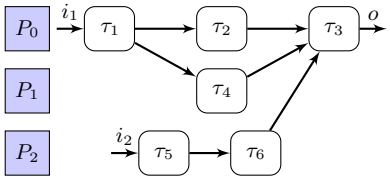

Figure 3: Example of a data flow program

memory (processor) demand over all traces for the task. As a result, the framework decouples response time analysis from a reliance on context independent WCET values (in isolation). Instead, the analysis formulates response times directly from the demands on different hardware resources. Such a separation of concerns trades different sources of pessimism. The simplifications used in [1] to make the analysis tractable are unable to take advantage of overlaps between processing and memory demands; however, this compromise is set against substantial gains acquired by considering the worst-case behaviour of hardware resources, such as the memory bus, over long durations equating to task response times, rather than summing the worst case over short durations such as a single accesses, as is the case with the traditional approach using context-independent WCETs.

With the MRTA framework, the response time R_i of task τ_i executing on core P_x is computed using the following recurrence relation:

$$R_i = \mathrm{PD}_i + I^{\mathrm{PROC}}(i, x, R_i) + I^{\mathrm{BUS}}(i, x, R_i) + I^{\mathrm{DRAM}}(i, x, R_i) \quad (1)$$

Where PD_i is the processor demand, which equates to the execution time of task τ_i in isolation assuming a perfect bus and memory with zero latency. I^{PROC} is the interference on the core due to higher priority tasks preempting or delaying task τ_i. I^{BUS} is the interference on the bus computed using a mathematical model of the bus arbiter. Finally I^{DRAM} is the interference due to DRAM refreshes. Equation (1) is solved as part of a larger fixed-point iteration which operates over the set of tasks, see Algorithm 1 in [1] for details.

The MRTA framework represents a generic and compositional solution for response time analysis. It allows the modelling of a wide range of different arbitration policies (and a combination of them), as well as different memory models (no cache, data and instruction cache, scratchpads, etc. and a combination of them). In this paper, we build upon the MRTA framework, instantiating it for different hardware components, bus arbitration policies, and application models.

2.3 The Synchronous Data Flow Model

Our aim is to obtain accurate bounds on the worst-case response time for data flow programs. A simple example of a data flow program is shown in Figure 3. In this work, we consider mono-rate programs, i.e. all tasks have the same period. In the case of a multi-rate program, we assume unfolded execution to the hyper-period (the least common multiple of the tasks' periods), effectively reducing the problem to a mono-rate one [1]. Also, we consider that all tasks in a cycle must complete before the end of the cycle, which is a common constraint when scheduling synchronous programs. As a consequence, scheduling can be done on one period (or hyper-period); the same schedule is then repeated indefinitely.

In terms of scheduling, the tasks in the data flow program are seen as an acyclic dependency graph. A task is released only when all its predecessors have finished their execution, i.e. when they produce tokens for the next tasks. In the example given in Figure 3, the output data of task τ_1 must be available to task τ_4 before it can execute. Hence, the release date of task τ_4 should be greater than the finish time of task τ_1. The data produced is written into a memory location where the consumer task can read it. In this case the memory bus is a shared resource and concurrent accesses may suffer from arbitration delays. In the example, tasks τ_2, τ_4, and τ_6 write to the memory of task τ_3 which creates potential interference within the response time of each task.

Our algorithm takes as input a fixed mapping of tasks to cores, and a fixed order for tasks mapped to the same core. We purposely delegate the mapping and ordering to a separate tool, dedicated to optimisation of the schedule and mapping, and focus on the analysis part. Mapping and ordering of tasks can also be done manually. We produce a completely static, time-driven schedule. There cannot be two tasks active on the same core at the same time, hence we do not use preemption and a task starts immediately when it is released. The schedule specifies the exact, fixed release date for each task. This is in a way pessimistic in the sense that each task waits for the worst-case response time of each of the tasks it is dependent on (it cannot start even if all of them have completed well before their deadline); however, our aim is to optimise the worst case, not the average case. The scheduling scheme has good properties for a hard-real time system. First, it enables the application to be executed without any operating system: we only require communication primitives, and one primitive to wait for a specified instant; they can be provided as a simple library. Also, it makes the whole execution highly predictable since the release date of a task does not depend on the execution time of previous tasks: we avoid any potential domino effects in timing.

We introduce the following additional notation used in our analysis: Each task τ_i has a release date rel_i (effectively an offset relative to the start of the data flow program) $\Theta = \{rel_1, ..., rel_n\}$ is the set of release dates and $\mathcal{R} = \{R_1, ..., R_n\}$ is the set of upper bound response times of tasks in Γ. Note that there is no order relation between rel_i and rel_{i+1} (resp. R_i and R_{i+1}) in the set Θ (resp. $\mathcal{R}$). Recall that each task is statically mapped to a core.

