Hiding Communication Delays in Contention-Free Execution for SPM-Based Multi-Core Architectures

Benjamin Rouxel, Stefanos Skalistis, Steven Derrien, Isabelle Puaut

Hiding Communication Delays in Contention-Free Execution for SPM-Based Multi-Core Architectures

Benjamin Rouxel
Univ Rennes, Inria, CNRS, IRISA, France
benjamin.rouxel@irisa.fr

Stefanos Skalistis
Univ Rennes, Inria, CNRS, IRISA, France
stefanos.skalistis@irisa.fr

Steven Derrien
Univ Rennes, Inria, CNRS, IRISA, France
steven.derrien@irisa.fr

Isabelle Puaut
Univ Rennes, Inria, CNRS, IRISA, France
isabelle.puaut@irisa.fr

Abstract

Multi-core systems using ScratchPad Memories (SPMs) are attractive architectures for executing time-critical embedded applications, because they provide both predictability and performance. In this paper, we propose a scheduling technique that jointly selects SPM contents off-line, in such a way that the cost of SPM loading/unloading is hidden. Communications are fragmented to augment hiding possibilities. Experimental results show the effectiveness of the proposed technique on streaming applications and synthetic task-graphs. The overlapping of communications with computations allows the length of generated schedules to be reduced by 4% on average on streaming applications, with a maximum of 16%, and by 8% on average for synthetic task graphs. We further show on a case study that generated schedules can be implemented with low overhead on a predictable multi-core architecture (Kalray MPPA).

2012 ACM Subject Classification Computer systems organization → Embedded and cyber-physical systems; Computer systems organization → Real-time systems

Keywords and phrases Real-time Systems, Contention-Free Scheduling, SPM multi-core architecture

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.25

Funding This work was partially supported by ARGO (`http://www.argo-project.eu/`), funded by the European Commission under Horizon 2020 Research and Innovation Action, Grant Agreement Number 688131.

1 Introduction

The race for computer performance has always been limited by the memory bottleneck. To overcome this issue, hardware [28], software [23] and hybrid [20] prefetching methods have been proposed in the past to bring data closer to the processor before it is needed. However, most prefetchers are not designed for time-critical applications, where predictability is essential.

Compared to cache-based architectures, multi-cores with a private ScratchPad Memory (SPM) per core are a very attractive alternative for time-critical embedded applications. Via software-managed SPMs, they offer sufficient computational power and the necessary predictability. Software-managed SPMs enable data-movement decisions, from/to main

memory, to be scheduled at design time (off-line), thus restricting or avoiding contention on shared resources. Examples of such architectures are the Cell multi-core architecture [19], the academic core Patmos [32] or the Kalray MPPA [11].

Efficient and predictable management of SPMs are facilitated by application models that offer a high-level view of parallel programs. We focus on applications modelled as directed acyclic task-graphs (DAGs), consisting of dependent tasks that exchange data through shared FIFO channels. In such application models, tasks are executed in three phases: 1) they *read* data from their input FIFOs, 2) *execute* their computation, and 3) *write* the results to their output FIFOs. This order of execution is in accordance with the PRedictable Execution Model (PREM) [29, 2] and the Acquisition Execution Restitution (AER) execution model [26]. Such execution models are well-suited for SPM-based architectures, as tasks can prefetch their input FIFOs from the shared memory into the private SPM and, after the task's execution, write-back the produced data to their output FIFOs. Using proper scheduling techniques, this can result in contention-free execution. These DAGs do not necessarily need to be built from scratch, which would require an important engineering effort. Automatic extraction of parallelism, for instance from a high level description of applications in model based design workflows [12], seems a much more promizing direction.

We believe that this combination of software (DAG with PREM) and hardware (SPM-based multi-cores) is essential to build efficient and predictable systems. In this paper, we propose a scheduling strategy that hides such delays by executing communications in parallel with computation. Our scheduling strategy relies in advancing (resp. postponing) the execution of read (resp. write) phase of a task such that it overlaps with the execution phase of another task, thus hiding the communication delay. The proposed scheduling strategy aims at minimizing the makespan of the total execution and includes an SPM allocation strategy ensuring that there is enough space in SPM at all times. The resulting schedules are contention-free to the shared bus, similarly to [3]. Additionally, and in comparison with most related works (such as [30, 8, 24, 37]), we fragment communication phases to augment communication hiding possibilities. In contrast with most other works dealing with SPM, e.g. [13, 4], that allow some information to stay in global main memory, our SPM allocation scheme imposes that *all* information accessed by a task is prefetched into SPM beforehand. In summary, the contributions of this work are the following:

1. We propose a strategy to map and schedule a task graph onto cores coupled with an SPM allocation scheme. The generated static contention-free non-preemptive schedules allow, when possible, to overlap communications and computations, through *non-blocking* loading/unloading of information into/from SPM. Communication phases are *fragmented* to maximize the duration of overlapping between communications and computations. The proposed strategy is formulated as a heuristic based on list-scheduling to produce schedules very fast.
2. We provide an experimental evaluation showing our method improves the overall makespan, up to 16%, compared to equivalent schedules generated with blocking communications.
3. We evaluate the impact of different granularities for communication fragments on the schedule makespan.
4. We experimentally show on a use case that generated schedules can be implemented with a low overhead on a predictable multi-core architecture (Kalray MPPA [11]).

The rest of this paper details the proposed strategy and is organized as follows. A motivating example is presented in Section 2, as well as the assumptions made on the hardware and software. Then, Section 3 presents the basic principles of the SPM allocation scheme. The scheduling/mapping/allocation heuristic technique is then detailed in Section 4. Section 5 presents experimental results, including an implementation on the Kalray MPPA platform. Finally, Section 6 presents related works, before concluding in Section 7.

2 Motivating Example and assumptions

2.1 Architecture Model

We consider multi-core architectures, where every core has access to a private dual-ported ScratchPad memory (SPM). Cores are connected through an arbitrated bus to a global external shared memory. Access requests are enqueued (one queue per core) and served according to the bus arbitration policy. While in the rest of the paper, we will assume a FAIR-round-robin arbitration [21], the proposed method is directly applicable for other policies with arbitration based on requests (e.g. first-come-first-served, fixed priority, etc.) and not on time (e.g. time division multiple accesses). All communications are non-preemptable and go through the shared global memory (no SPM to SPM communication). We further assume that the architecture supports loading of information in the dual-ported SPM, in parallel with computations. Provided support may be a hardware Direct Memory Access (DMA) engine or a specific core acting as a DMA software engine as in [9]. These assumptions are met in both academic and commercial processors (e.g. Patmos [32], Kalray MPPA [11]).

Communications can be implemented in *blocking* mode or *non-blocking* mode. In *blocking* mode, the CPU is in charge of transfers between SPM and the shared memory, and is then stalled during every transfer. In *non-blocking* mode, transfers are managed asynchronously, allowing the CPU to execute other jobs during memory transfers.

2.2 Application Model

We consider applications modeled as directed acyclic graphs (DAGs). A graph G is a pair (V, E) where the vertices in V represent the application's tasks and the edges in E represents the data dependencies between the tasks. This work supports multiple DAGs with same period as is, which is omitted due to space limitations. Extending our work to applications with different periods is deemed as a rather direct transposition, by making schedule generation operate on the hyperperiod. This extension is however left for future work.

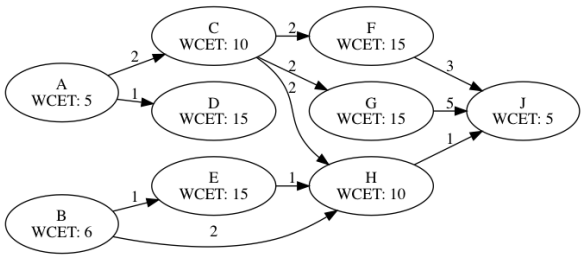


Figure 1 An example of a task-graph.

According to the semantics defined in PREM [29, 1] or AER [26], each task is divided in three phases, namely *read*, *exec* and *write*. The *read* phase reads/receives the mandatory code and/or data from the main memory to the SPM, such that the *exec* phase can proceed without access to the shared bus. Finally, the *write* phase writes the resulting data back from the SPM to the main memory. Using such an application and execution model is central in our method, as it allows to perform offline scheduling which precisely controls resource contention. The *exec* phase of tasks does not access the shared bus, and thus contentions when accessing the shared bus do not exist between *exec* phases and *read/write* phases; the off-line scheduler is in charge of scheduling communication phases in such a way that they

do not conflict with one another; finally, the presence of a dual-ported SPM per core allows calculations and communications to proceed in parallel, provided that they access different address ranges.

Note that considered DAGs with *read-exec-write* semantics need not be built from scratch. They can be extracted automatically either from a high-level description of applications in model based design workflows [12], or from legacy code with [27].

