Proceedings of the first international workshop on Investigating dataflow in embedded computing architectures (IDEA 2015), January 21, 2015, Amsterdam, The Netherlands

Technische Universiteit Eindhoven
Department of Mathematics and Computer Science

Proceedings of the First International Workshop on Investigating Dataflow in Embedded computing Architectures (IDEA 2015)

15/02

ISSN 0926-4515

editors: prof.dr. P.M.E. De Bra
prof.dr.ir. J.J. van Wijk

Reports are available at:
http://library.tue.nl/catalog/TUEPublication.csp?Language=dut&Type=ComputerScienceReports&Sort=Author&level=1 and
http://library.tue.nl/catalog/TUEPublication.csp?Language=dut&Type=ComputerScienceReports&Sort=Year&Level=1

Computer Science Reports 15-02
Eindhoven, May 2015

Proceedings of the

First International Workshop on Investigating Dataflow in Embedded computing Architectures (IDEA 2015)

January 21, 2015,
Amsterdam, The Netherlands

Held in Conjuction with the

10th HiPEAC Conference,
January 19 - 21, 2015

Preface

IDEA '15 held at HiPEAC 2015, Amsterdam, The Netherlands on January 21st, 2015 is the first workshop on Investigating Dataflow in Embedded computing Architectures. This technical report comprises of the proceedings of IDEA '15.

Over the years, dataflow has been gaining popularity among Embedded Systems researchers around Europe and the world. However, research on dataflow is limited to small pockets in different communities without a common forum for discussion. The goal of the workshop was to provide a platform to researchers and practitioners to present work on modelling and analysis of present and future high performance embedded computing architectures using dataflow.

Despite being the first edition of the workshop, it was very pleasant to see a total of 14 submissions, out of which 6 papers were selected following a thorough reviewing process. All the papers were reviewed by at least 5 reviewers.

This workshop could not have become a reality without the help of a Technical Program Committee (TPC). The TPC members not only did the hard work to give helpful reviews in time, but also participated in extensive discussion following the reviewing process, leading to an excellent workshop program and very valuable feedback to authors. Likewise, the Organisation Committee also deserves acknowledgment to make this workshop a successful event. We take this opportunity to thank everyone who contributed in making this workshop a success.

Waheed Ahmad
Robert de Groote
Alok Lele
Orlando Moreira

Organisation

Technical Program Committee

- Benny Åkesson (Czech Technical University, Prague)
- Marco Bekooij (NXP Research, Eindhoven)
- Shuvra S. Bhattacharyya (University of Maryland, College Park)
- Pieter J. L. Cuijpers (Eindhoven University of Technology, Eindhoven)
- Michael Glaß, (Friedrich-Alexander-Universität, Erlangen-Nürnberg)
- Kim Grüttner (OFFIS - Institute for Information Technology, Oldenburg)
- Alix Munier Kordon (INRIA / LIP6, Paris)
- Orlando Moreira (Ericsson, Eindhoven)
- Luis Miguel Pinho (CISTER, Porto)
- Petro Poplavko (VERIMAG, Grenoble)
- Gerard Smit (University of Twente, Enschede)
- Sander Stuijk (Eindhoven University of Technology, Eindhoven)
- Jean-Pierre Talpin (INRIA, Rennes)
- Xue-Yang ZHU (SKLCS, Beijing)

Organising Committee

- Waheed Ahmad (University of Twente, Enschede)
- Robert de Groote (University of Twente, Enschede)
- Alok Lele (Eindhoven University of Technology, Eindhoven)

Contents

Energy-Aware Mapping and Scheduling of Large-Scale Macro Data-Flow Applications

Jörg Walter
OFFIS – Institute for Information Technology,
Oldenburg, Germany
joerg.walter@offis.de

Wolfgang Nebel
University of Oldenburg
Oldenburg, Germany
wolfgang.nebel@uni-oldenburg.de

***Abstract*—Predicting the performance of parallel programs for large-scale parallel platforms is difficult due to the disparity between development system and target platform. Additionally, energy efficiency is becoming a universal concern, and platforms move towards highly heterogeneous systems containing GPUs, FPGAs, and other unconventional processing elements.**
In this paper we propose a static macro data-flow mapping and scheduling tool that is able to handle large parallel applications targeting heterogeneous platforms. It optimizes overall run time and energy consumption at the same time with a user-configurable cost function, allowing a selectable trade-off between both properties.

I. Introduction

Energy efficiency is a universal concern by now: the world's fastest supercomputers exhibit a de facto 20 MW power limit, in mobile computing it affects battery life time; even high-end workstations are constrained due to size and noise requirements.

An equally universal concern is parallel application design. Well-established in the high-performance computing (HPC) world as in embedded system level design, general-purpose computing has embraced heterogeneous parallelism as well.

Experience shows that embedded systems can be expected to reach the computational power of today's supercomputers within ten years. Consequently, they will face the same challenges as supercomputers: a developer workstation will struggle to model, map, and simulate an application running on hundreds of processing elements at today's level of detail.

Our current research includes a design and optimisation flow for large parallel applications on heterogeneous platforms. It works on task precedence graphs, or macro data-flow graphs, and treats tasks in an abstract way so that it can handle larger applications than usually addressed in embedded system design.

In this paper we present our tool to solve the mapping problem, i. e. how to assign tasks to processing elements (PEs) and how to schedule tasks on a given PE. While our mapper is based on well-known algorithms, it has two main advantages over existing tools:

- Instead of just minimizing overall run time time, it optimizes energy efficiency by minimizing the energy delay product or any other function of these two values.
- It handles real-world applications consisting of thousands of tasks mapped to hundreds of heterogeneous processing elements with much better results than simulated-annealing-based tools.

This paper has the following structure: The next section lists research related to our methodology. Section III shows the context for which we designed the mapper. Section IV defines input data that it operates on. We explain the actual algorithm in Section V, followed by Section VI, where we highlight open issues and possible solutions. Finally, Section VII summarizes our findings and gives an outlook to ongoing research.

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement Nº 609757 (FiPS - Developing Hardware and Design Methodologies for Heterogeneous Low Power Field Programmable Servers).

II. Related Work

Mapping and scheduling of task graphs is a well-researched topic. Many algorithms exist, and [8], [9] give a comprehensive overview; the latter also evaluates some of them. Most algorithms use very simple assumptions about computation and communication behaviour. We base our tool on a combination of such well-understood algorithms, but since we map to real-world heterogeneous cluster architectures, our performance and power models are more detailed.

Few schedulers are energy aware in any case, and they usually minimize energy under fixed deadlines, like [5], [6] or reduce PE idle consumption by stretching tasks via voltage/frequency scaling [2], but the actual scheduling still optimizes for time only. Our mapper minimizes a configurable cost function, by default energy times delay, and thus is able to perform an arbitrary trade-off between time and energy.

III. A Design and Optimisation Flow

Fig. 1 shows the application design and optimization flow for which we designed the mapper.

Designers have a golden application model, i. e. sequential program code. They separate the code into several compute-intensive kernels and extract them. With these stand-alone kernels, they perform a one-time characterisation process; later stages of the flow use its results.

Designers then build a task graph so that each task executes exactly one kernel; one kernel usually corresponds to multiple task instances. Task graph plus kernel code should be functionally equivalent to the source application.

Our mapping and scheduling algorithm uses task graph, platform description, and characterisation results to map each

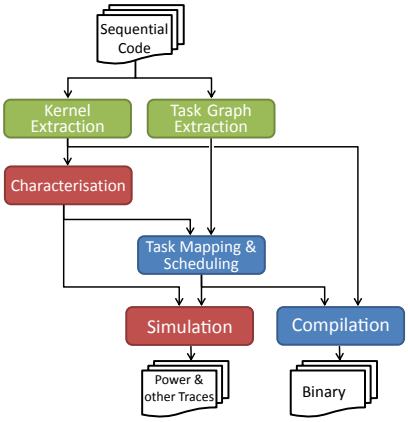


Figure 1. Overview of our design flow.

task of the task graph to a processing element (PE) on the target platform and generates a queue of tasks for each PE.

Simulation then uses this mapping in conjunction with characterisation results to predict makespan (i. e. overall run time) and energy consumption, as well as other metrics provided by simulation models.

With these predictions, designers can optimize the application in various ways: they can restructure the task graph, e. g. by changing the amount of parallelism, they can change the application's separation into kernels, and they can even start over and rewrite the application using different algorithms. This effectively constitutes a design space exploration loop. When satisfied, designers can create an executable version of the parallelised application from the mapped task graph in conjunction with the extracted kernels.

IV. Mapping Inputs

Our mapper works on an application model and a platform model, much like embedded design space exploration tools. We specifically target large parallel applications on loosely coupled (cluster-like) architectures.

A. Application Model

Applications are represented as task precedence graphs as commonly used in HPC research, e. g. in [1]. This application model is also known as macro data-flow, with slightly different terminology as main difference. Recent research has shown practical usefulness of this representation [4].

We add annotations in order to express computation and communication complexity with higher detail than simple edge/node weights could express: a task graph $TG = (T, D, K, \kappa, \delta)$ is an annotated directed acyclic graph, where T is the set of tasks (graph nodes), D is the set of precedence constraints (edges) between tasks, K is the set of known kernels (see Section III), function κ maps tasks to kernels, and function δ specifies communication volume for each edge.

A task can execute as soon as all predecessor tasks have completed execution, and all data objects associated with incoming dependencies have been received. Task execution is atomic (at least conceptually) and has no hidden side-effects, just communication as expressed by δ. Communication happens after the originating task has completed execution and before the destination task starts execution.

Our mapper expects that the task graph has exactly one start node, i. e. exactly one node with in-degree zero.

B. Platform Model

The mapper uses an abstract model for the target platform that provides a list of processing elements, their hardware architecture, and a graph of communication resources.

The platform communication graph $PCG = (C, L, M_C, \lambda)$ is an annotated directed graph, where C is the set of communication elements (CEs). Subset $P \subseteq C$ is the set of processing elements (PEs). Set L represents links between CEs. Function λ maps CEs to behavioural parameters contained in set M_C, most importantly start-up delay and usable bandwidth.

C. Characterisation Database

In addition to the task graph, the mapper relies on kernel execution times collected in a characterisation database. For each kernel k that is in the set K of all characterised kernels, it contains the average execution time for k running on each PE architecture, and the corresponding average power consumption. The database could hold synthetic numbers derived from power/performance simulation models, but it is intended to store actual measurements done on real hardware.

By splitting applications into kernels and characterising those independently of application and platform models, we get the flexibility to explore hypothetical (i. e. unfinished, non-existing) applications and platforms. A single processing element of each architecture is sufficient for characterisation.

V. Energy-Aware Mapping and Scheduling

Our mapping tool uses a constructive hybrid heuristic: An earliest-finishing-time-first list scheduler works on a linear task queue, while a modified version of simulated annealing (SA) explores random permutations of tasks in that queue by repeating the scheduling process for each permutation.

The mapper/scheduler only maps computation, not communication. We assume the platform has a shortest-path routing policy as is common with Ethernet interconnects. Furthermore, in its current state it does not model communication contention in any way (see Section VI).

A. Data Structures

In addition to the input models as explained in Section IV, our mapper uses the following important data structures:

1) Task Queue: The task queue is the central data structure. Initially, it contains a breadth-first traversal of the task graph, starting at the single start node. It must always contain a topological ordering of the task graph: for every task in the queue, all of its predecessors occur in front of it, and all of its successors appear after it. The optimizing heuristic always preserves this property.

2) Communication Matrix: The mapper assumes static shortest-path routing. From the platform communication graph it constructs a square matrix with one row and column per processing element (PE). It contains the cumulative start-up delay per transmission, and the maximum achievable bandwidth between each pair of PEs, which is the bandwidth of the slowest link on the route between them.

3) PE Schedule: The main output of the mapping process is a queue of tasks to be executed for each PE. A PE strictly adheres to this order of tasks, but it dynamically determines when the current task can start, i. e. when all inputs for a task are available. This is because the lack of communication congestion may lead to considerable timing inaccuracy.

B. Constructive Scheduler

The list-based mapper/scheduler traverses the task queue in order and maps each task to the PE that yields the earliest task finish time. It uses task execution times from the characterisation database and models communication delays via data taken from the communication matrix in conjunction with data size as recorded in the application model.

Even though this only considers time, there is a strong correlation between time and energy. As a consequence, we have achieved good results with this scheduler in conjunction with the actual optimizing heuristic (see Table I).

C. Optimizing Heuristic

The simulated-annealing (SA) part explores permutations of the task queue in order to minimize the user-configurable cost function, which can be any function of the two parameters total time and total energy. As its only move, the heuristic swaps the position of two adjacent tasks in the task queue. If that would violate the ordering restriction noted in Section V-A1, it chooses another pair until it finds a valid move.

It then lets the scheduler determine timing of the new task queue. After that, the mapper uses a two-state power model (active and idle) for processing elements to calculate total energy usage. Finally, it calculates the score using the user-configured cost function. It then decides whether to accept the solution with a typical SA probability function. Then it repeats the whole process; currently, it uses one million iterations.

1) Windowing: The mapper is intended for task graphs of a few hundred nodes as well as for ones with tens to hundreds of thousands of tasks. Therefore, it is difficult to set a fixed iteration count for simulated annealing. Instead, we subdivide the task queue into equal-sized windows. Each window overlaps with half of the preceding window and half of the following window. SA then works on a single window at a time, in increasing order, spending one million iterations each time.

In our experiments, a window size of about $3 \cdot |P|$ tasks has given best results; we observed a significant improvement in result quality over the all-at-once variant (given similar total mapping time).

2) Optimizations: We employed several optimisations in order to reduce mapping time. Most importantly, we do not reschedule the whole task graph on each iteration. We store intermediate scheduling results for every task in the task queue (using $O(|T| \cdot |P|)$ memory) and start rescheduling only the swapped tasks and the following ones. In average, this halves mapping time.

The list-based scheduler itself contains another useful optimisation: when searching for the best PE, it first checks if a given PE would be a viable candidate if it did not experience communication delays. If not, the scheduler skips calculation of communication delays entirely, which yields another significant speed-up (depending on target platform size).

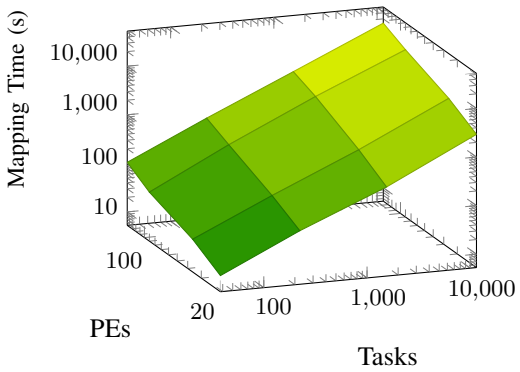


Figure 2. Mapping speed for various task graph and target platform sizes.

D. Evaluation

1) Mapping Speed: With a fixed iteration count, our mapper has an asymptotic run time of $O(|T| \cdot |P|)$, because the list scheduler iterates exactly once over the task queue on each annealing iteration, and it has to check all possible PEs for each task. This relies on the assumption that the average in-degree of all nodes is constant regardless of task graph size. It is easy to create task graphs that break this assumption, but for real-world parallel application patterns this holds, since kernels usually have a fixed, small number of inputs. With windowing this stays the same, because no matter how many windows there are, the amount of work per window decreases by that factor.

Fig. 2 shows the mapping time of some sample runs, each a mapping of Cholesky matrix decomposition in various degrees of parallelisation to various heterogeneous target platforms built from of three different types of processing elements. The mapper executed on a single core of an AMD Opteron processor running at 2.6 GHz. The figure shows that actual mapping times indeed scale linearly with number of PEs and tasks

2) Mapping Quality: We compared our mapper to a plain SA implementation as it is often found in literature, e. g. in [9], and to the plain list-based scheduler without annealing. Table I lists the results of mapping 1543 tasks to 90 PEs using the energy delay product as cost function, as suggested in [3]. As expected, the list-based scheduler is several orders of magnitude faster than the others. We selected SA parameters so it would run roughly the same time as our hybrid mapper.

