A study of predictable execution models implementation for industrial data-flow applications on a multi-core platform with shared banked memory

Matheus Schuh, Claire Maïza, Joël Goossens, Pascal Raymond, Benoît Dupont de Dinechin

A study of predictable execution models implementation for industrial data-flow applications on a multi-core platform with shared banked memory

Matheus Schuh[*,†]
[*] *Univ. Grenoble Alpes*
CNRS, Grenoble INP, VERIMAG
38000 Grenoble, France
matheus.schuh@univ-grenoble-alpes.fr

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

Joël Goossens
Univ. libre de Bruxelles
Faculté des Sciences
1050 Bruxelles, Belgique
joel.goossens@ulb.be

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

Benoît Dupont de Dinechin[†]
[†] *Kalray S.A.*
Montbonnot-Saint-Martin, France
benoit.dinechin@kalray.eu

***Abstract*—We study the implementation of data-flow applications on multi-core processor with on-chip shared multi-banked memory. Specifically, we consider the Kalray MPPA2 processor and three applications coded using the industrial toolchain SCADE Suite. We focus on the runtime environment assuming global static scheduling, time-triggered and non-preemptive execution of tasks. Our contributions include (i) a technique to implement SCADE applications compliant with execution models inspired by PREMs (PRedictable Execution Models), (ii) an exhaustive comparison of three execution models with and without isolation, and finally (iii) guidelines for predictable implementation of a data-flow application on multi-core processors with shared on-chip memory.**

I. Introduction

The implementation of critical applications must satisfy timing constraints, specifically bounded execution times of tasks and bounded communication delays for any potential interference. Implementation on multi-core processors classically uses spatial and temporal isolation to ensure the absence of interference. Any interference causes a potential delay that requires bounding. Ten years ago, this was seen as a main issue for time-predictability [1]. However, recent work has shown that such interference could be taken into account without scalability issues [2].

In this work, we implement PRedictable Execution Models (PREMs) [3] on a multi-core processor for data-flow applications developed with the industrial toolchain SCADE Suite [4]. SCADE applications are intrinsically phased with a data read phase before a local execution and a final write phase of processed data. On a multi-core processor, this limits the potential interference due to shared memory accesses to the data access phases (read and write). We target a multi-core processor with shared multi-banked on-chip memory. Furthermore, each memory bank is accessed through a dedicated memory bus arbiter with service guarantees. Taking advantage of these features enables higher performance while enforcing time-predictability through software-defined privatization of the local memory banks. Specifically, we compare PREMs by varying the following parameters: (i) a timing analysis that includes the delays of shared memory interference, versus an implementation with isolated phases to avoid any memory access interference; and (ii) the mapping of the inter-task communication buffers into the on-chip shared memory either distributed to memory banks assigned to the cores, or centralized into a dedicated memory bank.

PREMs were introduced to enable time-predictable execution of applications on COTS-based embedded systems. They are characterized by: division of jobs into a sequence of non-preemptive scheduling intervals; and time-predictable execution of some of these scheduling intervals by splitting them into shared memory access phases and local execution phases. The original PREM was motivated by the issue of distant and long-latency shared memory accesses with a lot of potential interferences. Further work has applied the PREM ideas to multi-core processors that include multiple on-chip local memories, in particular the Acquisition Execution Restitution (AER) model [5], which is a variant of 3-Phased execution models, also known as Read Execute Write (REW).

In this work, we are interested in the end-to-end execu-

tion time of data-flow applications. We propose scheduling algorithms that take into account the application constraints and the target multi-core architecture: a compute cluster of the Kalray MPPA2 processor, which is composed of multiple cores sharing a multi-banked local memory. We study the implementation of the different execution models on the MPPA2 processor for three case-studies: a simple data-flow example, an open-source avionics flight controller, and an industrial automotive control unit program. All three applications are developed using the industrial toolchain SCADE Suite. As discussed later, using SCADE gives additional constraints on the memory phases. We assume a time-triggered implementation with non-preemptive global static scheduling of tasks. A time-triggered implementation means that the tasks are initiated by their release dates (in opposition to event-triggered, where an event causes the release of a task). We focus on applications that fit into the multi-banked memory within a cluster: as observed by [5], this hypothesis is realistic for most highly-critical applications.

Our contributions are:

- a method to implement SCADE applications, inspired by PREM, on a multi-core processor with multi-banked memory, including new scheduling algorithms to isolate memory phases;
- a detailed comparison of studied execution models with and without memory temporal isolation;
- a discussion of predictable implementation for data-flow applications on multi-core processors with shared memory compliant with the proposed execution models.

The paper is organized as follows. Section II details the context of our work: data-flow applications, SCADE features, and a description of the Kalray MPPA2 processor. Section III introduces the execution models under study and illustrate them with examples. Section IV provides an overview of the existing methods we use for interference and Worst-Case Response Time (WCRT) analyzes. We survey related work and position our contributions in Section V. Section VI details our implementation of each execution model and the associated scheduling algorithms. Section VII presents the three case-studies and show experimental results. In the last section, we discuss what we learned from our experiments with guidelines, open questions and opportunities for future work.

II. Context

A. Data-flow application

Critical real-time applications are commonly developed using high-level data-flow languages, such as Simulink or SCADE. In such languages, programs are oriented graphs where nodes are computational units, and directed wires represent data transfers between nodes. We specifically consider here SCADE from the SCADE Suite toolchain, which is used for industrial critical applications, for instance in avionics or automotive. Sequential code generation for SCADE was developed in the 90s, and can be considered as state-of-the-art. The generated code intrinsically expose a REW structure, similar to the more recently defined AER model [5]. However, the data-flow semantics of SCADE is intrinsically parallel: nodes can be executed concurrently, as far as the scheduling constraints induced by data dependencies (wires) are fulfilled. In this paper, we consider the parallel implementation of SCADE programs: nodes are compiled into classical sequential tasks, and our goal is to map and schedule those tasks on a multi-core architecture.

The main features of SCADE Suite and its Multi-Core Code Generator (MCG) that are used in this paper are:

- a data-flow graph where each arrow represents a precedence constraint and a communication from a task to another, i.e. data-transfer;
- an intrinsic phased execution model that prepares the read/write phases to be executed sequentially with the execution phase (REW). Note that these memory-phases are filled with a sequence of read/write accesses subsequently referred to as "memory transactions". In this paper, we consider a memory phase as a sequence of memory transaction tasks that may be released independently;
- a mapping and scheduling of a set of tasks to each core: this mapping is an ordered set of tasks assigned to each core (ordered by chronological execution);
- a description of the data-structures used to communicate between tasks.

SCADE applications are periodic. In case of applications that are composed of tasks with different harmonic periods, we expand the application into a hyper-period to apply our framework. This ensures that even with a multi-rate initial program, a common denominator is found and, after this expansion, each period will follow the same implementation scheme and static schedule.

B. Multi-core processor with multi-banked local memory

The Kalray MPPA2 (Bostan) is a many-core processor whose architecture connects a set of multi-core entities named Compute Clusters (CCs). Each cluster contains 16 Processing Engines (PEs) cores for general purpose computing. These cores share access to a Shared MEMory (SMEM) of 2 MB, split into 16 independently arbitrated banks of 128 KB each. This SMEM is local to a cluster, physically close to the PEs, which enables low latency access.

