Efficient Contention-Aware Scheduling of SDF Graphs on Shared Multi-bank Memory

Hai Nam Tran
University of Brest, CNRS,
Lab-STICC - UMR 6285, France
hai-nam.tran@univ-brest.fr

Alexandre Honorat
Univ. Rennes, INSA Rennes, CNRS,
IETR - UMR 6164, France
alexandre.honorat@insa-rennes.fr

Jean-Pierre Talpin⋆
Inria, Univ. Rennes, CNRS,
IRISA - UMR 6074, France
jean-pierre.talpin@inria.fr

Thierry Gautier
Inria, Univ. Rennes, CNRS,
IRISA - UMR 6074, France
thierry.gautier@inria.fr

Loïc Besnard
Univ. Rennes, Inria, CNRS,
IRISA - UMR 6074, France
loic.besnard@irisa.fr

***Abstract*—Novel memory architectures have been introduced in multi/many-core processors to address the performance bottle neck due to shared memory accesses. Taking the advantages brought by these architectures in scheduling analysis is still an open challenge. In this article, we present a scheduling analysis technique that exploits a shared multi-bank memory architecture to efficiently schedule parallel real-time applications modeled as synchronous data flow (SDF) graphs by minimizing the memory access contentions. Our approach aims at producing a static time-triggered schedule with the objective of minimizing the makespan and buffer size requirements while respecting consistency and data dependency constraints. An Integer Linear Programming formulation of the scheduling problem is presented, as well as a heuristic with significantly lower time complexity. Experimental results are given using synthetic SDF graphs generated by the SDF3 tool and applications available in the StreamIt benchmark.**

I. Introduction

The increasing complexity of signal, image, and control processing applications demands higher computational power and performance to meet real-time constraints [1]. Multi/many-core processors with novel memory architectures are becoming applicable platforms for executing these applications as they provide both performance and power efficiency. At the moment, they demand the adoption of new programming paradigms to support efficient exploitations. Indeed, processes and threads have been used for decades by programmers, showing advantages but also limitations. Because of shared memory synchronization, processes and threads can yield unexpected behaviors, non-deterministic execution or deadlocks.

The data flow programming paradigm has been introduced to address these limitations. This model is commonly used in embedded system design to describe stream processing or control applications. Its simplicity allows the adaptation of automated code generation techniques to limit the problematic and error-prone task of programming real-time parallel applications. Among data flow models of computation, synchronous data flow (SDF) [2] is one of the most popular in the embedded system community. This model is deterministic in terms of communications: the topology and the amount of data sent and received are known (fixed). Communications between processing elements are clearly isolated through well identified FIFO channels. Task parallelism [3] in SDF can be exploited systematically: tasks can be mapped to available processing elements, dependencies can be enforced, and tasks on different branches can execute in parallel.

Problem statement: Scheduling analysis on multi/many-core platforms must account for the non-negligible delay due to accesses contention on shared memory [4]. Experimental results in [5] show that the execution time can be increased by a factor up to 8. Preliminary works on this scheduling problem only consider shared memory architectures with a single memory bank. They do not fully exploit the advantages of shared multi-bank memory as well as the deterministic communications exposed in SDF graphs to optimize memory access contention.

In addition, they conventionally involve the transformation of an SDF graph into its equivalent homogeneous form (HSDF). While this transformation helps fully expose task and data parallelism of the graph, it also presents several limitations such as an exponential increase of actors, memory consuming schedules and high buffer size requirements. A recent evaluation in [6] has shown that scheduling a partially expanded graph could be as efficient as its HSDF expansion. That motivates our proposal of a solution that explores the design space with the SDF representation.

Contribution: We propose a two-step approach to compute an efficient contention-aware scheduling of SDF graphs on one compute cluster consists of processing elements (cores) and a shared multi-bank memory. First, we compute an abstract schedule defining the dependencies amongst actor firings with the objective of minimizing buffer size requirements. Second, we construct an optimal time-triggered schedule that minimizes the *makespan*, which is the time difference between the start of the first scheduled actor and the end of the last scheduled actor in one iteration of the SDF graph [7]. Our approach aims at creating a time-triggered schedule that achieves a minimal makespan while trying to reduce the total

⋆ Jean-Pierre Talpin is partially supported by Nankai University and by the National Science Foundation of China, under grant 61672074

amount of buffer size requirements. We study a shared multi-bank memory architecture presented in [8]. This particular architecture already exists in industrial platforms such as the Kalray MPPA [9] many-core architecture.

The rest of this article is organized as follows. Section II presents related work and positions our contribution. Section III describes our system model, scheduler specification, and provides a brief description of our resource sharing platform model. Section IV provides an overview of our approach. Section V introduces our method of computing an abstract schedule that minimizes buffer space requirement. Then, section VI presents our approach of computing a time-triggered schedule with the objective of optimizing the makespan. The approach is evaluated in section VII with synthetic SDF graphs and applications taken from the StreamIt benchmark [10]. Finally, section VIII concludes the article and discusses future work.

II. Related Works

Our work is closely related to existing works in two domains: (1) scheduling SDF graphs onto multi/many-core architectures and (2) contention-aware real-time scheduling analysis.

First, we discuss SDF graphs scheduling. In [11], the authors presented a scheduling synthesis approach for SDF graphs on partitioned multi-core platform. The approach accounts for both functional properties (mapping) and timing parameters (period, release time and priority). In this work, the author modeled communication channels but did not take into account communication cost. Several conditions are described to guarantee the consistency of SDF graphs. While considering time-triggered scheduling rather than fixed priority and dynamic priority, our work tackles the same problem statement and can be considered as an extension of [11].

The work in [12] addressed the multiprocessor scheduling of synchronous programs under a time-triggered bus architecture. The authors assumed a model of communication in which the bus can only transmit one message at a time and collision of messages implies infeasible schedule. On the contrary, our work assumes a more complex communication model of multiple bus arbiters and accounts for contention.

In [13], the authors considered an implementation of data flow applications on multi-core systems based on the previous work in [14]. The authors focus on the optimization of an offline schedule taking into account non-functional properties such as start time and deadline. This work aims at eliminating the interference on shared resources by providing temporal isolation. In our approach, we account for the interference and hence may be complementary to this previous work.

A large proportion of works presented next are related to the Kalray MPPA many-core architecture. In this architecture, there are 256 processing cores organized in 16 compute clusters (16 cores per cluster) connected by a Network-on-Chip (NoC). These works are classified into either multi-cluster [15]–[18] or single-cluster [8], [19]–[21] context.

