A Framework for Periodic Scheduling Synthesis of Synchronous Data-flow Graphs

Hai Nam Tran[1], Alexandre Honorat[2], Shuvra S. Bhattacharyya[2,3], Jean-Pierre Talpin[4], Thierry Gautier[4], and Loïc Besnard[4]

[1] Lab STICC, CNRS, UMR 6285, Univ. Brest, Brest, France
hai-nam.tran@univ-brest.fr
[2] Univ Rennes, INSA Rennes, CNRS, IETR - UMR 6164, F-35000 Rennes, France
ahonorat@insa-rennes.fr
[3] University of Maryland, Maryland, USA
ssb@umd.edu
[4] Inria, CNRS, IRISA, UMR 6074, Univ. Rennes, Rennes, France
{jean-pierre.talpin, thierry.gautier, loic.besnard}@inria.fr

Abstract. Synchronous data-flow graphs (SDF) are widely used in the design of concurrent real-time digital signal processing applications on multiprocessor system-on-chip. The increasing complexity of these hardware platforms advocates the use of real-time operating systems and fixed-priority scheduling to manage applications and resources. This trend calls for new methods to synthesize and implement actors in SDF graphs as real-time tasks with computed scheduling parameters (periods, priorities, processor mapping, etc.). This article presents a framework supporting scheduling synthesis, scheduling simulation, and code generation of SDF graphs. The scheduling synthesis maps each actor to a periodic real-time task and computes the appropriate buffer sizes and scheduling parameters. The results are verified by a scheduling simulator and instantiated by a code generator targeting the RTEMS operating system. Experiments are conducted to evaluate the framework's performance and scalability as well as the overhead induced by the code generator.

1 Introduction

Data-flow models of computation are commonly used in embedded system design to describe stream processing or control applications. Their 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, synchronous data-flow (SDF) [1] is one of the most popular in the embedded community.

Multiprocessor system-on-chips, which are used to host real-time applications, are so complex that real-time operating systems (RTOSs) with fixed-priority preemptive scheduling are used to manage resources and host real-time tasks. The implementation of a data-flow application on an RTOS calls for methods to efficiently synthesize periodic real-time tasks from a data-flow model.

Problem statement: The work in [2] has established a strong theoretical background on scheduling synthesis of SDF graphs. The tool Affine Data-Flow Graph (ADFG) [2,3] was developed to support a large number of scheduling synthesis algorithms. From an input SDF graph, ADFG synthesizes a periodic task set preserving the consistency of the SDF graph with the objective of maximizing the throughput and minimizing the buffer size requirement. Nevertheless, some important elements are not covered in the scope of ADFG. First, scheduling in multiprocessor platforms can be highly impacted by extra parameters such as interference due to resource sharing (bus and memory) and preemption costs. To take these elements into account, a viable solution is to apply either dedicated feasibility tests or scheduling simulation to verify schedulability. Second, when the computed schedule is verified to be schedulable, one must take advantage of the SDF model to quickly and reliably generate the corresponding scheduler code. This requires us to have some knowledge of how actors are implemented and how to realize a specific scheduler by using the APIs provided by an RTOS.

Contribution: This article presents a framework to integrate the scheduling synthesized by ADFG into a scheduling simulator and a code generator targeting the RTEMS (Real-Time Executive for Multiprocessor Systems) [4] RTOS. Scheduling simulation is achieved by Cheddar [5], which is interoperable with ADFG. Automated scheduler code generation is achieved by exploiting the lightweight data-flow environment (LIDE) [6] to implement the core functionality of actors and then instantiate them as POSIX threads. Our framework provides novel capabilities for design space exploration and iterative tuning of real-time, SDF-based signal processing systems

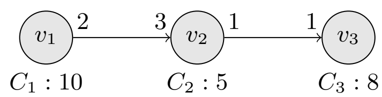
The rest of this article is organized as follows. Section 2 describes the background of our work. Section 3 provides a brief summary of scheduling synthesis and focuses on our approach to achieve scheduling simulation and code generation. Section 4 shows two experiments conducted to evaluate the performance of the framework. Finally, Section 5 presents related work and Section 6 concludes the article and discusses future work.

2 Background and Terminology

In this section, we present the SDF graph model, the periodic task model, and the notations used in the article. We briefly introduce our usage of scheduling simulation and code generation.