The SMEM address mapping can be configured in an *interleaved* mode or a *blocked* mode. The first distributes sequential memory addresses across memory banks at a 64-byte granularity, which is optimal for an average use-case. The second allows contiguous blocks of memory addresses (up to 128 KB) to be directed to a single memory bank. As each bank has its own arbiter, spatial isolation can be

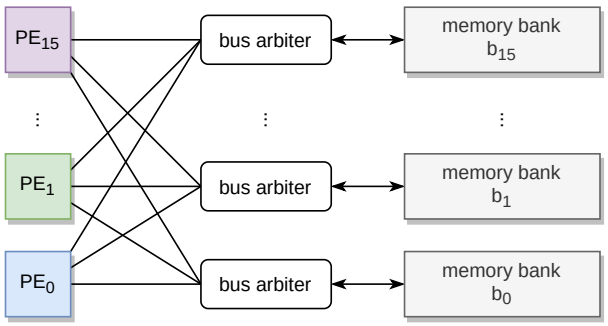

Figure 1. MPPA2 Cluster memory arbitration system

achieved if code and data are carefully placed. Figure 1 presents the cluster memory arbiters introduced here.

In this paper, we use one cluster of the MPPA2 processor and leverage its architectural features to facilitate the implementation of several execution models. All studied applications (see Section VII) fit into the SMEM, which avoids using the high latency distant global DDR memory. The SMEM is configured in *blocked* mode, and we manually place shared data according to the execution model (see Section VI) to better control memory interferences.

In a MPPA2 cluster, interference when accessing the SMEM is due to a sequence of bus arbiters. Note that there are instruction and data caches, but they are private to a core: there is a potential interference only in case of a miss of one of these cache memories. The Worst Case number of memory Accesses is referred to as WCA in this paper. Each memory access passes through 3 arbiters (2 Round-Robins and 1 Fixed-Priority). In this paper we use the interference analysis developed in [2] and explained in Section IV.

III. The studied execution models

In this section we present the execution models, inspired by PREMs, that we selected for our study. We compare their implementation according to two criteria:

- Memory partitioning
  1) a 2-Phased model with execute-write phases, see Figure 2. The memory is partitioned such that each partition is local to a core and the tasks may access another partition only during the write phases, to send the shared data. This way any read of shared data is done in the local memory during the execute phase.

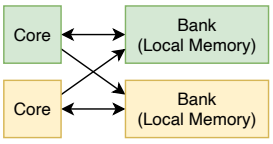

Figure 2. 2-Phased: Execute-Write

  2) a 3-Phased model with a local partition for each core accessed during the execution phase and one global *shared partition* accessed by each task during read and write phases, see Figure 3.

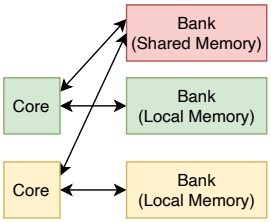

Figure 3. 3-Phased: Read-Execute-Write with Shared Bank

  3) a Memory-Centric 3-Phased model with a local partition for each core accessed during the execute phase and a global shared partition managed by a *dedicated core* that orchestrates the read and write phases for all tasks, see Figure 4.

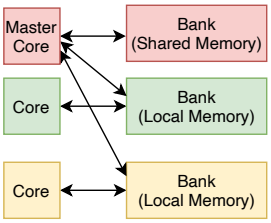

Figure 4. Memory-Centric 3-Phased with Master Core

- Memory interference
  a- no interference: the mapping and scheduling ensures no interference by a software isolation between memory phases;
  b- analyzed interference: an architecture model is used to estimate the interference delay and take it into account as part of the WCRT.

We use a simple data-flow application example to illustrate the implementation of the execution models under study. Figure 5 gives the Data-Flow Graph (DFG): each square represents a task (N_i for Node in data-flow terminology); each edge represents a communication (data transfer) and thus a precedence constraint.

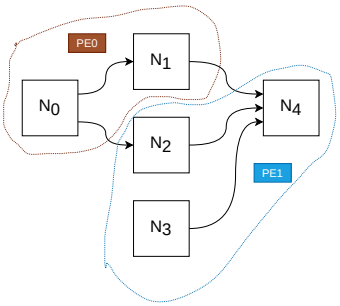

Figure 5. Example DFG

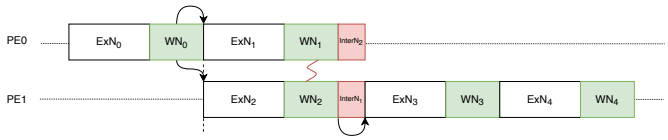

Figure 6. Example of scheduling for the 2-Phased model with interference cost

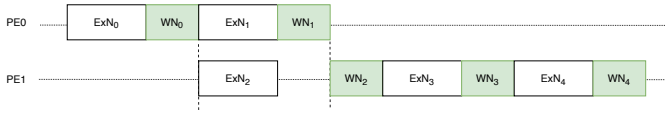

Figure 7. Example of scheduling for the isolated 2-Phased model

Figures 6 to 10 give a final schedule for the 5 execution model implementations: in white are the execute phases, in green the write phases, in yellow the read phases, and in red the delays due to interference. Figures 6 and 7 represent two final schedules for the 2-Phased model where each task reads data locally and writes data to the reader memory. We observe that when interference is considered (Figure 6) there may be additional delay to take into account. Here, the write phase of task N_1 and N_2 interfere due to the fact that they both write in the local memory of task N_4. In the isolated implementation model these two write phases cannot occur simultaneously, to prevent any interference. In Figure 7 we see that WN_2 starts once WN_1 is done.

For the 3-Phased execution models (Figures 8 and 9), there are the additional read phases as each task reads from the shared memory. We observe that there is additional interference in the read phases of tasks N_1 and N_2. We also see that a scheduling algorithm is required to achieve temporal isolation: here a priority is given to task N_1 to start its read phase RN_1 before task N_2.

For the Memory-Centric execution model implementation, there is no possibility of interference due to the fact that each memory transaction is done by the same core. Thus, there is only the isolated schedule given in Figure 10. Here we also observe that a scheduler is required: for instance, the read phase of task N_2 (RN_2) is scheduled

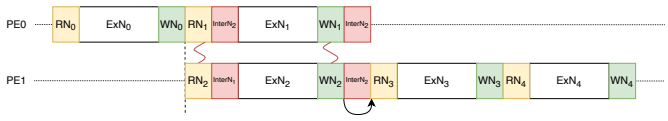

Figure 8. Example of scheduling for the 3-Phased model with interference cost

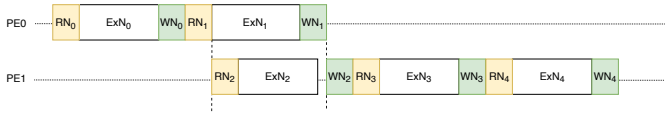

Figure 9. Example of scheduling for the isolated 3-Phased model

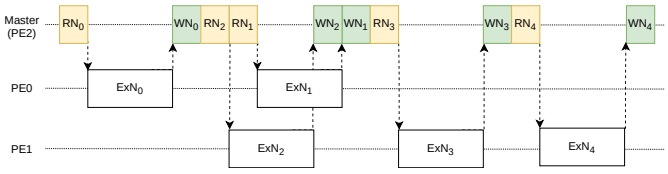

Figure 10. Example of scheduling for the isolated Memory-Centric model

before the read phase of task N_1 (RN_1).

In all schedules we observe that there is the same sequence of execute tasks on core PE_0 and on core PE_1: this is to illustrate that we work from an initial schedule given by SCADE that must be preserved. The freedom in the schedule is only between the memory phases. Note that these new schedules (isolated 3-Phased and Memory-Centric) are *global* because they must take into account the global data-dependencies and they are constrained by what is executed on other cores. For instance, in Figure 9 tasks RN_1 and RN_2 are executed on two distinct cores but must be executed in isolation nevertheless.