In [15], the authors presented an approach of mapping and scheduling *well-formed split-join* SDF graphs on all compute clusters of the Kalray MPPA by using a satisfactory modulo theory (SMT) solver. The well-formed split-join restriction in [15], which is relaxed in our work, facilitates the HSDF expansion and buffer size computation. Later, the work in [16] presented an optimization for the schedule generated by the SMT solver developed in [15]. The authors proposed an approach to identify and exclude the pessimistic estimation of contention by detecting the set of actors that cannot access the same resource at the same time. In our work, we use integer linear programming to find the optimal solution and also provide a heuristic with lower time complexity.

On the topic of contention-aware scheduling, the authors in [8] described a response time analysis for single-period application modeled as a directed acyclic graph (DAG) on a shared multi-bank memory architecture. This work applies to a single cluster of the Kalray MPPA. It is assumed that both mapping and execution order on each core are known. Later, [20] described a scheduling refinement technique that can reduce contention in the schedule of [8] by introducing processor idle time. In comparison with these works, our approach takes into account both mapping and scheduling problem instead of only scheduling.

In [17], the authors proposed a mixed-criticality scheduling response time analysis on the all compute clusters of the Kalray MPPA. The main difference with our approach is that the authors consider a "Flexible Time-Triggered Scheduling" model which divides time into frames and force a global synchronization barrier between frames. As pointed out in [8], this potentially creates core under-utilization while waiting for the barrier. Our proposed scheduling policy does not require any global barrier but is limited to only one compute cluster.

The work in [18] focused on the problem of mapping large multi-cluster applications consisting of periodic tasks while maintaining a strong temporal isolation from co-running applications. The application model and optimization objective in this work is different from our approach.

An ILP formulation and a heuristic aiming at scheduling periodic independent PRedictable Execution Model (PREM) based tasks [22] in a single cluster of the Kalray MPPAis proposed in [19]. The authors systematically create a contention-free schedule. Our work differs in the considered application model as well as the goal to reach. The authors consider sporadic independent tasks to which they aim at finding a valid schedule that meets deadlines. In contrast, we consider one iteration of an SDF graph and aim at finding the shortest schedule.

The authors in [21] also presented an ILP formulation and a heuristic to schedule and map DAG graphs in which each node is a PREM-based task [22] with the objective of minimizing the makespan. The authors took memory access contention into account but only assumed a single bank memory architecture. Comparing to this work, we assume a simpler task model while accounting for a more complex application model and memory architecture.

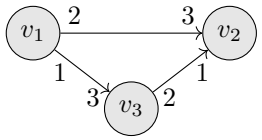

Fig. 1. A simple acyclic delayless SDF graph

We position our work in the single cluster context as the ones in [8], [19]–[21]. Comparing to the state of the art, the novelty of our work lies at the exploitation of the shared multi-bank architecture and the well determined communication model in SDF graphs to optimize memory access contention.

III. System model and assumptions

In this work, given an *acyclic delayless* synchronous data flow (SDF) graph, our aim is to compute the mapping and scheduling of one iteration of the graph on a resource-sharing platform model consisting of processing cores and a multi-bank shared memory. We first introduce the SDF model and present our scheduler specification in section III-A and III-B. In addition, we detail our assumptions regarding the acyclic and delayless property of SDF graphs. We also explain our motivation behind the choice of such model and scheduler. Then, we describe our resource sharing platform model in section III-C and illustrate the problem of memory access contention in section III-D.

A. Synchronous Data Flow Model

A synchronous data flow (SDF) graph is a directed graph $G = (V, E)$ consisting of a finite set of *actors* $V = \{v_1, ..., v_N\}$ and a finite set of one-to-one *channels* E. A channel $e_{ab} = (v_a, v_b, p, q) \in E$ connects the producer v_a to the consumer v_b such that the production (resp. consumption) rate is given by an integer $p \in \mathbb{N}$ (resp. $q \in \mathbb{N}$). Every time an actor fires, it consumes a fixed number of tokens from its input channels and produces a fixed number of tokens to its output channels.

An actor v_a is characterized by its execution profile denoted as $C_a = \{\text{PD}_a, \text{MD}_a\}$. In contrary to the notion of worst-case execution time (WCET) used in previous literature, the execution profile is decoupled into processor demand and memory demand. The processor demand (PD_a) denotes the computation or CPU time of v_a, without considering the time spent on fetching data from the memory. The memory demand (MD_a) denotes the number of memory accesses per actor firing. Such decoupling of the WCETs is proved to be feasible on fully timing compositional platform [23] such as the Kalray MPPA [8].

A channel e_{ab} has a bounded buffer size $\delta(e_{ab})$. If the buffer sizes for certain channels are too small, the SDF graph could deadlock. For example, with the graph in Figure 1, the channel e_{12} connecting v_1 and v_2 needs a minimum buffer size of 6. The reason is that v_1 must be fired at least 3 times before v_3 is fired and then v_2 can be fired. If $\delta(e_{12}) = 5$, the graph deadlocks after two firings of v_1 and no actors can be fired. There can be a number of initial tokens associated with each channel. It is called the delay of a channel since it can induce an offset in the execution of the consumer in relation to the producer [24]. We study only *delayless* SDF graphs in which the number of initial tokens on all channel is 0.

Definition 1 (Iteration [24]): The iteration of an SDF graph is a non-empty sequence of firings that returns the graph to its initial state. Two concepts that are related to an iteration of the graph are defined below.

Definition 2 (Makespan [7]): The makespan of an SDF graph is the time difference between the start of the first scheduled actor and the end of the last scheduled actor in one iteration of the SDF graph.

Definition 3 (Repetition Vector [24]): The repetition vector of an SDF graph is an array of length equal to the number of actors in SDF, such that if each actor is invoked for the number of times equal to its entry, the number of tokens on each edge of SDF remains unchanged.

For the graph in Figure 1, firing actor v_1 3 times, actor v_2 2 times and actor v_3 1 time forms an iteration. The repetition vector $\overrightarrow{z} = (3, 2, 1)$ indicates the number of actor firings per iteration. If such a vector exists, then the graph is said to be consistent. In this paper, we study only consistent acyclic SDF graphs. Inconsistent graphs are of less importance since they cannot be implemented with bounded memory without deadlock. The repetition vector can be computed efficiently by applying a depth-first search algorithm presented in [24].

