Open access • Journal Article • DOI:10.1145/3126496

Tightening Contention Delays While Scheduling Parallel Applications on Multi-core Architectures — Source link

Benjamin Rouxel, Steven Derrien, Isabelle Puaut

Institutions: University of Rennes

Published on: 27 Sep 2017 - Embedded Software

Topics: Job shop scheduling, Schedule, Scheduling (computing), Integer programming and Multi-core processor

Tightening Contention Delays While Scheduling Parallel Applications on Multi-core Architectures

Benjamin Rouxel, Steven Derrien, Isabelle Puaut

Tightening contention delays while scheduling parallel applications on multi-core architectures

Benjamin Rouxel [*1], Steven Derrien [†1], and Isabelle Puaut [‡1]

[1]University of Rennes 1/IRISA

September 19, 2017

Abstract

Multi-core systems are increasingly interesting candidates for executing parallel real-time applications, in avionic, space or automotive industries, as they provide both computing capabilities and power efficiency. However, ensuring that timing constraints are met on such platforms is challenging, because some hardware resources are shared between cores.

Assuming worst-case contentions when analyzing the schedulability of applications may result in systems mistakenly declared unschedulable, although the worst-case level of contentions can never occur in practice. In this paper, we present two *contention-aware* scheduling strategies that produce a time-triggered schedule of the application's tasks. Based on knowledge of the application's structure, our scheduling strategies precisely estimate the *effective* contentions, in order to minimize the overall makespan of the schedule. An Integer Linear Programming (ILP) solution of the scheduling problem is presented, as well as a heuristic solution that generates schedules very close to ones of the ILP (5 % longer on average), with a much lower time complexity. Our heuristic improves by 19% the overall makespan of the resulting schedules compared to a worst-case contention baseline.

Keywords — Real-time System, Contention-Aware Scheduling

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 increasing demand for computing power and low energy consumption is placing multi-/many-core architectures as increasingly interesting candidates for executing embedded critical systems. It becomes more and more common to find such architectures in automotive, avionic or space industries [2, 16].

Guaranteeing that timing constraints are met on multi-core platforms is a challenging issue. One difficulty lies in the estimation of the Worst-Case Execution Time (WCET) of tasks. Due to the presence of shared hardware resources (buses, shared last level of cache, ...), techniques designed for single-core architectures cannot be directly applied to multi-core ones. Since it is hard in general to guarantee the absence of resource conflicts during execution, current WCET techniques either produce pessimistic WCET estimates or constrain the execution to enforce the absence of conflicts, at the price of a significant hardware under-utilization.

[*] benjamin.rouxel@irisa.fr

[†] steven.derrien@irisa.fr

[‡] isabelle.puaut@irisa.fr

A second issue is the selection of a scheduling strategy which will decide where and when to execute tasks. Scheduling for multi-core platforms was the subject of many research works, surveyed in [8]. We believe that static mapping of tasks to cores (partitioned scheduling) and time-triggered scheduling on each core allow to have control on sharing of hardware resources, and thus allow to better estimate worst-case contention delays.

Some existing work on multi-core scheduling considers that the platform workload consists of independent tasks. As parallel execution is the most promising solution to improve performance, we envision that within only a few years from now, real-time workloads will evolve toward parallel programs. The timing behaviour of such programs is challenging to analyze because they consist of *dependent* tasks interacting through complex synchronization/communication mechanisms. We believe that models offering a high-level view of the behavior of parallel programs allow a precise estimation of shared resource conflicts. In this paper, we assume parallel applications modeled as directed acyclic task graphs (DAGs), and show that the knowledge of the application's structure allows to have precise estimation of tasks that effectively execute in parallel, and thus contention delays. 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 [10], looks to us a much more promising direction.

In this paper, we present two mapping and scheduling strategies featuring bus contention awareness. Both strategies apply to multi-core platforms where cores are connected to a round-robin bus. A safe (but pessimistic) bound for the access latency is to consider *NbCores - 1* contending tasks being granted access to the bus (with *NbCores* as the number of available cores). Our scheduling strategies take into consideration the application's structure and information on the schedule under construction to estimate precisely the *effective* degree of interference used to compute the access latency. The proposed scheduling strategies generate a non preemptive time-triggered partitioned schedule and select the appropriate level of contention to minimize the schedule length.

The first proposed scheduling method models the task mapping and scheduling problem as *constraints* on task assignment, task start times and communications between tasks. We demonstrate that the optimal schedule can only be found using quadratic equations due to the nature of the required information to model the communication cost. This modeling is then refined into an Integer Linear Programming (ILP) formulation that in some cases overestimates communication costs and thus may not find the shortest schedule. Since the solved scheduling problem is NP-hard, the ILP formulation is shown to not scale properly when the number of tasks grows. We thus developed a heuristic scheduling technique that scales much better with the number of tasks and is able to compute the accurate communication cost. Albeit not always finding the optimal solution, the ILP formulation is included in this paper, because it gives a non ambiguous description of the problem under study, and also serves as a baseline to evaluate the quality of the proposed heuristic technique.

The proposed scheduling techniques are evaluated experimentally. The schedule's length generated by our heuristic is compared to its equivalent baseline scheduling technique accounting for the worst case contention. The experimental evaluation also studies the interest of allowing concurrent bus accesses as compared to related work where concurrent accesses are systematically avoided in the generated schedule. Finally, we study the time required by the proposed techniques, as well as how schedule lengths vary when changing architectural parameters such as the duration of one slot of the round-robin bus. The experimental evaluation uses a subset of the StreamIT streaming benchmarks [25] as well as synthetic task graphs using the TGFF graph generator [11].

The contributions of this work are threefold:

1. First, we propose a novel approach to derive precise bounds on worst-case contention on a shared round-robin bus. Compared to previous methods, we use knowledge of the application's structure (task graph) as well as knowledge of tasks placement and scheduling to precisely estimate tasks that execute in parallel, and thus tighten the worst bus access delay.

2. Second, we present two scheduling methods that calculate a time-triggered partitioned schedule, using an ILP formulation and a heuristic. The novelty with respect to existing scheduling techniques lies on the ability of the scheduler to select the best strategy regarding concurrent accesses to the shared bus (allow of forbid concurrency) to minimize the overall makespan of the schedule.

3. Third, we provide experimental data to evaluate the benefit of precise estimation of contentions as compared to the baseline estimation where *NbCores - 1* tasks are granted access to the bus for every memory access. Moreover, we discuss the interest of allowing concurrency (and thus interference) between tasks as compared to state-of-the-art techniques such as [2] where contentions are systematically avoided.

The rest of this paper details the proposed techniques and is organized as follows. Section 2 presents related studies. Assumptions on the hardware and software are given in Section 3. Section 4 details the proposed method to calculate precise worst-case degree of interference when accessing the shared bus. Section 5 then presents the two techniques for schedule generation, using an ILP formulation and a heuristic. Section 6 presents experimental results. Concluding remarks are given in Section 7.