Table I
COMPARISON TO OTHER MAPPING HEURISTICS.

Type	Predicted Time (s)	Predicted Energy (kJ)	Cost (MJs)	Mapping Time (s)
Simulated Annealing	162	164	26.5	2260
List Scheduler	160	199	31.8	0.003
Hybrid	132	177	23.4	2385

Our hybrid mapper produces mappings that are 26 % better than the plain list-based scheduler (measured by the cost function). It is 12 % better than plain simulated annealing running for a similar amount of time. With an 8 % increase in energy usage over SA, it produced a solution that has 18 % less makespan. That solution is even faster than the time-only optimizing list scheduler.

VI. FUTURE IMPROVEMENTS

There are some open issues we want to solve in order to make our tool more suitable for its intended purpose. Unfortunately, it is already quite slow, so any improvement in accuracy must be essentially free, or we need to find further optimization opportunities in the algorithm.

A. Performance

One way to speed up the mapper is to use variable SA iteration counts. That way, users are able to make a speed/optimality trade-off.

Another common technique is parallelisation. There are existing implementations that promise easy parallelisation of SA-based heuristics [7]. We have not yet tried this, but will probably do so.

B. Data-Dependent Behaviour

In order to account for data-dependent behaviour that can be expressed statically, kernels can actually be parametrised, e. g. by the size of the inputs they process. In that case, tasks in the task graph also specify kernel parameter values.

We do not yet handle data-dependent behaviour that cannot be determined statically. Regarding execution time, histograms instead of constant task execution times are a possible solution. Since simulated annealing already encompasses repeatedly retrieving execution times, a simple random sample from these histograms on each iteration might be enough to get a realistic task execution time distribution. If this expectation turns out to be true, this means that essentially, we get stochastic modelling for free.

C. Communication Congestion

The worst accuracy problem we face is the lack of communication congestion modelling. As far as we are aware, there is no cheap solution. Since the scheduler iterates over the task queue once, we cannot (cheaply) readjust already-mapped tasks. This means that earlier tasks in the queue cannot account for communication of later tasks even though they might be contending for the same communication links.

The problem actually consists of two parts: how to get congestion data, and how to incorporate it.

1) Collecting Congestion Information: So far, we have had two ideas for gathering congestion data: As our mapper is part of an iterative design space exploration flow, congestion data generated by earlier iterations could influence future mappings.

Our windowing technique could also provide abstract link utilisation figures. A window could record how much data went over each link. The following (overlapping) window could use this for congestion modelling.

2) Accounting for Congestion: In order to account for such congestion, link bandwidth could simply be reduced. Alternatively, link bandwidth could be modelled stochastically, just like we intend to do for task execution times. The latter solution might work well with iterative feedback, and it might even be fed from each iteration of the annealing heuristic.

VII. CONCLUSION

In this paper, we have presented a mapping tool intended for mapping large task graphs onto highly heterogeneous parallel platforms. It uses a combination of two well-known heuristics to create better mappings than each one on its own. Furthermore, it scales linearly with the number of tasks and the number of processing elements. Its main drawback is a speed that is only suitable for static ahead-of-time mappings.

We are still investigating ways to improve the computation and communication model employed in the mapper. The most important missing element is communication congestion, as this is a major source of mismatch to real-world execution behaviour. Since the mapper is part of a full design space exploration flow, we expect that other parts of the flow can supply useful data to address this issue.

REFERENCES

[1] Vikram Adve and Rizos Sakellariou. Application representations for multiparadigm performance modeling of large-scale parallel scientific codes. *International Journal of High Performance Computing Applications*, 14(4):304–316, 2000.

[2] Sanjeev Baskiyar and Rabab Abdel-Kader. Energy aware DAG scheduling on heterogeneous systems. *Cluster Computing*, 13(4):373–383, 2010.

[3] Luca Benini and Giovanni de Micheli. System-level Power Optimization: Techniques and Tools. *ACM Trans. Des. Autom. Electron. Syst.*, 5(2):115–192, April 2000.

[4] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Herault, and Jack J Dongarra. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[5] Po-Chun Chang, I-Wei Wu, Jyh-Jiun Shann, and Chung-Ping Chung. ETAHM: An energy-aware task allocation algorithm for heterogeneous multiprocessor. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 776–779. IEEE, 2008.

[6] Jia Huang, Christian Buckl, Andreas Raabe, and Alois Knoll. Energy-aware task allocation for Network-on-Chip based heterogeneous multiprocessor systems. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pages 447–454. IEEE, 2011.

[7] Georg Kliewer. A general software library for parallel simulated annealing. *EURO Winter Institute on Metaheuristics in Combinatorial Optimisation, Lac Noir, Switzerland*, 2000.

[8] Yu-Kwong Kwok and Ishfaq Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, December 1999.

[9] Kaushik Ravindran. *Task allocation and scheduling of concurrent applications to multiprocessor systems*. ProQuest, 2008.

Pipelined Scheduling of Acyclic SDF Graphs using SMT Solvers

Pranav Tendulkar, Peter Poplavko, Oded Maler
VERIMAG Lab (CNRS,University of Grenoble), France

***Abstract*—We consider compile-time multi-core mapping and scheduling problem for synchronous dataflow (SDF) graphs, proved an important model of computation for streaming applications, such as signal/image processing and video/image coding. In general the real-time constraints for these applications include both the task periods / throughput and the deadlines / latency. The deadlines are typically larger than the periods, which enables pipelined scheduling, allowing concurrent execution of different iterations of an application. A majority of algorithms for scheduling SDF graphs on a limited number of processors do not consider both latency and period real-time constraints at the same time. For this problem, we propose an efficient method based on SMT (satisfiability modulo theory) solvers. We restrict ourselves to periodic scheduling and acyclic graphs, giving up some efficiency for the sake of simplicity. We present an approach to encode the pipelined scheduling problem and demonstrate its practicality on Kalray MPPA-256 multi-core platform by executing various benchmarks according to the optimal schedules.**

I. Introduction

Streaming applications process streams of data of indefinite length, where output stream(s) are function(s) of input streams. Typical examples are *digital signal processing* (DSP) applications, video/audio (de-)coding, digital radio and television applications [11]. Such applications have high computational demands and hence they are often implemented in dedicated hardware. However, the semiconductor technology advances make it worthwhile to port many such applications to programmable parallel architectures, such as multi-cores. To meet the performance targets on programmable hardware, it is crucial to make use of task parallelism through optimizing compiler tools. To this end, the designers represent their application by a model of computation that exposes the parallelism. The streaming applications can be conveniently expressed using dataflow models, such as *synchronous dataflow graph* (SDF) [4]. Several multi-core compilers for SDF and other dataflow models have been proposed, *e.g.,* StreamIt [11]. This paper contributes to SDF compiler optimization to satisfy real-time constraints on M identical shared-memory processors. For simplicity, we restrict ourselves to *acyclic graphs* (*i.e.,* all feedback loops are hidden inside the graph nodes).

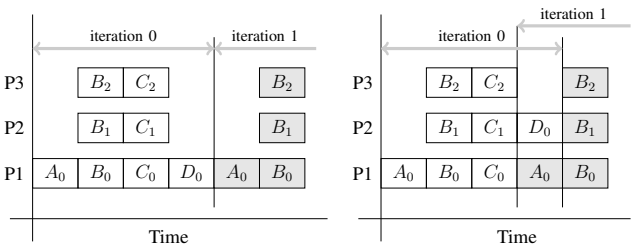
In real-time systems, for given limited set of processors the tasks should satisfy constraints on both throughput (*i.e.,* period) and latency (*i.e.,* response time, deadline). What makes the problem harder is the typical lack of support of task preemption in DSP multi-cores, which invalidates many real-time scheduling policies, such as EDF, making it computationally hard to analyze the schedulability. Moreover, even if preemptions were allowed, another problem is that DSP applications are *task graphs* and not independent tasks, which makes it hard to compute the response times. Therefore, many scheduling algorithms for DSP multi-cores are non-preemptive and they ignore latency and focus on throughput *e.g.,* [3]. Satisfying throughput, latency and processor count constraints at the same time is a hard combinatorial problem rarely addressed in the literature, especially if one tries to obtain or approximate the exact solution. For example, [6] approximates a similar problem using classical preemptive scheduling techniques.

Due to hardness of this problem, generic *constraint solving* techniques are typically applied for it, such as, SMT (Satisfiability Modulo Theory), ILP (integer linear programming), ASP (Answer Set Programming), and CP (constraint programming). For example [5] use SMT solvers and propose unfolding method for a problem similar to ours, but not considering specific constraints for SDF graphs. In our previous work [9], we apply SMT solvers for mapping and scheduling a (subclass of) acyclic SDF graphs, but we still focused on latency constraint and ignored the throughput constraint. Though we convert SDF graphs into task graphs (also known as homogeneous (HSDF) graphs), we propose task symmetry breaking constraints that use the information of the original (multi-rate) SDF graph actors to speed up the search for solutions. In this paper, we propose extensions of that work for period/throughput, assuming *pipelined scheduling*, *i.e.,* the period can be smaller than the latency. For simplicity, we restrict ourselves to *strictly periodic* schedules, *i.e.,* schedules where task graph iterations are spawned at equal time intervals. However, we believe that we do not loose much efficiency with this assumption because even self-timed solutions are eventually periodic, though not necessarily strictly periodic, but in general, multi-periodic, *i.e.,* imposing a period every K iterations for some $K \in \mathbb{N}$.

We propose a new technique called '*period locality*' for pipelined scheduling of SDF graphs. The proposed method represents the pipelined scheduling by a significantly simpler set of SMT constraints than the comparable encoding of unfolding [5] or modulo scheduling [10]. It also offers solutions that are sustainable to period variations for the fixed latency, in exchange of possible loss of optimality.

This technique was implemented in our tool StreamExplorer [7] and we perform experiments on the benchmarks from StreamIt and we validate our results by deploying them on a Kalray MPPA-256 multi-core processor architecture [1]. We observe that the error in prediction of period using a single cluster inside the platform is less than 15%.

Research supported by the European ICT Collaborative Project no. 288175 (CERTAINTY).



(a) Non-Pipelined, $P \geq \ell$ **(b)** Pipelined, $P < \ell$

Fig. 1: Periodic Schedule Examples for an SDF Graph

II. Synchronous Dataflow Graphs

Definition II.1 (Acyclic SDF Graph)**.** *An* acyclic SDF graph *is a tuple* $S = (V, E, d, r)$ *where* (V, E) *is a connected finite direct acyclic graph (DAG) whose nodes are repeatedly executed processes (actors) and edges are FIFO (first-in-first-out) channels,* $d : V \to \mathbb{R}_+$ *is a function assigning an execution time to each node,* $r : E \to \mathbb{N}_+ \times \mathbb{N}_+$ *assigns pairs of token production/consumption rates to channels. We use the notation* $r(u, v) = (\alpha(u, v), \beta(u, v))$*. The meaning of* α *is the number of data tokens produced to the channel at the end of each execution of actor* u*, and* β *is the number of data tokens consumed at the start of each execution of actor* v*. An SDF graph with* $r(e) = (1, 1)$ *for every* e *is called a* task-graph[1] *and is denoted by* $T = (U, \mathcal{E}, \delta)$*, renaming the first three tuple components and skipping the implicit component* r*.*

We deviate from the common definition of SDF graph by forbidding cyclic paths and initial tokens. This is not due to any fundamental restrictions, but certain parts of the theory, mentioned later, need to be extended to support these features in future work.

A practical SDF graph should satisfy the *consistency* property [4], namely, it should be possible to execute the actors such that the total amount of data produced on each channel is equal to the total amount of data consumed. Let $c(v)$ denote the number of times actor v is executed. The balance equation for an SDF channel (v, v') is written as:

$$c(v) \cdot \alpha(v, v') = c(v') \cdot \beta\ (v, v') \tag{1}$$

A graph is consistent if the balance equations have solutions $c(v)$, and only the smallest positive integer solutions are considered. Executing every actor $c(v)$ number of times is called *graph iteration*. The dependencies between actor executions in a graph iteration is modeled by equivalent task graph $(U, \mathcal{E}, \delta)$, where the nodes – called *tasks* – represent actor executions and edges represent precedence constraints. A consistent SDF graph can be expanded to a task graph, by well-known algorithm of deriving homogeneous SDF graph, see *e.g.,* [10]. In the derived task graph, every actor v is expanded into $c(v)$ tasks: $U_v = \{v_1, v_2, \ldots\}$.

III. SMT Encoding of the Scheduling Problem

A problem instance of the scheduling problem consists of an acyclic SDF graph S and the costs. Though for scheduling not the SDF graph itself, but the derived task graph is used, still we exploit the relation between these graphs for symmetry breaking in the solution space. The costs are the number of processors M, the latency ℓ, and, period P. The primary decision variables for the scheduling problem are the task start times, $s(u)$, and task mapping to processors, $\mu(u)$, assuming real $s(u) \in \mathbb{R}_{\geq 0}$ and integer $\mu(u) \in \mathbb{N}_+$. A scheduling interval for task u is interval $[s(u), e(u))$, where $e(u) = s(u) + \delta(u)$. We assume non-preemptive scheduling, and hence the task executes entirely inside this interval. Note that the scheduling is assumed periodic, so a task scheduled at $s(u)$ is also scheduled at $s(u) + P$, $s(u) + 2P$, etc., where P is period.

A schedule is *realizable*, if the tasks mapped to the same processor do not overlap in time. In addition, to be *feasible* it should respect tasks dependencies and the cost constraints. We define a realizable and feasible schedule in terms of constraints presented to the SMT solver tools. To express the scheduling constraints, it is convenient to define the following predicate:

$$\psi_{u,u'} : e(u) \leq s(u')$$

This predicate states that the scheduling interval of task u' follows after the interval of task u.

The following constraint is necessary to ensure that the schedule is realizable [5]:

$$\varphi_\mu : \bigwedge_{u \neq u' \in U} (\mu(u) = \mu(u')) \Rightarrow \psi_{u,u'} \vee \psi_{u',u}$$

φ_μ is called *mutual exclusion constraint*. It asserts that the scheduling intervals of two tasks running on the same processor are mutually exclusive.

The task graph dependencies are specified by precedence constraints:

$$\varphi_\epsilon : \bigwedge_{(u,u') \in \mathcal{E}} \psi_{u,u'} \tag{2}$$

We define two cost constraints: one for the *latency* (termination of the last task), denoted ℓ, and the other one for the number of *processors* used, denoted M:

$$\zeta_\ell : \bigwedge_{u \in U} e(u) \leq \ell \ \wedge \ \zeta_M : \bigwedge_{u \in U} \mu(u) \leq M$$

Putting all constraints together, we have the following encoding for the scheduling problem:

$$\Phi_{\mu\epsilon\ell M} : \varphi_\epsilon \wedge \varphi_\mu \wedge \zeta_\ell \wedge \zeta_M \tag{3}$$

In addition, we assert the processor and task symmetry breaking constraints in order to accelerate the search for solutions [9]. In particular, the task symmetry breaking constraints sort the schedule in the order compatible with the task index:

$$\bigwedge_{v \in V} \ \bigwedge_{v_h, v_{h+1} \in U_v} s(v_h) \leq s(v_{h+1})$$

where h is the index of task appearance in the 'classical' SDF graph sequential schedule with FIFO communication on the channels [10]. We prove a theorem that these constraints do not eliminate any feasible costs [9], [10]. Note that it is here where we exploit the connection of the derived task graph to its SDF origin. Note also that the task symmetry theorem would

[1] mostly referred to as *homogeneous* SDF graph

need to be revisited and generalized if we considered pipelined scheduling of SDF graphs that contain initial tokens. In fact, that is the reason why we do not yet support SDF graphs with feedback loops.

The encoding presented in this section is sufficient for non-pipelined scheduling, illustrated in Fig. 1a. However for pipelined scheduling, these constraints are not sufficient.