IV. Multi-Core Interference Analysis (MIA)

For the interference analysis we need a model of the MPPA2 memory system, a computation step to bound the interference and take it into account in the Worst-Case Response Time (WCRT). While there exists previous work on the interference sources on the MPPA2 processor [6]–[8], MIA[1] is, as far as we know, the only free and open-source tool that meets all our requirements.

MIA combines [2]:

- a model of the MPPA2 memory system: the idea is to analyze the interference sources using a model for each one of the bus arbiters before reaching the cluster memory;
- a WCRT analysis: it uses given Worst-Case Execution Time (WCET) and the WCA to estimate a global WCRT including interference bounds;
- a release date analysis for time-triggered implementation: from an initial schedule, a data-flow graph that gives precedence constraints and the WCRT analysis, MIA gives a release date for each task that preserves precedence constraints and the initial schedule.

There is still one open question about how to enforce isolation between memory tasks in MIA. The method we adopted from [2], [9] is to add dependencies in the data-flow graph so they do not overlap. For instance, on Figure 7 we see that adding a precedence constraint between tasks WN_1 and WN_2, enforces the latter to start only after the completion of the former, thus ensuring temporal isolation.

V. State of the art

In this section we present a short survey on research work according to our previously defined criteria: memory

[1]Available at https://gricad-gitlab.univ-grenoble-alpes.fr/verimag/synchrone/mia

partitioning and interference models. The PREM model has been introduced and extensively studied on mono-core in [3], [10], [11].

Regarding the memory partitioning, related work can be classified as:

- **2-Phased model:** This implementation model is only used in some works [2], [12] and is a specificity of architectures that provide bank or memory privatization features, such as the MPPA2 processor.
- **3-Phased model:** This model is the most used in related work. The shared resource to read/write is not always a memory and the PREM model may be used for I/O access [13]. The shared memory may be DRAM main memory [14]–[16] or scratchpad local memory [5], [7], [17]–[19]. It may also take into account the DMA load/unload [20]. In some articles, the architecture is not realistic but the focus is on the bus access model [21], [22].
- **Memory-Centric 3-Phased model:** This model is studied/used in [14], [16], [23], [24].

Concerning the memory interference, related work can be classified as:

- **no interference:** In [14], [19], [24], the mapping/scheduling ensures no interference by isolating memory phases. The isolation may be enforced by scheduling the task phases and may be combined to partitioning. This partitioning is used to isolate execution phases from memory access phases. The partitions may be based on time-division [13], [17], [20], [25] or round-robin software partitions. Also, the partition may be preemptive and combined with priority promotion technique.
- **analyzed interference:** The interference delay is estimated and taken into account as part of the WCRT in [21], [22]. In some related work, the interference is taken into account as a parameter to improve the scheduling on each core or the global mapping of tasks onto cores [15], [18], [23], [24]. In [7], the memory phases are even fragmented to improve the precision of the interference analysis. The scheduling may use a software partitioning to get a preciser interference analysis (less interfering tasks) [16], [26], [27].

In our work, we also focus on the orchestration of SCADE code. A few works also address the challenge of producing phased code as part of the code generation/compilation step [17], [20], [28] or in the operating system [14]. A distinct approach is to provide predictable execution using different models, such as the Logical Execution Time (LET) in [29] where synchronization points are used for write phases and the read phases are always executed at the beginning of a task's period.

In this paper, we consider applications described by a data-flow graph as in related work [7], [17], [18], [23].

Another characteristic of the PREM is that execute phases occur in memory isolation. Isolation is typically achieved by using the memory cache as a private memory [24], [30], [31]. In this work and some others, the target architecture provides a multi-banked local memory that allows to enforce isolation in a more straightforward way [19], [20], [23], [25].

In Section VI-B we present scheduling algorithms for the memory phases of tasks when they run in isolation. A comparison of the efficiency of such algorithms is presented in [30] but in a dynamic runtime and without precedence constraints between the tasks. [18] presents a forward list scheduling algorithm but their REW task model is said to be contiguous, limiting the flexibility of the schedule. In our study, we consider the impact of this *contiguous* REW, as it is similar to the SCADE intrinsic notion of PREM.

Implementation of data-flow industrial applications to multi-core platforms were presented in [5], [17]. These papers target different platforms, but with similar local shared memory. In both of them, isolation between memory access phases is implemented through hardware isolation (TDMA bus [17]) or software isolation [5]. Our work is complementary and our methodology could be applied to these hardware targets (TMS320C6678 in [5] and ARM based multi-core in [17]).

Our work is inspired by all the previous papers listed in this section and aims to give a comparison of execution models while providing new methods for their implementation.

VI. Our implementation

We study the implementation of phased execution models for SCADE applications on a multi-core processor with multi-banked local memory, specifically a compute cluster of MPPA2 processor. This avoids any resource contention that may arise when accessing the DRAM through the NoC or DMA components. Thus, the only memory contention model required for this study is of the cluster shared memory, which is already available in MIA. More details are given in Sections IV and VII.

A. Data-flow to PREM

First, we summarize our choices on how to implement the execution models onto the MPPA2 processor. For the implementation of the 2-Phased model with execute-write phases, each core accesses its local bank and may access the other banks during the write phase to communicate with tasks mapped onto other cores, as in [2]. For the 3-Phased model, each core accesses its local bank and one bank is used as shared memory. For the Memory-Centric 3-Phased model, each core access its local bank and a dedicated core is assigned a shared memory bank and runs the orchestration task. To implement our isolated model, the mapping/scheduling ensures no interference by a software isolation between memory phases. We introduce new scheduling algorithms for the 3-Phased and Memory-Centric cases. These algorithms are detailed in the next

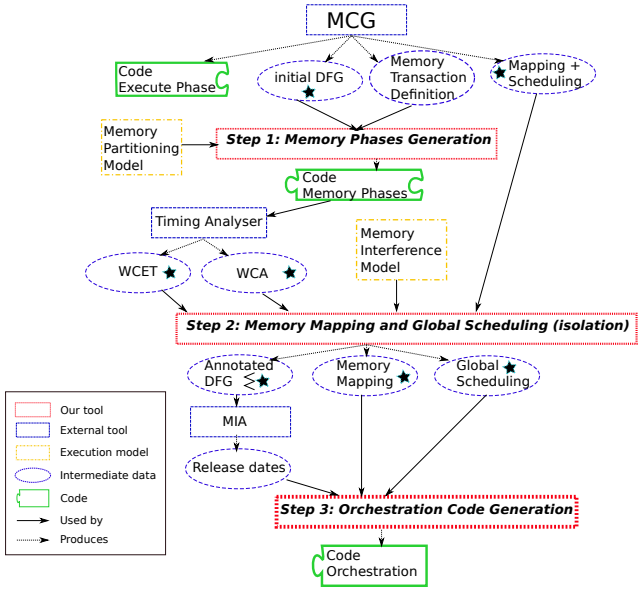

Figure 11. Implementation workflow

subsection. For the implementation of our execution models including interference, this delay is estimated and taken into account as part of the worst-case response time. To achieve this we use the MIA tool (see Section II).

The global workflow of our implementation is given in Figure 11 with a legend that identifies where external tools are used, the intermediate data and code produced, as well as the steps of our own tool.

The first input is provided by the SCADE MCG: a data-flow graph with precedence constraints, an initial mapping and scheduling from tasks to cores, the code for the execute phase and the data structures for the communication (size and type).