As an extension to the original PREM/AER model, we split each communication into *fragments*. A *fragment* is some division of the total amount of data that a task produces or consumes. How the data are divided into fragments is determined by the fragmentation scheme. The default fragmentation scheme assumed throughout this paper is to have one fragment for each task communication (edges in the graph). Thus, instead of a task reading/writing all of its inputs/outputs at once, it is done on a per-task basis with the size of the fragment being as the size of the communication. Other fragmentation strategies will be detailed in Section 5.4. A task τ_i is a tuple $\tau_i =< F_i^r, \tau_i^e, F_i^w >$, where τ_i^e is the *exec* phase, and F_i^r (resp. F_i^w) is the set of fragments read (resp. written) by the task. The f-th fragment of τ_i that is read (resp. written) is denoted as $\tau_{(i,f)}^r \in F_i^r$ (resp. $\tau_{(i,f)}^w \in F_i^w$).

An example of a task-graph is illustrated in Figure 1. The figure gives for each task its name, the Worst Case Execution Time (WCET) of its *exec* phase, and for each edge the amount of data exchanged, among the tasks, in bytes. The WCET of the *exec* phase, denoted C_i, can be estimated in isolation from the other tasks considering a single-core architecture, as there is no access to the main memory (all the required data and code have been loaded into the SPM before the task's execution). In general, *read* and *write* fragments could suffer from contentions caused by concurrent accesses to the shared bus, however in this paper the proposed technique produces contention-free schedules.

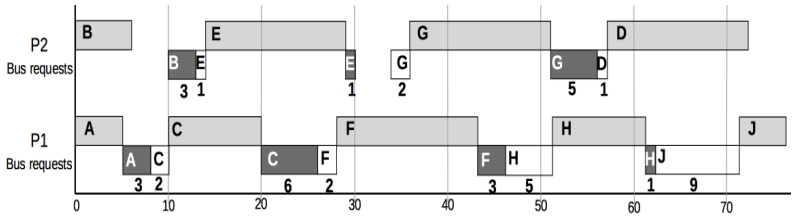
Since the code in our experimental evaluation, is generally small and likely to be reused along the execution of the application, for simplicity reasons we assume that the code is preloaded in the SPM at startup.

For simplicity when presenting the motivational example, we will assume the SPM to be large enough to store all information (code, data, communication buffers), this assumption will be relaxed in Section 3.

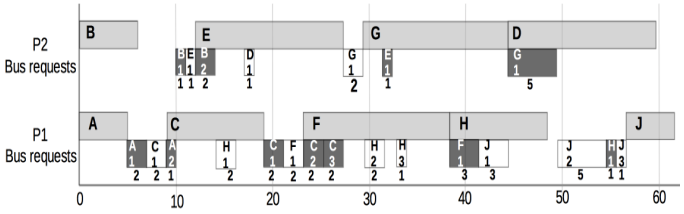
2.3 Motivating Example

Figure 2 motivates the use of *non-blocking, fragmented* communications for the application from Figure 1 assuming a dual-core architecture. Sub-figure 2b depicts the schedule obtained using non-blocking fragmented communications, with one fragment per outgoing edge in the graph, whereas sub-figure 2a depicts the schedule obtained using blocking communications. For each core, the top time-line depicts the scheduling of *exec* phases (grey boxes) and the bottom one depicts the scheduling of communications (*read*: white boxes, black font, *write*: dark boxes, white font). The communication cost is indicated below each communication phase.

In Figure 2a (*blocking* mode), all parts of the same task are scheduled contiguously on the same core, and the CPU is stalled when accessing the bus. The *read* and *write* phases are not fragmented as it would not bring any benefit in blocking mode. Precedence constraints are respected by ordering *read* phases after their preceding *write* phases, e.g. τ_C^r is scheduled after the completion of τ_A^w. There is no *read* phase for tasks A and B as they do not have predecessors, hence no data to fetch. The resulting schedule makespan (time at which the last task ends) is 76 time units. In Figure 2b (*non-blocking* mode), fragmented communication and *exec* phases overlap, e.g. $\tau_{(H,2)}^r$ and $\tau_{(H,3)}^r$ overlap with τ_F^e, thus hiding the communication delay. Having prefetched all required data into SPM, the *exec* phase of τ_H can start right after τ_F.



(a) Schedule in *blocking* mode (makespan of 76 time units).



(b) Schedule in *non-blocking* mode (makespan of 61 time units).

Figure 2 Schedules for the example task-graph on a dual-core. For each core, the top time-line depicts the schedule of *exec* phases (in grey), the bottom one depicts the schedule of *read* (in white) and write (in black) phases. The communication cost is indicated below each communication phase.

The gain in schedule length from Figure 2b is obtained by introducing the following flexibilities in the scheduling of communication fragments, while respecting *read-exec-write* phase's order: 1) communication phases of different tasks can have a different order than their respective *exec* phases, as long as there is no data dependencies between them, e.g. $\tau^r_{(D,1)}$ is scheduled before $\tau^w_{(E,1)}$, in reverse order compared to τ^e_D and τ^e_E. 2) communication phases and *exec* phase of the same task, do not need to be contiguous in time, e.g. $\tau^r_{(D,1)}$ and τ^e_D are not. 3) communications are fragmented, a task with multiple successors does not write all its data at once, e.g. τ_B has two successors, thus writing two fragments.

The last point (fragmented communications) is new compared to related work. Considering each fragment individually allows additional overlaps between communications and task execution that were impossible without fragmentation. In the example, it allows to hide part of the *write* phase of τ_A, and part of the *read* phase of τ_J, which was not possible without fragmentation. Thus, splitting communications allows each source/sink of the task graph to hide part of its communications. However, in the example from Figure 2b, the remaining first part A^w ($\tau^w_{(A,1)}$) can still not be hidden, but is however smaller than in Figure 2a. The overall makespan of the resulting schedule in *non-blocking* mode (Figure 2b) is 61 time units, resulting in a gain of 20%.

3 Principle of SPM allocation scheme

In our motivational example, we assumed the SPM large enough to store all information required to execute the entire application (code, data, communication buffers). To account for limited SPM capacity, our scheduling strategy comes with a SPM allocation strategy that allocates an SPM area (called hereafter *region*) to each communication fragment and execution phase. *Fragment-to-region mapping* is performed by the scheduler off-line. However, the same region can be used successively by different fragments, and the scheduler guarantees that the live ranges of the concerned fragments do not overlap. Region sizes vary according to the data stored by fragments/exec phases.

To isolate bus accesses from computation, we impose that *all* information accessed by a task is loaded into SPM beforehand. This comes in opposition to most SPM allocation policies that decide which information should be stored in the SPM and which information should remain in the global main memory (e.g. [13]). Our fragment-to-region mapping is inspired by the method proposed in [22].

The regions assigned to fragments F_i^r contain the input data, fetched from the main memory, which are required by the task's *exec* phase. These regions contain the data produced by all predecessor tasks. The unique region assigned to τ_i^e contains any kind of information used locally by the task (code, constants, local data, usually stack-allocated). The regions assigned to F_i^w contain the data produced by the task.

The size of a region obviously depends on the amount of data required by the associated fragment (i.e. amount of data produced by a predecessor in case of a *read* fragment). Considering a mapping of tasks to cores and a mapping of fragments to SPM regions, the sum of the sizes of regions on a core must not exceed the SPM size.

Let us consider the example of Figure 2b, in which for simplicity we concentrate on the communication fragments and ignore the execution phases. If the size of the SPM is 1 Kbytes then on processor $P2$ the SPM can be partitioned in seven regions $SPM = \{\tau^w_{(B,1)}, \tau^w_{(B,2)}, \tau^r_{(E,1)}, \tau^w_{(E,1)}, \tau^r_{(D,1)}, \tau^r_{(G,1)}, \tau^w_{(G,1)}\}$ with respective sizes in bytes $\{1, 2, 1, 1, 1, 2, 5\}$ (according to the amount of data exchanged between tasks, taken from Figure 1). The sum of the regions' sizes is 13 bytes, which is less than the SPM size. If we now restrict the SPM size to 10 bytes, the previous partitioning of SPM in regions is not valid anymore. However, once $\tau^w_{(B,1)}$ is completed, the data produced by τ_B has been committed to the global shared memory, therefore its assigned region can be reused. In this example, $\tau^r_{(G,1)}$ starts after the completion of $\tau^w_{(B,1)}$, as it is the case for $\tau^w_{(B,2)}$ and $\tau^w_{(G,1)}$. Thus, the fragments $\tau^w_{(B,1)}$,$\tau^r_{(G,1)}$ and $\tau^w_{(B,2)}$,$\tau^w_{(G,1)}$ can be assigned to the same SPM region, leaving a partitioning of five regions: $SPM = \{\{\tau^w_{(B,1)}, \tau^r_{(G,1)}\}, \{\tau^w_{(B,2)}, \tau^w_{(G,1)}\}, \tau^r_{(E,1)}, \tau^w_{(E,1)}, \tau^r_{(D,1)}\}$ with respective sizes (in bytes) $\{max(1,2), max(2,5), 1, 1, 1\}$. The sum of all regions sizes is 10 bytes, which can fit in the SPM.