IV. Pipelined Scheduling

In pipelined scheduling the graph iterations follow with a period that is smaller than the latency, so they can overlap in time, Fig 1b. The constraints φ_μ presented in the previous section ensure mutual exclusion inside every iteration but not between the iterations.

We introduce a novel approach of encoding mutual exclusion in order to produce a pipeline schedule. We call this method *period locality*. The idea is to use the same mutual exclusion constraints as the non-pipelined scheduling, but to restrict the schedule such that different iterations cannot compete for processors. For this we require that all task scheduling intervals assigned to the same processor fit within a timing interval of length P.

$$\varphi_\lambda : \bigwedge_{u,u' \in U} (\mu(u) = \mu(u')) \Rightarrow e(u) - s(u') \leq P$$

In a strictly periodic schedule with period P this condition eliminates the inter-iteration processor conflicts. Hence, we have the following encoding of the period locality method (if we ignore symmetry breaking):

$$\Phi_{\lambda\mu\epsilon\ell M} : \varphi_\lambda \wedge \varphi_\epsilon \wedge \varphi_\mu \wedge \zeta_\ell \wedge \zeta_M$$

The period locality is a heuristic, as it restricts the periodic schedule such that the iterations do not overtake each other on a processor. One can construct manual examples that show that this restriction may eliminate optimal periodic scheduling solutions. Nevertheless, for practical benchmarks, exact encoding methods such as unfolding and modulo scheduling do not show any advantage in quality of solutions, but require a much more complex encoding. Apparently, the higher complexity of the exact methods does not typically lead to significantly worse solver computation times in practice, though it may lead to higher solver memory demands [10]. The main advantage of period locality is, however, that it possesses *period monotonicity* property[2], meaning that if a given period is feasible then larger periods are feasible as well while reusing the same problem solution and thus keeping intact the other costs such as latency and processor count. Monotonicity is important for efficient design space exploration for cost trade-offs, because (in)feasibility of some points implies (in)feasibility for the dominated (or dominating) cost points [9].

For the cost trade-off exploration, in this paper we consider two costs: the number of processors M and the period P, fixing the latency to an upper bound $\ell_{\max}$, computed by [10]: $2(\Omega + 1)P$, where Ω is a maximal number of edges in an SDF graph path. The scheduling problem gets significantly more difficult if the latency constraint ℓ is below this value: $\ell < \ell_{\max}$, whereas when $\ell \geq \ell_{\max}$, one can decouple mapping and processor scheduling without compromising the latency constraint. The mapping would be done by load balancing, ensuring the sum of task execution times per processor does not exceed P [3]. The scheduling would be done after mapping by maximal re-timing, *i.e.,* splitting the time axis into equal intervals of length P and assigning every task to the interval[3] that follows immediately after the interval of its latest predecessor [10]. Comparing the SMT solver efficiency between this approach and period locality at $\ell_{\max}$ is future work. Note that for generalizing this method to cyclic SDF graphs one would have to reconsider the definition of $\ell_{\max}$.

V. Experiments

For pipelined scheduling problem, using our tool [7], we investigate the performance of SMT solver when applied for multi-criteria cost optimisation problems. We validate the computed solutions by deploying the application benchmarks on a single shared-memory cluster of the Kalray MPPA-256 platform [1]. Extending the pipelined scheduling to multiple clusters is a non-trivial task, requiring co-scheduling of tasks and communication transfers [8], which is currently limited to non-pipelined scheduling. From the solution obtained from the SMT solver, our framework uses the task-to-processor mapping and ordering and lets the tasks synchronize their communication at run-time. In a single cluster, we execute the application for a configured number of iterations in a self-timed way and measure the period in which every task executes. The maximum value over all tasks is taken into account.

Maximal actor execution times obtained from measurements are used in the scheduling constraints. The costs to minimize are the period and the number of allocated processors at $\ell_{\max}$ latency[4]. Within a certain predefined timeout a query to the SMT solver should provide a **sat** or **unsat** answer, *i.e.,* satisfiable (feasible) and non-satisfiable (unfeasible). The solver may also give a **timeout** answer when it cannot conclude on the feasibility within the given time. Our goal is to find the closest approximation of the Pareto front possible, for which we used a *grid based exploration strategy* [9]. Our benchmarks consist of JPEG decoder and number of benchmarks from StreamIt [11], [8].

All the experiments were performed using the Z3 Solver [2] version 4.1 running on a Linux machine with *Intel Core i7* processor at 1.73 GHz with 4 GB of memory. The time out per query is 3 minutes while we keep the global exploration timeout to be 10 minutes.

1) Radix Sort: We explain the experiments with the running example of Radix Sort benchmark, an application that sorts integers. It consists of chain of 11 *radix* actors connected between the source and sink actors.

Figure 2 shows the results obtained for the two-dimensional cost space exploration of the period and the processors used. We can observe the trade-off between the two. We show an example schedule in Figure 3 for two and four processors. We can see how the solver is able to pack multiple iterations

[2]probably related to so-called schedule sustainability

[3]positioning inside the interval is not important

[4]see [10] for experiments at $\ell < \ell_{\max}$

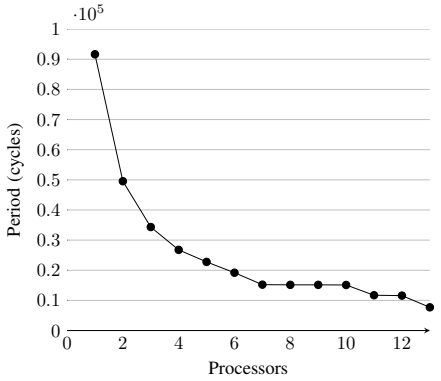
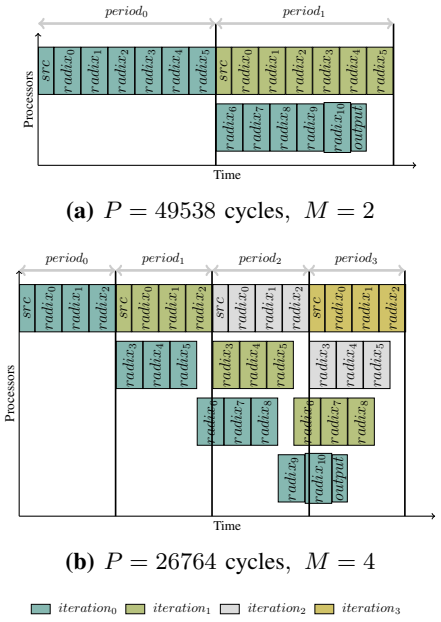


Fig. 2: Radix Sort : Processors used vs Period exploration



(a) $P = 49538$ cycles, $M = 2$

(b) $P = 26764$ cycles, $M = 4$

Fig. 3: Radix Sort : schedule for 2 and 4 processors

together. The amount of overlap between different iterations has increased when more processors are used. This also implies that the four-processor schedule requires larger communication buffers than the two-processor one, as more iterations run concurrently. However taking into account the communication buffer size together with the three other costs in pipelined scheduling is future work.

2) Other benchmarks: For the other benchmarks we perform the same experiment, *i.e.,* approximating the Pareto front and deploying the optimized solutions on the MPPA-256 cluster. Figure 4 shows the results for different benchmarks. We plot the number of solutions obtained for every benchmark and the maximum error as mismatch between the solver-predicted and measured period on the Kalray platform. The maximum error observed is 13.25% in case of BeamFormer application. There are two sources of error in our experiments. One is that we don't model the conflicts due to concurrent memory accesses by the processors. Secondly, Beamformer application has 53 tasks, which is relatively large. The cost space exploration experiences multiple solver timeouts, which leads to very loose predictions of feasible schedule periods. Since we execute them in a self-timed way the measured period is often much less than the predicted one in this benchmark.

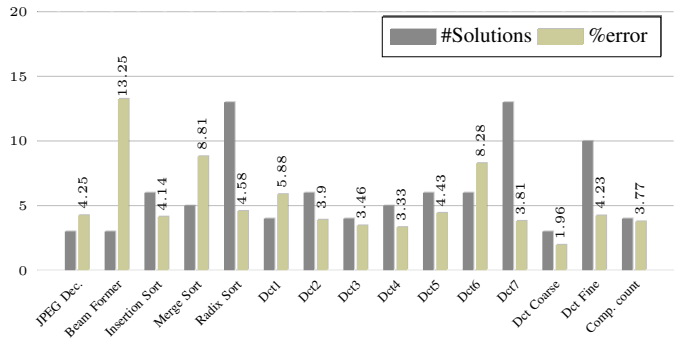


Fig. 4: Application benchmarks: maximum error for predicted vs. measured period

VI. Conclusions

In this paper we applied SMT solvers to address the pipelined scheduling problem for acyclic SDF graphs on shared-memory multi-cores with identical processors. We also evaluated our approach for a multi-core platform, showing good accuracy. Hereby, we considered *throughput* (*i.e.,* period), *latency* and *processor count* costs simultaneously, a problem that is rarely addressed in the literature.

We proposed the *period locality* heuristic, whose main advantage compared to exact methods is monotonicity, required for efficient design space exploration. We implemented this technique in our tool StreamExplorer [7] and evaluated it on a multi-core platform.

References

[1] B. de Dinechin et al. A clustered manycore processor architecture for embedded and accelerated applications. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6, 2013.

[2] L. de Moura and N. Bjorner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[3] M. V. Kudlur. *Streamroller : A Unified Compilation and Synthesis System for Streaming Applications*. PhD thesis, The University of Michigan, 2008.

[4] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75, 1987.

[5] J. Legriel and O. Maler. Meeting deadlines cheaply. In *ECRTS*, 2011.

[6] D. Liu, J. Spasic, J. T. Zhai, T. Stefanov, and G. Chen. Resource optimization for csdf-modeled streaming applications with latency constraints. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 188:1–188:6, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association.

[7] P. Tendular. Streamexplorer tool, http://www-verimag.imag.fr/ poplavko/streamexplorer.html.

[8] P. Tendulkar, P. Poplavko, I. Galanommatis, and O. Maler. Many-core scheduling of data parallel applications using SMT solvers. In *Conf. on Digital System Design, DSD 14, Proc.* IEEE, 2014.

[9] P. Tendulkar, P. Poplavko, and O. Maler. Symmetry breaking for multi-criteria mapping and scheduling on multicores. In *Formal Modeling and Analysis of Timed Systems (FORMATS'13)*, 2013.

[10] P. Tendulkar, P. Poplavko, and O. Maler. Strictly periodic scheduling of acyclic synchronous dataflow graphs using SMT solvers. Technical Report TR-2014-5, Verimag, 2014.

[11] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *PACT*, 2010.

Compile-Time Mapping of Dataflow Applications with Buffer Minimization

Youen LESPARRE
and Alix MUNIER-KORDON
Sorbonne Universités UPMC Univ Paris 06
UMR 7606, LIP6, F-75005, Paris, France
Email: name.lastname@alsoc.lip6.fr

Jean-Marc DELOSME
IBISC
Université d'Évry-Val-d'Essonne
91025 Évry, France
Email: delosme@ibisc.univ-evry.fr

***Abstract*—The problem of mapping dataflow applications on multi-core chips is notoriously difficult. This difficulty is compounded with the fact that the graphs describing the applications of interest are becoming very large. This paper proposes an approach which ensures that the size of the description of the mapping problem grows only polynomially with respect to the sizes of the graphs describing the application and the architecture. The application graphs considered in this paper are synchronous data flow graphs (SDFG) and the architectures are multi-cluster arrays of identical processors. The key idea is to apply a previously obtained polynomial condition of liveness to the SDFG describing the mapped application. This permits a formulation of the mapping problem as an Integer Linear Program whose objective is the minimization of buffer memory and whose size is polynomial in the sizes of the application and architecture graphs.**

I. Introduction

Keeping pace with the evolution of multi-core chips and the advent of many-core architectures [4], dataflow applications are broken down into numerous computation tasks, or actors, to be assigned to multiple on-chip processors. Finding an efficient algorithm to map these applications on clustered architectures, scalable for many cores, such as Kalray MPPA-256, STHORM or CoMPSoC, is difficult as this task encompasses the assignment of many actors to the processing resources and their scheduling under multiple resource constraints.

Synchronous Dataflow Graphs (SDFG), introduced in [9], are commonly used to model dataflow applications. They express an application with actors (nodes) and communications of data items between pairs of actors (arcs). Whenever an SDFG is consistent [9], it is possible to find an initial distribution of data items over the arcs, or initial marking, that ensures liveness and, hence, the existence of solutions to the scheduling problem [2]. In order to obtain good quality solutions, several methods have been recently proposed that jointly resolve the actor-to-resource assignment and the scheduling problem. Some authors [5], [8], [12] consider only homogeneous SDFGs (HSDFG), whose actors produce for (consume from) other actors a single data item at a time. By applying a simple transformation to a given consistent SDFG, an equivalent homogeneous SDFG may be obtained whose size is, however, exponential with respect to the SDFG's. This limits the scalability of HSDFG-based mapping methods, such as [3], where nodes and arcs are added to the HSDFG equivalent of the application's SDFG in order to model each candidate mapping and compute the associated throughput. An alternative is to bound the throughput from below by computing a schedule, see [14], and [6] for acyclic SDFGs.

To overcome the scalability problem associated to the use of the HSDFG model, a simplified model is used in [13] to evaluate the volume of the communications between two adjacent actors. Actors are assigned to processors under resource constraints in order to minimize the bandwidth between the processors. This technique gives coarser solutions with a better scalability (although the underlying problem remains NP-complete). Our model is similar but with a finer evaluation of the memory requirements.

We present in this paper a new analysis of the mapping problem on a clustered multi-core architecture using the SDFG model with bounded buffers. Instead of being transformed into an HSDFG, the SDFG is just "normalized" [11], [10]. Normalization, a simple scaling transformation applicable to any consistent SDFG, makes the weights on the arcs adjacent to an actor—the numbers of data items produced or consumed on each arc—all equal. It can be performed in polynomial time and does not increase the size of the model. Its main practical interest is that it simplifies live initial marking and throughput computation. Since it is reversible, all the mapping computations may be performed on normalized SDFGs, thus we shall assume in the paper that the SDFGs are normalized.

The architecture is a distributed memory architecture consisting of n equal tiles, the clusters, with limited memory (and limited processor count). They communicate by a NoC and each one contains an amount of memory equal to Δ^{max} to handle communications. Our goal is to minimize the overall memory required for communication between processor clusters while ensuring that the mapped application be live.

The application graph is supposed to fulfill the sufficient condition of liveness from [10]. Whenever a buffer is needed between two clusters, lower bounds on the amount of memory reserved in each cluster to implement the buffer are computed which guarantee that liveness is preserved. A simplified model issued from graph theory is then considered using these bounds to solve the global optimization problem.

The paper is organized as follows. The Synchronous Dataflow Graph model is introduced in Section II. In Section III, after a study of the memory needed when there is a single buffer between two clusters, a set of conditions on the memory required in order to fulfill the sufficient condition of liveness for the whole SDFG is derived, and a simple solution is proposed. Section IV presents two equivalent models for the global optimization problem. The first proceeds from graph

theory while the second formalizes the first using Integer Linear Programming. Section V is our conclusion.

II. Dataflow models

A. Synchronous Dataflow Graph model

In a Synchronous Dataflow Graph, nodes represent actors, which are programs that are executed repeatedly, and arcs represent data communications. Data are stored in a first-in first-out (FIFO) memory during communication. The SDFG is normalized as shown in [11]: as a result, an integer value $Z > 0$ is associated to each actor t so that, each time actor t is executed, Z data items are consumed (*resp.* produced) on each of its input (*resp.* output) buffers. The initial marking $M_0(a)$ of an arc a represents the normalized number of data items initially present in the associated buffer.

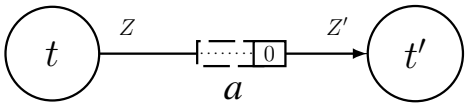


Fig. 1. A buffer a with initial marking $M_0(a) = 0$ between two actors of an SDFG, t, with production weight Z, and t', with consumption weight Z'.