The **Step 1** of our tool consists in generating the code for the memory transactions. Based on the description of the data structures used for communication (named channels in SCADE documentation), the communication graph and the execution model, the code for each memory transaction is produced. The kind of execution model influences this step due to the number of phases: 2-Phased only has a write phase, while 3-Phased and Memory-Centric have read and write phases.

At this point all purely functional code related to SCADE has been generated. Then, for each phase of each task, a timing analyzer provides its WCET and the associated WCA. The WCA corresponds to the memory accesses generated by data and instruction cache misses, which will be later incorporated into the WCRT by MIA.

In **Step 2**, we generate:

- a global scheduling: required if the implementation model is *without interference*. The schedule is used to enforce the isolation for the 3-Phased and Memory-Centric models. The algorithms are explained with details in Section VI-B;
- a memory mapping: this mapping script realizes the placement of data and code from each task in specific banks, according to the execution model, as explained in Section III;
- an annotated DFG: a file that summarizes all necessary information to be given to MIA. It incorporates all components tagged with a star in Figure 11: the initial DFG for precedence constraints, the mapping that provides task to core awareness, the initial or global scheduling and the timing bounds. The global scheduling adds additional dependencies between memory transactions to enforce isolation.

MIA is then used exclusively at this point to generate release dates that respect the information contained in this annotated DFG: the mapping, the scheduling and the dependencies.

In **Step 3** we generate the platform-specific code to orchestrate the application. The release dates are incorporated in this code with one orchestration function per core. Note that a cluster in the MPPA2 processor is mesochronous, thus we use a global barrier synchronization during an initialization phase to ensure that afterwards the execution flow and timings are respected.

B. Scheduling algorithms

This section presents 3 global scheduling algorithms to isolate the memory transactions of the memory phases. The general goal of these algorithms is to anchor the order between these transactions across all cores. This execution order is then given to MIA in the form of additional dependencies and priority in the data-flow graph, as explained in Section II, so that the computed release dates ensure temporal isolation between any task that access the memory of the system.

1) Background concepts: Before introducing the general intuition of the algorithms and their listings, we want to recall and properly define some terms that will be used onward.

In Section II memory transactions and phases have been introduced and it is important to clarify their difference: a *transaction* is an indivisible task that access the memory, while a *phase* can be composed of multiple distinct transactions. The algorithms here presented perform the schedule at the transaction level but must respect precedence constraints at the phase level.

The definition of a dependent memory transaction is strongly related to a typical data-flow task dependency, with some small particularities. A memory transaction m is constituted of a memory operation: either read or write, respectively m_r and m_w. It also has two compute transactions associated with it here called c_1 and c_2. Each transaction has an associated release (rel) and end (end) date. The definition then depends on the memory partitioning model:

- For the 2-Phased model: there are only write operations, so given a memory transaction m_w from c_1 to c_2, it is dependent on c_1 being finished, which imposes that $rel_{m_w} \geq end_{c_1}$. There is no algorithm for this model as the scheduling is trivial.
- For the 3-Phased and Memory-Centric models: there are read and write operations. Given a memory transaction m_w from c_1 to c_2, it is dependent on c_1 being finished, which imposes that $rel_{m_w} \geq end_{c_1}$. Given a memory transaction m_r from c_2 to c_1, it is dependent on its mirror write transaction, e.g. m_w, which imposes that $rel_{m_w} \geq end_{m_r}$.

2) Overview and shared utilities: All algorithms have the same input and output. The starting point is a set ℓ composed of read and write phases, ordered in the scope of a core, following the initial mapping and scheduling. The general idea is to pick transactions from ℓ and put them in the set g, which is the globally ordered set of memory transactions across all the cores. We start with the first transaction from the first phase of the first core, which always perform a write operation, meaning that the dataflow applications are either closed (they do not require external inputs) or already initialized. Then, according to the execution model and the algorithm, we either continue scheduling the other transactions from this phase or start looking for other transactions that can be scheduled because their dependencies are satisfied. The algorithms end as soon as all memory transactions are scheduled: either if ℓ and g have the same number of elements, or if we have already iterated through all memory transactions in ℓ.

In the algorithms we assume the existence of certain utility functions used in the listings and that have their core functionality explained here:

- `DepOk`(t) — checks if all the dependencies of a transaction t are already in g;
- `AreOnSameCore`($t1, t2$) — returns true if $t1$ and $t2$ are mapped to the same core;
- `GetWrPhase`(t) — returns the write phase associated with the execute task t;
- `GetRdPhase`(t) — returns the read phase associated with the execute task t;
- `GetMirror`(t) — returns the mirrored t transaction, i.e. for a *write* transaction between N_1 and N_0, its mirror transaction is a *read* transaction between N_0 and N_1. Note that in case of a mirror transaction the read is only subject to the precedence of the corresponding write. Thus, it is eligible for scheduling as soon as the write transaction ends.

3) Algorithms: For the 3-Phased execution, the memory phases belonging to a task are run on the same core as the execute phase. This restricts the algorithm, as they must be sequentially placed in this core due to the intrinsic SCADE execution model. Any memory transactions belonging to other tasks of the same core cannot be interleaved as they would violate the initial mapping and scheduling. To explore the impact of this restriction, while also proposing a solution, we introduce two variants of an algorithm for this 3-Phased model.

Algorithm 1 shows the first variant. As in [7], [18], the 3 phases (REW) are considered as a contiguous entity, without any idle time between the memory or execute transactions. The algorithm works as explained in the overview but it always schedules the remainder of a write phase after placing a write transaction on g (Line 5). Each write transaction unblocks its mirror read, which will only be scheduled if all other read transactions of the same phase are also unblocked. Otherwise, this read transaction waits in a leftover set.

Algorithm 1: 3-Phased contiguous memory phases, referred to as *Cont* in Section VII

```
1  w_sched = list(); r_leftover = list();
2  foreach t in ℓ do
3      if 'write' in t and t not in g then
4          g.append(t); w_sched.append(t);
5          foreach t2 in GetWrPhase(t) do
6              g.append(t2); w_sched.append(t2);
7          foreach t2 in w_sched do
8              if DepOk(t2) then
9                  foreach t3 in GetRdPhase(t2) do
10                     g.append(t3);
11                     if t3 in r_leftover then
                           r_leftover.remove(t3);
12                 foreach t3 in GetWrPhase(t2) do
13                     g.append(t3);
14             else r_leftover.append(t2) ;
15         w_sched.clear();
16     foreach t2 in r_leftover do
17         if DepOk(t2) and t2 not in g then
18             g.append(t2);
19             if t2 in r_leftover then
                   r_leftover.remove(t2);
```

Example 1: To illustrate the difference of behavior between the 3 algorithms we will use the program in Figure 5. We consider that the scheduling algorithm is at the point of deciding about the schedule of the write phase of the task N_0. This phase is composed of two transactions: `N0_write_N1` and `N0_write_N2`. We also know that N_0 and N_1 are mapped to the same core (PE_0) and N_2 is mapped to a distinct core (PE_1). Algorithm 1 would globally schedule the transactions: `N0_write_N1` → `N0_write_N2` → `N1_read_N0` → `N2_read_N0`. Note that with Algorithm 1 the PE_1 remains stuck until `N1_read_N0` finishes, even though the data dependency has already been satisfied.

Algorithm 2 shows the second 3-Phased variant. We remove the contiguous schedule constraint and add the possibility of introducing idle time between memory transactions of the same task and give priority to scheduling read transactions. This allows to unblock execute phases earlier than the previous algorithm and have a smaller response time. We use a double-ended queue (abbreviated here as deque) for the scheduling candidates that are popped at each iteration. Intuitively, once a transaction is scheduled, its mirror transaction may be:

- A read or write that needs to be placed contiguously with the transactions of the same phase;
- A read transaction belonging to another core that may be scheduled directly and unblocks this other core from an idle state.