B. Scheduler specification

In this work, we assume a *non-preemptive time-triggered scheduler* in a *partitioned* scheme of actor assignment. The term time-triggered scheduling refers to the fact that the overall behavior of the system is controlled by a recurring clock tick, the only event within the system which can trigger an action. There is an a-priori global schedule, which is computed offline, specifying the release times of all tasks. Time-triggered scheduling especially fits the needs of safety-critical applications. This special type of scheduling proved to be successful in such diverse areas as automotive electronics and aerospace industry [25]. Regarding the scheduling policy in each core, we consider a non-preemptive scheduler and hence do not deal with cache-related preemption delay or context-switch overhead.

Actors are assigned to a core at design time and are not allowed to migrate from theirs assigned core to another one at runtime. It implies that multiple simultaneous firings of an actor cannot be executed in parallel. As accounting for auto-concurrency can lead to a complex contention-aware analysis, it is not considered in this article. Interested readers can refer to [6], [26] for further detail.

C. Resource-Sharing Platform Model

We study the shared multi-bank memory introduced in [8]. This particular architecture already exists in industrial platforms such as the Kalray MPPA [9].

A compute cluster $\mathcal{P}$ consists of m homogeneous processing elements (cores), $\mathcal{P} = \{P_1, ..., P_m\}$. Each core in $\mathcal{P}$ has access

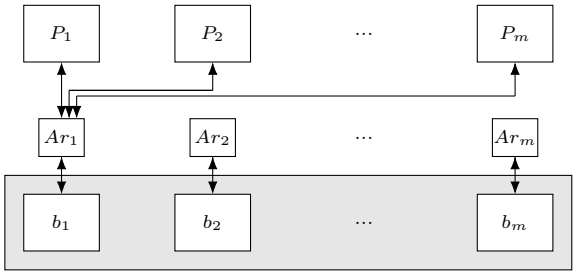

Fig. 2. Shared multi-bank memory architecture. Only buses connecting cores and arbiter 1 (Arb 1) are illustrated.

to a private cache memory and to a local shared memory $\mathcal{M}$. The shared memory $\mathcal{M}$ is divided into m memory banks, $\mathcal{M} = \{b_1, ..., b_m\}$, as illustrated in Figure 2. Each bank has an equal fixed size, denoted as s. Each core has a private path (bus) to the shared memory and each memory bank is accessed via a separate bus arbiter. The private paths of the cores are connected to all bank arbiters, as depicted abstractly for Arb 1 in Figure. 1. Under these assumptions, two concurrently executed tasks on different cores can perform parallel accesses to the shared memory without delaying each other provided that they access different banks.

Only one core can access a given bank at a time and once granted, bank access is completed within a fixed time interval noted memory delay (d), which is the same for read/write operations and for all banks. In the meantime, pending requests to the same bank from other cores stall execution on their cores until they are served.

D. Memory Access Contention

In our model, each actor has a local memory buffer, and the buffers of all actors running on the same core are mapped to the same memory bank. Buffer spaces for all input channels of an actor are allocated in the memory bank of this actor. Under these assumptions, read accesses are private but write requests may access to another core's memory bank. Contention may occur on the memory bus when two cores access to the same memory bank. This assumption is indeed an arbitrary choice. We are aware that optimizing buffer spaces allocation to minimize memory access contention is an interesting problem but it complex to tackle at the same time with the task mapping and scheduling problem.

Access requests from the cores inside one compute cluster are subject to a round-robin arbitration. For example, we assume that two firings of actors v_a and v_b access to the same memory bank and execute in parallel. In the absence of detailed information on the pattern of access requests within an actor, we consider that any two firings that overlap in time can interfere on each of their accesses. The memory contention between two firings, denoted as $\gamma_{a,b}$, is given by:

$$\gamma_{a,b} = \min(\text{MD}_a, \text{MD}_b) \cdot d \tag{1}$$

We take a simple example to illustrate the contention-aware scheduling problem on a shared multi-bank memory in Figure 3. Actors v_1 and v_2 are executed in parallel. Actor v_1 has

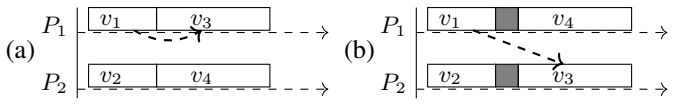

Fig. 3. (a) v_1 only accesses to memory bank 1, there is no contention (b) v_1 accesses to memory bank 1 and 2, memory access contention between v_1 (write) and v_2 (read) on bank 2

write accesses to the memory bank of actor v_3. In the (a) case, actor v_3 is mapped to core P_1 and there is no contention. In the (b) case, actor v_3 is mapped to core P_2. Thus, there is contention between actor v_1, which writes data to bank 2, and actor v_2, which reads data from bank 2. This contention can be effectively optimized when we compute the schedule.

In this section, we have introduced our system model and assumptions taken. In the next sections, we present our approach of computing an efficient contention-aware scheduling of SDF graphs on shared multi-bank memory.

IV. General Approach

Considering SDF graphs, we need to allocate storage spaces and schedule the actors so that deadlock, overflow and underflow do not occur. In addition, we need to compute timing parameters, processor mapping and to take into account the contention due to shared memory accesses. If all these variables are considered in a monolithic optimization problem, it would result in a very complex set of constraints and would be practically solvable for only a very small number of actors. From this observation, we propose a two-step approach in which each step focuses on solving a subset of problems.

First, we compute an abstract schedule defining firing dependencies between actors in one iteration of an SDF graph with the objective of minimizing storage space requirement. It is abstract in the sense that the schedule neither depends on processor mapping nor real-time scheduling, which are taken into account later. We show in the next section that our abstract schedule can be computed by simply solving the minimum buffer size scheduling problem [27].

Second, from the computed abstract schedule, we synthesize a time-triggered schedule that minimizes the makespan. This schedule defines processor mapping and timing parameters of all actor firings in one iteration of the SDF graph. Contention due to shared memory accesses is considered when we compute the schedule. To achieve the objective, we propose two solutions: an integer linear programming formulation that can be used to find the optimal schedule and a heuristic with significant lower complexity.

V. Abstract Schedule

In this step, first, we need to compute the minimum buffer sizes required for the channels. Second, based on the computed buffer sizes, we enforce the constraints related to firing dependencies so that the schedule will guarantee that neither overflow nor underflow will occur. At this step, we do not compute the precise mapping or timing activation of actors. Only the dependencies between actor firings are computed.