2 Related work

Tasks scheduling on multi-core platforms consists in deciding where (mapping) and when (scheduling) each task is executed. The literature on mapping/scheduling of tasks on multi-cores is tremendous as there exists plenty of different properties on, e.g. the input task set, the type of scheduling algorithm. According to the survey from Davis and Burns [8], the three main categories of scheduling algorithms are global scheduling, semi-partitioned, and partitioned scheduling. According to their terminology, the scheduling methods presented in this paper can be classified as static, partitioned, time-triggered and non-preemptive.

Shared resources in multi-core systems may be either shared software objects (such as variables) that have to be used in mutual exclusion or shared hardware resources (such as buses or caches) that are shared between cores according to some resource arbitration policies (TDMA, round-robin, etc). These two classes of resources lead to different analyses to ensure that there is neither starvation nor deadlock. Dealing with shared objects is not new, and there now exists several techniques adapted from the single-core systems. Most of them are based on priority inheritance. In particular Jarrett et al. [17] apply priority inheritance to multi-cores and propose a resource management protocol which bounds the access latency to a shared resource. Negrean et al. [20] provide a method to compute the blocking time induced by concurrent tasks in order to determine their response time.

Beyond shared objects, multi-core processors feature hardware resources that may be accessed concurrently by tasks running on the different cores. Typical hardware resources are the main memory, the memory bus or shared last-level cache. A contention analysis then has to be defined to determine the worst case delay for a task to gain access to the resource (see [12] for a survey). Some shared resources may directly implement timing isolation mechanism between cores, such as Time Division Multiple Access (TDMA) buses, making contention analysis straightforward.

To avoid resource under-utilization caused by TDMA, other resource sharing strategies such as round-robin offer a good trade-off between predictability and resource usage. Worst-case bounds on contention are similar to those of TDMA. However, knowledge about the system may help tightening estimated contention delays.

Approaches to estimate contention delays for round-robin arbitration differ according to the nature and amount of information used to estimate contention delays. For architectures with caches, Dasari et al. [6, 7] only assume task mapping known, whereas Rihani et al. [16] assume both mapping and execution order on each core known. Schliecker et al. [22] tightly determine the number of interfering bus requests. In comparison with these works, our technique jointly calculates task scheduling and contentions with the objective of minimizing the schedule makespan by letting the technique decide when it is necessary to avoid or to account for interference.

Further refinements of contention costs can be obtained by using specific task models. Pellizzoni et al. [21] introduced the PRedictable Execution Model (PREM) that splits a task in a *read* communication phase and an *execute* phase. This allows accesses to shared hardware resources to be precisely identified. In our work, we use a model very close to the PREM task model.

The PREM task model, or similar ones, was used in several research works [1, 2]. Alhammad and Pellizzoni [1] proposed a heuristic to map and schedule a fork/join graph onto a multi-core in a contention-free manner. They split the graph in sequential or parallel segments, and then schedule each segment. In contrast to us, they consider only code and local data access in contention estimation, leaving the global shared variable in the main external memory with worst concurrency assumed when accessing them. Moreover, we deal with any DAG not just fork/join graphs, and write back modified data to memory only when required. Becker et al. [2] proposed an ILP formulation and a heuristic aiming at scheduling periodic independent PREM-based tasks on one cluster of a Kalray MPPA processor. They systematically create a contention-free schedule. Our work differs in the considered task model as well as the goal to reach. They consider sporadic independent tasks to which they aim at finding a valid schedule that meets each tasks' deadline. In contrast, we consider one iteration of a task graph and we aim at finding the shortest schedule. In addition, our resulting schedule might include overlapping communications due to scheduler decision, while [1, 2] only allow synchronized communication.

Giannopoulou *et al* proposed in [13] a combined analysis of computing, memory and communication scheduling in a mixed-criticality setting, for cluster-based architectures such as the Kalray MPPA. Similarly to our work, the authors aim, among others, at precisely estimating contention delays, in particular by identifying tasks that may run in parallel under the FTTS schedule, that uses synchronization barriers. However, to our best knowledge they do not take benefit of the application structure, in particular dependencies between tasks to further refine contention delays.

In order to reduce the impact of communication delays on schedules, [4, 14] hide the communication request while a computation task is running. This accounts with the asynchronism implied by DMA requests. However they use a worst-case contention which could be refined by our study. In addition to the initial problem, shared resource interference can be accounted at schedule time in order to tighten the overall makespan of the resulting schedule.

To quantify memory interference on DRAM-banks, [19, 27] proposed two analyses, *request-driven* and *job-driven*. The former one bounds memory request delays considering memory interference on the DRAM bank, while the latter adds the worst-case concurrency on the data-bus of the DRAM. Their work is orthogonal to ours: the *request-driven* analysis would refine the access time part in our delay, while our method could refine their *job-driven* analysis by decreasing the amount of concurrency they use.

3 System model

3.1 Architecture model

We consider a multi-core architecture for which every core is connected to a main external memory through a bus. Each core either has private access to a ScratchPad memory (SPM) (e.g.: Patmos [23]) or there exists a mechanism for bank privatization (e.g.: Kalray MPPA [9]). Such a private memory allows, after having first fetched data/code from the main memory, to perform computations without any access to the shared bus. For each core, data is transferred between the SPM and the main memory through a shared bus.

Communications are *blocking* and *indivisible*. The sender core initiates a memory request, then waits until the request is fully complete (*blocking* communications), i.e. the data is transferred from/to the external memory. There is no attempt to reuse processor time during a communication by allocating the processor to another task (*indivisible* communication). Execution on the sending core is stalled until communication completion[1].

In case the sender and the receiver tasks execute on the same core, communications are performed directly using the SPM and no memory transfer is performed.

The shared bus is arbitrated using a round-robin policy. Access requests are enqueued (one queue per core) and served in a round-robin fashion. A maximum duration of T_{slot} is allocated to each core, to transfer D_{slot} data words to external memory (a data word needs T_{slot}/D_{slot} time units to be sent). If a core requires more time than T_{slot} to send all the data, then the data will be split in *chunks* to be sent in several intervals of length T_{slot} (see equation (1a)) plus some additional *remaining time* (see equation (1b)). If a full T_{slot} duration is not needed to send some data, the arbiter processes the request from the next core in the round. As an example, taking a D_{slot} of 2 data words and a core requesting a transfer request for data of 5 data words, results in two periods of duration T_{slot} and a remaining time of T_{slot}/D_{slot}.

In the worst case and for each chunk, a request will be delayed by $NbCores - 1$ pending requests from the other cores (with $NbCores$ being the number of available cores), see equation (1c). Overall, equation (1d) derives the worst latency to transmit some data with a round-robin arbitration policy.

The round-robin arbiter is predictable as the latency of a request can be statically estimated, as long as the configuration of the arbiter (parameters T_{slot} and D_{slot}) and the amount of data to be transferred ($data$) are known at design time [18].

$$\#chunks = \lfloor data/D_{slot} \rfloor \tag{1a}$$

$$remainingTime = (data\ mod\ D_{slot}) \cdot (T_{slot}/D_{slot}) \tag{1b}$$

$$\#waitingSlots = \lceil data/D_{slot} \rceil \tag{1c}$$

$$delay = \underbrace{T_{slot} \cdot \#waitingSlots \cdot (NbCores - 1)}_{\text{Total waiting time}} + \underbrace{T_{slot} \cdot \#chunks + remainingTime}_{\text{Total access time}} \tag{1d}$$

3.2 Task model

In this work, we consider an application modeled as a directed acyclic task graph (DAG)[2], in which nodes represent computations (tasks) and edges represent communications between tasks. A task graph G is a pair (V, E) where the vertices in V represent the tasks of the application. The set of edges E represents the data dependencies. An edge is present when a task is causally dependent on an other one, meaning the target of the edge needs the source to be completed prior to run. An example of simple task graph, extracted from the StreamIT benchmark suite [25], is presented by Figure 1 and corresponds to a radix-2 of a Fast Fourier Transform.