Thus, according to the mirror transaction operation and mapping, it is placed at different positions in the deque to be scheduled in the next iterations.

Algorithm 2: 3-Phased with idle memory phases, referred to as *Opt* in Section VII

```
1  sched_cand = deque(first_task_write_transactions);
2  while ℓ.size() ≠ g.size() do
3      c ← sched_cand.popleft();
4      if c not in g then
5          if DepOk(c) then
6              g.append(c);
7              m ← GetMirror(c);
8              if 'write' in c then
9                  tr ← GetWrPhase(c) +
                    GetRdPhase(c);
10                 idx ← −1 ;
11                 foreach c2 in sched_cand do
12                     if c2 in tr then idx ← c2.idx();
13                 if idx ≠ −1 then
                    sched_cand.insert(idx + 1, m);
14                 else
15                     if AreOnSameCore(c,m) then
                        sched_cand.append(m);
16                     else sched_cand.appendleft(m);
17             else if 'read' in c then
18                 foreach t in ℓ do
19                     if m in t then
                        sched_cand.append(t);
20         else sched_cand.append(c);
```

Example 2: For our illustrative program in Figure 5 and the moment of scheduling the write phase of the task N_0, Algorithm 2, instead of blindly sequentially scheduling all write transactions, searches for unblocked read transactions (mirror) mapped to another core. Due to SCADE code generation restrictions the algorithm cannot break the initial ordering between the transactions and interleave them if they belong to the same core. However, it introduces idle time and give priority to scheduling read tasks of other cores. Therefore, the global schedule given here is `N0_write_N1` → `N0_write_N2` → `N2_read_N0` → `N1_read_N0`. This allows PE_1 to start running the execute phase of node N_2 earlier, which gives an overall shorter response time.

Remember that both scheduling algorithms presented respect the SCADE semantics and preserve the DFG order as well as other constraints. The only optimization done in Algorithm 2 is to allow idle slots between memory and execute transactions.

For the Memory-Centric execution model, the sequential constraint of SCADE is loosened as memory transactions are mapped to a different core. Thus, there is room for a lot more flexibility when scheduling these memory transactions: we have the possibility to arrange them in any order as the execute phase will not happen on the same core. We can freely add idle intervals or not, interleave read/write transactions from other tasks and the execute phases will naturally follow the master core local scheduling due to the data-flow dependencies.

Algorithm 3 presents the method used. It follows the overview methodology but uses the mirror searching as in the Algorithm 2 to schedule read transactions as they are ready (Line 8), accelerating the parallelism deployment throughout all cores. If the dependencies for the read transactions are not satisfied they are placed in a leftover list that is revised in Line 13 before searching for the next write transaction in ℓ.

Algorithm 3: Memory-Centric isolation scheduling

```
1  sched_leftover = list();
2  foreach t in ℓ do
3      if 'write' in t and t not in g then
4          if DepOk(t) then
5              g.append(t);
6              m ← GetMirror(t);
7              if DepOk(m) then
8                  foreach t2 in GetRdPhase(m) do
9                      g.append(t2);
10                     if t2 in sched_leftover then
                        sched_leftover.remove(t2);
11             else sched_leftover.append(m) ;
12         else sched_leftover.append(t) ;
13     foreach t in sched_leftover do
14         if DepOk(t) then
15             g.append(t);
16             if t in sched_leftover then
                sched_leftover.remove(t);
```

Example 3: Coming back at the program in Figure 5 and the moment of scheduling the write phase of the task N_0, Algorithm 3, after scheduling a write transaction, searches for unblocked read transactions (mirror) mapped initially to any core (due to the dedicated core for memory transactions, the SCADE code generation restrictions are no longer applicable). Therefore, the global schedule is `N0_write_N1` → `N1_read_N0` → `N0_write_N2` → `N2_read_N0`. As the memory operations are mapped to a single core, this algorithm tends to behave worse than Algorithm 2.

4) Termination proofs:

- Algorithm 1: terminates because its main loop iterates over the elements of the finite set ℓ, which contains the memory transactions.
- Algorithm 2: terminates when g equals the size of ℓ, the initial set of memory transactions, meaning that it has successfully defined a global schedule for all tasks. There is also an auxiliary structure c that contains schedule candidates. At each iteration we pop one candidate from c and it is either added to g or put back into c. If it is added to g one or more transactions from ℓ are then added to c as candidates. The insertion process avoids any duplication. As the number of transactions is finite, once all candidates in c have been scheduled, the termination point is reached.
- Algorithm 3: same intuition as Algorithm 1.

5) Complexity Analysis: The complexity is given in terms of n which is the number of transactions to be scheduled.

- Algorithm 1: $O(n^3)$, as there are up to three nested loops (Line 2, Line 7 and Line 9)
- Algorithm 2: $O(n^2)$, as there are up to two nested loops (Line 2 and Lines 11;18)
- Algorithm 3: $O(n^2)$, as there are up to two nested loops (Line 2 and Lines 8;13)

To provide a comparison baseline we looked at the complexity of algorithms developed or referenced by [32] and [33]. Our three algorithms stay under $O(n^3)$ which is reasonable for an offline scheduling method. Moreover, similar algorithms found in these two references range between linear and cubic complexity, which reinforces for the $\mathcal{NP}$-hardness nature of the mapping/scheduling problem on multi-core architectures. In terms of scalability, [34] has showed that thousands of tasks can be scheduled in a reasonable time with an $O(n^2)$ complexity, which is the case for the majority of our algorithms.

VII. Experiments

We evaluate our implementation of the execution models proposed in Section III with three SCADE applications:

1) the simple example used to explain our method in Section I;
2) the ROSACE avionics case-study[2];
3) an industrial automotive Electronic Control Unit (ECU) program.

As the starting programming language of our implementation is SCADE, which is widely used in the industrial context, our goal with these applications is to evaluate the final schedule response time with real world scenarios. In particular, we are interested in comparing the possible timing improvements when taking interference into account against safe execution models that provide temporal isolation.

The implementation of the code generation steps of the workflow described in Figure 11 was done in Python with the Mako template library[3]. A bash script then ties together all the necessary steps of the workflow. The WCET and WCA of the execute and memory phases are measured after multiple runs on the processor. Timing analyzers such as OTAWA [35], Heptane [36] or aiT [37] can also be used here to estimate the WCET. For the interference estimation, it is important to know that on the MPPA2 processor, a blocking memory access requires 10 cycles to be completed and the multi-level arbitration system makes the maximal cost to be $\sum_{1\leq i\leq 15}(\min(I(P_0, P_i)) + \min((Y), (X)) + (R_x)$, where I gives the interference between specific cores, Y is the interference generated by any core, X is the interference between other initiators (T_x, RM, DSU) and R_x is the high priority initiator at the third and final level. Section IV contains more details about how the interference cost function is used.

The offline scheduling algorithms presented in Section VI-B have no significant runtime overhead in comparison with the time spent executing the whole workflow in the experiments conducted. For reference, the runtime ranges from 239ms to 525ms in the explored programs, while the complete workflow execution can take from 46s to 1m17s. In this section we will focus exclusively on the global WCRT computed by MIA and measured on the MPPA2, for each proposed execution model and application. The algorithms' runtimes show that their computational cost is reasonable, appropriate for our scenario and does not significantly impact the performance of the whole workflow.