Min buffer size analysis: Given a consistent acyclic SDF graph with its repetition vector, there exists schedules that

result in different buffer size requirements. It has been proved in [24] that the problem of computing a minimum buffer schedule for even an arbitrary SDF graph is NP-complete. However, a heuristic with polynomial complexity has been given in [27] where the authors established its optimality for acyclic, delayless SDF. To keep the article concise, we do not detail the algorithm. Interested readers can refer to [24] or [27] for a simple pseudo code and details. This algorithm helps us computing the minimum buffer size that allows for a schedule without deadlock.

After buffer sizes for all channels are identified, we can compute the firing dependencies between actors by applying the two analysis belows:

Underflow analysis: Underflow happens when an actor attempts to read and there are not enough tokens on the channel. Thus, we need to compute the firing dependencies that guarantee no overflow. For a channel $v_a \xrightarrow{p \quad q} v_b$, the n^{th} firing of v_b (denoted $v_b[n]$) is enabled if and only if the number of produced tokens is larger than $q \cdot n$. Hence, v_b has to wait for the l^{th} firing of v_a (denoted $v_a[l]$) such that:

$$l \cdot p - n \cdot q \geq 0 \tag{2}$$

The firing dependency between v_a and v_b is formalized by the following equation:

$$v_b[n] \geq v_a[l] \text{ with } l = \left\lceil \frac{n \cdot q}{p} \right\rceil \tag{3}$$

Overflow analysis: Overflow happens when an actor attempts to write and there are not enough empty spaces on the channel. For a channel $v_a \xrightarrow{p \quad q} v_b$ of size $\delta(e_{ab})$, the l^{th} firing of v_a (denoted $v_a[l]$) is enabled if and only if the number of produced tokens is smaller than or equal to the number of empty spaces. Hence, v_a has to wait for the n^{th} firing of v_b (denoted $v_b[n]$) such that:

$$l \cdot p - n \cdot q \leq \delta(e_{ab}) \tag{4}$$

The firing dependency between v_a and v_b is formalized by the following equation:

$$v_a[l] \geq v_b[n] \text{ with } n = \left\lceil \frac{l \cdot p - \delta(e_{ab})}{q} \right\rceil \tag{5}$$

At the end of this step, by applying the analysis based on equations 3 and 5 on the SDF graph in Figure 1, we have the following result:

Repetition Vector : $\overrightarrow{z} = (3, 2, 1)$
Buffer Size : $\delta(e_{12}) = 6,\ \delta(e_{13}) = 3,\ \delta(e_{32}) = 2$
Firing Dependencies : $v_1[2] > v_2[1]; v_1[3] > v_2[2]$; $v_1[3] > v_3[1]; v_3[1] > v_2[1]; v_3[1] > v_2[2]$

Firing dependencies between $v_a[l]$ and $v_a[l+1]$ is implicit. We recall that in our scheduler specification, auto-concurrency is not allowed. The l^{th} firing of actor v_a must finish before the $(l+1)^{th}$ firing starts.

The abstract schedule above requires a total buffer size of 11 and guarantees no overflow and underflow exception. However, this schedule does not precise the mapping and timing parameters of actors in the SDF graph. In the next step we present our approach to build a concrete schedule from the abstract one.

VI. Contention-Aware Schedule

This section presents two solutions to compute a concrete schedule and mapping from an abstract schedule. The first solution adopts an Integer Linear Programming formulation of the problem. The optimal schedule can be found with this solution. The second solution is a heuristic based on LIST scheduling, which allows us to find a valid schedule fast and generally close to the optimal one. The main outcome of both solutions is a static mapping and schedule for each core.

As we assume a time-triggered scheduler, it requires us to compute the timing parameters for all actor firings in one iteration of the SDF graph. In other words, we need to treat each actor firing as a real-time task. We first detail our transformation of actor firings to tasks in Algorithm 1.

Algorithm 1: Actor firings to tasks transformation

Input: SDF graph G, abstract schedule $\mathcal{S}_A$
Output: Task set $\mathcal{T}$

```
1 T ← ∅
2 foreach v_a ∈ G do
3     for l = 1 to z⃗(v_a) do
4         Create task τ_i
5         τ_i.C_i ← C_a
6         τ_i.A_i ← v_a
7         τ_i.prd(i) ← prd(v_a[l]) ∪ v_a[l − 1]
8         T = T ∪ {τ_i}
9 return T
```

Each firing of an actor v_a in one iteration of the SDF graph is mapped to a real-time task τ_i (line 2-8). We recall that one iteration of the SDF graph is defined by its repetition vector $\overrightarrow{z}$. Hence, the loop at line 3 is ranging from 1 to $\overrightarrow{z}(v_a)$, which is the number of firings of v_a in one iteration. The execution profile C_i of τ_i is identical to the one of the actor (line 5). The variable A_i is used to save the actor that τ_i represents. It is used in our algorithm to enforce partitioned mapping as an actor can only be mapped to one processor. Then, all firings of the actor must be on the same processor. The precedence dependencies of τ_i are computed by taking into account firing dependencies of $v_a[l]$, denoted $prd(v_a[l])$ (line 7). In addition, τ_i must start after the firing $v_a[l-1]$ with regard to our assumption about non-auto-concurrency.

This transformation effectively transforms our original problem into a scheduling problem of real-time tasks with dependencies. Next, our integer linear programing (ILP) formulation of the problem is presented.

A. Integer Linear Programming formulation

An Integer Linear Programming (ILP) problem consists of a set of integer variables constrained by linear inequalities.

TABLE I
DATA AND VARIABLES FOR THE ILP

Notation	Type	Description
Input Data		
$\mathcal{P} = c_1, ..., c_m$	Set	Set of cores
$\mathcal{T} = \tau_1, .., \tau_n$	Set	Set of tasks
A_i	Integer	τ_i is a firing of actor A_i
$prd(i)$	Set	Precedence dependencies of τ_i
$snk(i)$	Set	Set consumers of v_i in SDF graph
PD_i	Integer	Processor demand of τ_i (cycles)
MD_i	Integer	Memory demand of τ_i (# of accesses)
MR_i	Integer	Memory requirement of τ_i (# bytes)
d	Integer	Constant memory access delay
s	Integer	Memory bank size
Output Variables		
$Start_i$	Integer	Start time of task τ_i
End_i	Integer	Completion time of task τ_i
$p_{i,k}$	Binary	Task τ_i is mapped to core c_k
Internal Variables		
R_i	Integer	Response time of task τ_i
$o_{i,j}$	Binary	τ_i and τ_j are overlapped
$c_{i,j}$	Binary	τ_i and τ_j are on the same core
$a_{i,j}$	Binary	τ_i and τ_j access to same mem bank
$t_{i,j}$	Binary	τ_i and τ_j have memory interference
$b_{i,k}$	Binary	Task τ_i has access to bank k
$\gamma_{i,j}$	Integer	Mem access delay between τ_i and τ_j

Solving a problem consists in optimizing an objective which is a linear function of the variables. In our case, the objective is to obtain the shortest schedule while scheduling the real-time tasks on a fixed number of processors and accounting for memory contention. Table I summarizes the notations and variables needed by the ILP formulation.