An 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., $q \in \mathbb{N}$). A channel e_{ab} has a bounded *buffer size* δ_{ab} and can have a number of *initial tokens*. Every time an actor fires, it consumes q tokens from an input channel and produces p tokens to an output channel.

On the scheduling analysis front, we assume the classic periodic task model on a multiprocessor platform. A task τ_i is defined by a sextuple: (C_i, T_i, D_i,



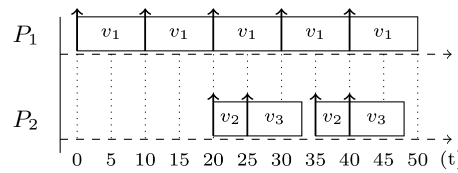


Fig. 1: An SDF graph of 3 actors, 2 channels and its periodic scheduling

Π_i, O_i, P_i). The parameter C_i, called the *capacity*, denotes the WCET of task τ_i when it executes non-preemptively. T_i, called the *period*, denotes the fixed interval between two successive releases of τ_i. D_i is the *deadline* of τ_i. In this article, we assume that tasks have implicit deadlines (i.e., $\forall i : D_i = T_i$). A task is assigned a *priority level* Π_i and makes its initial request at time O_i, called the *offset*. The last parameter P_i, called the *mapping*, denotes the processing unit in which the task is assigned to.

Figure 1 shows a simple SDF graph consisting of three actors and two channels. The notation C_i below each actor represents its WCET. For a channel $v_a \xrightarrow{p \quad q} v_b$, the production rate and consumption rate are provided. The channel sizes are not set here because these values depend on the computed schedule.

An *iteration* [1] of an SDF graph is a non-empty sequence of firings that returns the graph to its initial state. For the graph in Figure 1, firing actor v_1 3 times, actor v_2 2 times and actor v_3 2 times forms an iteration.

Periodic real-time scheduling of an SDF graph requires the mapping of each actor to a periodic task. The tasks must allow a *consistent* schedule for one iteration of the graph. By definition, a schedule is consistent if it has bounded buffer sizes, and if there is no deadlock, overflow, nor underflow. For a given actor v_i, we need to synthesize a periodic task τ_i and its scheduling parameters. Amongst those presented parameters, only the capacity is available in the SDF model. Other scheduling parameters must be computed to guarantee the consistency. In the next section, we introduce our approach and elaborate on how the approach supports consistent, real-time scheduling of SDF graphs.

3 Approach

Our framework consists of three main steps illustrated in Figure 2. First, from an input SDF graph, ADFG computes the required buffer sizes and synthesizes periodic scheduling parameters. The objective is to guarantee the consistency of the SDF graph while maximizing the throughput and minimizing the buffer size requirement. Second, a scheduling simulation is run by the Cheddar scheduling analyzer [5] to verify the schedulability and to give a thorough analysis of the synthesized schedule. Finally, after the schedule is verified, ADFG generates a real-time implementation of the computed schedule. We assume that actors are implemented in the lightweight data-flow environment (LIDE) [6] and we target the RTEMS RTOS. In the generated code, the input SDF graph is instantiated as a graph of LIDE actors and channels. Scheduling parameters are taken into account by using the POSIX thread API.

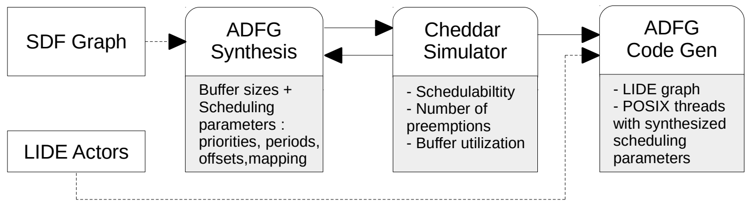


Fig. 2: Framework

3.1 Scheduling synthesis with ADFG

In this section, we present the ADFG tool and give a brief summary of the constraints and the objectives that we need to take into account in the scheduling synthesis step. ADFG [2, 3] is a free real-time scheduling synthesis tool for data-flow graphs. Periodic scheduling synthesis in ADFG takes into account the following two constraints to guarantee SDF graph consistency.

Underflow constraint: we have an underflow 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 underflow. For a channel $v_a \xrightarrow{p \quad q} v_b$, the n^{th} firing of v_b 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 such that: $l \cdot p - n \cdot q \geq 0$.

Overflow constraint: we have an overflow 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 δ_{ab}, the l^{th} firing of v_a 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 such that: $l \cdot p - n \cdot q \leq \delta_{ab}$.