A. Case-studies

1) Simple Data-Flow: This example application[4] was used to validate our methodology in terms of scheduling and implementation. As shown in Figure 5 the 5 task nodes are mapped to 2 cores, with N_0 and N_3 having no initial dependencies, while the others are connected to the data-flow that derives from them. The execute tasks perform basic arithmetic operations. Profiling information of this application for the 3-Phased partitioning model is:

[2]Available at https://forge.onera.fr/projects/rosace-case-study

[3]Documentation available at https://docs.makotemplates.org/

[4]Available at https://gricad-gitlab.univ-grenoble-alpes.fr/verimag/reproducible-research/simple-dflow-rtss-2020/

- Compute phases — WCET: from 260 to 326 cycles, WCA: from 2 to 7 accesses;
- Read phases — WCET: from 183 to 198 cycles, WCA: from 0 to 3 accesses;
- Write phases — WCET: from 183 to 202 cycles, WCA: from 0 to 3 accesses;
- Computation-Communication Ratio (CCR) — 1 732 compute cycles to 3 041 communication cycles.

2) ROSACE: This is an avionics open-source case-study developed by ONERA [38]. It contains a multi-periodic flight controller program that aims to be easily executed on a multi/many-core processor. The original code has been expanded into a hyper-period that normalizes the multi-periodic nature of the program. The application contains 10 nodes (execute tasks) and is mapped to 8 cores. The corresponding data-flow graph is given in Figure 12.

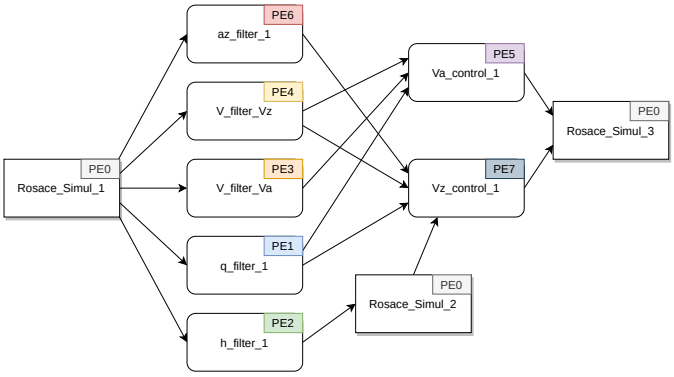

Figure 12. ROSACE data-flow graph and preliminary mapping

Profiling information of this application for the 3-Phased partitioning model is:

- Compute phases — WCET: from 624 to 74 343 741 cycles, WCA: from 7 to 195 accesses;
- Read phases — WCET: from 170 to 185 cycles, WCA: from 1 to 2 accesses;
- Write phases — WCET: from 170 to 187 cycles, WCA: from 1 to 3 accesses;
- CCR — 260 214 753 compute cycles to 5 972 communication cycles.

3) Automotive ECU: The third case-study is an industrial automotive program with 4 765 lines of functional code (not including the orchestration code). The 9 nodes are mapped to 6 cores. There is one initial task and one final task, the other 7 tasks depend only on the data produced by the initial task. This is a common structure with highly parallel periodic applications.

Profiling information of this application for the 3-Phased partitioning model is:

- Compute phases — WCET: from 465 to 2 653 cycles, WCA: from 10 to 68 accesses;
- Read phases — WCET: from 187 to 202 cycles, WCA: from 1 to 3 accesses;
- Write phases — WCET: from 187 to 202 cycles, WCA: from 1 to 3 accesses;
- CCR — 12 548 compute cycles to 5 416 communication cycles.

B. Experimental results

Experimental results are displayed in Figures 13, 14 and 15. For each case-study and each implementation we give the estimated *WCRT* computed by MIA for the code generated by our workflow and the corresponding *Measured* execution time (both in number of processor clock cycles). For the *2-Phased* and *3-Phased* implementation we give the results with *Interference* and in all cases the results for the implementation in *Isolation* are given. Remember that for the *Memory-Centric* implementation there is no interference between memory phases as they are all scheduled on the same core. Furthermore, we give the results using both isolated 3-Phased scheduling algorithms: *Cont* for the contiguous implementation of the three phases (see Algorithm 1) and *Opt* for the optimized version with potential idle time slots (see Algorithm 2).

We observe that the WCRT is always close to the measured one. This is due to the fact that we use a time-triggered implementation: the difference may only come from the difference between the WCET and the effective execution time of the last task, as the release date of the last task is identical to the one computed by MIA. Note that in case of interference, the release date of the last task is not the same as in the case of isolation. Furthermore, the difference between the measured and the bound on the WCRT may be larger due to potential interference taken into account during the execution of the last task.

From the *Interference* and *Isolation* WCRT values in all 3 figures we observe that **taking into account delays due to interference leads to shorter WCRT than isolating the memory phases.** This is attributable to the memory partitioning model limiting the interference during memory phases and also to the structure of the MPPA2 processor bus arbiter. The size of shared data is also small for all case-studies, usually less than the bus size: in this case a potential interference between

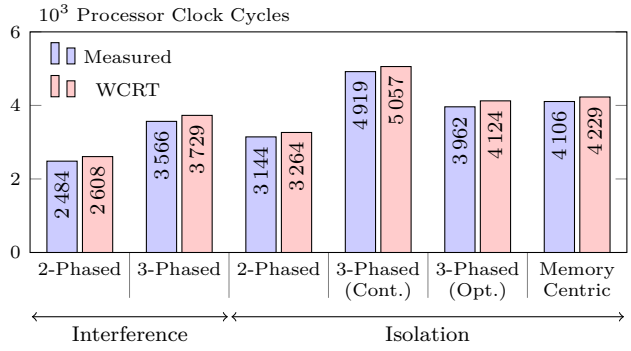

Figure 13. Simple Data Flow Results

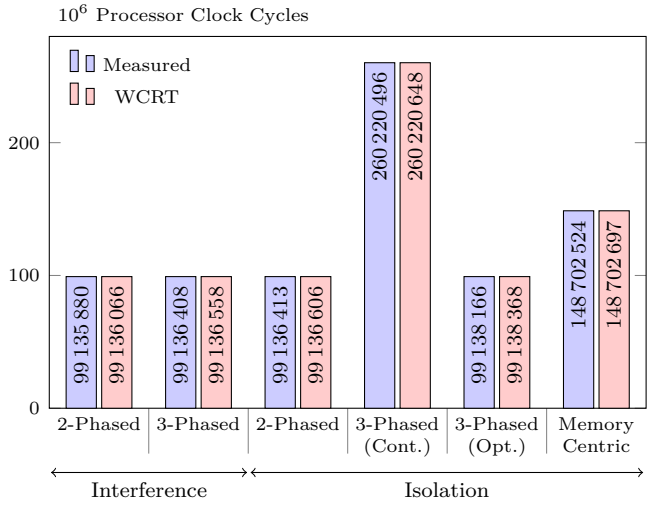

Figure 14. ROSACE Results

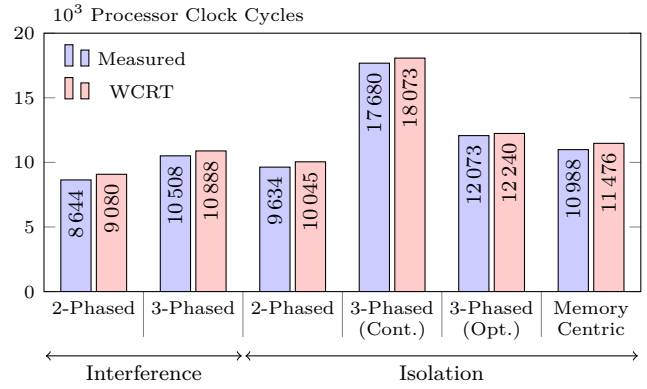

Figure 15. Automotive Use Case Results

two cores is accounted as only one additional cycle as it is simply a one-cycle wait for the round-robin arbiter. For the simple data-flow case-study, the additional cost of interference is of 2 cycles in the 2-Phased case and 3 cycles in the 3-Phased case. This happens because there are few potential interference points and only 2 cores are used. For ROSACE, in the 2-Phased implementation there is no additional cost due to interference. Note that ROSACE case-study has long execute phases and very small memory phases: this limits the probability of interference. For the 3-Phased implementation, the additional cost for ROSACE is of 2 cycles: here at least one of the additional read phases causes an interference. For the automotive case-study, the additional cost is of 2 cycles for the 2-Phased implementation and 10 cycles for the 3-Phased one. These low interference costs explains why the WCRT with interference is shorter: the cost of isolation is much heavier than the cost of interference.