In the example, both pairs $(\tau^w_{(B,1)}, \tau^r_{(G,1)})$ and $(\tau^w_{(B,2)}, \tau^w_{(G,1)})$ could share the same region, because their lifespan does not overlap. On the other hand, in Figure 2b, $\tau^r_{(D,1)}$ can not share the same region as $\tau^r_{(E,1)}$, because the data consumed by τ_E are in use from the start of the read phase F^r_E up to the end of the execution of τ^e_E. This leads to define the *live range* of regions for each type of fragment. Definition 1 defines the live range for a region assigned to a *read* fragment, while Definitions 2 and 3 give live ranges for regions assigned respectively to an *exec* and a *write* fragment.

▶ **Definition 1.** *Data fetched from the main memory by a read fragment are alive from its start time to the end of the corresponding exec phase.*

▶ **Definition 2.** *Local information used by an exec phase (code, stack data area) are alive for the whole execution time of the application.*

▶ **Definition 3.** *Data written back to main memory by a write fragment are alive from the start time of the corresponding exec phase to its transmission end time.*

We assume read/written data can be consumed/produced at any time in the *exec* phase of the task. Therefore, the live range in Definitions 1 and 3 include the duration of the *exec* phase.

The scheduler maps fragments to regions, but does not decide the addresses of the regions in the SPM, which is left to the compiler/code generator. Since the number and size of regions is decided off-line, address assignment is straightforward, and does cause

external fragmentation. Fragmentation of the SPM can only arise inside regions (*internal* fragmentation) when two (or more) phases are assigned to the same region but store different amounts of data.

4 Joint-mapping/scheduling and SPM allocation

This section presents a heuristic algorithm based on forward list scheduling that integrates *fragmented non-blocking* communication and *SPM allocation*. The main outcome of the proposed algorithm is a static mapping, scheduling and SPM fragment-to-region allocation, for a single application represented as a DAG. The objective is minimizing the overall schedule's makespan. The generated schedule is free from contention. According to the terminology given in [10], the proposed scheduling techniques are partitioned, time-triggered and non-preemptive. Schedule generation operates at *task-level* (as opposed to job-level as defined in [10]).

Heuristics based on forward list scheduling first order input elements (in our specific case *exec* and *communication* phases), then add them one by one in the schedule without backtracking. We experimented with three topological sorting algorithms. The first algorithm is a vanilla Depth First Search (DFS) algorithm to walk-through the task graph. Second, we use the same DFS algorithm but we postpone *read* fragments to avoid too early reading that might delay other fragments in the schedule (further details will be given when describing Algorithm 4.1). The last algorithm is a vanilla Breath First Search (BFS) algorithm. For all three sortings, we used the element memory footprint as tie breaking rule (larger footprint to be scheduled first). Since no sorting algorithm consistently outperforms the others, we generate three schedules, each resulting from one sorting algorithm, and selected the one resulting in the shortest schedule makespan as the heuristic's solution.

4.1 Notations and assumptions

Table 1 summarizes the notations that will be used to describe the scheduling algorithm (sets, utility functions and constants).

Calculation of constants $DELAY^{r}_{(i,f)}$ and $DELAY^{w}_{(i,f)}$ requires knowledge of the bus arbitration strategy and of concurrent accesses to the bus. The considered bus is characterized by a maximum duration of T_{slot} allocated to each core in a round-robin fashion, with a writing rate of D_{slot} data word per time unit. T_{slot} defines the duration a core is granted the bus, and D_{slot} defines the amount of data transmittable in a T_{slot} duration. For the scope of this paper, we generate contention-free schedules, thus no contention delay is paid, and the duration of a data transfer of d bytes is trivially calculated by equation (1). This equation could be refined to account for DRAM access cost, as done in [22].

$$delay = \lceil d/D_{slot} \rceil \cdot T_{slot} \tag{1}$$

In the description of the scheduling algorithm, the cost for setting up non-blocking memory transfers (DMA initialization in case of a hardware DMA engine) will not appear explicitly and is considered included in the WCET of the *exec phase*. Determination of this cost will be described in Section 5.5.

4.2 Scheduling algorithm

The scheduling algorithm is sketched in Algorithm 4.1. It uses the task graph as input, sorts the elements to schedule (*exec* phases and *communication* fragments) to create the list (line 2). Then a loop iterates on each element while there exists elements to schedule

Table 1 Notations.

Sets	T	set of tasks
	P	set of processors/cores
	R	set of regions
	F_i^r, F_i^w	sets of τ_i fragments
	$F = F_i^r \cup F_i^w, \forall i \in T$	sets of all fragments from all tasks in T
Funcs	$i = task(f)$	utility to retrieve the task of a fragment, fragment f belongs to τ_i
	$(j,q) \in pred((i,f))$	(j,q) means $\tau^X_{(j,q)}$ is a direct predecessor of $\tau^X_{(i,f)}$
Constants	SS_i	local (stack) data size of τ_i^e
	CS_i	code size of τ_i^e
	C_i	τ_i execute phase WCET computed in isolation as stated in Section 2
	$D^r_{(i,f)}, D^w_{(i,f)}$	size in bytes of $\tau^r_{(i,f)}$
	$DELAY^r_{(i,f)}$ $DELAY^w_{(i,f)}$	fragment f of τ_i, *read/write* latency from Equation (1)
	$SPMSIZE_c$	SPM size of core c
Variables	$\rho^r_{(i,p)}$, ρ^e_i, $\rho^w_{(i,q)}$	start times of $\tau^r_{(i,p)}$, τ^e_i and $\tau^w_{(i,q)}$

(lines 5-20). This heuristic uses an *As Soon As Possible* (ASAP) strategy when mapping an element. If the element to schedule is a communication fragment (line 8), then there is no need to map it on a core, but it still must be scheduled to avoid interference. If it is an *exec* phase, then a core is selected and the mapping with the shorter the makespan is selected (line 15).

SPM regions can be assigned to elements (*exec* phases and *communication* fragments) only when all of its phases are properly scheduled and mapped to a core (lines 18-20). When scheduling the read fragments, the core mapping information is not yet available. Additionally, when mapping the *exec* phase, we still do not have the information regarding the *write* fragments that have not been scheduled yet. While assigning the region (lines 18-20), the *exec* phase goes first then the communication phases. This order is motivated to better handle resident code in SPM and avoid SPM space to be stolen by communication fragments. For example, if there are 5 units of free space (not assigned yet) and the *exec* needs 5 units while a *read/write* need 2 units each. Then the task can still be mapped. The *exec* phase will take the remaining free space, while the communication fragments can share an already created, but available (in time), region (see Definitions 1 and 3).

Scheduling an element

Algorithm 4.2 sketches the method to determine the start time of the considered element (*exec* phase or *communication* fragment). First, each element must start after its causal predecessors (line 2) Then, lines 3-9 enforce that no *exec* phases overlap on the same core and no fragments overlap on the bus. Condition at line 4 enforces the type of *cur_elt* and e to be identical, and if both are exec phases then they must be mapped on the same core. Finally, line 9 postpones *cur_elt* start time if overlapping with e.

Algorithm 4.1: Scheduling algorithm.

```
   Input  : A task graph G and a set of processors
   Output : A schedule
1  Function ListSchedule(G = (T, E), P)
2      Qready ← Topological_Sort_Elements(G)
3      Qdone ← ∅
4      schedule ← ∅
5      while elt ∈ Qready do
6          Qdone ← Qdone ∪ {elt}
7          Qready ← Qready\{elt}
           /* tmpSched contain the best schedule for the current task   */
8          if elt is a read fragment ∨ elt is a write fragment then
9              Schedule_Element(Qdone, elt, null)
10         else if elt is an exec phase then
11             tmpSched ← ∅ with makespan = ∞
12             foreach p ∈ P do
13                 copy ← schedule
                   /* Set τ^e in copy on p the earliest in the schedule */
14                 Schedule_Element(Qdone, elt, p)
15                 tmpSched ← min_makespan(tmpSched, copy)
16             schedule ← tmpSched
17         if all fragments and exec phase of τ_i containing elt are in Qdone then
18             Assign_Region(schedule, Qdone, τ_i^e, SS_t + CS_t, 0, infinity)
19             ∀f ∈ F_i^r, Assign_Region(schedule, Qdone, f, D^r_(i,f), ρ^r_(i,f), ρ^e_i + C_i)
20             ∀f ∈ F_i^w, Assign_Region(schedule, Qdone, f, ρ^e_i, ρ^w_(i,f) + DELAY^w_(i,f))
21     return schedule
```

Allocation of SPM regions

Algorithm 4.3 associates a SPM region to an element (*exec* phase, fragment). If there is data to store in the SPM (line 2), then it first tries to reuse an existing region (lines 4-6), thus minimizing the required memory size. If no existing region can be shared, then a new one is created (lines 7-8). Sharing a region imposes that the selected region is big enough to handle the current amount of data and free for use at the required time interval (line 4).