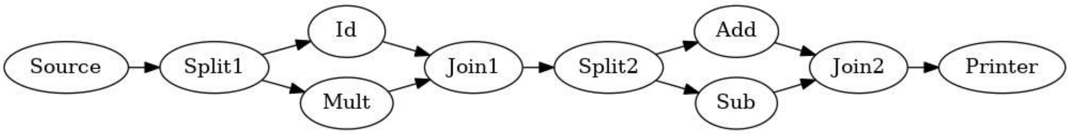


Figure 1: Task graph example – radix-2 case of Fast Fournier Transform from the StreamIT benchmark suite [25]

Each task is divided in three phases (or sub-tasks) according to the *read-execute-write* semantics,

[1] Non blocking communication using a DMA engine, is left for future work

[2] This work supports multiple DAGs with same periodicity as it is, however we skipped it for space considerations.

as first defined in PREM [21] and augmented in [1] with a *write* phase. The *read* phase reads/receives the mandatory code and/or data from main memory to SPM, such that the *execute* phase can proceed without access to the bus. Finally the *write* phase writes/sends the resulting data back to the main memory. In the following, *read* and *write* will refer to the communication phases of tasks. The obvious interest of the *read-execute-write* semantics is to isolate the sub-tasks that use the bus. Therefore, the contention analysis can focus only on these sub-tasks. For the sake of simplicity, this study considers that all code and data fits in all types of memory at any point of time. We also assume the code entirely fits into the SPM, but a simple extension could consider prefetching the code in the *read* phase.

A task i is defined by a tuple $< \tau_i^r, \tau_i, \tau_i^w >$ to represent its *read*, *execute*, and *write* phases. An edge is defined by a tuple $e =< \tau_s^w, \tau_t^r, D_{s,t} >$ where τ_s^w is the *write* phase of the source task s, τ_t^r is the *read* phase of the target task t. $D_{s,t}$ is the amount of data exchanged between s and t.

The WCET of the *execute* phase, noted C_i, can be estimated in isolation from the other tasks considering a single-core architecture, because 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). Conversely, the communication delay of the *read* and *write* phases (respectively noted $delay_i^r$ and $delay_i^w$) depend on several factors: amount of data to be transferred, number of potential concurrent accesses to the bus. Thus there are 2 possibilities to estimate the WCET of the *read/write* phase: either taking a pessimistic static bound, agnostic of task placement on cores (equation (1d)), or take into consideration the knowledge about the applications' structure (effective concurrency) and about task mapping and scheduling to obtain a more precise bound, as it will be detailed in Section 4.

4 Refining communication costs

The communication cost for a communication phase depends on how much interference this phase suffers from. The interference is due to tasks running in parallel on the other cores. The number of such tasks depends on scheduling decisions (task placement in time and space). Considering a task i, only tasks that are assigned to a different core may interfere with i, and only tasks that execute within a time window overlapping with that of i actually interfere. This section presents, using a top-down approach, how a precise estimation of communication costs is obtained. For the whole document, *concurrent* tasks is used for tasks with no data dependencies that *may* be executed in parallel, while *parallel* tasks is used for tasks that are scheduled to run in overlapping time windows.

4.1 Accounting for the actual concurrency in applications

Equation (1d) statically computes communication costs assuming all cores ($NbCores - 1$) execute a communicating phase and thus always delay every memory access, which is pessimistic. From the example in Figure 1, and assuming that the application is the only one executing on the architecture, no parallel request can arise at the time of the *read* phase of task *Split1* because of the structure of the application. Similarly on the same example, the *read* phase of task *Add* can only be delayed by the *read* and *write* phases of task *Sub*.

A pair of tasks is concurrent if they do not have data dependencies between each other, i.e. tasks that may be executed in parallel. As an example, in Figure 1, tasks *Add* and *Sub* are concurrent. Determining if two tasks are concurrent is usually NP-complete [24]. However, with the properties of our task model, in particular the presence of statically-paired communication points, evaluation of concurrency is polynomial. Two tasks are concurrent if there exists no path connecting them in the task graph. By building the transitive closure of the task graph, using for example the classical Warshall's algorithm [26], two tasks i and j are concurrent if there is no edge connecting them in the transitive closure. In the following, function *are_ conc(i,j)* will be used to indicate task concurrency according to the method described in this section. It returns *true* when tasks i and j are concurrent and *false* otherwise.

According to the knowledge of the structure of the task graph, equation (1d) can then be refined as follows. Instead of considering $NbCores - 1$ contentions for every memory access, the worst-case number of contenders with a task i will be *min(NbCores-1, | j | s.t. are_ conc(i,j)).*

4.2 Further refining the worst-case degree of interference

Keeping the example from Figure 1, if the two concurrent tasks *Add* and *Sub* are mapped on the same core and thus are executed in sequence, then their communication phases do not interfere anymore. Knowledge of tasks' scheduling (tasks placement and time window assigned to each task), when known, can further help refining the amount of interference suffered by a task.

Reasoning in reverse, two phases do *not* overlap if one ends before the other starts, which leads for tasks with *read-execute-write* semantics to equation (2). For two tasks i and j, taking two phases τ_i^X and τ_j^Y, where X and Y can either represent a *read* or a *write*, equation (2) states that if phase τ_j^Y ends before phase τ_i^X starts or vice versa, then the two phases τ_i^X and τ_j^Y do *not* overlap. We consider here the end date as the first discrete time point at which the task is over, thus no overlapping occurs when end_j^Y and $start_i^X$ are equal.

$$end_j^Y \leq start_i^X \vee end_i^X \leq start_j^Y \tag{2}$$

Then, by negating equation (2), we get equation (3) that will be true if two tasks have overlapping execution windows. In the following, $are\_OL(\tau_i^X, \tau_j^Y)$ returns *true* if the communication delay of phases τ_i^X and τ_j^Y overlap, and *false* otherwise.

$$are\_OL(\tau_i^X, \tau_j^Y) = \neg(end_j^Y \leq start_i^X \vee end_i^X \leq start_j^Y) \equiv (end_j^Y > start_i^X \wedge end_i^X > start_j^Y) \tag{3}$$

Assuming the schedule is known, the degree of interference a task can suffer from can be determined by counting the number of other tasks that overlap in the schedule. Only concurrent tasks (function *are_conc*) can overlap, because dependent (not concurrent) tasks have data dependencies.

As constrained by the task model (Section 3.2), only communication phases request accesses to the bus, thus only the amount of interference of the *read* and *write* phases needs to be computed. This leads to equations (4a) and (4b) that jointly compute the number of interfering tasks for each communication phase ($\#interf_i^r$ and $\#interf_i^w$ for respectively the *read* and *write*) of a task i by detecting overlapping executions in the set of concurrent tasks.