In addition, ADFG accounts for two objectives in order to optimize the synthesized schedule. Scheduling requires computing the task periods and buffer sizes such that there is no buffer underflow or overflow, while maximizing the throughput and minimizing the total buffer size. Then the total buffer size and the throughput are the main metrics to compare different scheduling parameter valuations. In [2], Bouakaz proved that the maximum throughput problem can be translated to a maximum processor utilization one.

Considering the SDF graph in Figure 1, ADFG computes the smallest possible actor periods to ensure the consistency. For example, if the targeted system has 2 processing units, ADFG finds the periods $T_1 = 10$, $T_2 = 15$, $T_3 = 15$ and offsets $O_1 = 0$, $O_2 = 20$, $O_3 = 20$. Actor v_1 is mapped alone on the first core, for a total processor utilization factor of $U = 1.87$ for two cores. Part of the resulting schedule is depicted in Figure 1. Actors v_2 and v_3 are released at the same time but ADFG sets a higher priority to v_2 and thus v_2 is executed first.

3.2 Scheduling simulation with Cheddar

In our approach, scheduling simulation is used in order to provide a thorough analysis of the schedule synthesized by ADFG. It allows us to verify not only the correctness of the results but also obtain additional information including the number of preemptions and the buffer utilization. The second advantage of scheduling simulation is that it allows a thorough analysis of interference due to shared resources such as caches and memory bus. While this data is not directly given by ADFG, we have the possibility of using an external static analysis tool to obtain a richer execution profile of an actor. In [7], a preliminary work has been implemented in ADFG to support time-triggered schedules with memory interference but not yet periodic schedules.

In the development of our proposed framework, we have extended ADFG with capabilities that enable interoperability with Cheddar [5]—an open-source real-time scheduling analyzer. Classical feasibility tests, scheduling algorithms and a simulator are implemented in Cheddar. System architectures are defined with the Cheddar Architecture Description Language (Cheddar-ADL). The periodic scheduling of periodic tasks with buffer communication is supported by the simulator and used to evaluate the results of ADFG in [3]. ADFG generates the scheduling synthesized to an XML file compliant to Cheddar-ADL.

If the schedule synthesized by ADFG is shown to be not schedulable with Cheddar due to interference, some adjustments must be made. For example, the cache related preemption delay [8], which is a well-studied source of interference in preemptive scheduling, can make a schedulable task set become unschedulable. A solution is to incorporate this delay in the WCET of actors and rerun the scheduling synthesis with ADFG. This is an example of the important kinds of design iteration that are facilitated by the proposed framework.

3.3 Code generation with LIDE

The final step is to generate the implementation of the graph with computed scheduling parameters. It consists of generating the graph implementation from a set of pre-implemented actors and instantiating them with scheduling parameters by using the APIs supported an RTOS.

ADFG supports automated code generation of the computed buffer sizes and scheduling parameters for data-flow applications that are implemented in the Lightweight Data-flow Environment (LIDE) [6]. LIDE is a flexible, lightweight design environment that allows designers to experiment with data-flow-based implementations directly. A data-flow graph consists of LIDE actors that can be initialized with parameters including channels and buffer sizes. The usage of LIDE allows a systematic way to instantiate data-flow graphs with the buffer size parameters computed by ADFG. In addition, it also allows us to separate concerns involving the implementation of actors and schedulers.

RTEMS [4] is an open-source real-time operating system that supports open standard application programming interfaces such as POSIX. We consider the usage of the RTEMS to generate the computed scheduling parameters. Actor invocations and fixed-priority periodic scheduling are achieved by the usage of the POSIX thread library.

The code generator's inputs are the results computed by ADFG, including: buffer sizes, periods, offsets, priorities and mapping. The generated code is cross-compiled and tested by using the QEMU tool to emulate an ARM platform with

RTEMS. Next, we introduce LIDE and demonstrate our method of taking into account the computed results in the code generation process.

Actors and channels Graph elements in LIDE are designed and implemented as abstract data types (ADTs) that provide C based, object-oriented implementation of actors and channels [6]. Each actor has an associated `context`, which encapsulates pointers to the FIFO channels that are associated with the edges incident to the actor. Four interface functions, namely `new`, `enable`, `invoke` and `terminate`, are required to create an actor. Designers can develop their own actors by appropriately specializing the prototype function templates.