5 Experimental evaluation

The first presented experiments (Section 5.2) aim at validating the quality of the proposed scheduling technique as compared to a scheduling strategy based on Integer Linear Programming (ILP, see Section 5.1) that provides the optimal solution (shortest schedule makespan). Then, we validate the benefits of hiding communications using the heuristic technique (Section 5.3). In the above-mentioned experiments, the default fragmentation strategy (one fragment per edge in the task graph) is used. We subsequently compare different ways to fragment communications (Section 5.4). Finally, we show in Section 5.5 on a case study that generated schedules can be implemented with low overhead on a Kalray MPPA platform [11]. In Sections 5.2 to 5.4, scheduler and communication implementation overheads are neglected, but they are considered in Section 5.5.

Experiments have been conducted both on real code, in the form of the open-source Refactored StreamIT benchmark suite STR2RTS [31] and on synthetic task graphs, generated using Task-Graph For Free (TGFF) [14].

Algorithm 4.2: Scheduling of an element (exec, fragment).

```
Input  : the list of scheduled element, the current element to schedule, the current core or
         null if the element is a fragment
Output :
1  Function Schedule_Element(Qdone, cur_elt, cur_proc)
       /* wct → Worst-Case Timing, DELAY_β^α or C_β                                  */
       /* X and Y depend on the type of the corresponding element                    */
2      ρ^X_{cur_elt} ← max_{p∈pred(cur_elt)}(ρ^Y_p + wct_p)
3      foreach e ∈ Qdone do
4          if cur_elt is a fragment and e is not a fragment
5          ∨ cur_elt is an exec phase and e is not an exec phase
6          ∨ cur_elt is an exec phase and e is not mapped on core cur_proc then
7              continue
8          if e overlaps in time with cur_elt then
9              ρ^X_{cur_elt} ← ρ^Y_e + wct_e
```

Algorithm 4.3: Allocation of a SPM region to a phase.

```
Input  : A schedule, the list of scheduled element, the current task and properties of the
         phase to map on a region
Output : A schedule
1  Function Assign_Region(schedule, Qdone, cur_elt, dataSize, start, end)
2      if data == 0 then return
3      proc ← getCore(schedule, cur_elt)
       /* Get the set of existing regions on core proc where : size ≥ dataSize ∧
          last reservation time ends before start                                    */
4      existing ← getExistingRegions(schedule, proc, dataSize, start)
5      if existing ≠ ∅ then
6          Assign the smallest existing region to cur_elt
7      else if free SPM size in proc ≥ dataSize then
           /* Create SPM region for cur_elt on proc with size data where the
              reservation time is [start; end]                                       */
8          CreateRegion(cur_elt, proc, dataSize, start, end)
9      else
10         Throw Unschedulable
```

The STR2RTS applications[1] are modeled using fork-join graphs and come with timing estimates for each task and amount of data exchanged between them. We did not use all the benchmarks and applications provided in the suite as some are not parallel, they are made of a linear chain of tasks (i.e. CFAR, FIR, ComplexFIR, FTT6), making them uninteresting for multi-core platforms. This leaves us 18 benchmarks with 73 tasks in average and average memory footprint of 4 KB.

The synthetic task-graphs were generated with the latest version of the TGFF generation software. Generated task-graphs include chains of tasks with different lengths and widths, fork-join graphs and more evolved structures (e.g. multi-DAGs). The resulting task graph characteristics are presented in Table 2. The table includes the number of task-graphs, their

[1] A table describing each used benchmark is available in the appendix.

number of tasks, the maximum width of the task-graph, the range of WCET values for each task and the range of amount of exchanged data in bytes between pairs of tasks, the range of code size and stack size for each task, and the global ratio of WCET per amount of exchanged data. The TGFF parameters (average and indicator of variability) are set in such a way that the average values for task WCETs and volume of data exchanged between pairs of tasks correspond to the analogous average values for the STR2RTS benchmarks.

Table 2 Task-graph characteristics for synthetic task-graphs.

#Task-graphs	50
#Tasks	5, 69, 22
Max. width	3, 17,8
Exchanged data	$[0; 192]$

WCET	$[5; 6000[$
Code size	$[3; 3920[$
Local size	$[1; 60]$
Ratio $\frac{WCET}{data}$	10

All reported experiments have been conducted on several nodes from an heterogeneous computing grid with 138 computing nodes (1700 cores). In all experiments, the duration a core is granted the bus (T_{slot}) is set to 3 as in [21] and shown in [30] to have little impact on the schedule length. The transfer rate is one word (4 bytes) per time unit.

5.1 Baseline: Integer Linear Programming scheduling

An Integer Linear Programming (ILP) formulation consists of a set of integer variables, a set of constraints and an objective function. Constraints describe the problem to solve in the form of linear inequalities. Solving a problem consists in finding a valuation for each variable satisfying all constraints with the goal of minimizing/maximizing the objective function. Table 3 summarizes the variables used in the ILP formulation. For a concise presentation of constraints, the two logical operators $\vee, \wedge$ are directly used in the text of constraints. These operators can be transformed into linear constraints in order to properly use ILP solvers using simple transformation rules from [5].

Objective function

The objective is to obtain the shortest schedule, and so to minimize the makespan Θ, Equation (2a). Equation (2b) constrains the completion time of all tasks (starting of all *write* fragment $\rho^w_{(i,f)}$, plus its latency $DELAY^w_{(i,f)}$) to be inferior or equal to the schedule makespan.

$$minimize\ \Theta \tag{2a}$$

$$\forall i \in T; \forall f \in F^w_i; \rho^w_{(i,f)} + DELAY^w_{(i,f)} \leq \Theta \tag{2b}$$

Problem constraints

Some basic rules of a valid schedule are expressed in the following equations. Equation (3a) ensures the unicity of a task mapping ($p_{i,c} = 1$ τ_i is mapped on core c). Equation (3b) defines if two tasks are mapped on the same core ($m_{i,j} = 1$). When $a^{ee}_{i,j} = 1$ then τ^e_i is scheduled before τ^e_j, thus Equation (3d) forbids an order of phases (resp. fragments) and its reversed order to be both active but imposes to choose one; one of the $a^{ee}_{i,j}, a^{ee}_{j,i}$ must be equal to 1, but both can not be equal to 1. Equations (3e) unifies Equations (3b) and (3d) to order *exec* phases only on the same core. In Equation (3d), no equation enforces

Table 3 ILP variables.

Int. variables	Θ	schedule makespan
	$\rho^r_{(i,p)}$, ρ^e_i, $\rho^w_{(i,q)}$	start times of $\tau^r_{(i,p)}$, τ^e_i and $\tau^w_{(i,q)}$
	$spmsr^c_z$	computed size of SPM region z on core c
	$\sigma_{(i,f)}$, σ_i	*spm reservation* start times of $\tau^X_{(i,f)}$, τ^e_i
	$\omega_{(i,f)}$, ω_i	*spm reservation* end times of $\tau^X_{(i,f)}$, τ^e_i
Binary variables	$p_{i,c} = 1$	τ^e_i is mapped on core c
	$m_{i,j} = 1$	τ^e_i & τ^e_j are mapped on the same core
	$a^{ee}_{i,j} = 1$	τ^e_i is scheduled before τ^e_j $(\rho^e_i \leq \rho^e_j)$
	$a^{XY}_{(i,f),(j,g)} = 1$	$\tau^X_{(i,f)}$ is scheduled before $\tau^Y_{(j,g)}$, in the sense $\rho^X_{(i,f)} \leq \rho^Y_{(j,g)}$ $XY \in \{rr, ww, rw, wr\}$
	$am^{ee}_{i,j} = 1$	same as $a^{ee}_{i,j}$ but on the same core
	$am^{XY}_{i,j} = 1$	same as $a^{XY}_{i,j}$ but on the same core $XY \in \{rr, ww, rw, wr\}$
	$spmp_{z,i} = 1$ $spmp_{z,(i,f)} = 1$	τ^e_i is allocated to SPM region z $\tau^X_{(i,f)}$ is allocated to SPM region z
	$spmm_{(i,f),(j,g)} = 1$	$\tau^X_{(i,f)}$ and $\tau^X_{(j,g)}$ are assigned to the same region (similar to $m_{i,j}$)
	$spma_{(i,f),(j,g)} = 1$	$\tau^X_{(i,f)}$ is causally before $\tau^X_{(j,g)}$ (similar to $a_{i,j}$)
	$spmam_{(i,f),(j,g)} = 1$	$\tau^X_{(i,f)}$ is causally before $\tau^X_{(j,g)}$, and both are assigned to the same region (similar to $am_{i,j}$)

to have the same ordering for *exec* phases as for with *read* phases, because the solver does not have to chose an order between them (see Section 2). The same remark applies to *exec* phases and *write* phases.