The approach we propose takes into account the interference on the bus as part of the response time analysis. Using the SDF model, we know which tasks could potentially execute at the same time and therefore be co-runners. We make use of this information to derive tight bounds on the amount of interference. Moreover, there is an

[1]The unfolding preserves the required minimum separation between jobs, since our scheduling scheme includes fixed release dates for each task. The unfolding process thus assigns proper release dates to multiple instances of the same task. We note the potentially large size of the hyper-period which may introduce a complexity issue.

implicit dependency between two successive periodic instances which allows us to limit the analysis to only one instance of the task graph. Our analysis, based on an adaptation of the MRTA framework, assumes static non-preemptive scheduling based on the task graph.

In summary, in addition to providing a model for the Kalray MPPA-256's bus arbiter, the main difference between the basic MRTA approach [1] and our approach is that MRTA considers sporadic tasks, but does not exploit any knowledge of dependencies or sequentiality between them. In contrast, we take into account the task dependencies (similarly to [6]) and the precise schedule including release dates and response times. As a consequence, we have to solve another fixed-point problem since the schedule depends on the response times of each task, and vice-versa.

3. ANALYSIS FOR SDF APPLICATIONS ON THE MPPA-256

In this section we first quantify the different sources of interference that need to be considered in analysing synchronous data flow applications running on a compute cluster of the Kalray MPPA-256. We then describe how bus interference can be computed using the task dependency graph for a synchronous data flow program, thus avoiding pessimism in the analysis caused by lack of information about co-runners. We then derive a mathematical model of the multi-level arbiter of the Kalray MPPA-256. Finally we describe our response time analysis algorithm.

3.1 Quantification of the Interference

We now highlight the main sources of interference that need to be considered as part of (1) when determining the response time R_i of task τ_i, given the hardware and application models considered.

The interference on the core $I^{\text{PROC}}(i, x, R_i)$ typically comes from delays or preemptions due to the execution of higher priority tasks on the same core. In our application model, we assume a static non-preemptive scheduler. Task release dates are set such that only one task is active per core at any given time. This effectively eliminates all interference from higher priority tasks executing on the same core. It also simplifies the analysis by removing all *cache-related preemption delays* [2].

The interference due to the DRAM is mainly due to refresh cycles. The Kalray MPPA-256 supports a DDR memory accessed through the I/O clusters. An access from a core in a compute cluster has to cross the NoC and the I/O cluster and finally the DDR controller. All these layers add to the complexity of the analysis and the access delay. For predictable operation, such accesses are generally avoided by pre-loading all of the code and data into the shared memory of the compute cluster. The on-chip RAM of the MPPA is 32 MB, 2 MB per cluster, which is sufficient for many applications. We therefore assume that $\forall t > 0, I^{\text{DRAM}}(i, x, t) = 0$.

The interference on the bus depends on the specific arbitration policy used. Cache misses in the private data and instruction caches issue requests to the shared memory that are granted according to the multi-level arbiter. A detailed derivation of the $I^{\text{BUS}}(i, x, \mathcal{R}, \Theta)$ function that depends on the set of release dates of all tasks is given in the following section.

Taking the above considerations into account, the response time formula given in (1) simplifies to:

$$R_i = \text{PD}_i + I^{\text{BUS}}(i, x, \mathcal{R}, \Theta) \tag{2}$$

Note, here $\mathcal{R}$ is the set of response times and Θ is the set of release dates for all tasks.

3.2 Bus Interference

In our application model, we consider a task dependency graph mapped to a set of cores. The hardware architecture allows the mapping of contiguous addresses to the same memory bank. Thus, concurrent accesses are independent as long as they are done in different memory banks, which reduces the bus interference. We exploit this by allocating the memory of each task running on the same core to the same bank. Tasks run on their locally reserved memory banks and access other locations only when writing data to the next successive task(s) in the task graph. We denote by MD_i^b the memory demand of task τ_i on memory bank b.

The bus interference is given by:

$$I^{\text{BUS}}(i, x, \mathcal{R}, \Theta) = \sum_{b \in B_i} \text{BUS}_b(i, x, \mathcal{R}, \Theta) \times d \tag{3}$$

where d is the latency of a bus access without interference, B_i is the set of memory banks accessed by task τ_i, and $\text{BUS}_b(i, x, \mathcal{R}, \Theta)$ is a function that, accounting for the arbitration policy, gives an upper bound on the number of accesses that can delay completion of task τ_i (running on core P_x) during a given time interval. Note that $\text{BUS}_b(i, x, \mathcal{R}, \Theta)$ includes both accesses of the task of interest and accesses performed by other tasks, since both delay the execution by the bus latency.

In order to derive $\text{BUS}_b(i, x, \mathcal{R}, \Theta)$, we need to compute an upper bound on all bus accesses during the response time of task τ_i. We define $S_i^{x,b}(\mathcal{R})$ as an upper bound on the number of accesses by the task of interest τ_i running on core P_x within its response time. Note that since the scheduler is non-preemptive the bus accesses from core P_x come only from the memory demand of task τ_i on the memory bank b (MD_i^b). Since we analyse one instance of task τ_i, we have:

$$S_i^{x,b}(\mathcal{R}) = \text{MD}_i^b \tag{4}$$

We define $\underset{i,k}{\Delta}(\mathcal{R}, \Theta)$ the overlap duration between tasks τ_i and τ_k. The computation of the overlap is trivial since we already know the release dates and response times of tasks τ_i and τ_k. Note that $\underset{i,k}{\Delta}(\mathcal{R}, \Theta)$ is 0 when the tasks do not overlap.