LIDE Prototype 1: Actor and FIFO functions

```
lide_c_<actor name>_context_type *lide_c_<actor name>_new(<FIFO pointer list>,
[parameter list])
boolean lide_c_<actor name>_enable (lide_c_<actor name>_context_type *context);
void lide_c_<actor name>_invoke (lide_c_<actor name>_context_type *context);
void lide_c_<actor name>_terminate (lide_c_<actor name>_context_type *context);
lide_c_fifo_pointer lide_c_fifo_new (int capacity, int token_size)
```

To implement a data-flow application we need to instantiate predefined actors, allocating channels and connecting them together. A channel is instantiated with the function `lide_c_fifo_new` which takes two input parameters: `capacity` and `token_size`. It allows us to apply buffer size results computed by ADFG in the code generation step.

- Capacity: the computed buffer size for a channel is given as an input parameter of the function `lide_c_fifo_new`. It is the number of tokens of which sizes are given as the second input parameter to the function.
- Token size: this information is given in the specification of the graph and passed directly to the code generator. In case of complex types, we assume that their specifications in C are also provided.

An actor is instantiated with the interface function `lide_c_<actor_name>_new`. The input parameters are FIFO channels that are connected to the actor. An example of the generated code is given in Figure 3. We generate a graph of two channels and three actors corresponding to the SDF graph in Figure 1.

```
lide_c_fifo_pointer v2_in = lide_c_fifo_new(6, sizeof(int));
lide_c_fifo_pointer v2_out = lide_c_fifo_new(1, sizeof(int));
lide_c_actor_context_type* actors[ACTOR_COUNT]; /* LIDE-C Actors */
actors[0] = (lide_c_actor_context_type*) (lide_c_v1_new (v2_in));
actors[1] = (lide_c_actor_context_type*) (lide_c_v2_new (v2_in, v2_in));
actors[2] = (lide_c_actor_context_type*) (lide_c_v3_new (v2_out));
```

Fig. 3: Actors and channels declaration with LIDE

```
rtems_actors[0].context = actors[0];
rtems_actors[0].name = "v1"
rtems_actors[0].priority = 1;
rtems_actors[0].period.tv_nsec = 10;
rtems_actors[0].processor = 1;
param.sched_priority=rtems_actors[0].priority; ;
pthread_attr_setschedparam(&attr,&param);
pthread_create(&id,&attr,lide_c_actor_start_routine,&rtems_actors[0]);
CPU_ZERO(&cpuset);
CPU_SET(processors[0], &cpuset);
pthread_setaffinity_np(id, sizeof(cpu_set_t), &cpuset);
```

Fig. 4: Generated code configuring the scheduling parameters in RTEMS

Scheduling parameters Actor firings are managed by a scheduler with scheduling parameters computed by ADFG. Four scheduling parameters are computed, namely: `priority`, `mapping`, `period` and `offset`. Code generation for these parameters is done by exploiting the POSIX thread API supported by RTEMS. The `priority` and `mapping` parameters are natively supported. The `period` and `offset` are taken into account by implementing a periodic scheduler.

- Priority: POSIX `set_affinity` function is used to set thread priorites.
- Mapping: POSIX provides the `cpu_set` property that allows us to choose the set of cores that a thread can execute on.
- Period: is implemented by exploiting the `nanosleep` function. The sleep duration is equal to the period minus the execution time of a thread.
- Offset: is generated by adding an idle period to the first execution of a thread.

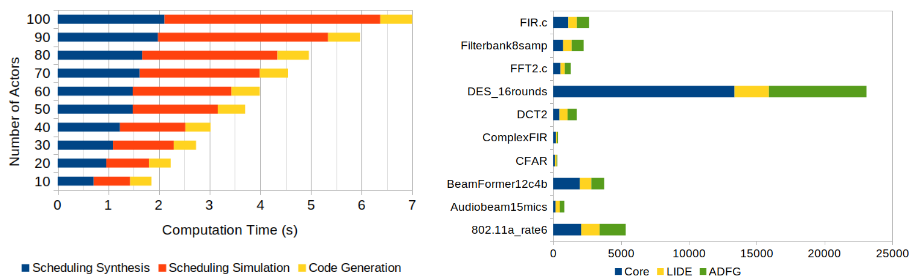
An example of the generated code is given in Figure 4. We create a data structure named `rtems_actor` that encapsulates a `lide_c_actor_context` and its scheduling parameters. Then, these parameters are passed to the attributes of a pthread accordingly. This example and its code can be duplicated systematically for all the actors in the graph and their corresponding threads in order to apply scheduling parameters computed by ADFG.