$\forall (i,j) \in T \times T; XY \in \{rr, ww, rw, wr\}; \forall f \in F^X_i; \forall g \in F^Y_j; i \neq j$

$$\sum_{c \in P} p_{i,c} = 1 \tag{3a}$$

$$m_{i,j} = \sum_{c \in P} (p_{i,c} \wedge p_{j,c}) \quad and \; m_{i,j} = m_{j,i} \tag{3b}$$

$$a^{ee}_{i,j} + a^{ee}_{j,i} = 1 \tag{3c}$$

$$a^{XY}_{(i,f),(j,g)} + a^{XY}_{(j,g),(i,f)} = 1 \tag{3d}$$

$$am^{ee}_{i,j} = a^{ee}_{i,j} \wedge m_{i,j} \tag{3e}$$

$$\rho^e_i + C_i \leq \rho^e_j + \mathcal{M} \times (1 - am^{ee}_{i,j}) \tag{3f}$$

$$\rho^X_{(i,f)} + DELAY^X_{(i,f)} \leq \rho^Y_{(j,g)} + \mathcal{M} \times (1 - a^{XY}_{(i,f),(j,g)}) \tag{3g}$$

Equation (3f) forbids the overlapping of two *exec* phases when mapped on the same core by forcing one to execute after the other. Equation (3g) forbids to have more than one active memory transfer at a time to produce contention-free schedules. Equations (3f) and (3g) must be activated only if the two elements are scheduled in a specific order. Thus, a nullification method is applied by using the classical big-M notation (the big-M notation allows to force a constraint to hold depending on a condition as further explained in [18]). The selected value for the big-M constant is the makespan of a sequential schedule on 1 core, the sum of tasks' WCETs and communication delays, which is the worst scenario that can arise.

Read-exec-write semantics constraints

Equations (4a) and (4b) constrain the order of all phases of a task to be *read* phase, then *exec* phase, then *write* phase. But, these phases will not necessarily be scheduled contiguously. The start date of τ_i^e (ρ_i^e) must be some time after the completion of all *read* fragments (start of *read* fragment $\rho^r_{(i,f)}$ + latency $DELAY^r_{(i,f)}$). Similarly, each *write* fragment starts ($\rho^w_{(i,f)}$) some time after the end of the *exec* phase (start of *exec* phase ρ_i^e + WCET C_i).

$\forall i \in T$,

$$\forall f \in F_i^r, \rho_i^e \geq \rho^r_{(i,f)} + DELAY^r_{(i,f)} \tag{4a}$$

$$\forall f \in F_i^w, \rho^w_{(i,f)} \geq \rho_i^e + C_i \tag{4b}$$

Data dependencies in the task-graph

Equation (5) enforces data dependencies by constraining all *read* fragments to start after the completion of all their respective predecessors. For a *read* fragment its predecessor is the *write* fragment of the task that produced the corresponding data.

$$\forall i \in T, \forall f \in F_i^r, \forall (j,g) \in pred(i,f) \quad \rho^w_{(j,g)} + DELAY^w_{(j,g)} \leq \rho^r_{(i,f)} \tag{5}$$

Assigning SPM regions

Equations (6a) & (6b) force every *element* (exec phase and fragments) from τ_i to be mapped on one and only one region z. Identically to [22], we initially consider the number of regions to be equal to the number of elements (number of *exec phase* + number of fragments). With the limited capacity of the SPM, the solver will then be able to minimize the number of effectively used regions.

$$\forall i \in T; \quad \sum_{z \in R} spmp_{z,i} = 1 \tag{6a}$$

$$f \in F, i = task(f); \quad \sum_{z \in R} spmp_{z,(i,f)} = 1 \tag{6b}$$

Equations (7a) and (7b) set the size ($spmsr_z^c$) of region z on core c to be the largest amount of data that will be stored in it. The data stored by an *exec* phase includes the code size (CS_t) and local data (SS_t, stack data). The data stored by a *read* or *write* fragment ($D^X_{(i,f)}$) includes all data consumed (or produced) by a task from one predecessor (or one successor). To store data into a given region of a core, both mapping variables for the region $spmp_{z,(i,f)}$ and the core $p_{i,c}$ must be set to 1.

$\forall c \in P, \forall z \in R, \forall i \in T$,

$$spmsr_z^c \geq \ (SS_i + CS_i) \ (spmp_{z,i} \wedge p_{i,c}) \tag{7a}$$

$$\forall \chi \in \{r,w\}, \forall f \in F_i^\chi; spmsr_z^c \geq \ D^\chi_{(i,f)} \ (spmp_{z,(i,f)} \wedge p_{i,c}) \tag{7b}$$

Equation (8) limits the sum of size for each region for a core to the available SPM size.

$$\forall c \in P, \quad \sum_{z \in R} spmsr_z^c \leq SPMSIZE_c \tag{8}$$

Delimiting the usage time of a region by an element relies on Definitions 1, 2 and 3. Equation (9a) sets the allocation start time $\sigma_{(i,f)}$ of $\tau^r_{(i,f)}$ to be equal to its schedule start time and the allocation end time $\omega_{(i,f)}$ to be the end of the corresponding *exec* phase. Equation (9b) forces the lifetime of the region used by the *exec* phase to be the whole duration of the schedule (recall that Θ represents the overall makespan). Equation (9c) sets the allocation start time $\sigma_{(i,f)}$ of $\tau^w_{(i,f)}$ equal to the beginning of the *exec* phase and the allocation end time $\omega_{(i,f)}$ equal to its start time.

$\forall i \in T$

$$\forall f \in F^r_i; \quad \sigma_{(i,f)} = \rho^r_{(i,f)} \quad and \quad \omega_i = \rho^e_i + C_i \tag{9a}$$

$$\sigma_i = 0 \quad and \quad \omega_i = \Theta \tag{9b}$$

$$\forall f \in F^w_i; \quad \sigma_{(i,f)} = \rho^e_i \quad and \quad \omega_{(i,f)} = \rho^w_{(i,f)} + DELAY^w_{(i,f)} \tag{9c}$$

Mapping elements (*exec* phases and communication fragments) to SPM regions is very similar to mapping tasks on cores. Therefore, following equations (10a), (10b), (10c) and (10d) mimic the behaviour of respectively (3b), (3d), (3e) and (3f) by replacing variables $m_{i,j}, a_{i,j}$ and $am_{i,j}$ with $spmm_{i,j}, spma_{i,j}$ and $spmam_{i,j}$. As a reminder, (10a) detects if two fragments are assigned to the same region from the same core, (10b) represents the causality of a fragment compare to another, and (10c) represents this causality on the same region. Finally, (10d) imposes the mutual exclusion of the reservation time.

$\forall (f,g) \in F \times F, f \neq g, i = task(f), j = task(g)$

$$spmm_{(i,f),(j,g)} = \sum_{z \in R} (m_{i,j} \wedge spmp_{z,(i,f)} \wedge spmp_{z,(j,g)}) \tag{10a}$$

$$spma_{(i,f)} + spma_{(j,g)} = 1 \tag{10b}$$

$$spmam_{(i,f),(j,g)} = spma_{(i,f)} \wedge spmm_{(i,j),(j,g)} \tag{10c}$$

$$\omega_{(i,f)} \leq \sigma_{(j,g)} + \mathcal{M} \times (1 - spmam_{(i,f),(j,g)}) \tag{10d}$$

5.2 Quality of the heuristic compared to the ILP

The following experiments aim at estimating the gap between makespans of schedules generated by the heuristic opposed to the optimal solutions provided by the ILP solver. We expect this gap to be small. Due to the intrinsic complexity of solving our scheduling problem using ILP, we need for these experiments a large number of small task-graphs, such that the ILP is solved in reasonable time. We thus used synthetic task graphs generated using TGFF (see Table 2). For each graph, we varied the number of cores in $\{2, 4, 8, 12\}$ and the sizes of the SPM vary in $\{2KB, 4KB\}$. SPM sizes allow to cover three situations: 1) all test-cases fit in the SPM (4KB size), 2) some test-cases do not entirely fit in SPM (2KB size), 3) some test-cases are too large, hence unschedulable (2KB size, biggest benchmarks).

The ILP solver used is CPLEX v12.7.1 configured with a timeout of 24 hours. The heuristic is implemented in C++ with a 60 minutes timeout.

Table 4 Degradation of the heuristic compared to the ILP on the synthetic task-graphs.

% of exact results (ILP only)	degradation <min,max,avg> %
68%	0%, 20%, 3%

Table 4 presents the combined results for all different configurations. First, it shows the number of optimal (including infeasible) results the ILP solver is able to find in the given timeout – 68%. The remaining 32% includes all other cases where the solver reaches the timeout without neither an optimal solution nor an infeasibility verdict. Then Table 4 presents the minimum/maximum and the average degradation induced by the heuristic over the ILP. As displayed, the average degradation is low thus showing the quality of our heuristic.

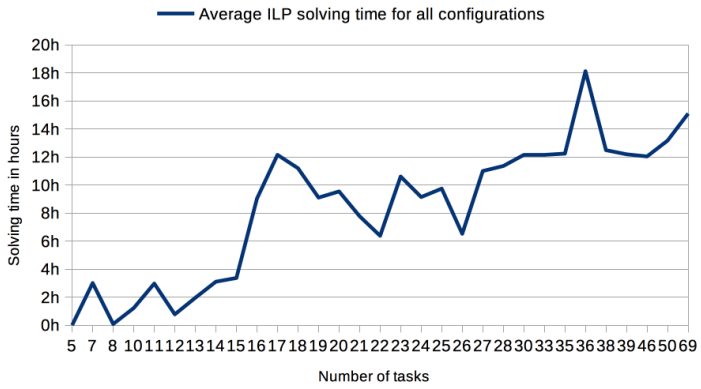


Figure 3 Average ILP solving time for all configurations per number of tasks.