B. Liveness

A dataflow graph is said to be live if all actors can be executed infinitely often. The liveness of a graph depends on its initial marking.

The most common ways to check the liveness of an SDFG are to compute a self-timed schedule or to expand the SDFG into its HSDFG equivalent [7]. However these methods have an exponential complexity, and their scalability is limited.

Another way is to evaluate the sufficient condition of liveness obtained in [10], which can be computed in polynomial time and, while not being necessary, is tight in practice. It has been implemented in the graph generator `Turbine` and used to generate live initial markings of SDFGs of up to 10,000 nodes [2]. In this paper this sufficient condition is supposed to be satisfied by the SDFG models of applications considered.

III. Evaluation of the overall memory for communications between clusters

This section presents a simple solution to the problem of evaluating the memory needed for buffers between two different clusters. Subsection III-A presents the problem and some notations. Subsection III-B expresses a condition when there is a unique buffer between two clusters. Subsection III-C extends the condition to several buffers between two clusters. Subsection III-D presents a feasible solution that will be used to express the subsequent global optimization problem.

A. Problem and notations

The application graph is an SDFG, $\mathcal{G} = (\mathcal{T}, \mathcal{B}, M_0)$, with $\mathcal{T}$ the set of actors, $\mathcal{B}$ the set of buffers and marking function M_0 giving the initial amount of data in the buffers. Any bounded buffer $a = (t, t') \in \mathcal{B}$, such that the number of data items it can contain is bounded by a fixed value $B(a)$, can be modeled by a backward arc $a' = (t', t)$ with $M_0(a') = B(a) - M_0(a)$, as shown in Fig. 2.

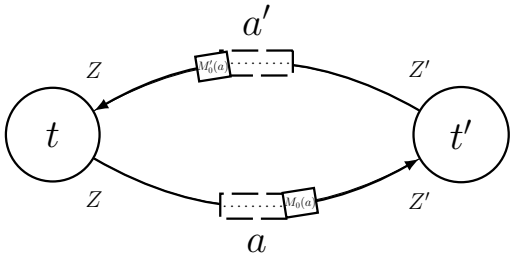


Fig. 2. A bounded buffer, with a the original arc and a' the backward arc.

Two assumptions are made on the buffer-aware SDFG $\overline{\mathcal{G}} = (\mathcal{T}, \overline{\mathcal{B}}, M_0)$ obtained when adding all the backward arcs:

1) The initial marking $M_0(a)$ of any buffer $a = (t, t')$ is divisible by $gcd_a = gcd(Z, Z')$, the greatest common divisor of the weights of the actors. Indeed, replacing $M_0(a)$ by $\lfloor M_0(a)/gcd_a \rfloor \cdot gcd_a$ does not change the behavior of the SDFG [10].
2) $\overline{\mathcal{G}}$ satisfies the sufficient condition of liveness found in [10]. Defining the height of an arc $a = (t, t')$ as $H(a) = M_0(a) + gcd_a - Z'$ and the height of a cycle μ as $H(\mu) = \sum_{a \in \mu} H(a)$, this condition states that the height of any cycle μ of $\overline{\mathcal{G}}$ satisfies $H(\mu) > 0$.

The set of clusters of the many-core architecture is denoted by $\mathcal{C}$, with $|\mathcal{C}| > 1$, and the overall memory available in each cluster is bounded by a fixed value Δ^{max}. The size of the data items stored in buffer $a \in \mathcal{B}$ is denoted by $\theta(a)$. If the two adjacent actors t and t' of $a = (t, t')$ are in a same cluster, the resource requirement for a is just $\theta(a) \cdot (M_0(a) + M_0(a'))$. Otherwise, the amount of memory needed in each cluster to ensure liveness is evaluated as shown in the rest of the section.

B. Case of a single bounded inter-cluster buffer

Consider now a bounded buffer $a = (t, t')$ with t and t' assigned to different clusters, c and c'. An actor $c(a)$ is inserted to perform the communication of data from c to c' via two bounded buffers, a_t and $a_{t'}$, as depicted in Fig. 3.

The weight $Z_{c(a)}$ of $c(a)$ and the initial markings of a_t and $a_{t'}$ are determined by minimizing the buffer sizes $B(a_t) = M_0(a_t) + M_0(a'_t)$ and $B(a_{t'}) = M_0(a_{t'}) + M_0(a'_{t'})$ while satisfying the sufficient condition of liveness. We shall see that if we set $Z_{c(a)} = gcd_a$ the equations will be greatly simplified.

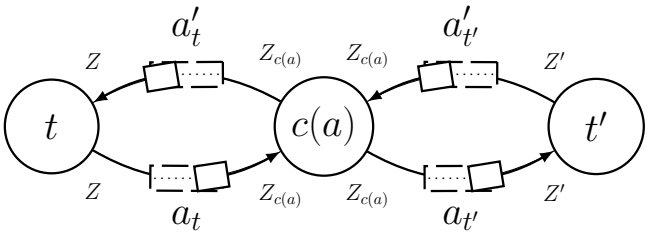


Fig. 3. A bounded inter-cluster buffer. The buffer $a = (t, t')$ is split between the two clusters. The part belonging to the cluster where actor t resides is represented by arc a_t (and backward arc a'_t) while the part belonging to the cluster where t' resides is represented by $a_{t'}$ (and $a'_{t'}$).

A sufficient condition of liveness for a single inter-cluster buffer is given in the following theorem.

Theorem 1: A single inter-cluster buffer $a = (t, t')$ with $Z_{c(a)} = gcd_a$ is live if $M_0(a_t) + M_0(a'_t) \geq Z$ and $M_0(a_{t'}) + M_0(a'_{t'}) \geq Z'$.

Proof: Let $\mu_1 = (t, a_t, c(a), a'_t, t)$ and $\mu_2 = (t', a'_{t'}, c(a), a_{t'}, t')$ be the two elementary cycles of the inter-cluster buffer. The sufficient condition of liveness of the buffer is that $H(\mu_1) > 0$ and $H(\mu_2) > 0$. Now, $H(\mu_1) = H(a_t) + H(a'_t) > 0$ is strictly equivalent to

$$M_0(a_t) + gcd(Z, Z_{c(a)}) - Z_{c(a)} + M_0(a'_t) + gcd(Z_{c(a)}, Z) - Z \quad > 0.$$

Since $Z_{c(a)} = gcd_a = gcd(Z, Z')$, $gcd(Z, Z_{c(a)}) = gcd_a$ and the inequality simplifies to

$$M_0(a_t) + M_0(a'_t) + gcd_a - Z > 0.$$

Since, by our first assumption on $\overline{\mathcal{G}}$, $M_0(a_t) + M_0(a'_t) - Z$ is divisible by gcd_a, this inequality is equivalent to

$$M_0(a_t) + M_0(a'_t) - Z \geq 0.$$

The other inequality results similarly from $H(\mu_2) > 0$. ■

C. General case

Define a communication-aware graph $\overline{\mathcal{G}}_c = (\mathcal{T}_c, \overline{\mathcal{B}}_c, M_0)$ as an SDFG obtained from the buffer-aware graph $\overline{\mathcal{G}}$ by splitting every bounded inter-cluster buffer $a = (t, t')$ as done in the previous subsection. The next theorem gives conditions on $\overline{\mathcal{G}}_c$'s initial marking M_0 ensuring that $\overline{\mathcal{G}}_c$ satisfies the sufficient condition of liveness from [10].

Theorem 2: Consider a communication-aware graph $\overline{\mathcal{G}}_c = (\mathcal{T}_c, \overline{\mathcal{B}}_c, M_0)$ associated to a live buffer-aware graph $\overline{\mathcal{G}} = (\mathcal{T}, \overline{\mathcal{B}}, M_0)$. The graph $\overline{\mathcal{G}}_c$ is live if, for every bounded inter-cluster buffer $a = (t, t') \in \mathcal{B}$, in addition to having $Z_{c(a)} = gcd_a$ and the two inequalities of Theorem 1 satisfied, the inequalities $M_0(a_t) + M_0(a_{t'}) \geq M_0(a)$ and $M_0(a'_{t'}) + M_0(a'_t) \geq M_0(a')$ are satisfied.

Proof: With the assumption $Z_{c(a)} = gcd_a$, the two inequalities of Theorem 1 ensure that the liveness condition is satisfied for the two elementary cycles associated to each bounded inter-cluster buffer $a = (t, t') \in \mathcal{B}$. Consider now all the other elementary cycles introduced when there is more than one inter-cluster buffer between two clusters. Under the assumption that $\overline{\mathcal{G}}$ satisfies the sufficient condition of liveness from [10], a sufficient condition of liveness for $\overline{\mathcal{G}}_c$ can be expressed as $H(a_t) + H(a_{t'}) \geq H(a)$ and $H(a'_t) + H(a'_{t'}) \geq H(a')$ for every bounded inter-cluster buffer $a = (t, t') \in \mathcal{B}$. The first inequality is equivalent to

$$M_0(a_t) + gcd(Z, Z_{c(a)}) - Z_{c(a)} + M_0(a_{t'}) + gcd(Z_{c(a)}, Z') - Z' \geq M_0(a) + gcd(Z, Z') - Z'.$$

Since $Z_{c(a)} = gcd(Z, Z') = gcd(Z, Z_{c(a)}) = gcd(Z_{c(a)}, Z')$, this inequality simplifies to $M_0(a_t) + M_0(a_{t'}) \geq M_0(a)$. The second inequality is obtained in a similar manner. ■

D. A live initial marking for a bounded inter-cluster buffer

The following corollary exhibits a simple solution to the inequalities of Theorem 2.

Corollary 1: Consider a live buffer-aware graph $\overline{\mathcal{G}} = (\mathcal{T}, \overline{\mathcal{B}}, M_0)$ and the associated communication-aware graph $\overline{\mathcal{G}}_c = (\mathcal{T}_c, \overline{\mathcal{B}}_c, M_0)$ with the initial marking $M_0(a_t) = M_0(a)$, $M_0(a'_t) = Z$, $M_0(a'_{t'}) = M_0(a')$ and $M_0(a_{t'}) = Z'$ for every inter-cluster buffer $a = (t, t') \in \mathcal{B}$. The graph $\overline{\mathcal{G}}_c$ is live. Moreover, the buffer sizes $B(a_t) = M_0(a_t) + M_0(a'_t)$ and $B(a_{t'}) = M_0(a_{t'}) + M_0(a'_{t'})$ verify

$$B(a_t) + B(a_{t'}) \leq 2(M_0(a) + M_0(a')) + gcd(Z, Z').$$

Proof: One can easily check that the initial marking considered verifies the inequalities of Theorem 1 and 2. Let $m = B(a_t) + B(a_{t'}) = M_0(a) + M_0(a') + Z + Z'$. Since the buffer-aware graph $\overline{\mathcal{G}} = (\mathcal{T}, \overline{\mathcal{B}}, M_0)$ verifies the sufficient condition of liveness, for the cycle of Fig. 2

$$M_0(a) + M_0(a') \geq Z + Z' - gcd(Z, Z'),$$

implying that $m \leq 2(M_0(a) + M_0(a')) + gcd(Z, Z')$. ■

The initial marking from Corollary 1 will be used in the next section to formalize the mapping problem.

IV. Formulation of the mapping problem

The mapping problem is formalized is this section. Subsection IV-A describes the problem, while Subsection IV-B models it using an Integer Linear Program of polynomial size.

A. Problem definition

The application is modeled by a live SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{B}, M_0)$. The buffer-aware SDFG $\overline{\mathcal{G}} = (\mathcal{T}, \overline{\mathcal{B}}, M_0)$ is derived from $\mathcal{G}$ by adding reverse arcs and associated initial markings so that the sufficient condition of liveness of [10] be satisfied.

The buffer memory requirements for the application can be modeled using an undirected multigraph $H = (\mathcal{T}, \mathcal{E})$ whose nodes are the actors. To each arc $a = (t, t') \in \mathcal{B}$ corresponds an edge $e = \{t, t'\} \in \mathcal{E}$ with three associated values:

- $S(e)$, the memory size required for arc a if t and t' are in the same cluster. $S(e) = \theta(a) \cdot (M_0(a) + M_0(a'))$, where $a' = (t', t)$ is the reverse of arc a in $\overline{\mathcal{G}}$.
- $S_t(e)$, and $S_{t'}(e)$, the memory sizes required for arc a in the cluster of t, and of t', if t and t' are not in the same cluster. $S_t(e) = \theta(a) \cdot B(a_t)$ and $S_{t'}(e) = \theta(a) \cdot B(a_{t'})$, where, from Corollary 1, these values are set to $B(a_t) = M_0(a) + Z$ and $B(a_{t'}) = M_0(a') + Z'$.

Note that $\mathcal{E}$ is a multiset; thus, when both arcs (t, t') and (t', t) belong to $\mathcal{B}$, there are two instances of $e = \{t, t'\}$ in $\mathcal{E}$. Sizes $S(e)$, $S_t(e)$ and $S_{t'}(e)$, are sums over all the instances of e.

A mapping consists in assigning the actors to the clusters. To each couple $(t, c) \in \mathcal{T} \times \mathcal{C}$ is associated a binary variable $x_{t,c}$ defined by

$$x_{t,c} = \begin{cases} 1 \text{ if actor } t \text{ is in cluster } c \\ 0 \text{ otherwise} \end{cases}$$

The mapping problem is to find the values $x_{t,c} \in \{0, 1\}$ for all the couples $(t, c) \in \mathcal{T} \times \mathcal{C}$ such that the total additional memory (due to buffers split into two different clusters) is minimized under the constraints that each actor is assigned to one and only one cluster, and the total memory assigned in each cluster is bounded by Δ^{max} (capacity constraint).

B. Integer Linear Programming Formulation

To each couple $(e, c) \in \mathcal{E} \times \mathcal{C}$ are associated two binary variables $s^e_{t,c}$, where $t \in e$, defined as:

$$s^e_{t,c} = \begin{cases} 1 \text{ if } x_{t,c} = 1 \text{ and } x_{t',c} = 0, \text{ where } t' \in e \text{ and } t' \neq t, \\ 0 \text{ otherwise} \end{cases}$$

Lemma 1 expresses the relationship between $s^e_{t,c}$ and $x_{t,c}$. It allows to compute $s^e_{t,c}$ using Integer Linear Programming:

Lemma 1: For any edge $e = \{t, t'\} \in \mathcal{E}$ and any cluster $c \in \mathcal{C}$, $s^e_{t,c} \geq x_{t,c} - x_{t',c}$.

Proof: If $x_{t,c} = x_{t',c}$ then $s^e_{t,c} = 0$ and the inequality is $s^e_{t,c} \geq 0$. Now, if $x_{t,c} = 1$ and $x_{t',c} = 0$, $s^e_{t,c} = 1$ and the inequality becomes $s^e_{t,c} \geq 1$. Lastly, if $x_{t,c} = 0$ and $x_{t',c} = 1$, $s^e_{t,c} = 0$ and the inequality becomes $s^e_{t,c} \geq -1$. The inequality $s^e_{t,c} \geq x_{t,c} - x_{t',c}$ is thus verified in all cases. ■

Lemma 2 allows to express the capacity constraint:

Lemma 2: For any couple $(c, e) \in \mathcal{C} \times \mathcal{E}$, the size of the memory allocated to edge e in cluster c is:

$$f(c, e) = \sum_{t \in e} s^e_{t,c}.S_t(e) + \sum_{t \in e}(x_{t,c} - s^e_{t,c})\frac{S(e)}{2}.$$

Proof: Three cases have to be considered:

1) If $x_{t,c} = x_{t',c} = 0$, neither t nor t' is assigned to cluster c. Then $s^e_{t,c} = s^e_{t',c} = 0$ and $f(c, e) = 0$.
2) If $x_{t,c} = x_{t',c} = 1$, both t and t' are in c and so is the buffer associated to e. Then $s^e_{t,c} = s^e_{t',c} = 0$ and $f(c, e) = S(e)/2 + S(e)/2 = S(e)$.
3) If $x_{t,c} \neq x_{t',c}$, suppose that $x_{t,c} = 1$ and $x_{t',c} = 0$ thus t is assigned to c but not t'. Then $s^e_{t,c} = 1$ and $s^e_{t',c} = 0$ hence $f(c, e) = S_t(e)$.