4 Evaluation

Experiments are conducted to evaluate our framework by three criteria. First, we show the time taken by each analysis step with SDF graphs of varied sizes. Second, we present the overhead induced by the code generator in terms of lines of code (LoC) added to a data-flow application. Finally, we discuss the run-time overhead introduced by the usage of our framework.

4.1 Framework performance and scalability

We evaluate the framework with synthetic SDF graphs generated by the SDF3 tool [9]. The number of actors varies from 10 to 100 in steps of 10. SDF3 takes many parameters besides number of actors; however, their values are chosen arbitrarily as they do not have a significant impact on the performance of our



(a) Computation time of the three steps (b) LoC added by the code generator

Fig. 5: Performance, scalability and code generation overhead

toolchain. The number of processing units is fixed at 4. This experiment is conducted on a PC with an Intel Core i7-8650U (1.90GHz $\times$ 8) processor, having 16 GBs of memory, and running Ubuntu 18.04.4.

Figure 5a shows the average computation time of each analysis step. It takes from 1.85s to 7.01s to synthesize, simulate, and generate periodic scheduling for the tested SDF graphs. We observe that the computation time grows linearly with the number of actors so the framework has an acceptable scalability. If we analyse each analysis step, scheduling synthesis and code generation have a better scalability than scheduling simulation. The differences between the max and min computation time of each step are 1.4s, 3.54s, and 0.22s. Scheduling simulation is also the analysis step taking the most time with up to 60% of the total computation time.

4.2 Code generation overhead

The code generator is evaluated by comparing the number of lines of code (LoC) added to a data-flow application. As our targets of hardware platforms are embedded systems, code size is an important metric to evaluate.

For a given application, we count the LoCs of its core functionality. Then we count the LoCs added in order to implement each actor with LIDE interface functions. Next, we count the LoCs generated for the parameters computed by ADFG. We have selected a subset of applications in the STR2RTS benchmark [10], which is a refactored version of the StreamIT benchmark.

The LoCs added to support LIDE interface functions are 30 LoCs per unique actor. In an SDF graph, we often observe that some actors are duplicated to exploit the parallelism. In other words, they are copies of an actor and they have identical functionalities As a result, they only need to be implemented once. The LoCs added to support the POSIX thread API are 14 LoCs per actor and 3 LoCs per buffer. On the contrary to the first category, we always need to generate the code for an actor, even if it is a duplication of another actor.

The result of the evaluation is given in Figure 5b. On average, the LoCs generated are about 60% of the complete application. As actors in the benchmark have simple functionality and do not support the implementation of multi-threaded

Table 1: WCET analysis of LIDE functions

	LIDE C functions	WCET analysis
1	`lide_c_<actor name>_enable`	165 cycles
2	`lide_c_fifo_write`	825 cycles (token size=8)
3	`lide_c_fifo_read`	815 cycles (token size=8)
4	`lide_c_<actor name>_invoke`	1640 cycles (token size=8)
5	`lide_c_<actor name>_terminate`	System call: free()
6	`lide_c_<actor name>_new`	System call: malloc()
7	`lide_c_fifo_new`	System call: malloc()

execution, this proportion is expected. This number is lower for SDF graphs with a high number of actors but a low number of unique ones such as DES_16round.

4.3 Run-time overhead

In this section, we discuss and evaluate the run-time overhead introduced by the usage of our framework. This overhead can be categorized in three sources: (1) RTOS overhead, (2) Periodic scheduling overhead, and (3) LIDE overhead.

RTOS overhead is due to the usage of an operating system such as RTEMS and its services instead of a bare-metal implementation. In [11], the authors provided an evaluation of RTEMS core characteristics. RTOS overhead depends on the choice of system designers and the evaluations of different embedded RTOS is not in the scope of our work. Periodic scheduling overhead is due-to the usage of a periodic scheduler instead of a rate optimal one. A comparison between the rate optimal schedule and the periodic schedule computed by ADFG has been presented in [2].

LIDE overhead is due to the code added when refactoring SDF actors and the usage of LIDE functions to read/write data in the channels. We present in Table 1 the WCET of the added functions. WCET analysis is done by the tool OTAWA [12] and the compiler used is arm-linux-gnueabi-gcc version 9.3.0. The token size, which is used to determine the loop bound when using the `memcpy` function to read/write data in the channels, is set to 8 bytes (integer token).