We use $\underset{i,k}{W}^b(\mathcal{R}, \Theta)$ to denote an upper bound on the number of accesses by task τ_k that may interfere with task τ_i at the memory bank b during its response time. In the absence of detailed information on the pattern of access requests within a task, we consider that any two tasks that overlap in time can interfere on each of their accesses. $\underset{i,k}{W}^b(\mathcal{R}, \Theta)$ is given by:

$$\underset{i,k}{W}^b(\mathcal{R}, \Theta) = \min(\text{MD}_k^b, \left\lceil \frac{\underset{i,k}{\Delta}(\mathcal{R}, \Theta)}{d} \right\rceil) \tag{5}$$

We use $A_i^{y,b}(\mathcal{R}, \Theta)$ to denote an upper bound on the number of accesses by all tasks running on core $P_y \neq P_x$ during the response time of task τ_i. The number of accesses

is bounded by the memory demand of each task on memory bank b. $A_i^{y,b}(\mathcal{R},\Theta)$ is therefore given by:

$$A_i^{y,b}(\mathcal{R},\Theta) = \sum_{k\in\Gamma_y} W_{i,k}^{b}(\mathcal{R},\Theta) \quad (6)$$

The terms $S_i^{x,b}(\mathcal{R})$ and $A_i^{y,b}(\mathcal{R},\Theta)$ are used to derive an upper bound on the number of accesses that contribute to the interference during the response time of task τ_i, which also depends on the bus arbitration policy. The multi-level arbiter of the Kalray MPPA-256 requires a combination of several policies (see Figure 2). We initially make the pessimistic assumption that the bus arbiter is *work-conserving*[2] and that the accesses from the task of interest τ_i are dealt with last of all. (We note that compared to the analysis given in [1] there is no +1 term accounting for an access from a lower priority preempted task. Since we assume a static non-preemptive schedule, any previous task and its accesses must have completed before task τ_i starts). Thus we have:

$$\mathrm{BUS}_b(i,x,\mathcal{R},\Theta) = S_i^{x,b}(\mathcal{R}) + \sum_{y\in G1\wedge y\neq x} A_i^{y,b}(\mathcal{R},\Theta) + A^{G2,b}(\mathcal{R},\Theta) + A^{G3,b}(\mathcal{R},\Theta) \quad (7)$$

(We recall that G1, G2 and G3 are the three levels of arbitration defined in Section 2.1).

3.3 Model of the Multi-level Bus Arbiter

In this section, we study the multi-level arbitration policy used in the Kalray MPPA-256 architecture. We consider the bus arbiter to a memory bank b as shown in Figure 2. The policy operates over 4 levels which we label $L1$ to $L4$ where $L1$ is the first (left-most) level, and $L4$ the final level which is based on fixed priority arbitration. Our analysis is built up following the hierarchy from level $L1$ to level $L4$.

Level L1: As input to the first level, we assume that the maximum number of accesses that can be generated by each source in the response time of a task can be determined. These values are as follows:

- First group (G1): this is a core and may be treated in the same way as the analysis given for a round-robin arbiter in [1]. Note that we do not need to distinguish between accesses that come via the Instruction Cache (IC) and those that come via the Data Cache (DC), since all must be processed before the task of interest τ_i can complete. Hence we may represent the output from this group as either $S_i^{x,b}(\mathcal{R})$ or $A_i^{y,b}(\mathcal{R},\Theta)$ depending on whether we are computing the accesses from the core that τ_i executes on, or from another core.
- Second group (G2): here we only need to compute the overall output from the group: $A_i^{G2,b}(\mathcal{R},\Theta) = A_i^{Tx,b}(\mathcal{R},\Theta) + A_i^{DSU,b}(\mathcal{R},\Theta) + A_i^{RM,b}(\mathcal{R},\Theta)$, since we are only interested in the interference it generates.
- Third group (G3): there is only one item, hence the output is the same as the input: $A_i^{G3,b}(\mathcal{R},\Theta) = A_i^{Rx,b}(\mathcal{R},\Theta)$.

Level L2: At level $L2$ the outputs (accesses) from all 16 processors are combined via a 16 to 1 Round-Robin (RR) arbiter. Note that each core has only one slot in the RR cycle. The number of accesses to bank b that can delay the execution of a task on core P_x at the output of $L2$ is given by:

$$\mathrm{BUS}_b^{L2}(i,x,\mathcal{R},\Theta) = S_i^{x,b}(\mathcal{R}) + \sum_{y\in G1\wedge y\neq x} \min\left(A_i^{y,b}(\mathcal{R},\Theta), S_i^{x,b}(\mathcal{R})\right) \quad (8)$$

where x is the index of the core P_x that task τ_i executes on, and similarly y ranges over the other 15 cores.