$\forall i \in T;$

$$\#interf_i^r = \sum_{j \in T | are\_conc(i,j)} are\_OL(\tau_i^r, \tau_j^r) + \sum_{j \in T | are\_conc(i,j)} are\_OL(\tau_i^r, \tau_j^w) \tag{4a}$$

$$\#interf_i^w = \sum_{j \in T | are\_conc(i,j)} are\_OL(\tau_i^w, \tau_j^w) + \sum_{j \in T | are\_conc(i,j)} are\_OL(\tau_i^w, \tau_j^r) \tag{4b}$$

The values of $\#interf_i^r$ and $\#interf_i^w$ from equations (4a) and (4b) can then replace the pessimistic value of *NbCores* −1 from equation (1d) to tighten the worst-case delay of the *read* and *write* phases, leading to equations (5a) and (5b).

$\forall i \in T;$

$$delay_i^r = T_{slot} \cdot \#waitingSlots \cdot \#interf_i^r + T_{slot} \cdot \#chunks + remainingTime \tag{5a}$$

$$delay_i^w = T_{slot} \cdot \#waitingSlots \cdot \#interf_i^w + T_{slot} \cdot \#chunks + remainingTime \tag{5b}$$

This last refinement depends on the knowledge of the schedule. The two scheduling techniques described in Section 5 use these formulas jointly with schedule generation.

5 Contention-aware mapping/scheduling algorithms

This section presents two scheduling techniques that integrate the precise interference costs calculated in the previous section, first as a constraints' system mapped to Integer Linear Programming (ILP) formulation, second as a heuristic method. The main outcome of both techniques is a static mapping and schedule for each core, for one single application. According to the terminology given in [8], the proposed scheduling techniques are partitioned, time-triggered and non-preemptive.

5.1 Integer Linear Programming technique

An Integer Linear Programming (ILP) problem consists of a set of integer variables constrained by linear inequalities. Solving an ILP problem then consists in optimizing (mimimizing or maximizing) a linear function of the variables. When scheduling and mapping a task graph on a multi-core platform, the objective is to obtain the shortest schedule. Table 1 summarizes the notations and variables needed by the ILP formulation.

Table 1: Notations & ILP variables

Sets	T	the set of tasks
	P	the set of processors/cores
Functions	$predecessors(i)$	returns the set of direct predecessors of task i
	$successors(i)$	returns the set of direct successors of task i
	$are\_conc(i, j)$	returns true if i and j are concurrent, as defined in Section 4.1
Constants	C_i	task i execute phase's WCET computed in isolation as stated in Section 3.2
	$D_{i,j}$	amount of data exchanged between task i and j
Integer variables	Θ	schedule makespan
	ρ_i^r, ρ_i, ρ_i^w	start times of *read*, *execute*, *write* phases of task i
	δ_i^r δ_i^w	total amount of data read/written by a *read/write* phase of i according to predecessors/successors' mapping
	$chunk_i^r$ $chunk_i^w$	the number of full slots to *read/write* δ_i^r/δ_i^w from eq. (1a)
	$remainingTime_i^r$ $remainingTime_i^w$	the remaining time to *read/write* that do not fit in a full slot from eq. (1b)
	$waitingSlots_i^r$ $waitingSlots_i^w$	the number of full slots a *read/write* phase must wait from eq. (1c)
	$interf_i^r$ $interf_i^w$	the number of interfering tasks of the *read/write* phases of i from eq. (4a) and (4b)
	$delay_i^r$ $delay_i^w$	task i *read*, *write* phases' WCET from equations (5a) and (5b)
Binary variables	$p_{i,c} = 1$	task i is mapped on core c
	$m_{i,j} = 1$	tasks i & j are mapped on the same core
	$a_{i,j} = 1$	task i is scheduled before task j, in the sense $\rho_i^r \leq \rho_j^r$
	$am_{i,j} = 1$	same as $a_{i,j}$ but on the same core
	$ov_{i,j}^{XY} = 1$	phase X of i overlaps with phase Y of j $XY \in \{rr, ww, rw, wr\}$

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 using the simple transformation rules from [3].

Objective function The goal is to minimize the makespan of the schedule, that is minimizing the end time of the last scheduled task. The objective function, given in equation (6a), is to minimize the makespan Θ. Equation (6b) constraints the completion time of all tasks (starting of *write* phase, ρ_i^w, plus its WCET, $delay_i^w$) to be inferior or equal to the schedule makespan.

$$minimize\ \Theta \tag{6a}$$

$$\forall i \in T; \rho_i^w + delay_i^w \leq \Theta \tag{6b}$$

Problem constraints Some basic rules of a valid schedule are expressed in the following equations. Equation (7a) ensures the unicity of a task mapping. Equation (7b) indicates if two tasks are mapped on the same core, with a simplification to decrease the number of constraints. Equation (7c) orders tasks such that a task scheduled before an other one can not also be scheduled after it, and also imposes an ordering between pairs of tasks. Finally equation (7d) calculates ordering of tasks assigned to the same core.

$$\forall i \in T; \quad \sum_{c \in P} p_{i,c} = 1 \tag{7a}$$

$$\forall (i,j) \in T \times T; i \neq j, m_{i,j} = \sum_{c \in P} (p_{i,c} \wedge p_{j,c}) \quad and\ m_{i,j} = m_{j,i} \tag{7b}$$

$$\forall (i,j) \in T \times T; i \neq j, a_{i,j} + a_{j,i} = 1 \tag{7c}$$

$$\forall (i,j) \in T \times T; i \neq j, am_{i,j} = a_{i,j} \wedge m_{i,j} \tag{7d}$$

Read-execute-write semantics constraints We impose each phase to execute contiguously, as expressed in equations (8a) and (8b). The start time of the *execute* phase of task i (ρ_i) is immediately after the completion of the *read* phase (start of *read* phase ρ_i^r + communication cost $delay_i^r$). Similarly, the *write* phase starts (ρ_i^w) right after the end of the *execute* phase (start of *read* phase ρ_i + WCET C_i).

$$\forall i \in T, \rho_i = \rho_i^r + delay_i^r \tag{8a}$$

$$\forall i \in T, \rho_i^w = \rho_i + C_i \tag{8b}$$

Absence of overlapping on the same core Equation (9) forbids the overlapping of two tasks when mapped on the same core by forcing one to execute after the other.

$$\forall i,j \in T \times T; i \neq j, \quad \rho_i^w + delay_i^w \leq \rho_j^r + \mathcal{M}\ (1 - am_{i,j}) \tag{9}$$

This constraint must be activated only if the two tasks are mapped on the same core. Thus, a nullification method is applied, with the use of a big-M notation [15]. The selected value for the big-M constant is the makespan of a sequential schedule on 1 core, the sum of tasks' WCETs (see equation (10)), which is the worst scenario that can arise.

$$\mathcal{M}\ = \sum_{i \in T} C_i \tag{10}$$

Data dependencies in the task graph Equation (11) enforces data dependencies by constraining all tasks to start after the completion of all their respective predecessors.

$$\forall i \in T, \forall j \in predecessors(i); \quad \rho_j^w + delay_i^w \leq \rho_i^r \tag{11}$$

Computing communication phases interference All the following equations implement, using linear equations, the refinement of contention duration presented in Section 4.2, with the use of the function $are\_conc(i,j)$ to exclude from the search space, tasks that never interfere with each other. Equations (12a) to (12d) implement function are_OL derived from equation (3). For each pair of communication phases, the equations indicate if they are overlapping in the schedule ($ov^X_{Y,Z} = 1$, with $X \in \{rr, ww, rw, wr\}$).