WCET analysis cannot be done for the functions 5, 6, 7 in Table 1 because the usage of the system calls `free` and `malloc`, which cannot be analyzed by the WCET analyzer. These functions are only called once at the initialisation/termination step and are not used when the system enters the steady-state.

We compare the obtained results with the average WCETs of actors found in the StreamIT [13] benchmark to have a quantitative evaluation. As presented in [10], the average WCETs of actors in the benchmark is varied between 273 - 2.94e5 cycles. Comparing to this result, the overhead of LIDE functions varies between 12.6 - 0.01 times the actors's WCET. This high variance exists because in the benchmark, there are both fine-grained actors with only few lines of code and coarse-grained ones. Coarse-grained SDF applications are the main targets of our framework as we consider the usage of RTOS and periodic scheduling.

5 Related Work

In this section, we position our contribution by providing discussions on SDF graph analysis tools. Many tools are able to analyze SDF graphs, to derive a few properties (e.g. mapping and buffer size), and finally to generate the glue code of the schedule automatically: DIF-GPU [14], PREESM [15], MAPS [16], Diplomat [17], Gaspard [18], PeaCE [19], and Ptolemy [20]. But these tools either do not consider real-time executions, periodic scheduling, or do not perform all syntheses automatically.

Another line of work is to build a complete data-flow compilation toolchain. In [13] and [21], the authors both introduce their own programming languages, namely, StreamIT and ΣC. Many analysis steps are applied to compute a static time-triggered schedule and generate an executable. Our approach differs from the two in terms of the choice of programming language and scheduling strategy. First, we refactor programs written in the C programming language to facilitate the generation of scheduling parameters. Second, we aim to generate a periodic scheduler instead of static time-triggered ones as described in [22–24].

The most related work to ADFG is the DARTS tool [25]. It allows to compute the strictly periodic scheduling parameters achieving the best throughput under earliest deadline first or rate monotonic scheduling policies. The main difference is that DARTS considers a non-constrained number of available processors on the target system and requires a constraint on the maximum total buffer size.

Compared to prior work on data-flow analysis tools, such as those summarized above, and to prior work on tools for real-time embedded systems (e.g., [26], [27], and [28]), our proposed framework provides a novel integration of real-time execution, periodic scheduling, resource-constrained mapping, and capabilities for iterative tuning and optimization of scheduling parameters.

6 Conclusions

In this article, we present a framework for periodic scheduling synthesis of SDF graphs. The framework is built from three open-source tools: ADFG [3], Cheddar [5], and LIDE [6]. Starting from an SDF graph, we synthesize its periodic schedule with ADFG and then verify the result with Cheddar. We assume that actors are implemented in LIDE and generate the implementation of the graph and the computed schedule by targeting the RTEMS RTOS. Our experiments have shown that the proposed framework has an acceptable performance, thus it can be used in the early stage of system design when changes occur quite frequently. For future works, we want to extend the scheduling synthesis in ADFG to take into account interference in order to provide more precise results. In addition, we aim to extend the framework with the integration of static resource analysis tools to directly obtain the timing and memory footprint of actors instead of relying on external sources of information.

Engineering effort has been put in this framework to assure that tools interoperability are achieved by data file export/import and a set of scripts. Nevertheless, the process of installing and configuring all the tools presented can be complex and time-consuming. We are investigating options to make a ready-to-use setup of the framework such as a pre-configured virtual machine.