In order to keep the presentation of ILP formulations short and concise, we use two logical operators $\vee$ and $\wedge$ directly when describing the constraints. These operators can be transformed into linear constraints using simple transformation rules [28]. Some ILP solvers such as IBM CPLEX support automatic transformation of these operators.

Objective function: Our goal is to minimize the makespan of the schedule. In other words, it is minimizing the end time of the last scheduled task. The objective function, given in equation 6, is to minimize the makespan Θ. Equation 7 constrains the completion time of all tasks to be inferior or equal to the makespan.

$$\text{Minimize: } \Theta \tag{6}$$

$$\Theta \geq End_i, \ \forall \tau_i \in \mathcal{T} \tag{7}$$

Problem constraint: Basic rules of a valid schedule are expressed in the following equations. Equation 8 enforces that a task is mapped on exactly one core. Equation 9 enforces that all tasks representing firings of an actor must be mapped to the same core. Equation 10 enforces that the completion time of a task is equal to its start time plus the response time R_i.

$$\sum_{k=1}^{m} p_{i,k} = 1, \ \forall \tau_i \in \mathcal{T} \tag{8}$$

$$p_{i,k} = p_{j,k}, \ \forall \tau_i, \tau_j \,|\, A_i = A_j \tag{9}$$

$$End_i = Start_i + R_i \tag{10}$$

Dependency constraints: Equation 11 enforces data dependencies, a task τ_i can only be released when all tasks $\tau_j \in prd(i)$ are completed. The precedence dependencies of the tasks are computed from the abstract schedule as we mention in Algorithm 1.

$$Start_i \geq End_j, \forall \tau_j \in prd(i) \tag{11}$$

Absence of overlapping on the same core: Equations 12, 13 and 14 forbid the overlapping of two tasks when mapped on the same core. This constraint means that task τ_i cannot overlap any task τ_j running on the same processor as τ_i. The overlapping condition of two intervals $[Start_i, End_i]$ and $[Start_j, End_j]$, denoted $o_{i,j}$, is computed by Equation 13. The condition of whether τ_i and τ_j are mapped on the same core is computed by Equation 14. We note that the value of $c_{i,j}$, which is a binary variable, can only be either 1 or 0 as the unicity of task mapping is guaranteed by Equation 8.

$$\mathrm{o}_{i,j} \wedge c_{i,j} = 0, \ \forall \tau_i, \tau_j \in \mathcal{T}; i \neq j \tag{12}$$

with

$$o_{i,j} = (Start_i < End_j) \wedge (End_i > Start_j) \tag{13}$$

$$c_{i,j} = \sum_{k=1}^{m} (p_{i,k} \wedge p_{j,k}) \tag{14}$$

Bank access constraints: Equations 15 and 16 imply that if task τ_i is mapped to core p_k, it has accesses to memory b_k. In addition, τ_i also has write accesses to the memory bank of the consumers of actor A_i. We emphasize the fact that we use the consumer of actor A_i which is extracted from the SDF graph in this constraint. Equation 17 determines whether two tasks access to the same memory bank.

$$b_{i,k} = p_{i,k} \tag{15}$$

$$b_{i,k} = p_{j,k}, \forall \tau_j \,|\, A_j \in snk(i) \tag{16}$$

$$a_{i,j} = \bigvee_{k=1}^{m} (b_{i,k} \wedge b_{j,k}) \tag{17}$$

Equation 18 enforces that the total memory space requirement of actors mapped to one processor cannot be larger than the size of the memory bank associated to this processor. We assume that channels are allocated in the memory bank corresponding to the processing core where the consumers are mapped to. Because all tasks (firings) of an actor are mapped to the same processor, we only need to choose one task of each actor to generate this constraint.

$$\sum_{A_i \in G} p_{i,k} \cdot MR_i \leq s, \ \forall k \in [0, m] \tag{18}$$

Computing memory access interference: Equation 19 computes whether two tasks τ_i and τ_j that are overlapped in time also access to the same memory bank. Equation

20 computes the memory access interference between two tasks. Finally, the response time R_i is computed by applying Equation 21.

$$t_{i,j} = o_{i,j} \wedge a_{i,j} \tag{19}$$

$$\gamma_{i,j} = t_{i,j} \cdot \min(MD_i, MD_j) \cdot d \tag{20}$$

$$R_i = PD_i + MD_i \cdot d + \sum_{\tau_j \in \mathcal{T}} \gamma_{i,j} \tag{21}$$

Processor symmetry break: The presence of symmetry is common in many job-shop scheduling type problems that consider multiple identical machines. We apply a technique presented in [29] to break the processor symmetry in our problem. We let processor 1 run actor v_1 then processor 2 run the lowest index actor not running on processor 1, etc. This constraint is added to limit the search space and improve the performance of the ILP solver. It can be easily seen that the cost regarding buffer space requirements and memory contentions remains unchanged with such restrictions.

$$p_{i,k} \leq \sum_{j=1}^{i-1} p_{j,k-1}, \ \forall (i,k), i \geq k \geq 2 \tag{22}$$

The result of the scheduling problem after being solved by an ILP solver is then defined by two sets of variables. Task mapping is defined by variables $p_{i,k}$. The static schedule on every core is defined by variables $Start_i$ and End_i, which define the start and end time of tasks.

B. Heuristic

The basic idea of the proposed heuristic, based on list scheduling, is to order tasks from an SDF graph, find a set of ready tasks and then to schedule each task one by one without backtracking, while keeping the goal of minimizing the overall makespan of the schedule. Tasks in all queues are sorted in a topological order based on the position of their corresponding actors in the SDF graph. We are fully aware that it is an arbitrary choice; however, task ordering is beyond the scope of this article.

LIST Scheduling: The heuristic is sketched in Algorithm 2. In order to keep the description short and concise, we only detail important functions of the algorithm while providing a short description for other functions.