Note that when there is no data for the considered communication phase ($\delta^r_i = 0, \delta^w_i = 0$), then there is no possible overlapping, and then each $ov^X_{Y,Z}$ is constrained to be equal to 0.

$\forall i \in T, \forall j \in are\_conc(i,j)$;

$$ov^{rr}_{i,j} = \delta^r_i > 0 \land \delta^r_j > 0 \land \rho^r_i < \rho_j \land \rho^r_j < \rho_i \tag{12a}$$

$$ov^{ww}_{i,j} = \delta^w_i > 0 \land \delta^w_j > 0 \land \rho^w_i < \rho^w_j + delay^w_j \land \rho^w_j < \rho^w_i + delay^w_i \tag{12b}$$

$$ov^{rw}_{i,j} = \delta^r_i > 0 \land \delta^w_j > 0 \land \rho^r_i < \rho^w_j + delay^w_j \land \rho^w_j < \rho_i \tag{12c}$$

$$ov^{wr}_{i,j} = \delta^w_i > 0 \land \delta^r_j > 0 \land \rho^w_i < \rho_j \land \rho^r_j < \rho^w_i + delay^w_i \tag{12d}$$

Equations (13a) and (13b) implement equations (4a) and (4b) by counting the number of interfering tasks in $interf^r_i$ and $interf^w_i$ for respectively the *read* and *write* phases of task i.

$\forall i \in T$;

$$interf^r_i = \sum_{j \in T | are\_conc(i,j)} ov^{rr}_{i,j} + \sum_{j \in T | are\_conc(i,j)} ov^{rw}_{i,j} \tag{13a}$$

$$interf^w_i = \sum_{j \in T | are\_conc(i,j)} ov^{ww}_{i,j} + \sum_{j \in T | are\_conc(i,j)} ov^{wr}_{i,j} \tag{13b}$$

Finally, equation (14) contains two optimizations that constrain the overlapping variables, to improve the solving time.

$$ov^{rr}_{i,j} = ov^{rr}_{j,i} \qquad ov^{wr}_{i,j} = ov^{rw}_{j,i} \tag{14}$$

Estimation of worst-case communication duration To estimate the time needed for the communication phases, the volume of data read/written respectively by the *read* or *write* phases is required (δ^r_i, δ^w_i). This volume of data must account for the task mapping, as no communication overhead should be charged when both the producer and consumer are mapped on the same core ($m_{i,j} = 1$). This leads to equations (15a) and (15b) that sum the data (constant $D_{i,j}$ extracted from the application) read or written depending on the mapping of the predecessors and successors of a task.

$\forall i \in T$;

$$\delta^r_i = \sum_{j \in predecessors(i)} D_{j,i} \cdot (1 - m_{j,i}) \tag{15a}$$

$$\delta^w_i = \sum_{j \in successors(i)} D_{i,j} \cdot (1 - m_{i,j}) \tag{15b}$$

The next group of equations (16a)-(16d) encodes the round-robin bus arbitration policy, equations (1a)-(1d), and later refined in equation (5). For conciseness, we skip equations related to the write phase as they are equivalent to those for the read phase with some trivial substitutions.

Equation (16a) computes the number of full slots needed as in (1a), according to the volume of data effectively transmitted (δ^r_i) and the transfer rate (D_{slot}). Equation (16b) determines the remaining time as in (1b), equation (16c) computes the number of waiting slots of T_{slot} duration as in (16c), and finally equation (16d) estimates the communication delay required by the read phase of i as in equation (5).

$\forall i \in T;$

$$chunks_i^r = \lfloor \delta_i^r / D_{slot} \rfloor \tag{16a}$$
$$remainingTime_i^r = (\delta_i^r \ mod \ D_{slot}) \cdot (T_{slot}/D_{slot}) \tag{16b}$$
$$waitingSlots_i^r = \lceil \delta_i^r / D_{slot} \rceil \tag{16c}$$
$$delay_i^r = T_{slot} \cdot waitingSlots_i^r \cdot interf_i^r + T_{slot} \cdot chunks_i^r + remainingTime_i^r \tag{16d}$$

The reader may note that equations (16a)-(16c) are not linear. They can however easily be linearized, without any loss of information, into the set of equations (17a)-(17c).

$\forall i \in T;$

$$\delta_i^r = chunks_i^r \cdot T_{slot} + remainingTime_i^r \tag{17a}$$
$$\delta_i^r = waitingSlots_i^r \cdot T_{slot} - unused\_rest_i^r \tag{17b}$$
$$unused\_rest_i^r \geq remainingTime_i^r \tag{17c}$$

A remaining issue emerges in the cost model provided in equation (16d), because this equation is quadratic and non-convex (with the term $waitingSlots_i^r * interf_i^r$, both operands being problem's variables as defined by equations (16c) and (13a)). To model our problem as an instance of ILP, we make the choice of using a safe linear approximation of equation (16d), in which we substitute variable $waitingSlots_i^r$ by a constant $WAIT_i^r$ that may overestimate the number of waiting slots. We do so by considering the worst case scenario in terms of transmitted data, that is, when all data exchanged between dependant tasks occur through the shared bus, which happens for a *read* phase when all the predecessors of the task are mapped on different cores. $WAIT_i^r$ is thus determined by the sum of all data read $D_{j,i}$ as in equation (18).

$$\forall i \in T; \quad WAIT_i^r = \lceil (\sum_{j \in predecessors(i)} D_{j,i}) / T_{slot} \rceil \tag{18}$$

This over-approximation of communication costs induces the solver to map tasks in sequence on the same core or isolate communication phases to avoid interference.

5.2 Heuristic technique based on list scheduling

The basic idea of the proposed heuristic, based on forward list scheduling, is to order tasks from the task graph, and then to add each task one by one in the schedule without backtracking, while keeping the goal of minimizing the overall makespan of the schedule. In the following, tasks are sorted in topological order. Task ordering is a topic on its own and will not be further discussed in this paper.

The method is sketched in algorithm 1. It uses the task graph as input, sorts the nodes to create the list (line 1), and then a loop iterates on each task while there exists tasks to schedule (lines 4-18). This heuristic uses an *As Soon As Possible* (ASAP) strategy when mapping a task. It tries to schedule the task as early as possible on every processor, and then selects the processor where the mapping minimizes the overall makespan (line 15).

As previously explained, the communication cost is dependent on task placement. Thus, after scheduling each task, the communication costs in relation with the newly scheduled task must be recomputed and tasks must be moved on the time line of each involved core to ensure a valid schedule, i.e. a schedule accounting for all interference (lines 11 and 14). Moreover, the heuristic also enforces *read/execute/write* phases to be scheduled contiguously.