References

1. S. S. Bhattacharyya, E. A. Lee, and P. K. Murthy, *Software Synthesis from Dataflow Graphs*. Norwell, MA, USA: Kluwer Academic Publishers, 1996.
2. A. Bouakaz, "Real-time scheduling of dataflow graphs. (ordonnancement temps-réel des graphes flots de données)," Ph.D. dissertation, University of Rennes 1, France, 2013. [Online]. Available: https://tel.archives-ouvertes.fr/tel-00945453
3. 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.
4. "Rtems real time operating system (rtos)," https://www.rtems.org/.
5. F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar: a flexible real time scheduling framework," in *ACM SIGAda Ada Letters*, vol. 24, no. 4, 2004, pp. 1–8.
6. S. Lin, Y. Liu, K. Lee, L. Li, W. Plishker, and S. S. Bhattacharyya, "The DSPCAD framework for modeling and synthesis of signal processing systems," *Handbook of Hardware/Software Codesign*, pp. 1185–1219, 2017.
7. H. N. Tran, A. Honorat, J.-P. Talpin, T. Gautier, and L. Besnard, "Efficient contention-aware scheduling of SDF graphs on shared multi-bank memory," in *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2019, pp. 114–123.
8. S. Altmeyer and C. Maiza Burguière, "Cache-related preemption delay via useful cache blocks: Survey and redefinition," *Journal of Systems Architecture*, vol. 57, no. 7, pp. 707–719, 2011.
9. S. Stuijk, M. Geilen, and T. Basten, "Sdfˆ 3: Sdf for free," in *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*. IEEE, 2006, pp. 276–278.
10. 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.
11. F. G. Nicodemos, O. Saotome, G. Lima, and S. S. Sato, "A minimally intrusive method for analysing the timing of rtems core characteristics," *International Journal of Embedded Systems*, vol. 8, no. 5-6, pp. 391–411, 2016.
12. C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "Otawa: An open toolbox for adaptive wcet analysis," in *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer, 2010, pp. 35–46.
13. W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *International Conference on Compiler Construction*. Springer, 2002, pp. 179–196.
14. S. Lin, J. Wu, and S. S. Bhattacharyya, "Memory-constrained vectorization and scheduling of dataflow graphs for hybrid CPU-GPU platforms," *ACM Transactions on Embedded Computing Systems*, vol. 17, no. 2, pp. 50:1–50:25, January 2018.
15. M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi, "Preesm: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming," in *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, Sept 2014, pp. 36–40.

16. J. Castrillon, R. Leupers, and G. Ascheid, "MAPS: Mapping concurrent dataflow applications to heterogeneous MPSoCs," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 527–545, Feb 2013.
17. B. Bodin, L. Nardi, P. H. J. Kelly, and M. F. P. O'Boyle, "Diplomat: Mapping of multi-kernel applications using a static dataflow abstraction," in *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Sept 2016, pp. 241–250.
18. A. Gamatié, S. Le Beux, E. Piel, R. Ben Atitallah, A. Etien, P. Marquet, and J.-L. Dekeyser, "A model-driven design framework for massively parallel embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 4, Nov. 2011.
19. S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y.-P. Joo, "PeaCE: A hardware-software codesign environment for multimedia embedded systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 3, pp. 24:1–24:25, May 2008.
20. L. Guo, Q. Zhu, P. Nuzzo, R. Passerone, A. Sangiovanni-Vincentelli, and E. A. Lee, "Metronomy: A function-architecture co-simulation framework for timing verification of cyber-physical systems," in *2014 International Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS)*, Oct 2014, pp. 1–10.
21. P. Aubry, P.-E. Beaucamps, F. Blanc, B. Bodin, S. Carpov, L. Cudennec, V. David, P. Doré, P. Dubrulle, B. D. De Dinechin *et al.*, "Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor," *Procedia Computer Science*, vol. 18, pp. 1624–1633, 2013.
22. Ching-Yi Wang and K. K. Parhi, "High-level dsp synthesis using concurrent transformations, scheduling, and allocation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 3, pp. 274–295, 1995.
23. C.-Y. Wang and K. K. Parhi, "Resource-constrained loop list scheduler for dsp algorithms," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 11, no. 1, pp. 75–96, 1995.
24. K. K. Parhi and D. G. Messerschmitt, "Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding," *IEEE Transactions on Computers*, vol. 40, no. 2, pp. 178–195, 1991.
25. M. Bamakhrama and T. Stefanov, "Hard-real-time scheduling of data-dependent tasks in embedded streaming applications," in *International Conference on Embedded Software (EMSOFT)*, 2011.
26. M. Chéramy, A.-M. Déplanche, P.-E. Hladik *et al.*, "Simulation of real-time multiprocessor scheduling with overheads," in *International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH)*, 2013.
27. Y. Chandarli, F. Fauberteau, D. Masson, S. Midonnet, M. Qamhieh *et al.*, "Yartiss: A tool to visualize, test, compare and evaluate real-time scheduling algorithms," in *Proceedings of the 3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2012, pp. 21–26.
28. R. Urunuela, A.-M. Déplanche, and Y. Trinquet, "STORM: a simulation tool for real-time multiprocessor scheduling evaluation," in *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on.* IEEE, 2010, pp. 1–8.