The input of the algorithm is an abstract schedule and a set of processors. At the beginning all tasks are in the waiting queue (QWait); other queues are empty (lines 1-4). Then a loop iterates until all tasks are scheduled (lines 5-17).

Inside the loop, we first find tasks that can be scheduled immediately without any dependencies to be satisfied and put them into the ready queue (QReady). Function GetReadyQueue (line 6) computes this information based on tasks in the waiting queue and complete queue. Another loop iterates while there exists tasks ready to be scheduled (lines 10-17).

Algorithm 2: List Scheduling

Input: Abstract schedule $\mathcal{S}_\mathcal{A}$, set of processors $\mathcal{P}$
Output: A concrete schedule: $\mathcal{S}_\mathcal{C}$

```
1  QWait ← GetWaitingQueue()
2  QReady ← ∅
3  QComplete ← ∅
4  S_C ← ∅
5  while QWait ≠ ∅ do
6      QReady ← GetReadyQueue(QWait,QComplete)
7      QWait = QWait \ QReady
8      foreach τ_i ∈ QReady do
9          tmpSched.makespan ← ∞
10         foreach p ∈ P do
11             pSched ← S_C
12             MapEarliestStart (pSched, τ_i, p)
13             UpdateSchedule (pSched, τ_i, p)
14             tmpSched ← MinMakespan(tmpSched,
                pSched)
15         S_C ← tmpSched
16         QReady = QReady \ {τ_i}
17         QComplete = QComplete ∪ {τ_i}
18 return S_C
```

Function MapEarliestStart (line 12) tries to schedule the task as early as possible on a processor. This heuristic uses an As Soon As Possible (ASAP) strategy when mapping a task. It is important to note that task mapping has an effect on memory contention. After scheduling each task, the contention 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.

This problem is illustrated in Figure 4. We assume that τ_1 and τ_2 both produce data to τ_3. Before the mapping of task τ_3 on processor P_1, there is no contention between two tasks τ_1 and τ_2. However, after τ_3 is mapped to processor P_1, because τ_1 and τ_2 both write to the memory bank of τ_3, there must be contention added to the execution time of the two tasks. As a result, we have to recompute the response time in order to account for the added delay.

Function UpdateSchedule (line 13), which is described in Algorithm 3, handles this problem. This algorithm is based on the contention-aware multi-core response time analysis presented in [8].

Functon MinMakespan (line 14) compares the new schedule with the previously constructed schedule and returns the one with shorter makespan. Our algorithm chooses to map a task on a core that results in a schedule with the shortest makespan.

When a task is scheduled, it is removed from ready queue (line 16) and moved to complete queue (QComplete, line 17). The heuristic terminates when all tasks are scheduled.

Update Schedule: Algorithm 3 computes the memory contention that can delay a given task. The implementation of function GetContention is based on Equation 20. Since the

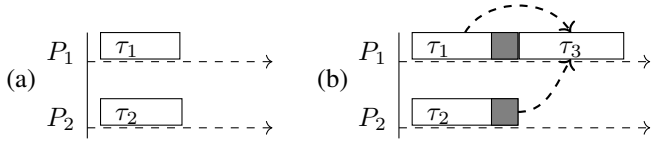

Fig. 4. (a) The schedule before the mapping of τ_3, there is no contention. (b) The schedule must be recomputed after τ_3 is mapped to processor P_1 because τ_1 and τ_2 write data to the memory bank of τ_3

Algorithm 3: Update Schedule

Input: τ_i, pSched, p
Output: Updated Schedule

```
prd(i) = prd(i) ∪ GetLastTask(p)
do
    do
        R'_i ← R_i
        foreach τ_i ∈ pSched' do
            R_i = PD_i + MD_i * d +
              GetContention(τ_i, pSched)
    while (exist τ_i | R_i ≠ R'_i);
    Start'_i ← Start_i
    foreach τ_i ∈ pSched do
        Start_i = max(Start_j + R_j | j ∈ prd())
while exist τ_i | Start_i ≠ Start'_i;
return pSched
```

potential contention depends on the start time ($Start_i$) and response time (R_i), and the response times depend on the interference, this requires a fixed-point iteration. After computing the response times, the schedule we get may not respect the firing dependencies and sequential constraints. We modify the start time so that each task is released immediately after each of the tasks it depends on is guaranteed to have completed. Modifying the start times may change the contention, hence we have to recompute it until a fixed point is reached. The proof of convergence for these fixed-point iterations can be found in [8].

The output of the heuristic is a time-triggered schedule that gives the start times and response times of tasks on each core. Indeed, the makespan can be easily computed from this schedule. In the next section, we compare the ILP formulation and the proposed heuristic by evaluating their scalability and the computed makespans.

VII. Implementation & Evaluation

Implementation of our approach is realized in ADFG [30], which is a real-time scheduling analysis tool for data flow graphs. SDF graphs model and several consistency verification methods are already provided by ADFG, which gives us a good facility to integrate our contribution. We extended the tool so that it computes the abstract schedule and exports constraints that can be presented to an ILP solver. The ILP solver used is IBM CPLEX v12.7.1 with a timeout of 1 hour. The solver stops after one hour even if the optimal solution is not found.

TABLE II
Parameters of synthetic SDF graphs

	# Actors	# Firings	# Cores	Width	Rate
Small Set	[5,15]	[10,40]	4	[1,5]	[1,5]
Large Set	[40,60]	[100,1000]	16	[1,16]	[1,30]

Experiments have been conducted to evaluate our approach, with the particularity of using shared multi-bank memory. In section VII-A, we evaluate the scalability of the ILP solution and the heuristic with regard to the size of the scheduling problem. Then, in section VII-B, we evaluate the quality of the heuristic by comparing its result to the ILP solution. We also provide a comparison between single-bank and multi-bank memory in section VII-C.

We performed experiments with synthetic SDF graphs generated by the SDF3 graph generator [31] as well as a subset of applications taken from a refactored version of StreamIt benchmark [32]. SDF3 tool allows us to generate SDF graphs with various parameters such as: number of actors, number of firings per iteration of the SDF graph and actor's WCET. In addition, the generator allows us to set the max width of the graph and choose whether a graph is acyclic or not.

The general configuration of synthetic graphs is as follows. We generated two sets of synthetic SDF graphs as described in Table II. The small set is used to evaluate the ILP solution's scalability as well as comparing the ILP solution with the heuristic. The large set is used to evaluate the heuristic's scalability. For each set, 100 graphs were generated. Experiments with small set and large set are conducted with different numbers of cores. This choice is made mostly because SDF graphs in the small set do not benefit from a high number of cores.