The worst-case situation occurs when each access in $S_i^{x,b}$ is delayed by each core $P_y \neq P_x$ for 1 slot. Given the round-robin arbiter, interference by core P_y is limited to the minimum of the number of accesses from P_y and from P_x, i.e. $\min\left(A_i^{y,b}(\mathcal{R},\Theta), S_i^{x,b}(\mathcal{R})\right)$.

Level L3: At level $L3$, the output from the level $L2$ arbiter, i.e. (8), is combined with that from the second group, i.e. $A^{G2,b}(\mathcal{R},\Theta)$, again via a round-robin arbiter, hence we have:

$$\mathrm{BUS}_b^{L3}(i,x,\mathcal{R},\Theta) = \mathrm{BUS}_b^{L2}(i,x,\mathcal{R},\Theta) + \min\left(A_i^{G2,b}(\mathcal{R},\Theta), \mathrm{BUS}_b^{L2}(i,x,\mathcal{R},\Theta)\right) \quad (9)$$

Again, the worst-case situation occurs when each access in $\mathrm{BUS}_b^{L2}(i,x,\mathcal{R},\Theta)$ is delayed by the output of $G2$ for 1 slot. Interference by the output of $G2$ is limited to $A_i^{G2,b}(\mathcal{R},\Theta)$.

Level L4: Finally, at level $L4$, the output from the level $L3$ arbiter, i.e. (9), is combined with the output from $G3$, i.e. $A_i^{Rx,b}(\mathcal{R},\Theta)$. As this is done via a fixed priority arbiter with higher priority given to $A_i^{Rx}(\mathcal{R},\Theta)$, we have:

$$\mathrm{BUS}_b^{L4}(i,x,\mathcal{R},\Theta) = \mathrm{BUS}_b^{L3}(i,x,\mathcal{R},\Theta) + A_i^{G3,b}(\mathcal{R},\Theta) \quad (10)$$

Finally, given the bus latency d, the bus interference is given by:

$$I^{\mathrm{BUS}}(i,x,\mathcal{R},\Theta) = \sum_{b\in B_i} \mathrm{BUS}_b^{L4}(i,x,\mathcal{R},\Theta)\times d \quad (11)$$

3.4 Response Time Analysis

Our response time analysis algorithm (Algorithm 1) is based on the MRTA approach [1]. First, we augment the framework with the model of the multi-level arbiter of the Kalray MPPA-256, and modify the way computation of the potential interference is performed: the original MRTA framework uses a model with sporadic tasks with minimum inter-arrival times and without dependencies, while we analyze a single period of a single-rate application, with a static schedule. Knowing the release dates and response times, Algorithm 1 computes the number of accesses that can delay a given task in a given time interval. Since the potential interference depends on the release dates and response times, and the response times depend on the interference, this requires a fixed-point iteration (line 10).

After computing the response times, the schedule we get may not respect the dependencies and sequentiality constraints. We modify the release dates so that each task is released immediately after each of the tasks it depends on is guaranteed to have completed (Algorithm 2). Modifying the release dates may change the interference, hence we have to re-compute it using Algorithm 1, and so on, until a fixed point is reached (Algorithm 3).

Algorithm 1 solves the recursive equation (2) using a fixed-point iteration, and computes the response times $\mathcal{R}$ of all

[2]A work-conserving bus arbiter will not idle the bus as long as there are pending requests.

Algorithm 1 Response Time Analysis Given a Set of Release Dates

```
1: function MultiCoreRTA(Θ)
2:     l = 1
3:     ∀_i : R^l[i] = PD_i + MD_i · d
4:     do
5:         for all i do
6:             R^{l+1}[i] = PD_i + I^BUS(i, x, R^l, Θ)
7:                                  ▷ I^BUS is given by equation 11
8:         end for
9:         l = l + 1
10:    while R^l ≠ R^{l−1}
11:    return R^l
12: end function
```

Algorithm 2 Update Release Times to Start After All Dependencies

```
1: function UpdateReleases(Θ_min, Θ, R)
2:     for all i do
3:         Θ[i] = max(Θ_min[i], {Θ[k] + R[k] | k ∈ deps(i)})
4:     end for
5:     return Θ
6: end function
```

tasks given the release dates Θ. Algorithm 2 ensures the dependency constraints between tasks are satisfied. It is parameterised by Θ_{min} which gives the earliest release date for each task: $\Theta_{min}[i] = t$ means that task τ_i cannot start before t. A task τ_i is released only when all the tasks it depends on (denoted by $deps(i)$) are guaranteed to have finished. We statically schedule every release date, hence we set the release date of each task to the maximum of the *worst-case* finish time of each task it depends on. Algorithm 3 uses MultiCoreRTA (Algorithm 1) to compute the response times of tasks in Γ given a set of release dates (line 5). Then, UpdateReleases (Algorithm 2) is used to verify and update the dependency constraints. Algorithm 3 starts from initial release dates (bounded by the SDF period) (InitRelease, line 3) and performs a fixed-point iteration.