Figure 3 shows that solving an ILP problem does not scale with the growing number of tasks. In contrast, we believe that the proposed scheduling technique does, given its low running times: for the synthetic graphs the average schedule generation times are always less than one second, while for the SRT2RTS benchmarks (up to 340 tasks), the heuristic needs 4 minutes on average.

5.3 Blocking vs non-blocking communications

To compare the benefit of hiding communication latency, the proposed scheduling technique must be opposed to a scheduler that does not hide it. We preferred to modify our heuristic to implement both the *blocking* and *non-blocking* methods instead of reusing a state-of-the-art algorithm. The main reason, as detailed in Section 6, is that related work have characteristics that are hardly compatible with our proposal: different task model [35], SPM big enough to store all code/data [30, 3], lack of information on SPM management [4], different interconnect [16]. Another reason for this choice is to guarantee that the deviation between the results from the two communication modes will not be affected by any other technical implementation decision (e.g.: sorting algorithm).

To summarize the modifications applied to the heuristic in order to get the blocking mode: 1) we forbid to have more than one phase active at a time (both communication and computation as in the example of Figure 2a) 2) we do not fragment communications. We varied the number of cores in $\{2, 4, 8, 12\}$, and the SPM sizes in $\{4KB, 2MB\}$ ($2MB$ is the SMEM (Shared MEMory) size in one cluster of the Kalray MPPA [11]). All aforementioned three situations regarding the SPM size are covered with these configurations. Note that STR2RTS benchmarks are larger in term of memory space than synthetic benchmarks. We then calculate the gain of the *non-blocking* mode versus the *blocking* mode that we expect to be positive.

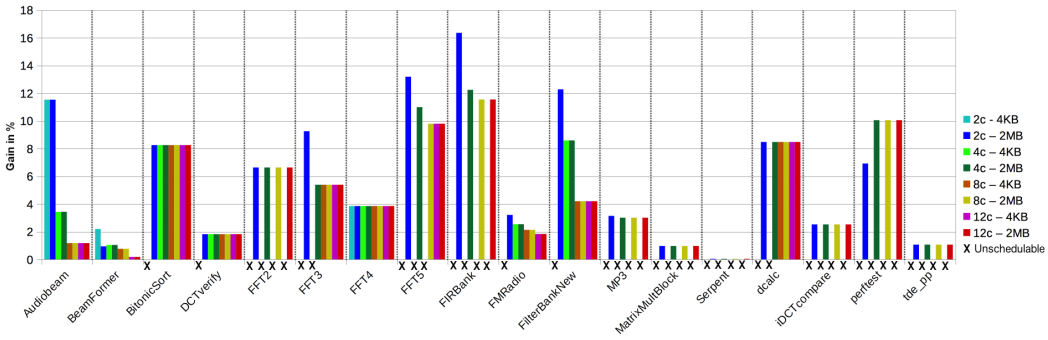


Figure 4 Gain of *non-blocking* communications over *blocking* on STR2RTS benchmarks per cores/SPM configuration – avg: 4%.

Figure 4 presents the average gain per benchmark for all configurations, e.g. *2c-2MB* stands for *2-cores and SPM size of 2MB*. Unfeasible configurations are denoted by the symbol "x". The maximum gain is 16% (FIRBank on 2 cores with 2MB SPM), whereas the average is 4%.

Figure 4 shows that some benchmarks are *unschedulable* for some configurations, e.g. FFT2 with 2c-4KB. This comes from a lack of SPM space to place all code and all data. This might be relaxed with code pre-fetching in *read* phase, which is left for future work.

Lower gains are observed when the amount of parallelism is low due to the lack of opportunity to hide communications. For example, Serpent is a chain of fork-joins containing 2 concurrent tasks only, as opposed to FIRBank which includes only one fork-join construct with several long chains of tasks. In addition, higher gains are observed on hardware configurations with lower number of cores – i.e. 6% on average with 2-cores as opposed to 4% with 12-cores.

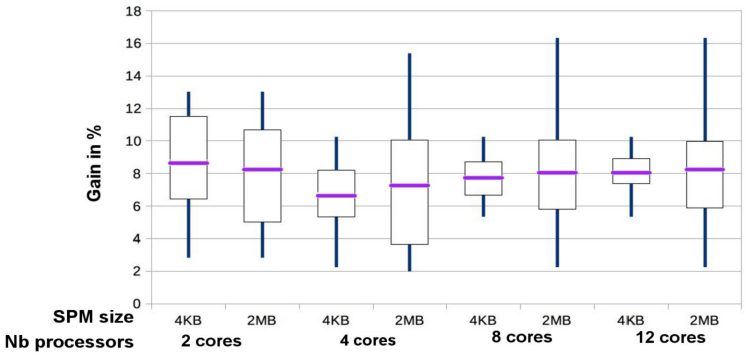


Figure 5 Gain of *non-blocking* communications over *blocking* on TGFF benchmarks.

To evaluate the impact of graph shapes on gains, we experimented our heuristic technique on synthetic task graphs, the ones used previously to validate the heuristic. In contrast to STR2RTS graphs, that are fork-join graphs, synthetic task graphs are arbitrary directed acyclic graphs. Results are depicted in Figure 5. We observe these graphs offer more opportunities to hide communication, with an average gain of 8% in total.

5.4 Impact of fragmentation strategy

Through the paper, we have split *read/write* phases according to tasks dependencies (one fragment per edge in the task). We experimented with two more fine-grain splitting strategies:

- splitting by D_{slot}: each fragment will fit in a T_{slot} bus period, each transmitting D_{slot} bytes – a task transmitting 5 floats (20 bytes) with a $T_{slot} * D_{slot}$ of $3 * 4$ bytes per request will result to 2 fragments, generating 2 communications.
- splitting by *data-type unit (DTU)*: an application exchanging only *floats* will have a DTU of 1 float (4 bytes). If a task produces 5 floats, then there is 5 fragments.

We conducted the experiments by applying our heuristic on the STR2RTS benchmarks, with the very same experimental setup as before. We include in the comparison scheduling in *non-blocking* mode without communication fragmentation (label *no frag* in Figure 6). We expect the gain to increase as the fragment granularity gets smaller.

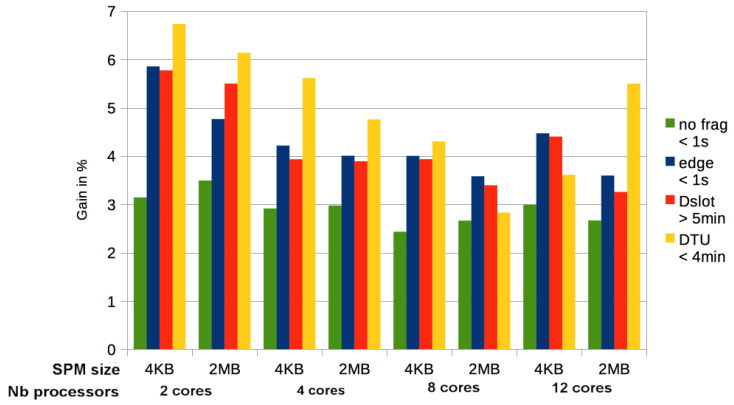


Figure 6 Average gain of *non-blocking* over *blocking* depending on fragmentation strategy.

Figure 6 presents the results with four granularities: *no frag*, *edge* (default configuration), D_{slot} (12 bytes) and *DTU* (4 bytes). Fragmenting communications always result in shorter schedules than the *no frag* configuration. In addition, in most cases the smaller granularity results in higher gains. However the better the results are, the higher the schedule generation time is, as given in the legend of the figure. Schedules are generated in less than 1 second on average for *no frag* and *edge*, whereas several minutes are required on average for fine-grain fragments.

5.5 Schedule implementation on a Kalray MPPA platform

We successfully implemented schedules generated with our heuristic targeting one cluster of a Kalray MPPA Bostan platform [11]. The final code is largely auto-generated (only the code of the exec phase of each task has to be inserted manually in the generated code). At the time of writing, we managed to run benchmark *BeamFormer_12ch_4b* from the STR2RTS benchmark suite [31]. Benchmark *BeamFormer_12ch_4b* is made of 56 tasks with a DAG width of 12. The Kalray MPPA platform includes 16 clusters, each containing 16 cores and a SMEM of 2MB. Four I/O clusters, containing 4 cores each, access either the off-chip global memory or the Ethernet. Clusters are connected through a Network On Chip (NoC).

Following is a summary of the implementation on the Kalray MPPA. Implementation was done at the *bare metal* level. The SMEM is configured in *banked* address mapping mode (consecutive addresses are mapped to the same memory bank), with memory banks are split between computing cores at compile time to have a single-bank considered as SPM per core, as assumed in Section 2.