Actors in synthetic SDF graphs were generated so that their memory demand made up 10% to 30% of their WCETs. For actors taken from the refactored StreamIt benchmark [32], memory requirement is computed by their generated code size, local variables and buffer spaces of input channels. In addition, memory demand of an actor is computed based on its production/consumption rate and the amount of data transfered per memory access, which is 64 bytes in our case. Memory access delay is set to 10 cycles as obtained from internal specifications [8].

A. Scalability

Solving an ILP problem for a mapping/scheduling problem on multi-core platform is known to be NP-hard. Thus, the running time of our ILP formulation is expected to explode as the number of tasks grows. The configuration with the **small** set of SDF graphs is used in this experiment.

Figure 5.a displays the average ILP 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 that produces schedules much faster. Similar observations were made on the StreamIt benchmark, for which an exact solution cannot be found for all applications in the benchmark.

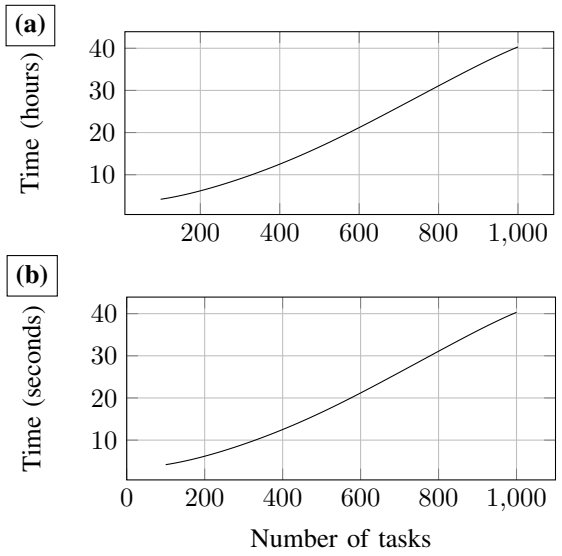

Fig. 5. Average ILP (a) and heuristic (b) computation time. Time units are different in 5.a and 5.b.

We also evaluate the scalability of our heuristic with the **large** set of SDF graphs. Figure 5.b shows the average heuristic computation time. The heuristic has a significant lower time complexity and better scalability compared to the ILP solution. For the smallest generated SDF graph with 40 actors and 100 firing, it takes less than 5 seconds to compute the schedule. For the largest generated SDF graph with 100 actors and 1000 firings, it only takes around 40 seconds to compute the schedule.

B. Quality of the heuristic

Two baselines are used to evaluate the quality of our heuristic: (1) a naive solution which is a simple forward LIST scheduling and (2) the optimal solution computed by applying the ILP formulation. For this experiment, we use the small set configuration.

A naive solution is a simple forward LIST scheduling that only assigns a task to a core but does not verify whether the assignment results in the shortest makespan in a contention-aware context. As depicted in Figure 3, it can lead to unnecessary contention that could be either avoided or reduced. After the schedule is computed, the actual makespan is obtained by applying the contention-aware WCRT analysis presented in [8].

We denote the makespan of the schedule obtained with the naive solution as M_n, the heuristic as M_{aware} and the ILP as M_{ILP}. The difference of the makespan in percentage is computed by $(M_{aware} - M_n)/M_n$ when comparing with the naive solution and $(M_{aware} - M_{ILP})/M_{ILP}$ when comparing with the ILP solution. The results are shown in Table III.

Comparing with the naive solution, the average speedup achieved with the heuristic is 18.3%. Considering that we only generate SDF graphs with low memory demand, the contention that could be reduced by the heuristic is relevant.

TABLE III
HEURISTIC QUALITY COMPARED TO THE NAIVE AND THE ILP SOLUTION

	Min	Max	Average
vs. Naive	0%	- 29.4%	- 18.3%
vs. ILP	0%	14.5%	5.7%

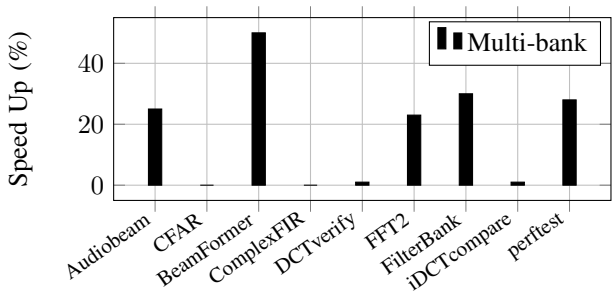

Fig. 6. Performance gain comparing to single-bank memory architecture achieved with the heuristic

Comparing with the ILP solution, in the worst case, the heuristic performs 14.5% worse. As we can observe, the average degradation is low, only 5.7%, which means our heuristic has an acceptable quality. The difference in the length of computed makespan between two solutions comes from two reasons. First, in the heuristic, tasks are allocated to processors without any backtracking. We know that different methods of ordering tasks in ready queue can affect the schedule; however, this is a complex problem and not considered in the scope of the work. Second, we have observed that the ILP solution allows the existence of processor idle time in the schedule to avoid memory contention. In other words, instead of overlapping the execution time of two tasks on two processors, the schedule waits until one task is completed then schedules another task.

From the experiments in Section VII-A and VII-B, we can see that the heuristic has better scalability compared to the ILP solution while provides acceptable results.

C. Multi-bank and single-bank

The objective of this experiment is to compare the difference in performance between shared memory with single-bank and multi-bank. In the experiment, we apply our heuristic to a subset of applications in the StreamIt benchmark. Memory contention is accounted for in two ways: single-bank and multi-bank. With shared single-bank memory, any tasks that are overlapping in time can introduce contention. The makespan of the schedule obtained with shared multi-bank memory is denoted as M_{multi} and the other is M_{single} The speedup factor is computed by $(M_{single} - M_{multi})/M_{single}$.

The result is provided in Figure 6. The key reasons of the speedup come from the utilization of shared multi-bank memory. It allows actors to access shared memory in parallel without contention. Our first observation is that the speedup varies significantly between applications. In fact, it depends on the width of the SDF graph or application's parallelization level. For applications such as CFAR or ComplexFIR, the width of the SDF graph is 1. Thus, shared multi-bank

memory does not bring any advantage. For applications such as BeamFormer in which the graph is highly parallelized, we can see a significant speedup up to 50%.