Algorithm 3 terminates when UpdateReleases does not change the release dates. When this happens, the response times $\mathcal{R}^{l+1}$ computed before the call to UpdateReleases remain valid afterwards, and hence are valid at the end of the loop. Termination of Algorithm 1 is guaranteed: we limit the computation to one period of the task graph; the number of bus accesses is bounded which implies that the amount of interference seen by a task is also bounded. The response time computation of task τ_i is a monotonically increasing and bounded function, thus Algorithm 1 converges for any values in Θ. Termination of Algorithm 3 is non-trivial to show: the intuition is that a task cannot interfere with its past. At each iteration, release dates of tasks released before some instant of time t become fixed and remain the same for all subsequent iterations, with t advancing by at least one release date at each iteration. Note this means the number of iterations of Algorithm 3 is at most (`number of tasks` $- 1$). The complete proof is given in the technical report [20].

Since Algorithm 3 is parameterised by a function InitRelease, one might think that the choice of InitRelease could impact the precision of the result. However, we prove in the technical report [20] that the fixed point of the composition of MultiCoreRTA and UpdateReleases is unique, hence the algorithm will return the same schedule for any function InitRelease, and there is no point trying to optimise it. In our implementation, we start with $\Theta^0 = \Theta_{min}$, i.e. all release dates set to zero.

Algorithm 3 Adapt Release Dates to Meet Real-Time Constraints

```
1: function ComputeRT(Θ_min)
2:     l = 0
3:     Θ^l = InitRelease(), R^l = ⊥
4:     do
5:         R^{l+1} = MultiCoreRTA(Θ^l)
6:         Θ^{l+1} = UpdateReleases(Θ_min, Θ^l, R^{l+1})
7:         l = l + 1
8:     while Θ^l ≠ Θ^{l−1}
9:     if ∀_i : (Θ^l[i] + R[i]^l) ≤ D_i then
10:        return schedulable
11:    else return not schedulable
12:    end if
13: end function
```

4. EVALUATION

In this section we evaluate our approach using different configurations. We show how the application model as well as the architecture configuration may affect the estimation of the WCRT. We analyse a didactic micro-benchmark and a case study of a flight management system controller to validate our approach. More experiments with detailed results can be found in the long version of this paper published as a technical report [20].

4.1 Experimental Setup

Static analysis tools such as, OTAWA [3] and aiT [24][3], do not yet support the Kalray MPPA-256 Bostan. For this reason, we establish the task profiles from measurement-based techniques. Each task is executed in isolation while profiling processor cycles and the number of cache misses. Several measurements are performed for each task and the results show a variance approaching zero. This reflects the efforts made in the design of the Kalray MPPA-256 targeting real time applications. In our experiments, we consider a bus delay $d = 10$ cycles obtained from internal specifications (we ignore transaction pipelining and TLB cache misses). We also consider that the context switch delay is included in the task execution. We assume that the Resource Manager (RM), which loads the application onto the cores before operation starts, does not interfere with running tasks ($A_i^{RM,b} = 0, \forall i, b$). Finally, the Debug Support Unit (DSU) is disabled during operation ($A_i^{DSU,b} = 0, \forall i, b$).

Bus Model: We assume the benchmark in Section 4.2 is run in isolation on a compute cluster, i.e. accesses from the NoC do not occur during the execution of the application of interest. As a consequence, by setting $A^{Rx,b} = A^{Tx,b} = 0$ in (11), the interference is simplified to one level of round-robin arbitration. We do consider the accesses from the NoC in the benchmark in Section 4.3

Execution Model: We first consider a single-phase execution model where we make no assumptions about the distribution of read and write accesses between the start and end of a task. In our code generation scheme for the SDF model, tasks execute computations, then write the result to a shared memory location where the next task can

[3]To the best of our knowledge, aiT supports the first generation Kalray MPPA-256 Andey only.

task	PD (cycles)	MD (accesses)	dependencies
τ_1	5	42	$\emptyset$
τ_2	8	30	$\{\tau_1\}$
τ_3	20	18	$\{\tau_2, \tau_4, \tau_6\}$
τ_4	5	52	$\{\tau_1\}$
τ_5	8	30	$\emptyset$
τ_6	20	58	$\{\tau_5\}$

Table 1: Task profiles of SDF example in Figure 3

read it. Similar to [13], this execution model allows each task to be split into a first *execution* phase limited to reading the input and doing computations, and then a *write* phase where the output is sent to the next task. In the execution phase, the accesses are to the local memory bank of the task whereas in the write phase, requests may access a remote memory bank. We exploit this execution model in our analysis. We consider the two phases of a task as separate subtasks with a direct dependency relation. Using our analysis technique we compare the single-phase model with the two-phase model.

Experiments: We explore and compare a number of setups for the experimental evaluation so as to determine the effectiveness of various techniques that form part of the schedulability analysis. In the first experiment **E1**, we use our approach taking into account a two-phase execution model. Experiment **E2** also applies our approach, but using a single-phase execution model. In experiment **E3**, we use a simplified approach that discards the release dates of tasks, meaning that all tasks potentially overlap, and considers the tasks using the two-phase execution model. The same approach as **E3** is used in **E4**, but using the single-phase execution model. Finally, we consider in experiment **E5** that co-runners continuously interfere with the task of interest. This is a pessimistic analysis that assumes the worst-case interference on each memory access. Note that, this may result in unbounded interference due to the fixed priority level of the MPPA bus. In this case, we consider the upper bound on the number of accesses by all higher priority components during the analysed execution instance. Then, we assume that each task access is delayed by all the higher priority accesses. In the following, we compare the different analyses with different arbitration policies for each benchmark.