Data exchanges between cores and the off-chip memory walk through the architecture's NoC, which in our experiments is free from interferences as we only use one cluster of the architecture[2]. Communications are implemented using the Kalray *channel connectors* (one channel for reading, one channel for writing), kept open for the whole execution of the application.

Each core runs one thread, in charge of implementing the schedule of *exec* phases generated off-line, by interleaving a *sched* function between *exec* phases. The *sched* uses *busy-waiting* (reads the local clock of the core to wait for tasks' start time). The worst-case measured overhead of the *sched* function due to clock reading is 32 cycles. An *ad hoc* protocol using barriers is used to re-synchronize local clocks at application start. A specific core is reserved to act as a software DMA engine and is in charge of implementing the schedule of communications (*read* and *write* phases) determined off-line, in a contention-free manner. Implementation of communication phases schedule is identical to the one of computation phases. Moreover, the I/O receiving core follows the schedule to receive and store data to the main memory or to send it to the cluster.

We were able to generate the following versions of the benchmark: 1) *blocking* mode (S_{bl}), 2) *non-blocking* mode without communication fragmentation (S_{nbl}), 3) *non-blocking* mode with fragmentation by edges (S_{nbl}^{edge}), 4) *non-blocking* mode with fragmentation by D_{slot} (12 bytes) (S_{nbl}^{dslot}), 5)*non-blocking* mode with fragmentation by DTU, fragment size is one 4-bytes word (1 float) (S_{nbl}^{dtu}).

In terms of implementation overheads, there is no overhead to set up the software-implemented DMA at run-time, since *channel connectors* are initialized only once at application start. The overhead of 32 cycles due to the scheduler implementation is taken into account.

For this experiments, WCETs of computations and communications were estimated using measurements, adding an arbitrarily chosen margin of 20% for safety. Taking into account implementation overheads, as expected, the overall schedule makespans are: $S_{bl} > S_{nbl} > S_{nbl}^{edge} > S_{nbl}^{dslot}$. The gain of S_{nbl} schedule over the S_{bl} schedule is 1%, the gain of S_{nbl}^{edge} schedule over the S_{nbl} schedule is 36%, and the gain of S_{nbl}^{dslot} over S_{nbl} is 22%.

However, the finer fragmentation policy suffers from an overhead on this platform. The degradation of S_{nbl}^{dtu} over S_{nbl}^{edge} is 24%. The source of this overhead mainly originates from *read* phases measured time where reading one float takes as much time as reading four floats. Nevertheless, we observe a small decrease in *write* phases measured time depending on the amount of data exchanged (approximately 1000 cycles on average).

[2] Note that our abstract architecture model from Section 2 uses a bus. Using a NoC in the Kalray MPPA only changes the overall communication delay computed in Equation 1 since the NoC is free from contentions.

6 Related work

Accessing the global shared memory has always been a performance bottleneck. To overcome this issue, prefetching mechanisms bring information closer to the processor before it is actually needed. Hardware prefetchers will speculatively request data or instructions based on memory access patterns [28]. Software prefetchers give control to the developer or compiler to introspect the code and add prefetching instructions [23]. In this paper we propose a *software* prefetcher that adds prefetching based on a schedule generated off-line.

Most of other works considering SPM aim at deciding what should be stored into the SPM and when to evict data, and in cases some information cannot be stored in SPM it stays in main memory. Considered metrics for SPM allocation are *average-case* performance [15, 25], *power consumption* [36], *WCET* [13], or schedule makespan [4]. In contrast to these studies, our work, in order to control resource contention, requires all information to be stored in SPM.

Wasly and Pellizzoni [38] add a hardware component, named RSMU, to manage the SPM. This RSMU acts similarly as a Memory Management Unit (MMU), except it also uses a previously computed schedule for loads/unloads of code/data from mixed-critical tasks. To use our method, no specific hardware component needs to be added. Giorgi et al. [17] introspect the code to add control of the RSMU, in order to prefetch global data from the global external memory into a local memory on a many-core architecture. They modified the compiler to isolate loads into specific basic blocks and added synchronization if the mandatory data are not yet ready for use. However, their study does not include any real-time guarantee on blocking times. We can guarantee the data will be ready for use without blocking time.

Kim et al. [22] present an algorithm to map a function to a specific SPM region, that inspired our phase to region mapping step. They aim at storing the basic blocks into the SPM in order to improve the WCET of an application on a single-core. We improve their work to map multiple tasks on multi-cores.

Cheng et al. [7] derive a speed-up factor and a resource augmentation factor when partitioning memory banks with minimum interference. At the opposite we have a complete off-line schedule with phase to region allocation on single bank SPM memory.

The PRedictable Execution Model (PREM) from Pellizzoni et al. [29, 1] exposes parallelism by splitting tasks in communication/computation phases. PREM has been widely used – e.g. [38, 3, 39, 4] – because it increases the predictability of an application by isolating memory accesses. Coupling this principle with a software-managed memory (SPM) drastically improves the predictability of the application and so improves its estimated WCET. The authors of [27] present a method to automatically adapt any application to the PREM model, which then allows the application of any SPM load/unload technique including ours. The studies we could find exploiting both the SPM and the PREM model usually fuse the *write* phase of a task with the next activated *read* phase on the same core [38, 39, 2]. As opposed to them, we follow the Acquisition-Execution-Restitution principle from [26] which adds more freedom to schedule generation.

On a single-core, using PREM, Soliman et al. [34] hide the communication latency at the basic-block level thanks to a modification of the LLVM compiler toolchain. Wastly and Pellizzoni [39] proposed to dynamically co-schedule, without preemption, DMA accesses and sporadic tasks on a SPM-based single-core. The SPM is split in 2 parts: one assigned to the currently executing task, while the other load information for the next scheduled task. Our work makes a better use of the SPM by allowing more than two regions alive at the same time. This last work has been extended to multi-core in [2].

Rouxel et al. [30] presented a co-scheduling and mapping of computation and communication phases from task-graph for multi-cores. They limited their work to *blocking* communication whereas we use *non-blocking* ones and we fragment them to add flexibility in the schedule. They assume an infinite SPM size, which looks to us unrealistic, therefore we relaxed this assumption in our scheduling method. In addition, they showed that their scheduling method with an accurate contention model exhibits similar gain and a larger solving times than contention-free ones. Hence, we use a contention-free model in this paper.

The technique proposed in [4] generates contention-free off-line schedules with periodic dependant tasks. Dealing with the SPM, they aim at deciding if a task should be resident in SPM or be fetched before each execution from the global memory. Unfortunately they do not provide information on SPM allocation, raising questions about address allocation and SPM fragmentation. With our region allocation scheme, an SPM allocation scheme that manages fragmentation is proposed.

A technique to hide transfers behind calculations is presented in [35]. Similarly to [39] and [2], the SPM is split in two regions, one used by the application while the other is being loaded. Our work differs from the work in [35] by the task model under use (dependant tasks in our work, sporadic independent tasks in their work). Moreover, our work make better use of SPM by allowing more than two SPM regions to be alive simultaneously.

The work presented in [16] proposes an off-line scheduling scheme for flight management systems using a PREM-like task model. The proposed schedule avoid interferences to access the communication medium. However, in contrast to our work, there are still interferences in their schedule, due to communications between tasks assigned to different cores.

Other works very close to our research, such as [24, 9, 37, 33], statically schedule applications represented by synchronous data flow graphs with some form of buffer checking. However, they do not use the PREM/AER model like us [37, 33], and none of them fragment the communications, which allows us to drastically increase the hiding opportunities. The research presented in [6] proposes a feasibility test that verifies whether scratchpad memories are large enough to contain the maximum memory backlog that may be generated by an application modeled as a task graph. In contrast to [6], our work focuses not only on memory usage feasibility but also on timing feasibility.

7 Conclusion

In this work, we have shown how to minimize the impact of the communication latency when mapping/scheduling a task graph on a multi-core, by overlapping communications and computations. We also argued this kind of technique should always be coupled with a memory allocation scheme to guarantee the integrity of the accessed data. Thus we formulated such allocation scheme in our scheduler. Our experimental results show that, compared to a scenario not overlapping communications and computations, our approach improves the schedule makespan by 4% on average on streaming application (8% on synthetic task graphs). As future work, we plan to improve the accesses of the global main memory such as the DRAM where the scheduler accounts for the locality in this memory. For example, the fragments could be designed to exploit DRAM row locality and read/write switching of the communications. In the near future, we intend to extend this work to applications integrating multiple DAGs. Finally, we plan to strengthen our implementation on the Kalray MPPA platform, especially on the SMEM management.

References

1 Ahmed Alhammad and Rodolfo Pellizzoni. Time-predictable execution of multithreaded applications on multicore systems. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE, 2014.

2 Ahmed Alhammad, Saud Wasly, and Rodolfo Pellizzoni. Memory efficient global scheduling of real-time tasks. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pages 285–296. IEEE, 2015.

3 Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Akesson, Vincent Nélis, and Thomas Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *Real-Time Systems (ECRTS), 2016 28th Euromicro Conference on*, pages 14–24. IEEE, 2016.