The lemma is thus verified. ■

The mapping problem may now be expressed as an Integer Linear Program as follows:

$$\text{minimize} \sum_{c \in \mathcal{C}} \sum_{e \in \mathcal{E}} \sum_{t \in e} s^e_{t,c} \cdot S_t(e)$$

under the constraints

$$\sum_{c \in \mathcal{C}} x_{t,c} = 1, \ \forall t \in \mathcal{T}, \tag{1}$$

$$s^e_{t,c} \geq x_{t,c} - x_{t',c}, \ \forall t \in e, \ \forall e = \{t, t'\} \in \mathcal{E}, \ \forall c \in \mathcal{C}, \tag{2}$$

$$\sum_{e \in \mathcal{E}} f(c, e) \leq \Delta^{max}, \ \forall c \in C, \tag{3}$$

$$x_{t,c} \in \{0, 1\}, \forall c \in \mathcal{C}, \ \forall t \in \mathcal{T} \tag{4}$$

$$s^e_{t,c} \in \{0, 1\}, \forall e \in \mathcal{E}, \ \forall c \in \mathcal{C}, \forall t \in \mathcal{T}. \tag{5}$$

The objective is the minimization of the memory used for the communications between clusters. Constraints (1) ensure that each actor is assigned to exactly one cluster. Constraints (2) express the relationships between the binary variables. Constraints (3) ensure that the limit Δ^{max} of total memory in each cluster is not exceeded. Further constraints, such as $\sum_{t \in \mathcal{T}} x_{t,c} \geq \sum_{t \in \mathcal{T}} x_{t,c'}$ where c immediately precedes c' in a total order on the clusters, could be added to reduce the number of equivalent solutions.

A key feature of our formulation is that the size of the program is polynomial. Indeed, the number of variables is $|\mathcal{T}| \cdot |\mathcal{C}| + |\mathcal{T}| \cdot |\mathcal{C}| \cdot |\mathcal{E}|$ while the number of equations is $|\mathcal{T}| + |\mathcal{T}| \cdot |\mathcal{C}| \cdot |\mathcal{E}| + |\mathcal{C}|$.

V. Conclusion and perspectives

The problem of finding a mapping of an SDFG on a clustered architecture that minimizes the overall memory has been formulated as an Integer Linear Program of polynomial size in terms of the SDFG and architecture model sizes. The first perspective is to test the scalability of this method using a solver and develop efficient heuristics to solve the program on large instances. The evaluation of the memory size presented in Corollary 1 should also be improved. Two other interesting issues are the extension of this method to more widely applicable models such as the Cyclo-Static DataFlow Graphs [1] and the incorporation of a minimum throughput guarantee.

References

[1] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean A. Peperstraete. Cyclo-static data flow. *IEEE Transactions on Signal Processing*, pages 3255–3258, 1995.

[2] Bruno Bodin, Youen Lesparre, Jean-Marc Delosme, and Alix Munier-Kordon. Fast and efficient dataflow graph generation. In *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems*, pages 40–49. ACM, 2014.

[3] Alessio Bonfietti, Luca Benini, Michele Lombardi, and Michela Milano. An efficient and complete approach for throughput-maximal SDF allocation and scheduling on multi-core platforms. In *13th Design, Automation & Test in Europe Conference (DATE)*, pages 897–902, 2010.

[4] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, pages 746–749. ACM, 2007.

[5] Thomas Carle, Manel Djemal, Dumitru Potop-Butucaru, Robert De Simone, Zhen Zhang, et al. Off-line mapping of real-time applications onto massively parallel processor arrays. Research Report INRIA RR-8429, December 2013.

[6] Yuankai Chen and Hai Zhou. Buffer minimization in pipelined sdf scheduling on multi-core platforms. In *17th Asia and South Pacific Design Automation Conference*, pages 127–132. IEEE, 2012.

[7] Amir Hossein Ghamarian, MCW Geilen, Twan Basten, Bart D Theelen, Mohammad Reza Mousavi, and Sander Stuijk. Liveness and boundedness of synchronous data flow graphs. In *Formal Methods in Computer Aided Design, 2006. FMCAD'06*, pages 68–75. IEEE, 2006.

[8] Tae ho Shin, Hyunok Oh, and Soonhoi Ha. Buffer optimal static scheduling with a throughput constraint for synchronous dataflow applications on multiprocessors. In *2010 International SoC Design Conference (ISOCC)*, pages 298–301, Nov 2010.

[9] Edward A. Lee and David G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[10] Olivier Marchetti and Alix Munier-Kordon. A sufficient condition for the liveness of weighted event graphs. *European Journal of Operational Research*, 197(2):532–540, September 2009.

[11] Olivier Marchetti and Alix Munier-Kordon. Cyclic scheduling for the synthesis of embedded systems. In Yves Robert and Frédéric Vivien, editors, *Introduction to scheduling*, pages 135–164. CRC Press, 2010.

[12] Orlando Moreira, J-D Mol, Marco Bekooij, and Jef Van Meerbergen. Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix. In *11th Real Time and Embedded Technology and Applications Symposium, RTAS 2005.*, pages 332–341. IEEE, 2005.

[13] Oana Stan, Renaud Sirdey, Jacques Carlier, and Dritan Nace. A GRASP for placement and routing of dataflow process networks on many-core architectures. In *8th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, pages 219–226. IEEE, 2013.

[14] Zheng Zhou, K. Desnos, M. Pelcat, J.-F. Nezan, W. Plishker, and S.S. Bhattacharyya. Scheduling of parallelized synchronous dataflow actors. In *2013 International Symposium on System on Chip (SoC)*, pages 1–10. IEEE, Oct 2013.

Towards Translating FSM-SADF to Timed Automata

Mladen Skelin
Department of Engineering Cybernetics,
Norwegian University of Science and Technology
mladen.skelin@itk.ntnu.no

Erik Ramsgaard Wognsen, Mads Chr. Olesen,
René Rydhof Hansen, Kim Guldstrand Larsen
Department of Computer Science, Aalborg University
{erw,mchro,rrh,kgl}@cs.aau.dk

***Abstract*—Dataflow formalisms play a significant role in the areas of design and analysis of embedded streaming applications. These formalisms can roughly be split into static and dynamic ones. Static dataflow formalisms are highly analyzable, but due to their static nature are not able to capture the dynamism inherent to modern embedded streaming applications. Dynamic dataflow formalisms on the other hand provide a sufficient level of expressiveness to capture the application dynamism at the cost of reduced analyzability. The recently introduced finite state machine-based scenario aware dataflow (FSM-SADF) formalism provides a good trade-off between expressiveness and analyzability. This paper reports on the translation of the FSM-SADF formalism to timed automata (TA). In short, we propose a compositional translation from FSM-SADF to TA that enables computation of some quantitative and qualitative properties of the model not supported by the existing tools, in the UPPAAL model checker. We demonstrate our approach on an MPEG-4 case study which is a typical example of a streaming application from the multi-media domain.**

I. Introduction

Dataflow formalisms are widely used to design and analyze embedded streaming applications running on distributed platforms such as MPSoCs (Multi-Processor System on Chips). In general, dataflow formalisms take the form of a directed graph which consists of *actors* as vertices and *channels* as edges. Actors are computational entities that usually represent application sub-tasks, while channels are communicational entities used to communicate application, control and synchronisation data between actors. In dataflow, an actor *firing* is an indivisible quantum of computation during which an actor consumes a certain number of data values from its input channels and produces a certain number of data values on its output channels. These data values are abstracted into *tokens* and the consumption and production numbers are called *rates*. In timed dataflow formalisms, it takes some time for the actor firing to complete and this time duration is called the actor *firing duration*.

Modern embedded streaming applications exhibit a high level of dynamism. The consequence is that the workload of such applications changes over time. How difficult it is to model such a dynamic application will depend on the particular dataflow formalism used. Not all dataflow formalisms provide us with a sufficient level of expressiveness needed to capture the dynamism of the application. On the other hand, those that do, pay the price in terms of significantly reduced analyzability. A good comparison between expressiveness and analyzability for various dataflow formalisms can be found in [6].

This work is supported by the 7th EU Framework Program under grant agreement 318490 (SENSATION).

The scenario aware dataflow (SADF) formalism [8] models an application as a collection of different behaviours called *scenarios* in which consumption and production rates and the actor firing durations change within one scenario and from one scenario to the other. A stochastic approach is used to model variance in firing durations and scenario ordering. The SADF formalism can therefore model dynamic applications and comes equipped with algorithms able to decide on the its qualitative and quantitative properties. These algorithms are implemented in the SDF^3 tool [5].

Finite state machine-based SADF (FSM-SADF) [7], [3] is a subset of SADF that abstracts from the stochastic aspects of firing durations and scenario ordering. In FSM-SADF every scenario is represented by a synchronous dataflow (SDF) graph [4], while scenario occurrence patterns are given by a finite state machine. These restrictions render the FSM-SADF more analyzable and implementation efficient than the general SADF model and very well positioned on the expressiveness vs. analyzability trade-off chart [6]. All FSM-SADF analysis and implementation algorithms can be found in the SDF^3 tool.

However, tools such as SDF^3 can be too specialized in the sense that they can only handle predefined properties, thus lacking support for user-defined properties. Although work has been done to check general properties by translating to model checkers [10], [9], this has only been done for the probabilistic SADF formalism. To circumvent this limitation, in this paper we propose a translation of the FSM-SADF formalism to timed automata (TA) as the first step to enable more general verification. Using TA has a number of advantages, in that very efficient abstractions exist. For example, temporal logics can express many of the properties common in reasoning about timed systems with concurrency. Furthermore, TA models of dataflow specifications can be easily extended to add costs such as energy and include the underlying implementation fabric models. This would in the future give us the possibility of using FSM-SADF for reachability analysis of embedded dynamic streaming applications through an optimal control formulation using model-checking techniques. We demonstrate our approach using an MPEG-4 case study modeled as an FSM-SADF graph for which we compute important quantitative and qualitative properties, some of which are not supported by the SDF^3 tool. We use the UPPAAL [2] state-of-the-art tool. The closest related work is the work of Ahmad et al. [1], although this only tackles the SDF formalism and is more concerned with modelling lower-level details of the scheduling on a given execution platform.

II. Definition of FSM-SADF

Here we give a more concise definition of FSM-SADF than the one given in [7]. Specifically, since the sets of ports and detectors have a simpler structure than in the general SADF, it is not necessary to represent them explicitly.

Definition 1 (FSM-SADF graph)**.** *An FSM-SADF graph is a tuple $G = (\mathcal{S}, \mathcal{K}, \mathcal{B}, E, R_p, R_c, \mathbb{S}, \mathbb{T}, \iota, \Phi, t, \phi_\iota, \psi_\iota)$, where*

1) *$\mathcal{S}$ is the nonempty finite set of scenarios,*
2) *$\mathcal{K}$ is the nonempty finite set of kernels,*
   - *$\mathcal{P} = \mathcal{K} \cup \{d\}$, where $d \notin \mathcal{K}$ denotes the unique detector, is the set of processes,*
3) *$\mathcal{B} \subseteq \mathcal{K} \times \mathcal{P}$ is the set of buffers,*
4) *$E : \mathcal{P} \times \mathcal{S} \to \mathbb{N}_0$ is the execution time for each process in each scenario,*
5) *$R_p, R_c : \mathcal{B} \times \mathcal{S} \to \mathbb{N}_0$ is the production (consumption) rate of the kernel producing to (process consuming from) each buffer in each scenario,*
6) *$(\mathbb{S}, \mathbb{T}, \iota, \Phi)$ is the FSM of the detector, where $\mathbb{S}$ is the nonempty set of states, $\mathbb{T} : \mathbb{S} \to 2^{\mathbb{S}}$ is the transition function, $\iota \in \mathbb{S}$ is the initial state, and $\Phi : \mathbb{S} \to \mathcal{S}$ associates each state with a scenario,*
7) *$t : \mathcal{K} \times \mathcal{S} \to \mathcal{S}^+$ is the string of scenarios sent to the FIFO of each kernel in each scenario of the detector,*
8) *$\phi_\iota : \mathcal{B} \to \mathbb{N}_0$ is the initial buffer status,*
9) *$\psi_\iota : \mathcal{K} \to \mathcal{S}^*$ is the initial control status.*

The detector is connected to every kernel by an explicitly ordered (FIFO) control channel. We further define $In(p) = \{b \in \mathcal{B} \mid \pi_r(b) = p\}$, where π_r is the right projection function, to be the set of buffers that process p consumes from (that input into p). Similarly, $Out(k) = \{b \in \mathcal{B} \mid \pi_l(b) = k\}$.

In anticipation of the next section we define $\varnothing$ to be the empty multiset, $\mathbb{P}$ to be the set of all submultisets of its input set, $\uplus$ to be the multiset sum, and $\setminus$ to be the zero-truncated asymmetric multiset difference. For example let $A = \{1, 1\}$ and $B = \{1, 2\}$. Then $A \cup B = \{1, 1, 2\}$ (maxima of multiplicities), $A \uplus B = \{1, 1, 1, 2\}$ (sums of multiplicities), $A \setminus B = \{1\}$, and $B \setminus A = \{2\}$. For strings $\sigma, \tau, \nu \in \mathcal{S}^*$ we define σ_i to be the ith element of σ, $\sigma + \tau$ to be the concatenation of σ and τ, and, if $\nu = \sigma + \tau$, then $\nu - \sigma = \tau$.

A. Operational Semantics

The behavior of an FSM-SADF graph is defined as a transition system where states are configurations.

Definition 2 (Configuration)**.** *A configuration of an FSM-SADF graph $G = (\mathcal{S}, \mathcal{K}, \mathcal{B}, E, R_p, R_c, \mathbb{S}, \mathbb{T}, \iota, \Phi, t, \phi_\iota, \psi_\iota)$ is a tuple $(\phi, \psi, \kappa, \delta)$, where ϕ is a buffer status, ψ a control status, κ a kernel status, and δ a detector status:*

- *A buffer status is a function $\phi : \mathcal{B} \to \mathbb{N}_0$ from each buffer to the number of tokens it stores,*
- *A control status is a function $\psi : \mathcal{K} \to \mathcal{S}^*$ from each kernel to the string of scenarios (control tokens) its FIFO stores,*
- *A kernel status is a function $\kappa : \mathcal{K} \to \mathbb{P}(\mathcal{S} \times \mathbb{N}_0)$ that to each kernel assigns a multiset of ongoing firings and their remaining execution times,*
- *A detector status is a pair $\delta \in \mathbb{S} \times (\mathbb{N}_0 \cup \{-\})$ that represents the state of the FSM and the remaining execution time of the ongoing firing, or, if there is no ongoing firing, the value $-$.*

The initial configuration of G is $(\phi_\iota, \psi_\iota, \kappa_\iota, \delta_\iota)$, where ϕ_ι and ψ_ι are defined in G, $\kappa_\iota = \mathcal{K} \times \{\varnothing\}$ and $\delta_\iota = (\iota, -)$.

Five types of configuration transitions are distinguished.