4.2 Didactic Example

We analyse the example given in Figure 3. Table 1 summarises each task profile that consists of the processor demand and the memory demand. Figure 4 gives a static schedule computed by our approach which accounts for the bus interference and the dependencies between tasks. Figure 5 compares the overall estimated response times obtained with different analyses. We note that taking into account the memory banks always yields a better estimation of the overall response time. Further, taking into account the two-phase execution model ($E1$), the estimation is 1.5 times smaller than the pessimistic approach ($E5$) while the analysis with the single-phase execution model ($E2$) is 1.49 times smaller. The approaches that discard the release dates ($E3$ and $E4$) are as pessimistic as $E5$ in the analysis that discards the memory banks, and only 1.06 times smaller while taking into account the memory banks.

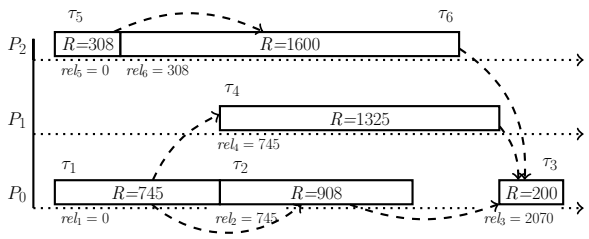

Figure 4: Static scheduling of the example in Figure 3 considering 3 memory banks

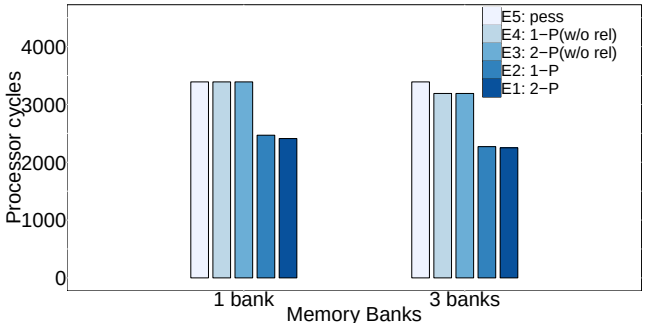

Figure 5: Comparison between the overall response time obtained with different analyses of the SDF example in Figure 3

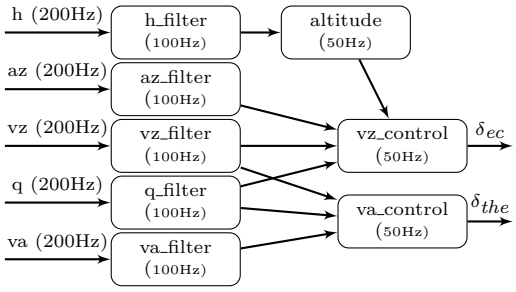

Figure 6: Flight Management System controller

4.3 ROSACE (Flight Management System)

Pagetti et al. [16] provide a case study of a Flight Management System (FMS)[4]. The case study consists of a multi-rate controller and an environment simulator. In this kind of application, the input (sampled from physical sensors) is transmitted to a controller which, after computation, sends commands to the actuators. Figure 6 illustrates the set of tasks in the SDF application, their inputs, outputs, dependencies, and their rates. We established task profiles by executing each task in isolation and measuring a trace of its execution. The profiles are given in Table 2. The inputs from sensors and the commands to the actuators are sent through the NoC via the Rx and Tx components. Since there are multiple rates, we unfold the SDF program over a hyper-period in order to

[4]Open source implementation available on the svn repository https://svn.onera.fr/schedmcore/branches/schedmcore-RTAS2014/Case_Study_RTAS

Task	PD (cycles)	MD(accesses)
altitude	275	22
az_filter	274	22
h_filter	326	24
va_control	303	24
va_filter	301	23
vz_control	320	25
vz_filter	334	25