Similarly, reducing the number of phases, limits the number of interference and the number of tasks to isolate with the global scheduling algorithms: a **2-Phased implementation is always more efficient than a 3-Phased one.** Note that this is also a specificity of the multi-banked memory that allows to execute locally with a predictable shared data memory (in contrast to a shared cache memory). The difference between the 2-Phased and 3-Phased is not only due to the number of phases, but also due to the distributed shared memory among the banks vs. one shared memory bank. The shared memory bank seems to be a good idea for better isolation of shared memory access. Nevertheless, our experiments show that a distribution of the shared memory on all banks is more efficient for the applications and target processor we use. Therefore, a 2-phase model with execute-write operations is preferable when it can be applied to the program, the architecture contains a multi-banked shared memory and the number of generated memory phases can be controlled.

The Memory-Centric implementation behaves most of the time worse than the 3-Phased Opt. The reason for this is the mapping of all memory phases to the same core, which forbids any concurrency between the memory phases. Note that the Memory-Centric implementation of PREM has been introduced for external shared memory where the memory access time is significantly longer.

Our optimized algorithm that introduces idle intervals instead of enforcing contiguous REW phases, yields the best results among the 3-Phased models, except for the automotive use case. This is a result of read phases being scheduled earlier by this algorithm and as consequence the execute phase may also start sooner. This is at a price of inserting idle times between memory transactions, but it is important to reinforce that the semantics of SCADE are still preserved. The exception on the automotive use case is a small difference (less than 1 000 cycles) and probably comes from its CCR profile. Another possible reason is a good timing in the scheduling between memory and execution phases that does not degrade the global execution time, even if the memory transaction are serialized on one core.

VIII. Discussion

In this section we discuss the results of our study and indicate open questions or potential future work that may extend this work.

Among our results, the first point we highlight is the interest of the 2-Phased model. This solution is easy to implement on any processor that provides multi-banked shared memory. The question raised by this is why the use of these 2-Phased method is so scarce, even if it appears to be efficient in providing good execution time. We see here two main reasons. The code may be intrinsically 3-Phased and the 2-Phased implementation would require changes in the way the code is generated. Similarly, in the real-time community, the theory usually works with 3 phases independently of the target processor. In case of the MPPA2 processor, our study shows that even for SCADE code, **a 2-Phased implementation is more efficient.**

A second point is about the interferences. We observed that **the potential of interference is quite low** in our study, so why are they generally avoided instead of analyzed? Likely, the answer is the hypothesis of a timing compositional processor[5]. With this hypothesis, the additional cost due to interference may be added with a guaranteed final estimated bound. The MPPA2 processor is assumed to be timing compositional [39]. Unfortunately, for this processor and all other industrial ones, there exists yet no proof of such compositionality. This leads to an important open question: is it possible to write such a proof and guarantee that the compositionality is ensured? A formal proof would be largely beneficial for certification and thus industrial use of the interference analysis.

What we observed about SCADE intrinsically phased execution is that enforcing a sequence of REW for each task may have a cost and it is better to introduce idle slots between phases or memory transactions to improve the global response time. Furthermore, the initial mapping and scheduling may not be optimal in some cases, as we observed for our simple program (see Figure 5) where node N_3 could be scheduled before node N_2 without losing any precedence constraints nor functional property.

The Memory-Centric implementation is beneficial mainly when an external memory with longer memory transactions is used. However, we included it in our study as it seems as a good solution to separate memory phases and execute phases. Even though this seems to be a reasonable argument for predictability, our experiments shows that a good mapping of a 2- or 3-Phased implementation delivers better efficiency even in isolation.

Our work may be generalized out of the SCADE context. As you can see in our workflow from Figure 11, it may be applied to any data-flow application, as Simulink ones for example. The minimal initial information for our method is a data-flow graph, a definition of the communication data (the structure size and which task access what) and an initial mapping/scheduling. For the initial scheduling/mapping, we could use any state-of-the-art method if it is not supplied with the code. Finally, a difference with minor impact is that the generated `C` code from SCADE is not identical to the generated `C` code by other data-flow languages, which may require modifications on the generated orchestration code.

The Kalray MPPA2 processor appears as a friendly target for time-critical systems, and in particular for PREM implementation, as the multi-banked memory of a compute cluster is large enough for the applications studied. Using more than one cluster is possible and has been exploited in [40]. However, it introduces additional latency for NoC data transfers and leads to longer memory transactions. Inside a cluster, the memory transactions are similar to the ones exploited in our study, but each inter-cluster data transfer may enlarge the memory transaction and lead to different results. It may be interesting to use a large external memory. With this configuration, further work is needed as the memory phases may last significantly longer than the execute phases, unlike in our study.

Another approach to improve the methodology presented here is to incorporate runtime adaptation of the generated time-triggered schedule, as proposed in [41]. Such mechanism can reduce the pessimism introduced by computing bound WCRT looking at the Actual Execution Time (AET) of tasks, regardless of their phases.

To conclude, on our experiments we have shown that, in a timing-compositional architecture and for the applications we have studied, the 2-phased model combined with analyzed interference yields the shortest end-to-end WCRT that guarantees a time-predictable execution.

[5]No timing anomaly or with bounded effects such that any delay may be added without any loss of a guaranteed global bound.

Acknowledgment

This work was performed in the scope of the ES3CAP research project, under the Bpifrance Invest for the Future Program (Programme d'Investissements d'Avenir – PIA).

References

[1] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, and R. Wilhelm, "Predictability considerations in the design of multi-core embedded systems," in *Proceedings of Embedded Real Time Software and Systems*, May 2010.

[2] 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," in *RTNS*. ACM, 2016, pp. 67–76.

[3] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A Predictable Execution Model for COTS-Based Embedded Systems," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011, pp. 269–279.

[4] J.-L. Colaço, B. Pagano, and M. Pouzet, "Scade 6: A formal language for embedded critical software development," in *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE, 2017, pp. 1–11.

[5] G. Durrieu, M. Faugère, S. Girbal, D. Gracia Pérez, C. Pagetti, and W. Puffitsch, "Predictable Flight Management System Implementation on a Multicore Processor," in *Embedded Real Time Software (ERTS'14)*, TOULOUSE, France, Feb. 2014. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01121700

[6] S. Skalistis and A. Simalatsar, "Worst-case execution time analysis for many-core architectures with noc," in *International Conference on Formal Modeling and Analysis of Timed Systems*, 08 2016, pp. 211–227.