Definition 3 (Kernel Start Action)**.** *A kernel start action transition $(\phi, \psi, \kappa, \delta) \xrightarrow{\text{start}(k)} (\phi', \psi', \kappa', \delta)$ represents the start of a firing of kernel k. Let $s = \psi(k)_1$ denote the scenario of the firing (if it is defined). The transition is enabled if $|\psi(k)| \geq 1$ and $\forall b \in In(k) : \phi(b) \geq R_c(b, s)$. The resulting statuses are defined as*

$$\begin{aligned}
&\phi' = \phi[b \mapsto \phi(b) - R_c(b, s)] \quad \textit{for all } b \in In(k) \\
&\psi' = \psi[k \mapsto \psi(k) - s] \\
&\kappa' = \kappa[k \mapsto \kappa(k) \uplus \{(s, E(k, s))\}]
\end{aligned}$$

Definition 4 (Kernel End Action)**.** *A kernel end action transition $(\phi, \psi, \kappa, \delta) \xrightarrow{\text{end}(k)} (\phi', \psi, \kappa', \delta)$ is the end of a firing of kernel k. It is enabled if $\exists s \in \mathcal{S} : (s, 0) \in \kappa(k)$. The resulting buffer and kernel statuses are*

$$\begin{aligned}
&\phi' = \phi[b \mapsto \phi(b) + R_p(b, s)] \quad \textit{for all } b \in Out(k) \\
&\kappa' = \kappa[k \mapsto \kappa(k) \setminus \{(s, 0)\}]
\end{aligned}$$

Definition 5 (Detector Start Action)**.** *A detector start action transition $(\phi, \psi, \kappa, \delta) \xrightarrow{\text{start}(d)} (\phi', \psi, \kappa, \delta')$ represents the start of a firing of the detector, d. It is enabled if there is no ongoing firing $\exists s \in \mathbb{S} : \delta = (s, -)$ and all inputs are available $\forall b \in In(d) : \phi(b) \geq R_c(b, \Phi(s))$. The resulting statuses are*

$$\begin{aligned}
&\phi' = \phi[b \mapsto \phi(b) - R_c(b, \Phi(s))] \quad \textit{for all } b \in In(d) \\
&\delta' = (s, E(d, \Phi(s)))
\end{aligned}$$

Definition 6 (Detector End Action)**.** *A detector end action transition $(\phi, \psi, \kappa, \delta) \xrightarrow{\text{end}(d)} (\phi, \psi', \kappa, \delta')$ is enabled if $\exists s \in \mathbb{S} : \delta = (s, 0)$, and the resulting statuses are*

$$\begin{aligned}
&\psi' = \psi[k \mapsto \psi(k) + t(k, \Phi(s))] \quad \textit{for all } k \in \mathcal{K} \\
&\delta' = (s', -) \quad \textit{for some } s' \in \mathbb{T}(s)
\end{aligned}$$

In [7] time transitions are defined very generally, such that to account for given scheduling/resource constraints one needs to instantiate the time transitions needed. In the following we will assume a unconstrained execution, namely that all ongoing firings advance at the same pace.

Definition 7 (Time Transition)**.** *A time transition $(\phi, \psi, \kappa, \delta) \xrightarrow{\text{time}(t)} (\phi, \psi, \kappa', \delta')$ represents time progressing t time units. It is enabled if no kernel end or detector end transition is enabled, and t is the smallest remaining execution time of any ongoing firing. The resulting kernel status is*

$$\kappa' = \kappa[k \mapsto \{(s, n - t) | (s, n) \in \kappa(k)\}] \quad \textit{for all } k \in \mathcal{K}$$

using multiset comprehension. The detector status $\delta = (s, n)$ is updated as $\delta' = (s, n - t)$, unless $n = -$ in which case it is unchanged, $\delta' = \delta$.

B. Overtaking Problem

A closer look at the definition of the kernel status and the kernel start action in Section II-A reveals that the operational semantics of FSM-SADF allows the possibility of multiple simultaneous firings of a kernel, i.e. auto-concurrency. If these simultaneous firings of a kernel occur in different scenarios, due to the potential difference in kernel execution times in different scenarios, tokens may "overtake" each other. The result of that is that some kernel might consume tokens in a different scenario than the one these were produced in. This phenomenon makes it hard to ensure determinacy. One way of ensuring determinacy under auto-concurrency is considered in [3] by the introduction of the (max,+) algebraic semantics of FSM-SADF. In this paper, we assure determinacy by prohibiting auto-concurrency in the TA translation introduced in the following section.

III. Translation of FSM-SADF to TA

To be able to model check an FSM-SADF specification, we encode the operational semantics of Section II-A in the UPPAAL model checker. We refer to [2] for the full formalism and introduce it only briefly here due to space constraints. The correctness of the translation (with auto-concurrency being prohibited) follows from the construction itself as explained in the remainder of this section. In UPPAAL, a system is modeled as a network of TA that is extended with bounded discrete variables that are part of the state. We recall the definition of TA where we use $\mathcal{B}(\mathcal{C})$ to denote the set of constraints defined over a finite set of real-valued variables $\mathcal{C}$ called *clocks* and where $\Sigma = \{a!, a?, \ldots\}$ is a finite alphabet of synchronization actions.

Definition 8 (Timed automaton (TA))**.** *A timed automaton $\mathcal{A}$ is a tuple (L, l^0, E, I), where L is a finite set of locations (nodes), l^0 is the initial location, $E \subseteq L \times \mathcal{B}(\mathcal{C}) \times \Sigma \times 2^{\mathcal{C}} \times L$ is the set of edges and $I : L \to \mathcal{B}(\mathcal{C})$ assigns invariants to locations. We shall write $l \xrightarrow{g,a,r} l'$ when $(l, g, a, r, l') \in E$.*

The configuration is modelled such that the kernel and detector statuses are encoded in the states of the TA, while the buffer and control statuses are modelled explicitly using discrete variables. These do not add to the expressive power of the formalism, and for presentation purposes, we do not encode their use in the following, but we show the token consumptions and productions in the UPPAAL models in Fig. 1.

Given an FSM-SADF graph G, we generate a parallel composition of TA $System = \mathcal{A}_{k_1} \parallel \ldots \parallel \mathcal{A}_{k_n} \parallel \mathcal{A}_d$, where $k_i \in \mathcal{K}$ and $n = |\mathcal{K}|$. As there is only one instance of each kernel, no auto-concurrency exists, and determinacy is ensured. Fig. 1a shows the UPPAAL model of any kernel and Fig. 1b shows the detector of an FSM-SADF graph with $\mathcal{S} = \{A, B\}$ and the FSM defined to generate the language $(AB)^*$.

Every kernel $k_i \in \mathcal{K}$ is translated to the TA $\mathcal{A}_{k_i} = (L_i, l_i^0, E_i, I_i)$ where $L_i = \{\texttt{Initial}, \texttt{Configure}, \texttt{Fire}\}$, $l_i^0 = \texttt{Initial}$, and E_i and I_i are given as follows. The path

$$\texttt{Initial} \xrightarrow{|\psi(k_i)|\geq 1,\texttt{asap!},\emptyset} \texttt{Configure} \xrightarrow{\forall b \in In(k_i): \phi(b) \geq R_c(b,s),\emptyset,\{x_i\}} \texttt{Fire}$$

corresponds to the kernel start action. It is split into two TA edges because the kernel must first receive its configuration from the detector to know, depending on the current scenario s, how many tokens must be available in its input buffers. The edge $(\texttt{Initial}, \ldots, \texttt{Configure})$ synchronizes by sending on the urgent broadcast channel `asap` (which has no receivers), thus ensuring that the transition is taken as soon as a scenario is available. The kernel end action is encoded by

$$\texttt{Fire} \xrightarrow{x_i = E(k_i,s),\emptyset,\emptyset} \texttt{Initial}$$

and the invariant $I(\texttt{Fire}) = x_i \leq E(k_i, s)$ which together assure that the system stays in the location `Fire` for exactly the execution time $E(k_i, s)$ of the kernel k_i in scenario s. Thus, time transitions are encoded implicitly in the operation of the network of TA, for which time progresses in unison.

The detector TA uses the structure of its FSM, but embeds in each transition a firing location wherein time can pass between the events of consuming the input tokens and producing the output tokens. We encode it as $\mathcal{A}_d = (L_d, l_d^0, E_d, I_d)$, where $L_d = \mathbb{S} \cup \{(s, s') \mid s, s' \in \mathbb{S} \wedge s' \in \mathbb{T}(s)\}$ and $l_d^0 = \iota$. The edge set E_d is defined such that each transition $s_i \to s_j$ described by $\mathbb{T}$ is translated into a detector start edge followed by a detector end edge:

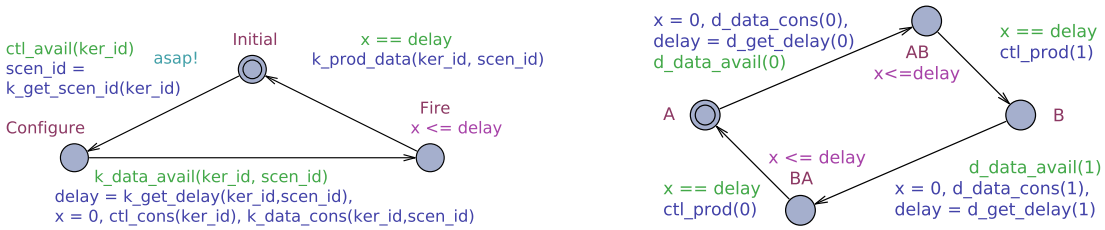
$$s_i \xrightarrow{\forall b \in In(d): \phi(b) \geq R_c(b,\Phi(s)),\emptyset,\{x_d\}} (s_i, s_j) \xrightarrow{x_d = E(d,\Phi(s)),\emptyset,\emptyset} s_j$$

The invariant function I_d is defined such that each firing location (s_i, s_j) maps to the invariant $x_d \leq E(d, \Phi(s_i))$.

IV. Model Checking of TA Model

In this section we demonstrate examples of qualitative and quantitative analysis of the MPEG-4 FSM-SADF specification from [6] using the UPPAAL model checker. The types of analysis, the associated time and memory usage and whether or not the respective type of analysis can be found in the SDF^3 tool are shown in Table I. The experiments were performed on an Intel Core i7-3520M CPU running UPPAAL 4.1.18 64-bit on Linux. The default settings were used and UPPAAL was restarted between each query.

SDF^3 can analyze deadlock freedom, buffer occupancy, inter-firing latency, response delay and throughput of an FSM-SADF specification. Table I reveals that using UPPAAL we can analyze all these properties, except throughput which is a subject of future work. Using UPPAAL we obtained the same results as SDF^3 on the analysis types supported by both frameworks. Analyzing for deadlock freedom is achieved by the UPPAAL query (`A[] not deadlock`). Maximum buffer occupancy analysis is performed using the UPPAAL supremum operator, e.g. (`sup:` b_i). Maximum inter-firing latencies can be obtained as clock suprema. For example, maximum latency of the detector process can be computed using the query (`sup:` x_d). We can check the relationship between the maximum response delay of a process p and a constraint r using the query (`E<> !`p`.bFirstFirCompleted and` y_p `>=` r), where y_p is a clock that is never reset, and `bFirstFirCompleted` is a variable set to `true` when p completes its first firing within a scenario. In this experiment, p = MC, and r = 3510. We can also check interleaving patterns of process firings, e.g. "between two consecutive firings of the process p, process q fires at least n times", etc.



(a) Kernel in UPPAAL

(b) Example detector in UPPAAL

Fig. 1: FSM-SADF UPPAAL model

TABLE I: MPEG-4 verification time and virtual memory usage

Analysis	SDF^3	Time [s]	Mem [MB]
Deadlock freedom	Yes	5.87	452
Maximum buffer occupancy, all buffers	Yes	2.44	452
Maximum inter-firing latency, detector	Yes	3.71	455
Maximum response delay, MC	Yes	1.44	252
Interleaving patterns of process firings	No	4.89	460
Maximum delay between process firings	No	4.18	455

For this we use a *leads to* query (whenever a eventually b) and a counter variable: (p`.Fire --> `q`.FireCount >=` n). In the experiment, p = MC, q = RC, and $n = 1$. We can also check whether the maximum delay between the end times of firings of two processes within a scenario is greater than, less than or equal to a predefined value by constructing a query monitor TA that synchronizes with the events of firing completions of the processes it monitors. In the case of kernels, this synchronization takes place when the edges `(Fire,...,Initial)` are taken. In the experiment we verify that the MC-RC delay is always smaller than 5000.

UPPAAL allows us to check the model against various properties, many of which are not supported by the SDF^3 tool-set, therefore justifying the use of a general verification tool such as UPPAAL as a complement to specialized tools. The flexibility of the UPPAAL's TCTL based query language and the possibility of construction of various query monitor automata allows the user to easily and in almost no time compute various qualitative and quantitative properties of the model. In contrast to UPPAAL, doing the same in a specialized tool like SDF^3 would involve the user into a process of software development.

V. Conclusion and Future Work

FSM-SADF is a powerful dataflow formalism that is able to capture the dynamic behaviour of modern streaming applications while offering a good trade-off between expressiveness, analyzability and implementation efficiency. However, the formalism is currently only supported by the SDF^3 tool-set which implements a predefined set of properties that can be analysed/verified. In this paper we propose a translation of FSM-SADF to TA, thereby enabling the use of the UPPAAL model checker for analysing and verifying user-defined properties in a straightforward manner. As future work we plan to develop methods for worst-case throughput analysis, investigate the scalability of our translation and also to give performance comparison with SDF^3. Furthermore, we wish to further investigate auto-concurrency and the overtaking problem and explore policies that would assure determinacy, such as the (max,+) one [3]. As our translation also sets the first milestone towards enabling the use of FSM-SADF in a wider context, e.g. cost-optimal analysis, we also plan to investigate reachability analysis of applications modeled by FSM-SADF through an optimal control formulation using the UPPAAL family of model-checkers.

References

[1] W. Ahmad, E. d. Groote, P. K. F. Hölzenspies, M. I. A. Stoelinga, and J. C. v. d. Pol. Resource-constrained optimal scheduling of synchronous dataflow graphs via timed automata. Technical Report TR-CTIT-13-17, University of Twente, Enschede, January 2014.

[2] G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal. In *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, 2004.

[3] M. Geilen and S. Stuijk. Worst-case performance analysis of synchronous dataflow scenarios. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, Oct 2010.

[4] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9), Sept 1987.

[5] S. Stuijk, M. Geilen, and T. Basten. SDF^3: SDF for free. In *Sixth International Conference on Application of Concurrency to System Design (ACSD 2006), 28-30 June 2006, Turku, Finland*, 2006.

[6] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, July 2011.

[7] S. Stuijk, A. Ghamarian, B. Theelen, M. Geilen, and T. Basten. FSM-based SADF. Technical report, Eindhoven University of Technology, Department of Electrical Engineering.

[8] B. Theelen, M. C. W. Geilen, T. Basten, J. P. M. Voeten, S. V. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings. Fourth ACM and IEEE International Conference on*, July 2006.

[9] B. D. Theelen, M. Geilen, and J. Voeten. Performance Model Checking Scenario-Aware Dataflow. In *Proceedings of the 9th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, volume 6919 of *Lecture Notes in Computer Science*, Aalborg, Denmark, 2011. Springer.

[10] B. D. Theelen, J.-P. Katoen, and H. Wu. Model checking of Scenario-Aware Dataflow with CADP. In W. Rosenstiel and L. Thiele, editors, *Proceedings of the 2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Dresden, Germany, 2012. IEEE.

Towards Generally-Timed Energy SADF

Arnd Hartmanns and Holger Hermanns
Saarland University – Computer Science, Saarbrücken, Germany

I. Introduction

Scenario-aware dataflow (SADF [11]) is an extension of synchronous dataflow (SDF [9]) supporting the description of variations in data processing (e.g. changes in speed, or disabling of components) according to predefined scenarios. As such, it extends the applicability of dataflow formalisms, which have traditionally seen heavy use in digital signal processing applications, to the setting of modern dynamic embedded applications where behaviour may change drastically depending on changes in input data or the operational environment.

The execution times of the actors in an SADF graph are a key aspect of the model with direct impacts on characteristics such as throughput, buffer occupancy, or energy usage. So far, SADF has been considered with execution times (or processing delays) either chosen from discrete (in fact, finite-support) probability distributions [12], or sampled from the (negative) exponential distribution with certain (scenario-dependent) rates [7], [10]. The latter has been called exponentially-timed SADF, or eSADF for short.