Table 2: Task profiles of the FMS controller

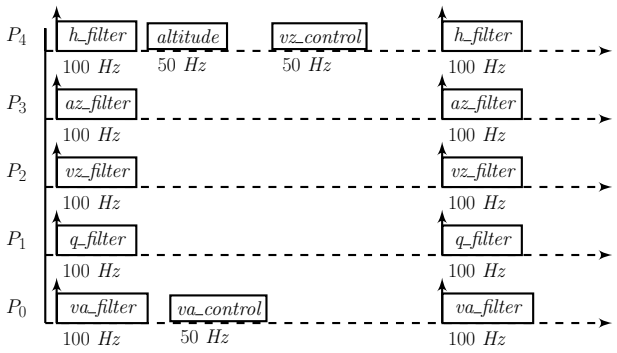

Figure 7: Task-to-core mapping and unfolding of tasks in the FMS controller

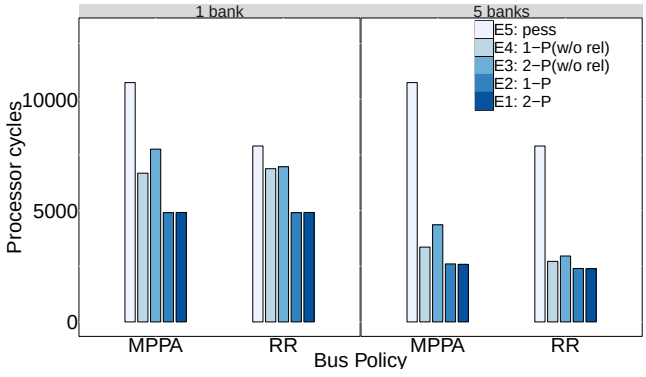

Figure 8: The smallest schedulable period obtained with different analyses

make the model compatible with our approach. In this case, tasks with a frequency of 100 Hz execute twice within the hyper-period, while tasks with a frequency of 50 Hz execute only once. Further, the Rx component writes the inputs (h, az, vz, q, va) to the shared memory four times (200 Hz) within the hyper-period, while the Tx component reads and transmits the outputs $(\delta_{ec}, \delta_{the})$ once (50 Hz). Our experiments consider a time window with the length of the hyper-period that starts with the Tx accesses from the previous execution of the program.

There are several possible mappings for the multi-rate application. We choose the mapping described in Figure 7 and evaluate its schedulability with the previously defined analyses. We also consider a single-level round-robin bus (RR) as well as the multi-level arbiter (MPPA). This allows us to compare the performance of the MPPA against the conventional RR arbitration policy using our approach. Figure 8 gives the smallest period, in processor cycles, for which the mapping in Figure 7 is schedulable. This is equivalent to finding the slowest processor clock frequency that satisfies the scheduling requirements.

The results in Figure 8 show that accounting for the memory banks improves the estimation with a factor of 1.77 to 2.52 in $E1, E2, E3, E4$ (5 banks vs. 1 bank). Our refined approach that takes into account the number of memory banks and the release dates can verify schedulability with a hyper-period of 2588 cycles ($E1$) and 2604 cycles ($E2$) assuming the MPPA bus. This represents an improvement by a factor of 4.15 ($E1$) compared to the pessimistic approach in $E5$ with 10748 cycles. The gain achieved by considering release dates is a factor of 1.68 in $E1$ (respectively 1.23 in $E2$) when compared against $E3$ (respectively $E4$) which ignores release dates. Our analysis with the RR bus gives an estimation of 2388 cycles in $E1$ (respectively 2400 cycles in $E2$) which corresponds to a gain of a factor 3.3 when compared to the pessimistic approach in $E5$ that has 7900 cycles. Note that the two-phase model is more pessimistic in this case than the single-phase model. This is due to accumulated pessimistic considerations on the write phase and the execution phase which may lead in some cases to a higher estimation than when the execution is considered as a single phase. We address this in more detail in the technical report [20]. The analysis of the RR arbiter provides slightly better performance than that for the multi-level arbiter. Any pessimistic assumption in the analysis have a higher effect on the multi-level arbiter than the RR arbiter. This is due to the fixed priority level that pessimistically counts all highest priority accesses at each bus access.

Finally, we comment on the run-time of our approach. The analysis of the FMS controller takes 0.15 seconds (Intel 2.4 GHz CPU). The analysed hyper-period has 18 tasks. The analysis in $E2$ (single-phase execution model) takes 4 iterations in Algorithm 3 and at most 20 iterations at each execution of Algorithm 1. In $E1$, the analysed hyper-period has 31 subtasks/tasks and takes 6 iterations in Algorithm 3 and at most 31 iterations at each execution of Algorithm 1.

5. CONCLUSIONS AND FUTURE WORK

We presented an analysis able to compute a valid static schedule of a synchronous data flow application on the Kalray MPPA-256 multi-core architecture with shared memory and a multi-level arbiter. We start the analysis with a given mapping, set of dependencies between tasks and precedence constraints: the choice of the mapping and the order of tasks on a given core can either be defined manually or delegated to a separate allocation algorithm.

The analysis we derived was based on the Multicore Response Time Analysis (MRTA) framework [1]. We extended this framework by deriving a mathematical model of the multi-level bus arbitration policy used by the Kalray MPPA-256. Further, we refined the analysis to account for the release dates and response times of co-runners, and the use of memory banks. Improvements to the precision of the analysis may be achieved by splitting each task into two sequential phases, with the majority of the memory accesses in the first phase, and a small number of writes in the second phase. Our experimental evaluation focussed on the ROSACE avionics case study. Using measurements from the Kalray MPPA-256 as a basis, we showed that the new analysis introduced in this paper leads to response times that are a factor of 4.25 smaller for this application,

compared to the default approach of assuming that each access is subject to the worst-case interference.