4 Matthias Becker, Saad Mubeen, Dakshina Dasari, Moris Behnam, and Thomas Nolte. Scheduling multi-rate real-time applications on clustered many-core architectures with memory constraints. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 560–567, January 2018. doi:10.1109/ASPDAC.2018.8297382.

5 Gerald G Brown and Robert F Dell. Formulating integer linear programs: A rogues' gallery. *INFORMS Transactions on Education*, 7(2):153–159, 2007.

6 Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio C. Buttazzo. Memory Feasibility Analysis of Parallel Tasks Running on Scratchpad-Based Architectures. In *2018 IEEE Real-Time Systems Symposium, RTSS 2018, Nashville, TN, USA, December 11-14, 2018*, pages 312–324, 2018.

7 Sheng-Wei Cheng, Jian-Jia Chen, Jan Reineke, and Tei-Wei Kuo. Memory Bank Partitioning for Fixed-Priority Tasks in a Multi-core System. In *Real-Time Systems Symposium (RTSS), 2017 IEEE*, pages 209–219. IEEE, 2017.

8 Junchul Choi, Hyunok Oh, Sungchan Kim, and Soonhoi Ha. Executing synchronous dataflow graphs on a spm-based multicore architecture. In *Proceedings of the 49th Annual Design Automation Conference*, pages 664–671. ACM, 2012.

9 Yoonseo Choi, Yuan Lin, Nathan Chong, Scott Mahlke, and Trevor Mudge. Stream compilation for real-time embedded multicore systems. In *Code generation and optimization, 2009. CGO 2009. International symposium on*, pages 210–220. IEEE, 2009.

10 Robert I. Davis and Alan Burns. A survey of hard real-time scheduling algorithms for multiprocessor systems. *in ACM Computing Surveys*, 2011.

11 Benoît Dupont De Dinechin, Duco Van Amstel, Marc Poulhi'es, and Guillaume Lager. Time-critical computing on a single-chip massively parallel processor. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE, 2014.

12 Steven Derrien, Isabelle Puaut, Panayiotis Alefragis, Marcus Bednara, Harald Bucher, Clément David, Yann Debray, Umut Durak, Imen Fassi, Christian Ferdinand, Damien Hardy, Angeliki Kritikakou, Gerard Rauwerda, Simon Reder, Martin Sicks, Timo Stripf, Kim Sunesen, Timon ter Braak, Nikolaos Voros, and Jürgen Becker. WCET-Aware Parallelization of Model-Based Applications for Multi-Cores: the ARGO Approach. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2017*. IEEE, 2017.

13 Jean-Francois Deverge and Isabelle Puaut. WCET-directed dynamic scratchpad memory allocation of data. In *Real-Time Systems, 2007. ECRTS'07. 19th Euromicro Conference on*, pages 179–190. IEEE, 2007.

14 Robert P Dick, David L Rhodes, and Wayne Wolf. TGFF: task graphs for free. In *Proceedings of the 6th international workshop on Hardware/software codesign*, pages 97–101. IEEE Computer Society, 1998.

15 Boubacar Diouf, Can Hantacs, Albert Cohen, "Ozcan "Ozturk, and Jens Palsberg. A decoupled local memory allocator. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):34, 2013.

16 Guy Durrieu, Madeleine Faugere, Sylvain Girbal, Daniel Gracia P'erez, Claire Pagetti, and Wolfgang Puffitsch. Predictable flight management system implementation on a multicore processor. In *Embedded Real Time Software (ERTS'14)*, 2014.

17 Roberto Giorgi, Zdravko Popovic, and Nikola Puzovic. Exploiting DMA to enable non-blocking execution in Decoupled Threaded Architecture. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

18 Igor Griva, Stephen G. Nash, and Ariela Sofer. *Linear and Nonlinear Optimization, Second Edition*. Society for Industrial Mathematics, 2008.

19 James A Kahle, Michael N Day, H Peter Hofstee, Charles R Johns, Theodore R Maeurer, and David Shippy. Introduction to the cell multiprocessor. *IBM journal of Research and Development*, 49(4.5):589–604, 2005.

20 Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. Inter-core Prefetching for Multicore Processors Using Migrating Helper Threads. *SIGPLAN Not.*, 46(3):393–404, March 2011. doi:10.1145/1961296.1950411.

21 Timon Kelter, Tim Harde, Peter Marwedel, and Heiko Falk. Evaluation of resource arbitration methods for multi-core real-time systems. In *WCET*, pages 1–10, 2013.

22 Yooseong Kim, David Broman, Jian Cai, and Aviral Shrivastaval. WCET-aware dynamic code management on scratchpads for software-managed multicores. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 179–188. IEEE, 2014.

23 Alexander C Klaiber and Henry M Levy. An architecture for software-controlled data prefetching. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 43–53. ACM, 1991.

24 Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *ACM SIGPLAN Notices*, volume 43, pages 114–124. ACM, 2008.

25 Lian Li, Jingling Xue, and Jens Knoop. Scratchpad memory allocation for data aggregates via interval coloring in superperfect graphs. *ACM Transactions on Embedded Computing Systems (TECS)*, 10(2):28, 2010.

26 Cl'audio Maia, Luis Nogueira, Luis Miguel Pinho, and Daniel Gracia P'erez. A closer look into the aer model. In *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*, pages 1–8. IEEE, 2016.

27 Renato Mancuso, Roman Dudko, and Marco Caccamo. Light-PREM: Automated software refactoring for predictable execution on COTS embedded systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, pages 1–10. IEEE, 2014.

28 Pierre Michaud. Best-Offset Hardware Prefetching. In *International Symposium on High-Performance Computer Architecture*, Barcelona, Spain, March 2016. doi:10.1109/HPCA.2016.7446087.

29 Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for COTS-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279. IEEE, 2011.

30 Benjamin Rouxel, Steven Derrien, and Isabelle Puaut. Tightening contention delays while scheduling parallel applications on multi-core architecture. In *Embedded Software (EMSOFT), 2017 International Conference on*. ACM, 2017.

31 Benjamin Rouxel and Isabelle Puaut. STR2RTS: Refactored StreamIT Benchmarks into Statically Analyzable Parallel Benchmarks for WCET Estimation & Real-Time Scheduling. In Jan Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 57 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–12, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2017.1.

32 Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. Patmos reference handbook. *Technical University of Denmark, Tech. Rep*, 2015.

33 Stefanos Skalistis and Alena Simalatsar. Near-optimal deployment of dataflow applications on many-core platforms with real-time guarantees. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 752–757. IEEE, 2017.

34 Muhammad Refaat Soliman and Rodolfo Pellizzoni. WCET-Driven Dynamic Data Scratchpad Management With Compiler-Directed Prefetching. In Marko Bertogna, editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:23, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECRTS.2017.24.

35 Rohan Tabish, Renato Mancuso, Saud Wasly, Ahmed Alhammad, Sujit S Phatak, Rodolfo Pellizzoni, and Marco Caccamo. A real-time scratchpad-centric os for multi-core embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE*, pages 1–11. IEEE, 2016.

36 Hideki Takase, Hiroyuki Tomiyama, and Hiroaki Takada. Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 1124–1129. IEEE, 2010.

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

38 Saud Wasly and Rodolfo Pellizzoni. A dynamic scratchpad memory unit for predictable real-time embedded systems. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 183–192. IEEE, 2013.

39 Saud Wasly and Rodolfo Pellizzoni. Hiding memory latency using fixed priority scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 75–86. IEEE, 2014.

A STR2RTS benchmark suite

Following Table 5 characterise the used benchmarks from STR2RTS benchmark suite. The first column presents the number of tasks and the second column the width of the graph. Then it gives the average data in bytes sent along all edges. Following is the average memory footprint of all tasks withing a benchmark, it includes the code size and the stack size. Last column shows the average, among all tasks, of WCET estimates. All this information are shipped with the benchmark suite and target a Patmos single core architecture [32].

Table 5 Benchmarks characteristics.

Name	#Tasks	Width	avg data (bytes)	avg task's memory footprint	avg task's WCET
Audiobeam	20	15	12 B	108 B	41
Beamformer	56	12	18 B	246 B	2718
BitonicSort	122	8	49 B	109 B	30
DCTverify	7	2	513 B	506 B	10045
FFT2	26	2	551 B	2 KB	618
FFT3	82	16	84 B	208 B	120
FFT4	10	2	6 B	32 B	11
FFT5	115	16	52 B	1 KB	38
Firbank	340	12	505 B	2 KB	670
FMRadio	67	20	6 B	191 B	235
FilterbankNew	53	8	35 B	180 B	144
MP3	116	36	3502 B	19 KB	12222
MatrixMultiBlock	23	2	793 B	1 KB	726
Serpent	234	2	1013 B	709 B	922
dcalc	84	4	106 B	685 B	174
IDCTcompare	13	3	454 B	685 B	4557
perftest	16	4	8267 B	21 KB	5269
tde_pp	55	2	25344 B	16 KB	2931