VIII. Conclusions

In this article, we present an approach of achieving efficient contention-aware scheduling of SDF graphs on a shared multi-bank memory architecture. The approach consists of two steps. First, we compute an abstract schedule defining firing dependencies with the objective of minimizing the buffer size requirement. Then from this abstract schedule we synthesize a time-triggered scheduling table with the objective of minimizing the makespan. The scheduling table can be computed by either the ILP formulation or the proposed heuristic based on LIST scheduling. Our experiments have shown that the proposed heuristic offers acceptable results, with only 5.7% degradation on average, while having a significant lower complexity.

For future works, we would like to take into account different schedulers such as periodic fixed priority. Furthermore, we want to refine the contention model, by more accurately capturing the actual duration of contention phases between communicating tasks. A possible research direction is taking into account a refined task execution model such as PRedictable Execution Model (PREM) [22] that splits a task into a read phase and an execute phase.

References

[1] T. Grandpierre and Y. Sorel, "From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations," in *1st International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, 2003, pp. 123–132.

[2] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on computers*, vol. 100, no. 1, pp. 24–35, 1987.

[3] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 5, pp. 151–162, 2006.

[4] R. Pellizzoni and M. Caccamo, "Toward the predictable integration of real-time cots based systems," in *28th International Real-Time Systems Symposium (RTSS)*. IEEE, 2007, pp. 73–82.

[5] V. Nélis, P. M. Yomsi, and L. M. Pinho, "The variability of application execution times on a multi-core platform," in *OASIcs-OpenAccess Series in Informatics*, vol. 55. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[6] G. F. Zaki, W. Plishker, S. S. Bhattacharyya, and F. Fruth, "Implementation, scheduling, and adaptation of partial expansion graphs on multicore platforms," *Journal of Signal Processing Systems*, vol. 87, no. 1, pp. 107–125, 2017.

[7] S. Sriram and S. S. Bhattacharyya, *Embedded multiprocessors : scheduling and synchronization*. Boca Raton: CRC Press: Taylor and Francis, 2009.

[8] H. Rihani, M. Moy, C. Maiza, R. I. Davis, and S. Altmeyer, "Response time analysis of synchronous data flow programs on a many-core processor," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. ACM, 2016, pp. 67–76.

[9] B. D. De Dinechin, D. Van Amstel, M. Poulhiès, and G. Lager, "Time-critical computing on a single-chip massively parallel processor," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2014.

[10] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *International Conference on Compiler Construction*. Springer, 2002, pp. 179–196.

[11] A. Bouakaz, "Real-time scheduling of dataflow graphs," Ph.D. dissertation, Université Rennes 1, 2013.

[12] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert, "From simulink to scade/lustre to tta: a layered approach for distributed embedded applications," in *ACM Sigplan Notices*, vol. 38, no. 7. ACM, 2003, pp. 153–162.

[13] T. Carle, D. Potop-Butucaru, Y. Sorel, and D. Lesens, "From dataflow specification to multiprocessor partitioned time-triggered real-time implementation," *Leibniz Transactions on Embedded Systems*, vol. 2, no. 2, 2015.

[14] D. Potop-Butucaru, R. De Simone, Y. Sorel, and J.-P. Talpin, "Clock-driven distributed real-time implementation of endochronous synchronous programs," in *Proceedings of the seventh ACM international conference on Embedded software*. ACM, 2009, pp. 147–156.

[15] P. Tendulkar, P. Poplavko, I. Galanommatis, and O. Maler, "Many-core scheduling of data parallel applications using smt solvers," in *17th Euromicro Conference on Digital System Design (DSD)*. IEEE, 2014, pp. 615–622.

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

[17] G. Giannopoulou, N. Stoimenov, P. Huang, L. Thiele, and B. D. de Dinechin, "Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources," *Real-Time Systems*, vol. 52, no. 4, pp. 399–449, 2016.

[18] Q. Perret, P. Maurere, E. Noulard, C. Pagetti, P. Sainrat, and B. Triquet, "Mapping hard real-time applications on many-core processors," in *24th International Conference on Real-Time and Network Systems (RTNS 2016)*, Brest, France, Oct. 2016, pp. pp. 235–244. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01692702

[19] M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nélis, and T. Nolte, "Contention-free execution of automotive applications on a clustered many-core platform," in *28th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2016, pp. 14–24.

[20] S. Martinez, D. Hardy, and I. Puaut, "Quantifying wcet reduction of parallel applications by introducing slack time to limit resource contention," in *25th International Conference on Real-Time Networks and Systems*, 2017.

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

[22] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *17th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2011, pp. 269–279.

[23] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 7, pp. 966–978, 2009.

[24] S. S. Battacharyya, E. A. Lee, and P. K. Murthy, *Software Synthesis from Dataflow Graphs*. Norwell, MA, USA: Kluwer Academic Publishers, 1996.

[25] K. Schild and J. Würtz, "Scheduling of time-triggered real-time systems," *Constraints*, vol. 5, no. 4, pp. 335–357, 2000.

[26] M. Kudlur and S. Mahlke, "Orchestrating the execution of stream programs on multicore platforms," in *ACM SIGPLAN Notices*, vol. 43, no. 6. ACM, 2008, pp. 114–124.

[27] M. Čubrić and P. Panangaden, "Minimal memory schedules for dataflow networks," in *International Conference on Concurrency Theory*. Springer, 1993, pp. 368–383.

[28] G. G. Brown and R. F. Dell, "Formulating integer linear programs: A rogues' gallery," *INFORMS Transactions on Education*, vol. 7, no. 2, pp. 153–159, 2007.

[29] J. Ostrowski, M. F. Anjos, and A. Vannelli, *Symmetry in scheduling problems*. Groupe d'études et de recherche en analyse des décisions, 2010.

[30] A. Honorat, H. N. Tran, L. Besnard, T. Gautier, J.-P. Talpin, and A. Bouakaz, "Adfg: a scheduling synthesis tool for dataflow graphs in real-time systems," in *25th International Conference on Real-Time Networks and Systems*, 2017.

[31] S. Stuijk, M. Geilen, and T. Basten, "SDF^3: SDF For Free," in *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*. IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006, pp. 276–278. [Online]. Available: http://www.es.ele.tue.nl/sdf3

[32] B. Rouxel and I. Puaut, "Str2rts: Refactored streamit benchmarks into statically analysable parallel benchmarks for wcet estimation & real-time scheduling," in *OASIcs-OpenAccess Series in Informatics*, vol. 57. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.