While we accurately model the 3-level bus arbiter and the set of memory banks in the Kalray MPPA-256 architecture, some work is still needed to model the end-to-end delay for a complete application. We focused on local interferences between cores when accessing the memory. We will later work on a model of the NoC traffic, i.e. $A_i^{Rx,b}$ and $A_i^{Tx,b}$. Any formula returning a number of accesses for a given time interval can be plugged into their computation: we can use the hardware configuration of the packet shaper (at the exit of each MPPA cluster), which limits the allowed bandwidth for each compute cluster on the NoC. Computations based on Network Calculus can also provide the worst-case number of accesses for a time window [8]. We can also use knowledge of the application's architecture (e.g. compute the amount of data that is written to and from the cluster during each clock cycle of the SDF application). The resource manager will also require consideration, as in general it can access the shared memory and is itself a shared resource to consider in response time computations.

We considered that each access to the memory takes 10 cycles, and may delay other accesses by 10 cycles. This is actually a worst case as 10 cycles elapse between the start and the end of a memory transaction, but a more precise model should take transaction pipelining into account: during each of the 10 cycles above, only one stage of the pipeline is used by the transaction. In the best case, a core and memory couple can execute one transaction per cycle. Taking this into account should lead to a dramatic improvement in the precision of our analysis, but is non-trivial since interference at different levels of the arbiter happen at different stages of the pipeline. Other details of the architecture will be modeled more finely in the future: for example, write transactions are asynchronous, while read transactions block the core until completion (no speculation).

Acknowledgments

This work was funded in part by the grant CAPACITES (PIA-FSN2 n°P3425-146798) from the French *Ministère de l'économie, des finances et de l'industrie*, the EPSRC project MCC (EP/K011626/1) and the INRIA International Chair program. EPSRC Research Data Management: No new primary data was created during this study.

6. REFERENCES

[1] S. Altmeyer, R. I. Davis, L. Indrusiak, C. Maiza, V. Nelis, and J. Reineke. A generic and compositional framework for multicore response time analysis. In *RTNS 2015*, pages 129–138.

[2] S. Altmeyer, R. I. Davis, and C. Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, 48(5):499–526, 2012.

[3] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: An open toolbox for adaptive WCET analysis. In *SEUS 2010*, pages 35–46.

[4] G. Berry. *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems: Proceedings of the GM R&D Workshop*, chapter SCADE: Synchronous Design and Validation of Embedded Control Software, pages 19–33. 2007.

[5] T. Carle, D. Potop-Butucaru, Y. Sorel, and D. Lesens. From dataflow specification to multiprocessor partitioned time-triggered real-time implementation. *LITES*, 2(2):01:1–01:30, 2015.

[6] J. Choi, D. Kang, and S. Ha. Conservative modeling of shared resource contention for dependent tasks in partitioned multi-core systems. In *DATE 2016*, pages 181–186.

[7] D. Dasari, V. Nelis, and B. Akesson. A framework for memory contention analysis in multi-core platforms. *Real-Time Systems*, pages 1–51, 2015.

[8] B. D. de Dinechin, Y. Durand, D. van Amstel, and A. Ghiti. Guaranteed services of the NoC of a manycore processor. NoCArc 2014, pages 11–16.

[9] B. D. de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager. Time-critical computing on a single-chip massively parallel processor. In *DATE 2014*, pages 97:1–97:6.

[10] G. Giannopoulou, N. Stoimenov, P. Huang, L. Thiele, and B. D. de Dinechin. Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources. *Real-Time Systems*, 52(4):399–449, 2016.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[12] K. Lampka, G. Giannopoulou, R. Pellizzoni, Z. Wu, and N. Stoimenov. A formal approach to the WCRT analysis of multicore systems with memory contention under phase-structured task sets. *Real-Time Systems*, 50(5-6):736–773, 2014.

[13] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo. Memory-processor co-scheduling in fixed priority systems. In *RTNS 2015*.

[14] J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *ECRTS 2014*, pages 109–118.

[15] V. Nélis, P. M. Yomsi, and L. M. Pinho. The variability of application execution times on a multi-core platform". In *WCET 2016*.

[16] C. Pagetti, D. Saussie, R. Gratia, E. Noulard, and P. Siron. The ROSACE case study: From simulink specification to multi/many-core execution. RTAS 2014, pages 309–318.

[17] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *DATE 2010*, pages 741–746.

[18] W. Puffitsch, E. Noulard, and C. Pagetti. Mapping a multi-rate synchronous language to a many-core processor. In *RTAS 2013*, pages 293–302.

[19] P. Radojković, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F. J. Cazorla. On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. *TACO 2012*, 8(4):34.

[20] H. Rihani, M. Moy, C. Maiza, R. I. Davis, and S. Altmeyer. Response time analysis of synchronous data flow programs on a many-core processor—full version. Technical Report TR-2016-1, Verimag Research Report, 2016. http://www-verimag.imag.fr/Technical-Reports,264.html?lang=en&number=TR-2016-1.

[21] A. Schranzhofer, J.-J. Chen, and L. Thiele. Timing analysis for TDMA arbitration in resource sharing systems. RTAS 2010, pages 215–224.

[22] P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *RTAS 2016*, pages 161–172.

[23] J. Walter and W. Nebel. Energy–aware mapping and scheduling of large–scale macro data–flow applications. In *IDEA 2015*.

[24] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008.