[7] B. Rouxel, S. Skalistis, S. Derrien, and I. Puaut, "Hiding communication delays in contention-free execution for spm-based multi-core architectures," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Leibniz International Proceedings in Informatics, 2019.

[8] Q. Perret, P. Maurere, E. Noulard, C. Pagetti, P. Sainrat, and B. Triquet, "Predictable composition of memory accesses on many-core processors," in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.

[9] S. Skalistis and A. Simalatsar, "Near-optimal deployment of dataflow applications on many-core platforms with real-time guarantees," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2017, pp. 752–757.

[10] S. Wasly and R. Pellizzoni, "Hiding memory latency using fixed priority scheduling," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014, pp. 75–86.

[11] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo, "Memory-Processor Co-Scheduling in Fixed Priority Systems," in *Proceedings of the International Conference on Real-Time Networks and Systems (RTNS)*, 2015, pp. 87–96.

[12] A. Graillat, M. Moy, P. Raymond, and B. D. De Dinechin, "Parallel code generation of synchronous programs for a many-core architecture," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1139–1142.

[13] J. Kim, M. Yoon, R. Bradford, and L. Sha, "Integrated Modular Avionics (IMA) partition scheduling with conflict-free I/O for multicore avionics systems," *Proceedings - International Computer Software and Applications Conference*, pp. 321–331, 2014.

[14] J. M. Rivas, J. Goossens, X. Poczekajlo, and A. Paolillo, "Implementation of memory centric scheduling for cots multi-core real-time systems," in *31st Euromicro Conference on Real-Time Systems*. Leibniz International Proceedings in Informatics, 2019.

[15] A. Alhammad and R. Pellizzoni, "Schedulability analysis of global memory-predictable scheduling," in *Proceedings of the IEEE & ACM International Conference on Embedded Software (EMSOFT)*, Oct 2014, pp. 1–10.

[16] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo, "Memory-centric scheduling for multicore hard real-time systems," *Real-Time Systems*, vol. 48, no. 6, pp. 681–715, nov 2012. [Online]. Available: http://link.springer.com/10.1007/s11241-012-9158-9

[17] C. Pagetti, J. Forget, H. Falk, D. Oehlert, and A. Luppold, "Automated generation of time-predictable executables on multicore," in *Proceedings of the International Conference on Real-Time Networks and Systems (RTNS)*. ACM, 2018, pp. 104–113.

[18] B. Rouxel, S. Derrien, and I. Puaut, "Tightening contention delays while scheduling parallel applications on multi-core architectures," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, p. 164, 2017.

[19] M. Becker, D. Dasari, B. Nicolic, B. Åkesson, V. Nélis, and T. Nolte, "Contention-Free Execution of Automotive Applications on a Clustered Many-Core Platform," in *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, 2016, pp. 14–24.

[20] M. R. Soliman and R. Pellizzoni, "Prem-based optimal task segmentation under fixed priority scheduling," in *2019 31st Euromicro Conference on Real-Time Systems*, 2019, pp. 1–24.

[21] G. Giannopoulou, K. Lampka, N. Stoimenov, and L. Thiele, "Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems," in *Proceedings of the tenth ACM international conference on Embedded software*. ACM, 2012, pp. 63–72.

[22] 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*, vol. 50, no. 5, pp. 736–773, 2014. [Online]. Available: https://doi.org/10.1007/s11241-014-9211-y

[23] M. Becker, S. Mubeen, D. Dasari, M. Behnam, and T. Nolte, "Scheduling multi-rate real-time applications on clustered many-core architectures with memory constraints," in *Proceedings of the 23rd Asia and South Pacific Design Automation Conference*. IEEE Press, 2018, pp. 560–567.

[24] G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo, "Global Real-Time Memory-Centric Scheduling for Multicore Systems," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2739–2751, sep 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7328709/

[25] T. Carle, D. Potop-Butucaru, Y. Sorel, and D. Lesens, "From Dataflow Specification to Multiprocessor Partitioned Time-triggered Real-time Implementation," *Leibniz Transactions on Embedded Systems (LITES)*, vol. 2, no. 2, pp. 01:1—-01:30, 2015.

[26] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele, "Scheduling of mixed-criticality applications on resource-sharing multicore systems," in *Proceedings of the IEEE & ACM International Conference on Embedded Software (EMSOFT)*, Sept 2013, pp. 1–15.

[27] 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 *Proceedings of the International Conference on Real-Time Networks and Systems (RTNS)*, 2015, pp. 129–138. [Online]. Available: http://doi.acm.org/10.1145/2834848.2834862

[28] B. Forsberg, M. Mattheeuws, A. Kurth, A. Marongiu, and L. Benini, "A synergistic approach to predictable compilation and scheduling on commodity multi-cores," in *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2020, pp. 108–118.

[29] P. Pazzaglia, A. Biondi, and M. Di Natale, "Optimizing the functional deployment on multicore platforms with logical execution time," in *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2019, pp. 207–219.

[30] S. Bak, G. Yao, R. Pellizzoni, and M. Caccamo, "Memory-Aware Scheduling of Multicore Task Sets for Real-Time Systems," in *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, aug 2012, pp. 300–309. [Online]. Available: http://ieeexplore.ieee.org/document/6300162/

[31] A. Alhammad and R. Pellizzoni, "Time-predictable execution of multithreaded applications on multicore systems," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2014, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6800243

[32] M. Bertogna, M. Cirinei, and G. Lipari, "Schedulability analysis of global scheduling algorithms on multiprocessor platforms," *IEEE Transactions on parallel and distributed systems*, vol. 20, no. 4, pp. 553–566, 2008.

[33] J. Goossens, S. Funk, and S. Baruah, "Priority-driven scheduling of periodic task systems on multiprocessors," *Real-time systems*, vol. 25, no. 2-3, pp. 187–205, 2003.

[34] M. D. de Dinechin, M. Schuh, M. Moy, and C. Maïza, "Scaling up the memory interference analysis for hard real-time many-core systems," in *Design, Automation and Test in Europe Conference (DATE)*, 2020.

[35] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "Otawa: an open toolbox for adaptive wcet analysis," in *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer, 2010, pp. 35–46.

[36] D. Hardy, B. Rouxel, and I. Puaut, "The heptane static worst-case execution time estimation tool," in *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[37] C. Ferdinand and R. Heckmann, "ait: Worst-case execution time prediction by static program analysis," in *Building the Information Society*. Springer, 2004, pp. 377–383.

[38] C. Pagetti, D. Saussie, R. Gratia, E. Noulard, and P. Siron, "The ROSACE case study: From simulink specification to multi/many-core execution," in *Real-Time and Embedded Technology and Applications Symposium, 2014 IEEE 20th*, ser. RTAS 2014, 2014, pp. 309–318.

[39] S. Saidi, R. Ernst, S. Uhrig, H. Theiling, and B. D. de Dinechin, "The shift to multicores in real-time and safety-critical systems," in *2015 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2015, Amsterdam, Netherlands, October 4-9, 2015*, G. Nicolescu and A. Gerstlauer, Eds. IEEE, 2015, pp. 220–229.

[40] A. Graillat, C. Maiza, M. Moy, P. Raymond, and B. D. de Dinechin, "Response time analysis of dataflow applications on a many-core processor with shared-memory and network-on-chip," in *Proceedings of the 27th International Conference on Real-Time Networks and Systems*, 2019, pp. 61–69.

[41] S. Skalistis and A. Kritikakou, "Timely fine-grained interference-sensitive run-time adaptation of time-triggered schedules," in *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2019, pp. 233–245.