Finding the best solution between overlapping and mutual exclusion In the ILP formulation, to minimize the overall makespan, the ILP solver had the opportunity to select, on a per task basis, the best solution between two options: synchronize every communication phase (perform

ALGORITHM 1: Forward list scheduling

```
Input   : A task graph $G = (T, E)$ and a set of processors $P$
Output : A schedule
1  Qready ← TopologicalSortNode(G)
2  Qdone ← $\emptyset$
3  schedule ← $\emptyset$
4  while $t \in$ Qready do
5      Qready ← Qready $\setminus\{t\}$
6      Qdone ← Qdone $\cup\{t\}$
       /* tmpSched contains the best schedule for the current task */
7      tmpSched ← $\emptyset$ with makespan $= \infty$
8      foreach $p \in$ P do
9          $copy_{eff}$ ← schedule
           /* Set $t$ in $copy_{eff}$ on $p$ the earliest in the schedule */
10         MapTaskEarliestStartTime($copy_{eff}, t, p$)
11         AdjustSchedule($copy_{eff}$, Qdone, $t$)
12         $copy_{mutex}$ ← schedule
           /* Set $t$ in $copy_{mutex}$ on $p$ the earliest in mutual exclusion with others */
13         MapTaskEarliestStartTime($copy_{mutex}, t, p$)
14         AdjustSchedule($copy_{mutex}, Qdone, t$)
15         tmpSched ← $min_{makespan}$(tmpSched, $copy_{mutex}, copy_{eff}$)
16     end
17     schedule ← tmpSched
18 end
19 return schedule
```

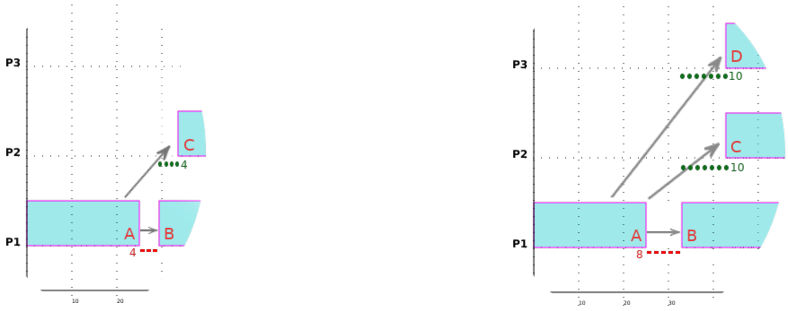
them in mutual exclusion) to obtain a contention-free schedule, or enable concurrency if it results in a shorter global schedule. A similar approach is used in the heuristic. Two schedules are computed: one allowing overlapping between concurrent tasks (lines 9-11) and the other one avoiding it (lines 12-14). Then the shortest of the two schedules is selected (line 15).

Updating the schedule to cope with interference Each time a task is added, new interferences caused by the addition of the task in the schedule must be added, and the delay of some communication phases must be recomputed.

As an example, Figure 2a depicts a partial initial schedule where arrows depict causality between tasks, red dashes draw the writing delay $delay^w_A$, and green dots draw the reading delay $delay^r_C$. A task A mapped on $P1$ sends 4 data items to a task C mapped on $P2$ with a $T_{slot} = 3$ and $D_{slot} = 1$, thus $delay^w_A = delay^r_C = 4$ (equation (5), in this situation none of them suffers from any concurrence). Figure 2b sketches the addition of task D on $P3$, it reads 4 data items written by A. Thus, the new writing delay for task A becomes $delay^w_A = 8$ (still no concurrence on task A). The first consequence is to move task B to guarantee the blocking communication restriction (Section 3.1). Second, the reading delay $delay^r_C$ must be adjusted now to account for the interference introduced by the reading delay $delay^r_D$ and becomes $delay^r_C = 10$ (according to equation (5)), then task C is delayed accordingly.

Whenever a communicating task that was already mapped needs to be rescheduled/delayed, it may change the number of interfering tasks. The communication delay of all tasks impacted by this change must therefore be recomputed, since they may in turn also create interference. The partial communication delay calculation must therefore proceed iteratively until no task is impacted. Convergence is always reached since, at worst, every concurrent task will interfere.

To reduce the number of tasks impacted by each adjustment, algorithm 2 first computes the set



(a)

(b)

Figure 2: Example of adjustments that occur while scheduling. (2a) initial schedule of 3 tasks. (2b) adjusted communication phase delays after the addition of task D

of *related* tasks (line 1), i.e. the tasks that can be impacted by the addition of the current task. The set is constructed by looking into the schedule for the earliest scheduled predecessor of the current task, then it includes all tasks scheduled after this predecessor on all processors.

To propagate these changes, algorithm 2 recomputes the delay of each communication phase (line 2). Then, it remaps each task ASAP (line 3) with respect of the previous choice considering synchronizations (explained in the previous paragraph). Due to previous tasks' movement on the processor time lines, lines 6-13 shift forward tasks that need to be either re-synchronized, or because one earlier mapped extends itself on it. This process is then repeated until the length of the schedule becomes stable.

Precision of the estimation of communication costs Compared to the ILP formulation, the heuristic does not suffer from the aforementioned over-estimation, thus the communication cost can be computed as accurately as possible using equation (5) from Section 4.2 and the effective amount of data according to tasks' placement. For conciseness, the function to compute the communication cost of the *read* and *write* phases (line 2 in algorithm 2) are not detailed as they are just the application of equations (1) refined with equation (5).

6 Experiments

Experiments were conducted on real code in the form of a subset of the StreamIT benchmark suite [25], as well as on synthetic task graphs generated using the TGFF [11] graph generator.

Applications from the *StreamIT benchmark suite* are modeled using fork-join graphs and come with timing estimates for each task and amount of data exchanged between them. Table 2 summarizes the benchmarks we used for our experiments. We were not able to use all the benchmarks and applications provided in the suite due to difficulties when extracting information (task graph, WCET, ...) or because some test cases are linear chains of tasks with no concurrency. For each benchmark the table includes its number of tasks, the width of the graph (maximum number of tasks that may run in parallel) and the average amount of bytes exchanged between pairs of tasks. All average values given in the rest of the paper are *arithmetic* means.

Task Graph For Free (TGFF) was used when there is a need to generate a large number of task graphs. It is first used to evaluate the quality of our heuristic against the ILP formulation. Due to the intrinsic complexity of solving our scheduling problem using ILP, we need for that experiment small task graphs such that the ILP is solved in reasonable time. TGFF was also used to test our

ALGORITHM 2: AdjustSchedule : Updating the schedule to cope with interference

```
Input  : An incomplete schedule schedule, the list of already mapped tasks Qdone and the newly mapped
         task cur_task
Output : An updated schedule
related_set ← BuildRelated(schedule, Qdone, cur_task)
Compute read/write phases' delay $\forall t \in$ related_set
/* WCET of read/write might have changed, maybe we should move backward/forward some tasks */
Remap task $t \in$ related_set as early as possible according to previous decision regarding synchronization
while schedule.length is not stable do
    Compute read/write phases' delay $\forall t \in$ related_set
    foreach $t \in$ related_set do
        foreach $t' \in$ related_set| start time of $t'$ > start time of $t$ do
            if t and t' are on the same processor $\wedge$ t extends itself on t' then
                Add delay to $t'$ to start after $t$
            else if t and t' are on different processor $\wedge$ are_OL(t,t') $\wedge$ is_synchronized(t') then
                Add delay to re-synchronize $t'$
        end
    end
end
```

heuristic technique for applications larger than the StreamIT benchmarks.