In this abstract, we generalise both notions of SADF to *generally-timed energy SADF*, or xSADF for short. This allows not only execution times following arbitrary probability distributions (including discrete ones as used by Theelen et al. [12] and continuous ones such as the continuous uniform, normal, or exponential distribution), but also the nondeterministic choice of delays over given (sets of) time intervals, and any combination thereof. Additionally, we include in xSADF a way to specify the energy consumed by an actor during idle times and during different processing modes as a core feature.

To equip our enriched variant of SADF with a formal semantics, we need a formalism that supports the necessary combination of quantitative aspects: soft and hard real-time behaviour, discrete and continuous probabilistic decisions, and nondeterministic choices. We find support for all of these aspects in the model of stochastic timed automata (STA [1]). Despite their expressiveness, STA can be seen as a straightforward generalisation of existing, well-known formalisms such as timed automata or Markov decision processes. Our semantics takes advantage of the fact that STA support compositional modelling through a standard parallel composition operator. The analysis of STA by means of model checking is challenging, but first techniques and implementations have appeared recently [4]. This means that it is possible today to create xSADF models, convert them to their STA semantics, and perform a fully automated model checking analysis.

II. Generally-Timed Energy SADF

In this section, we define generally-timed energy SADF (xSADF) by example. Our definitions are based on the one of Katoen and Wu for eSADF [7], and our notation is mostly consistent with theirs. xSADF extends eSADF by allowing arbitrary deterministic, nondeterministic and stochastic processing times and by adding information about the energy consumption of kernels and detectors in different scenarios. Our example extends the one introduced by Theelen et al. [12]:

In Figure 1a, we display the exemplary structure of an xSADF graph. It consists of the *kernels* $\mathbf{A}$ and $\mathbf{B}$ and the *detectors* $\mathbf{D}$ and $\mathbf{D}'$. They are connected by *channels*: Data channels, drawn with solid lines, carry untyped data tokens, while control channels, drawn with dashed lines, carry typed scenario tokens. In this abstract, we denote the channel that carries tokens from X to Y by ch_Y^X. The types of tokens that a control channel ch can carry are given by $\Sigma(ch)$; here, we have $\Sigma(ch_{\mathbf{D}}^{\mathbf{D}'}) = \{\, u \,\}$ and $\Sigma(ch_{\mathbf{A}}^{\mathbf{D}}) = \Sigma(ch_{\mathbf{B}}^{\mathbf{D}}) = \{\, v, w \,\}$. For data channels, we specify the initial number of tokens on the channel; in our example, we initially have 2 tokens on $ch_{\mathbf{A}}^{\mathbf{B}}$ and 1 token on $ch_{\mathbf{D}}^{\mathbf{B}}$. Control channels are initially empty.

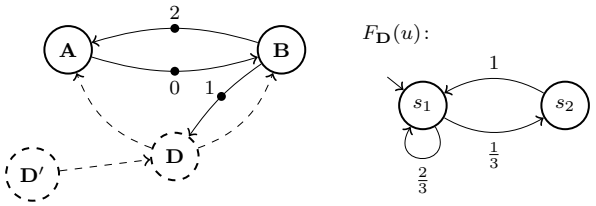
A kernel processes information: It takes a specified number of data tokens from all its incoming data channels, processes the data for some time, and then outputs a number of new data tokens on its outgoing data channels. In each such step, a kernel also consumes a scenario token from each of its incoming control channels; together, these tokens determine the scenario that the kernel operates in. The current scenario determines the number of data tokens consumed and produced, the processing time, and the energy consumption during processing.

A detector is a more powerful kind of kernel in that it is also supports output of scenario tokens. It may still have incoming control channels, but its behaviour is governed by subscenarios: After consuming scenario tokens and thus determining the current scenario, a scenario-specific state machine performs one step (originating from its previous state), with its new state being the current subscenario. We use *discrete-time Markov chains* (DTMC) for this purpose, but other state-based formalisms such as labelled transition systems or Markov decision processes could be used instead without requiring conceptual extensions to the STA semantics we develop in this abstract. The subscenario DTMC that $\mathbf{D}$ uses when it is in scenario u is shown in Figure 1b. The set of subscenarios of $\mathbf{D}$ is thus $\{\, s_1, s_2 \,\}$.

To complete our example, we need to specify the numbers of data tokens consumed and produced by the kernels and detectors, the types of scenario tokens produced by the detectors, the processing times, and the energy consumption rates. For each of these pieces of information and each kernel and detector, we define a function that depends on the current scenario. The production and consumption of data tokens is given by

$$R_{\mathbf{D}} = \{\, \langle s_1, ch_{\mathbf{D}}^{\mathbf{B}} \rangle \mapsto 1, \langle s_2, ch_{\mathbf{D}}^{\mathbf{B}} \rangle \mapsto 1 \,\},$$
$$R_{\mathbf{A}} = R_{\mathbf{B}} = \{\, \langle v,\ ch_{\mathbf{B}}^{\mathbf{A}} \rangle \mapsto 1, \langle v,\ ch_{\mathbf{A}}^{\mathbf{B}} \rangle \mapsto 2,$$
$$\langle w, ch_{\mathbf{B}}^{\mathbf{A}} \rangle \mapsto 0, \langle w, ch_{\mathbf{A}}^{\mathbf{B}} \rangle \mapsto 0 \,\}$$

This work is supported by the EU 7th Framework Programme under grant agreements 295261 (MEALS) and 318490 (SENSATION), the DFG Transregional Collaborative Research Centre SFB/TR 14 AVACS, and by the CAS/SAFEA International Partnership Program for Creative Research Teams.



(a) Graph structure (b) Subscenario DTMC for $\mathbf{D}$

Figure 1: Components of an example xSADF graph

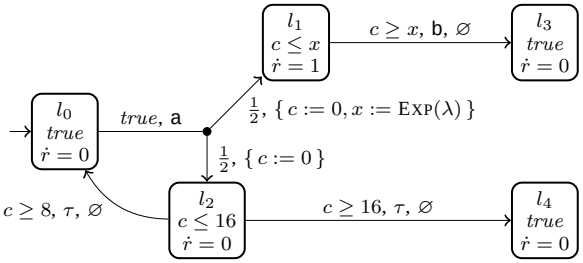


Figure 2: An example stochastic timed automaton [4]

while the type of scenario tokens produced by $\mathbf{D}$ is given by

$$T_{\mathbf{D}} = \{\, \langle s_1, ch^{\mathbf{D}}_{\mathbf{A}} \rangle \mapsto v,\ \langle s_1, ch^{\mathbf{D}}_{\mathbf{B}} \rangle \mapsto v,\\ \langle s_2, ch^{\mathbf{D}}_{\mathbf{A}} \rangle \mapsto w, \langle s_2, ch^{\mathbf{D}}_{\mathbf{B}} \rangle \mapsto w \,\}.$$

To specify the processing times incurred by each kernel or detector, we associate to their possible scenarios an *expression* that characterises the delay:

$$E_{\mathbf{A}} = \{\, v \ \mapsto \textsc{Nondet}([1,3[), \quad w \mapsto \textsc{Det}(1) \,\},$$
$$E_{\mathbf{B}} = \{\, v \ \mapsto \textsc{Sample}(\textsc{Uni}(1,3)), \, w \mapsto \textsc{Det}(1) \,\},$$
$$E_{\mathbf{D}} = \{\, s_1 \mapsto \textsc{Sample}(\textsc{Exp}(\tfrac{1}{2})), \quad s_2 \mapsto \textsc{Det}(1) \,\}$$

where Det denotes a fixed, i.e. deterministic, delay, Nondet denotes a delay that is nondeterministically chosen out of the specified interval, and finally Sample specifies that the delay is to be sampled from the specified probability distribution. In this example, we use $\textsc{Uni}(1,3)$, the continuous uniform distribution over the interval $[1,3]$, and $\textsc{Exp}(\frac{1}{2})$, the (negative) exponential distribution with rate parameter $\frac{1}{2}$. In this model, every node thus waits for exactly one time unit (e.g. 1 second) in the "switched-off" scenario w resp. s_2, while the processing time in the "active" scenario v is nondeterministic for $\mathbf{A}$ and stochastic with mean 2 for $\mathbf{B}$ and $\mathbf{D}$. We note that the STA semantics we introduce in this abstract supports arbitrary probability distributions (e.g. the (truncated) normal distribution, or discrete probability distributions as used by Theelen et al. [12]) as well as more complex expressions such as a nondeterministic choice of delay over an interval whose upper bound has been sampled from some probability distribution.

Finally, in xSADF, we add a notion of quantifiable energy consumption: We specify, for each kernel and detector, how much energy they consume per unit of time in each (sub)scenario and when idle:

$$P_{\mathbf{A}} = \{\, v \mapsto 4,\ w \mapsto \tfrac{1}{8}, \perp \mapsto \tfrac{1}{8} \,\},$$
$$P_{\mathbf{B}} = \{\, v \mapsto 2,\ w \mapsto \tfrac{1}{8}, \perp \mapsto \tfrac{1}{8} \,\},$$
$$P_{\mathbf{D}} = \{\, v \mapsto \tfrac{1}{4}, w \mapsto \tfrac{1}{4}, \perp \mapsto \tfrac{1}{4} \,\}$$

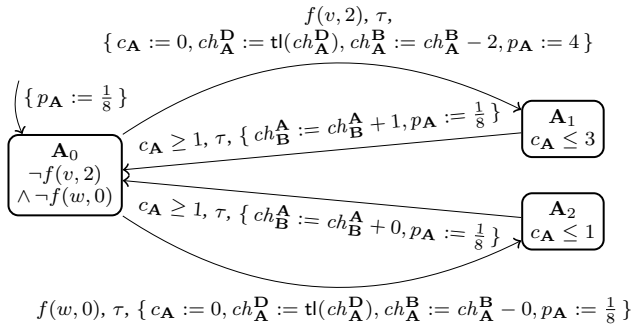
where $\perp$ refers to the idle state when a kernel or detector is waiting for scenario or data tokens to become available. Intuitively, in this example, we thus say that the detector $\mathbf{D}$ always has an energy consumption of 0.25 (e.g. measured in watts when one time unit is taken to correspond to one second), while the detectors both consume $0.125\,\mathrm{W}$ in idle and when "switched off" in scenario w, and $2\,\mathrm{W}$ per when processing data per token in scenario v.

We have not described the detector $\mathbf{D}'$ so far. This is because its only purpose is to illustrate that detectors can receive scenario tokens that determine the subscenario DTMC, and how this is achieved in the STA semantics later on.

III. Stochastic Timed Automata

Stochastic timed automata (STA) are a very expressive automata-based formalism. They allow nondeterministic decisions, real time aspects, continuous and discrete probabilistic choices, and any combination thereof. STA had been introduced as the original formal semantics of the high-level compositional modelling language Modest [1]. They can be viewed as probabilistic timed automata (PTA [8]) with the additional capability to sample from arbitrary (in particular also continuous) probability distributions, or as generalised semi-Markov processes (GSMP) extended with discrete and continuous nondeterminism. In an STA, it is possible to represent all the kinds of delays that we used in our example in the previous section, and more.

Figure 2 shows an example STA. It consists of a set of *locations*, l_0 through l_4, connected by *edges*, and a set of *variables* and *rewards*. Here, we have two variable c and x, where c is a *clock*, and one reward r. Variables have some type, e.g. real numbers or FIFO queues, where clocks are particular in that they have type real, but advance over time at rate 1 (as in timed automata). Each location is associated with an *invariant* (second line) and a number of *rate rewards* (third line) of the form $\dot{v} = x$ for a *reward* v and $x \in \mathbb{R}$. The invariants specify when time is allowed to advance; so in location l_2, time can progress until clock c reaches the value 16, at which point the location has to be left via an edge before time can pass again. If that is not possible, a timelock occurs. When time is spent in some location, the values of all rewards increase at the rate given by the location's rate rewards. Edges are labelled with a *guard* and an *action*, and lead from one location to a symbolic probability distribution over pairs of a target location and a set of assignments. Guards specify when an edge is enabled, i.e. when it can be taken, while the action labels are used in parallel composition (see below). The only outgoing edge of l_0 has guard $true$ and is labelled with action a. It leads to l_1 or l_2 with probability $\frac{1}{2}$ each. In both cases, clock c is *reset*, and when we go to l_1, the real-valued variable x is updated with a randomly selected value according to the exponential distribution with rate λ. In general, the probabilities for an edge's targets can be given via expressions that may refer to variables and that are interpreted as weights. When an edge has a single target location, we omit the branching in the graphical representation of the STA; likewise, we may omit $true$ guards.



with $f(sc,n) \stackrel{\text{def}}{=} |ch^{\mathbf{D}}_{\mathbf{A}}| \geq 1 \wedge \mathsf{hd}(ch^{\mathbf{D}}_{\mathbf{A}}) = sc \wedge ch^{\mathbf{B}}_{\mathbf{A}} \geq n$

Figure 3: STA semantics of the kernel $\mathbf{A}$

In our example STA, we have several types of delays and decisions: The choice of target location when leaving l_0 is *probabilistic*, i.e. it follows a discrete probability distribution. When we arrive in l_2, the edge back to l_0 can be taken after a delay *nondeterministically* chosen between 8 and 16 time units. If we choose to wait for the full 16 time units, there is an additional (discrete) *nondeterministic* choice of going to l_4 instead. When we go from l_0 to l_1, the update of x combined with the way x is subsequently used in the invariant of l_1 and the guard of the edge to l_2 means that the amount of time spent in l_1 follows the exponential distribution with rate λ, i.e. it is a *stochastic* delay.

Given two STA, we can define their parallel composition using a standard interleaving semantics. STA support a CSP-like parallel composition operator $\|$ that forces edges with the same action label to synchronise. In particular, in the parallel composition $M_1 \parallel M_2$, if STA M_1 wants to take an edge labelled a and M_2 also has an edge with the same label at some point, then M_1 is forced to either wait until M_2 is also ready to take an edge labelled a, resulting in synchronisation, or to take an edge with a different label if possible. We also allow variables to be shared between STA that run in parallel. This means that it is possible to specify *inconsistent* assignments, i.e. two edges that synchronise and assign different values to the same variable. We treat this as a modelling error and do not allow models where this is possible. We call a given set of STA $\{ M_1, \ldots, M_n \}$ a *network of STA* (NSTA) and identify it with the parallel composition $M_1 \parallel \ldots \parallel M_n$.

IV. Semantics

We can now give a semantics for xSADF in terms of NSTA. We broadly follow the ideas of the existing semantics of eSADF in terms of Markov automata [7], in particular its use of variables to model channels and its compositional approach. We support the additional features of xSADF, namely its very general execution times and the energy consumption annotations, by transforming them into the corresponding STA constructs that are not available in Markov automata: assignments involving arbitrary general probability distributions for stochastic delays in combination with guards and invariants for continuous nondeterministic choices over time, and rate rewards to model the rate of energy consumption. We again proceed by example by showing the parts of the semantics of the xSADF graph of Section II necessary to understand the approach, with the overall semantics of an entire graph being the network of the STA corresponding to its components.

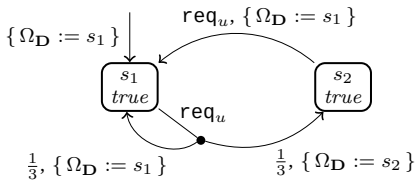


Figure 4: STA for the subscenario DTMC of detector $\mathbf{D}$

The semantics of a kernel is a single STA. For kernel $\mathbf{A}$, it is shown in Figure 3. It has one local variable: $c_{\mathbf{A}}$, a clock. The data processing takes place in locations $\mathbf{A}_1$ and $\mathbf{A}_2$, whose invariants combined with the guards of the outgoing transitions ensure that the correct delay according to $E_{\mathbf{A}}$ is incurred. We see that no edge synchronises with any other STA since they are all labelled with the special silent action τ. The communication with the other components of the semantics of the xSADF graph happens through the channels, which are modelled as shared variables that have the name of the channel itself. We use integer variables for data channels, keeping track of the number of data tokens in the channel. For control channels, the order of the typed scenario tokens matters, so we use variables of type FIFO queue. Operation `hd` returns the first item of a queue, `tl` returns the queue without its first item, and the dot operator $.$ appends an item to the queue.