We generated two sets of task graphs: one with relatively small task graphs (referred as STG), and another with bigger graphs (referred as BTG). For both sets, we used the latest version of the TGFF task generation software to generate task graphs with tasks' chains of different lengths and widths, including both fork-join graphs and more evolved structures (e.g. multi-DAGs). Their resulting parameters are presented in Table 3. The table includes for both sets the number of task graphs, their number of tasks, the width of the task graph, the range of WCET values for each task[3] and the range of amount of exchanged data in bytes between pairs of tasks. The TGFF parameters for STG (average and indicator of variability) are set in such a way that the average values for task WCETs and volume of data exchanged between task pairs correspond to the analogous average values for the StreamIT benchmarks.

All reported experiments have been conducted on several nodes from an heterogeneous computing grid with 138 computing nodes (1700 cores). In all experiments T_{slot} is precised, and a transfer rate of one data word (32 bits) per time unit is used.

6.1 Scalability of the ILP formulation

Solving an ILP problem for a mapping/scheduling problem on multi-cores is known to be NP-hard [5]. Thus, the running time of our ILP formulation is expected to explode as the number of tasks grows. To evaluate the scalability of the ILP formulation with the number of tasks, a large number of different configurations is needed, explaining why we used synthetic task graphs for the evaluation. For each task graph in set STG we vary the number of cores in interval $[2; 15]$ and vary T_{slot} in interval $[1; 10]$. With those varying parameters, the total number of scheduling problems to solve is $200 \cdot 10 \cdot 14 = 28000$. The ILP solver used is CPLEX v12.6.0[4] with a timeout of 11 hours.

Figure 3 draws the average solving time per number of tasks in each graph. As expected, when the number of tasks grows, the average solving time explodes, thus motivating the need for a heuristic

[3] The reader may notice that the WCET average value is not perfectly in the middle of the min and max values. This is due to the generation of random numbers in TGFF (pseudo-random, not perfectly random) combined to the limited number of values generated.

[4] https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

Table 2: StreamIT benchmark suite scheduling

Name	#Tasks	Width	avg data (bytes)
802.11a	113	7	7048
audiobeam	20	15	60
beamformer	56	12	96
compCnt	31	8	64
dct_comp	13	3	768
dct_verif	7	2	512
fft2	26	2	1024
fft3	82	16	256
fft4	10	2	8
fft5	115	16	128
fhr	29	7	2048
filterbank	53	8	256

Name	#Tasks	Width	avg data (bytes)
fm	67	20	24
hdtv	150	24	106e4
merge	31	8	64
mp3	116	36	36e3
mpd	201	5	66e3
mpeg2	43	3	101e4
nokia	178	32	1920
perftest4	16	4	66e3
tconvolve	75	36	98e6
tde	15	12	48
vocoder	115	17	240

Table 3: Task graph parameters for synthetic task graphs

	#Task graphs	#Tasks	Width	WCET	Amount of bytes exchanged
		<min,max,avg>			
STG	200	3, 34, 14	1, 11, 3	$[1; 70]$	$[0; 11]$
BTG	1000	9, 687, 228	1, 21, 8	$[8; 999]$	$[0; 70]$

that produces schedules much faster. Similar observations were made on the StreamIT benchmarks, for which an exact solution was found for only 17 out of the 23 benchmarks.

6.2 Quality of the heuristic compared to ILP

The following experiments aim at estimating the gap between makespans of schedules generated by the heuristic (see Section 5.2) opposed to solutions found by the ILP formulation. We expect this gap to be small. To perform the experiments we used the 200 task-graphs from the STG task set with the same parameters' variation as previously: number of cores $\in [2; 15]$ and $T_{slot} \in [1; 10]$. The heuristic is implemented in C++ and CPLEX was configured with a timeout of 11 hours.

Table 4: Degradation of the heuristic compared with the ILP (synthetic task graphs / STG)

% of exact results (ILP only)	degradation <min,max,avg> %
98%	-8%, 43%, 5%

Table 4 summarizes the results. The first column of Table 4 presents the percentage of exact results the ILP solver is able to find in the granted time. We only refer to the exact solutions for the comparison as the feasible ones (i.e not exact) might bias the conclusion on the quality of the heuristic compared to the ILP. The next column presents the minimum, maximum and average degradation in percent, computed using makespans with formula $(heuristic - ILP)/ILP$. Positive values mean a degradation of the heuristic against the ILP formulation, while negative values show an improvement which is due to the over-approximation of the communication delay in the ILP formulation (see Section 5.1).

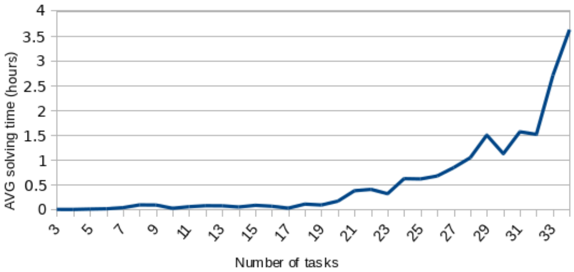


Figure 3: Scalability of ILP formulation (synthetic task graphs / STG)

As we can observe, the average degradation is low, which means our heuristic has acceptable quality. A deeper analysis of the distribution of degradation, not included for space consideration, shows that 80% of the heuristic schedules are less than 10% worse than the ILP formulation solutions. We also observed a schedule generation time far much lower than the ILP, < 1 second on average, with a maximum observed of 2 seconds. Solving time is dependent on the number of re-adjustments the heuristic must perform to cope with effective amount of interference.

The influence of task sorting has a significant impact on the heuristic output. We chose a topological order with random tie breaking as stated in Section 5.2. This choice of sorting algorithm rather than simple explains the under-performance of 43% on the worst-case. We did not include a comparative study on sorting algorithm, considered as out of the scope of the paper.

6.3 Quality of the heuristic compared to basic contention analysis

We estimate the gain when using our method to tighten communication delays over the same heuristic using the pessimistic estimation of interference from equation (1). The higher is the gain the tighter is the proposed estimation of communication delays. The experiment was performed on the StreamIT benchmarks. The target architecture configuration includes 15 cores, and a value of $T_{slot} = 3$ is used as in [18].

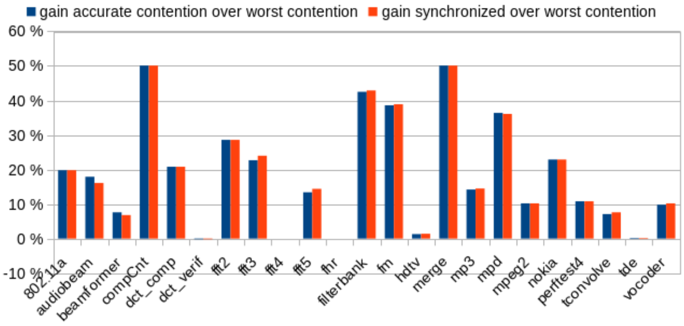


Figure 4: Gain in % obtained by precise contention analysis (heuristic, StreamIT benchmarks)

Results are depicted in Figure 4 by blue bars. The gain is computed using equation 19.

$$\frac{worst\ concurreny - accurate\ interference}{worst\ concurrency} \tag{19}$$

Results show that the gain to use the accurate degree of interference decreases the overall makespan of 19% on average over the worst case concurrency, demonstrating the benefits of precisely computing the degree of interference at schedule time.

6.4 Quality of the heuristic compared to synchronized communication

Recent papers [21, 1, 2] suggested to build contention-free schedules to nullify interference cost. Due to the different task models and system models in the aforementioned works, a direct comparison with them is hard to achieve. Thus, we modified our heuristic to produce a schedule without any contention and to be as close as possible to the ideas defended in the mentioned papers. The gain of a contention-free heuristic against a worst contention one is depicted for the StreamIt benchmarks in Figure 4 by red bars.

Among the contention-aware and contention-free variants of our heuristic, no method outperforms the other for all benchmarks. Moreover, the difference between the schedule makespans using the two variants is very small. The average difference is 0,08%, with a worst value of 0.3%. Synchronized execution (red bars) gives better results for *fft3, fft5, filterbank, fm, hdtv, mp3, tconvolve* and *vocoder*; our proposed heuristic (blue bars) gives better for *audiobeam, beamformer* and *mpd*; the results for all other benchmarks are identical. Regarding schedule generation duration for the StreamIT benchmarks, *contention-free* solutions are found in less than 30 seconds on average, while *contention-aware* once need less than 3 minutes on average. The shortest schedule generation times were obviously observed when generating contention-free schedules, because no estimation of interference costs has to be performed at all. We believe that our contention-aware scheduling heuristic will be better suited to task models in which communications are not separated from calculations, i.e. *non-PREM* task-set. Quantitative evaluation of the obtained benefit for such a task model is left for future work.

6.5 Impact of T_{slot} on the schedule

With our heuristic, we finally studied the influence of the duration T_{slot} on the overall makespan, assuming the overhead negligible when switching between slots. We chose to fall back on synthetic task graphs to benefit from a wider range of different test cases. Here the BTG task set is employed. For each graph, we generated three versions of the same topology but with different amount of exchanged data between tasks to study the influence of the duration T_{slot} on graphs that exchanges few data $[0; 5]$, reasonable amount of data $[5; 15]$ and large amount of data $[15; 70]$. The duration T_{slot} is in the range $[1; 40]$ as it covers all scenarios to exchange data in one or several chunks.

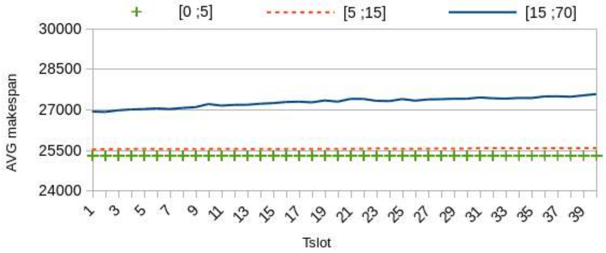


Figure 5: Average makespan when varying T_{slot} (synthetic task graphs / BTG)

To compute the results of this experiments, we set a timeout to 1h, leaving us 75.6% of the initial number of task graphs. Results are presented in Figure 5 where the three curves correspond to the average makespan of each category over the value of T_{slot}. We observe that T_{slot} has very little impact on task graphs with few communications (crossed line). While there is an impact on task graphs with bigger amount of data exchanged (continuous line). The exposed results confirm that it is a better choice to keep this T_{slot} small to reduce the waiting time between each slot even if there are several chunks. This allows small packets of data to be handled faster when in competition with bigger packets.

7 Conclusion

In this work, we show how to take advantage of the structure of a parallel application, along with its target hardware platform, to obtain tight estimates of contention delays. Our approach builds on a precise model of the cost of bus contention for a round-robin bus arbitration policy, which we use to define two scheduling and mapping strategies. Our experimental results show that, compared to a scenario where we account for worst case contention, our approach improves the schedule makespan by 19% on average.

One of the limitation in our approach is its restriction to blocking communications. A natural extension of this work is therefore to relax this constraint and introduce support for asynchronous communications, which are notoriously more challenging to support in a real-time context. Another possible research direction is to further refine the contention model, by more accurately capturing the actual duration of contention phases between communicating tasks. Extensions to architectures with local caches is another direction for future research.

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] 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.

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

[4] 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.

[5] Edward G Coffman Jr, Michael R Garey, and David S Johnson. Approximation algorithms for bin packing: a survey. In *Approximation algorithms for NP-hard problems*, pages 46–93. PWS Publishing Co., 1996.

[6] Dakshina Dasari and Vincent Nélis. An analysis of the impact of bus contention on the WCET in multicores. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 1450–1457. IEEE, 2012.

[7] Dakshina Dasari, Vincent Nelis, and Benny Akesson. A framework for memory contention analysis in multi-core platforms. *Real-Time Systems*, 52(3):272–322, 2016.

[8] RI Davis and A Burns. A survey of hard real-time scheduling algorithms for multiprocessor systems. *in ACM Computing Surveys*, 2011.

[9] 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.

[10] 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.

[11] 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.

[12] Gabriel Fernandez, Jaume Abella Ferrer, Eduardo Qui nones Moreno, Christine Rochange, Tullio Vardanega, Francisco Javier Cazorla Almeida, et al. Contention in multicore hardware shared resources: Understanding of the state of the art. 2014.

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

[14] Michael I Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S Meli, Andrew A Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, et al. A stream compiler for communication-exposed architectures. In *ACM SIGPLAN Notices*, volume 37, pages 291–303. ACM, 2002.

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

[16] Rihani Hamza, Moy Matthieu, Maiza Claire, Davis Robert I., and Altmeyer Sebastian. Response time analysis of synchronous data flow programs on a many-core processor. In *In proceedings of the 24th International Conference on Real-Time Networks and Systems (RTNS 2016)*. ACM, 2016.

[17] Catherine E Jarrett, Bryan C Ward, and James H Anderson. A contention-sensitive fine-grained locking protocol for multiprocessor real-time systems. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, pages 3–12. ACM, 2015.

[18] 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.

[19] Hyoseung Kim, Dionisio De Niz, Bj"orn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 145–154. IEEE, 2014.

[20] Mircea Negrean, Simon Schliecker, and Rolf Ernst. Response-time analysis of arbitrarily activated tasks in multiprocessor systems with shared resources. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 524–529. European Design and Automation Association, 2009.

[21] 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.

[22] Simon Schliecker, Mircea Negrean, and Rolf Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Proceedings of the conference on design, automation and test in Europe*, pages 759–764. European Design and Automation Association, 2010.

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

[24] Richard N Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19(1):57–84, 1983.

[25] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer, 2002.

[26] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM (JACM)*, 9(1):11–12, 1962.

[27] Heechul Yun, Rodolfo Pellizzon, and Prathap Kumar Valsan. Parallelism-aware memory interference delay analysis for cots multicore systems. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 184–195. IEEE, 2015.