We use one reward p to keep track of the total energy consumption across the entire xSADF graph. We therefore cannot use rate rewards inside the semantics of the kernels and detectors, and instead keep track of their current rate of energy usage through real-valued variables like $p_{\mathbf{A}}$ in kernel $\mathbf{A}$. We then add an extra STA with a single location and no edges that specifies the aggregate rate reward; in this case, it is $\dot{p} = p_{\mathbf{A}} + p_{\mathbf{B}} + p_{\mathbf{D}} + p_{\mathbf{D}'}$.

The semantics of a detector consists of two STA in order to separate the selection of the subscenario from its higher-level behaviour, which is similar to that of a kernel. The semantics of the subscenario selection DTMC of detector $\mathbf{D}$ when in scenario u is shown in Figure 4. It synchronises with the high-level STA for $\mathbf{D}$, given in Figure 5, via actions `req` indexed with the current scenario and the shared variable $\Omega_{\mathbf{D}}$. The synchronisation via `req` on the edge from location $\mathbf{D}_0$ is used to let the DTMC perform one step as soon as the required scenario tokens are available; the selected subscenario is then communicated back in $\Omega_{\mathbf{D}}$. If $\mathbf{D}$ had more than one scenario, $\mathbf{D}_0$ would have multiple outgoing edges with different indexes on `req`. Otherwise, the semantics of $\mathbf{D}$ is largely similar to that of a kernel, with locations $\mathbf{D}_1$, $\mathbf{D}_2$ and $\mathbf{D}_3$ performing the same basic tasks as $\mathbf{A}_0$, $\mathbf{A}_1$ and $\mathbf{A}_2$, respectively, for kernel $\mathbf{A}$.

V. Analysis

There are currently two possible ways to analyse xSADF using its STA semantics: First, if there is no nondeterminism in the execution times (and subscenarios are selected by DTMC), then all nondeterminism that is present in the concrete semantics of the network of STA is due to the interleaving of the automata, and in particular does not affect the analysis results. In this case, we can use statistical model checking

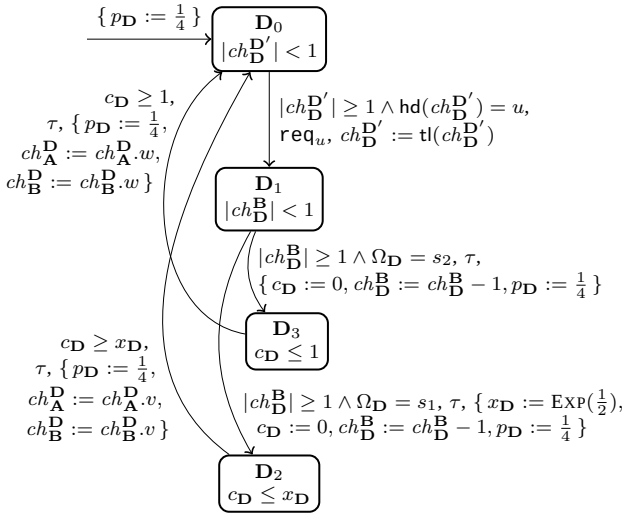


Figure 5: STA for the higher-level behaviour of detector $\mathbf{D}$

(a.k.a. simulation) for an efficient but approximative analysis. In case the xSADF model contains "true" nondeterminism, due to nondeterministic delays or owed to the use of nondeterministic machinery such as Markov decision processes to select subscenarios, we need to perform STA model checking, which is a challenging problem. However, a first approach that allows model checking of general STA has recently been developed [4]. It enables the computation of upper bounds on maximum and lower bounds on minimum probabilistic reachability and expected accumulated reward properties, for both time-bounded and -unbounded properties. It can thus be used to compute, for example, bounds on the worst-case (i.e. maximum) probability of exceeding a channel's capacity, on the (maximum/minimum) probability of having a certain number of tokens in a channel within a time bound, on the (minimum/maximum) expected time until a certain number of data tokens has been processed by a kernel, or on the (maximum/minimum) number of tokens processed with some time bound. The analysis of STA models is supported by the MODEST TOOLSET [6], where the modes tool handles statistical model checking and the recently developed mcsta tool implements the new STA model checking technique [4].

VI. FUTURE WORK

This abstract introduced xSADF and its semantics by example. We are working on the full formal definitions, actually aimng at a proof that all nondeterminism outside of delays and subscenario selection does not affect analysis results, evaluating the approach on case studies of concrete and larger models, and possibly implementing support for the SDF^3 file format in the MODEST TOOLSET, we see the potential for a significant extension in the way energy usage is currently handled in xSADF:

Currently, energy consumption can be observed in (reward-based) properties during the analysis of the model, but this information is not available to the kernels and detectors during their execution within the model itself. Allowing them to observe and react to data such as the current (aggregate) energy consumption rate or the available energy supply could make it possible to represent power management schemes like dynamic voltage and frequency scaling (DVFS) inside an SADF model, and study their effects in the analysis of that model. Of particular interest would be the inclusion of battery models, which would allow a detector to, for example, switch off components when the battery runs low. Adding features like these means that the semantics of such an SADF graph becomes a network of stochastic hybrid automata (SHA [2]). While SHA are well-understood and their analysis w.r.t. the same properties as for STA is possible in theory [3], tools (such as prohver [5]) are currently still in a prototypical stage and limited to very small state spaces.

VII. CONCLUSION

We have presented generally-timed energy SADF, or xSADF for short. By allowing general stochastic and nondeterministic processing delays, it is an extension of both the discrete probabilistically-timed SADF of Theelen et al. [12] and the exponentially-timed eSADF of Katoen and Wu [7]. In addition, it supports scenario-dependent energy usage annotations. We have outlined a compositional semantics of xSADF in terms of stochastic timed automata (STA), which in turn can be analysed using recently developed model checking techniques for STA [4]. We look forward to experimenting with this new model and studying its applicability in terms of both modelling expressiveness and scalability of the analysis techniques.

REFERENCES

[1] H. C. Bohnenkamp, P. R. D'Argenio, H. Hermanns, and J.-P. Katoen, "MoDeST: A compositional modeling formalism for hard and softly timed systems," *IEEE Trans. Software Eng.*, vol. 32, no. 10, pp. 812–830, 2006.

[2] M. Fränzle, E. M. Hahn, H. Hermanns, N. Wolovick, and L. Zhang, "Measurability and safety verification for stochastic hybrid systems," in *HSCC*. ACM, 2011, pp. 43–52.

[3] E. M. Hahn, "Model checking stochastic hybrid systems," Ph.D. dissertation, Universität des Saarlandes, 2013.

[4] E. M. Hahn, A. Hartmanns, and H. Hermanns, "Reachability and reward checking for stochastic timed automata," *Electronic Communications of the EASST*, vol. 70, November 2014.

[5] E. M. Hahn, A. Hartmanns, H. Hermanns, and J.-P. Katoen, "A compositional modelling and analysis framework for stochastic hybrid systems," *Formal Methods in System Design*, vol. 43, no. 2, pp. 191–232, 2013.

[6] A. Hartmanns and H. Hermanns, "The Modest Toolset: An integrated environment for quantitative modelling and verification," in *TACAS*, ser. LNCS, vol. 8413. Springer, 2014, pp. 593–598.

[7] J.-P. Katoen and H. Wu, "Exponentially timed SADF: Compositional semantics, reductions, and analysis," in *EMSOFT*, 2014.

[8] M. Z. Kwiatkowska, G. Norman, R. Segala, and J. Sproston, "Automatic verification of real-time systems with discrete probability distributions," *Theor. Comput. Sci.*, vol. 282, no. 1, pp. 101–150, 2002.

[9] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow: Describing signal processing algorithm for parallel computation," in *COMPCON*. IEEE Computer Society, 1987, pp. 310–315.

[10] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, 2nd ed. CRC Press, 2009.

[11] B. D. Theelen, M. Geilen, T. Basten, J. Voeten, S. V. Gheorghita, and S. Stuijk, "A scenario-aware data flow model for combined long-run average and worst-case performance analysis," in *MEMOCODE*. IEEE, 2006, pp. 185–194.

[12] B. D. Theelen, M. Geilen, and J. Voeten, "Performance model checking scenario-aware dataflow," in *FORMATS*, ser. Lecture Notes in Computer Science, vol. 6919. Springer, 2011, pp. 43–59.

State-Based Real-Time Analysis of SDF Applications on Multi-Cores

Maher Fakih*, Kim Grüttner*, Martin Fränzle† and Achim Rettberg†
*OFFIS – Institute for Information Technology, Oldenburg, Germany
†Carl von Ossietzky Universität Oldenburg, Germany

***Abstract*—The timing predictability of multi-core platforms with hard real-time applications is much more challenging than that of traditional platforms due to their large number of shared processing, communication and memory resources. Yet, this is an indispensable challenge for guaranteeing their safe usage in safety critical domains (avionics, automotive).**

In this paper, a real-time analysis based on model-checking is proposed. The model-checking based method allows to guarantee timing bounds of multiple Synchronous Data Flow Application (SDFA) implementations. This approach utilizes Timed Automata (TA) as a common semantic model to represent WCET of software components (SDF actors) and shared communication resource access protocols for buses, DMA, private local and shared memories of the multi-core platform. The resulting network of TA is analyzed using the UPPAAL model-checker for providing safe timing bounds of the implementation.

We demonstrate our approach using several image processing algorithms and a multi-phase electric motor control algorithm mapped to Infineon's TriCore-based Aurix multi-core hardware platform with two different inter-core communication styles: burst- and single-beat transfer. Our approach shows a significant precision improvement compared with the worst-case bound calculation based on analytical upper-bound delays for every shared resource access.

I. Introduction and Motivation

The growing computational demand of real-time applications (in automotive, avionics and multimedia) requires extensions in the traditional design process to support multi-core architectures. Due to their significantly increased performance and Space Weight and Power (SWaP) reductions, multi-cores offer an appealing alternative to traditional single core architectures. Yet, the timing analysis of hard real-time applications running on multi-core platforms is much more challenging compared to traditional single processor. This comes from the large number of shared processing, communication and memory resources available in today's multi-core platforms.

There are mainly two performance analysis approaches for embedded applications: simulative and formal methods. Both methods have their assets and drawbacks. Even exhaustive simulations never guarantee completeness (i.e. they provide no guarantee that all interesting corner cases are covered), but simulations are capable to handle systems with a huge state space. Formal methods guarantee completeness, but suffer from state explosion and scalability issues. For giving safe timing guarantees under all conditions, a formal approach is needed to calculate safe upper bounds based on Worst-Case-Execution Times (WCETs) of the application and platform dependent upper bounds for communication.

In this work, a mathematical (static) real-time analysis methodology for a subset of data flow oriented applications using model-checking is proposed. Model-checking techniques are capable of verifying the performance properties of a system with rigor, in contrast to simulative approaches. Furthermore, for unmet timing properties counter examples are provided. Unfortunately, these techniques do not scale well with the number of applications and hardware resources of full featured multi-core systems (considering preemption and contention on shared memories and on-chip communication media).

For this reason we constrain our hardware platform to a multi-core architecture where each core has its own instruction and data memory, called a "tile". Tiles are connected through one (or more) arbitrated shared interconnect(s) (bus(s), shared DMA(s)). Communication between tiles is realized through FIFO-style message passing on a shared memory. The multi-core is represented as Architecture Resource Graph (ARG). We also limit applications to the Synchronous Dataflow Flow (SDF) [12] Model of Computation (MoC). In the context of multi-core research [12, 11, 6, 8, 10, 13], the SDF MoC is gaining consideration due to its analyzability (e.g. deadlocks and bounded buffer properties are decidable for such models). In an SDF specification, parallelism is represented explicitly and static schedules can be obtained. Furthermore, SDF semantics supports a clean separation between computation and communication since no communication (resource access) is allowed during the computation phase. This enables a compositional timing analysis where SDF actor execution times can be analyzed independently from communication delays of message passing between SDF actors. A mapping relation between SDF graphs and the multi-core ARG describes the implementation of the application on the multi-core architecture. With these constraints and assumptions an analytical timed-automata model can be constructed as a common semantic model to represent execution time boundaries (Best Case and Worst Case Execution Times) of SDF actors and communication FIFOs, as well as their mapping and utilization of multi-core resources, such as scheduling of SDFs, shared communication resource access protocols for buses, local and shared memories. The resulting network of TA is analyzed using the UPPAAL model-checker for obtaining safe timing bounds of the chosen implementation.

In this paper, we briefly present our state-based real-time analysis framework, which enables (using the UPPAAL model-checker) calculating safe timing bounds of multiple (hard real-time) SDF-based applications on multi-core platforms (represented as network of TA), considering variable access delays due to the contention on shared communication resources. We will summarize the most relevant achievements

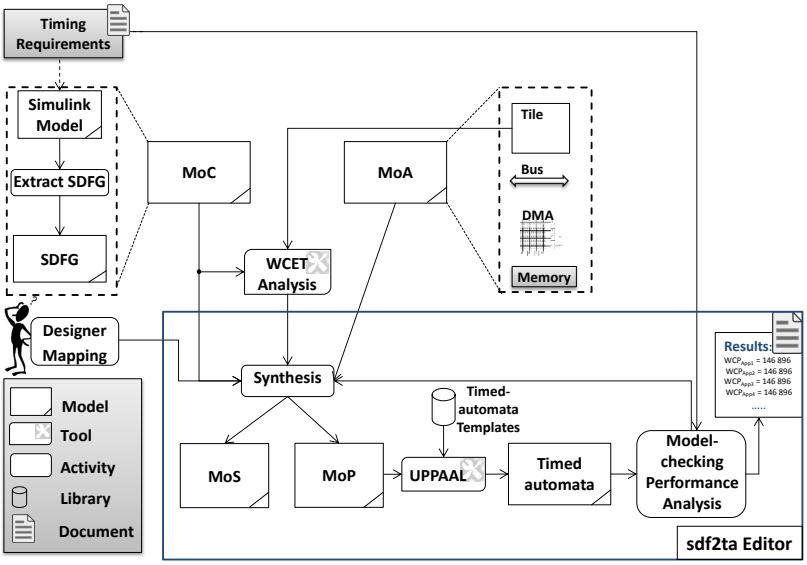


Figure 1: Overall model-based real-time analysis flow

of this approach, focusing on the developed SDF2TA tool. Due to space limitations, the reader is encouraged to refer to [3, 4] for more information.

II. Model-based Real-time Analysis of SDFGs

Fig. 1 depicts the overall analysis flow proposed in this work. Typically, the focus of the system designer is on the representation of the application in the proper MoC and mapping it to the available platform resources. The mapping constraints considered in this article depend mainly on the timing requirements. The input of our analysis flow is an SDF Model of Computation (MoC) and a Model of Architecture (MoA). If the functional input model is described in Matlab/Simulink, an SDFG can be obtained from the Matlab/Simulink model using the translation procedure described in [2]. This transformation transforms the top-level Simulink blocks into SDF actors, while keeping the structure as well as the precedence/activation relationships of all blocks.

The target platform is represented in the MoA by combining tiles (processor with private memories) with other hardware components (DMA, buses, shared memories see Fig. 1 top right). Afterwards a WCET analysis (using aiT [5] or chronos [9]) is performed for all combinations of actors and available processing elements (PEs) of the platform. It is important to note that this WCET analysis estimates lower/upper bounds of the actors' execution time on single PEs without considering the timing effects of inter-core communication on shared communication resources in the platform. To enable this timing analysis, C-code implementation of each actor is needed. In the case the application is available as a Simulink model, C-code can be generated from it using a code-generator (e.g. Simulink Coder). Next, a synthesis activity takes place, which takes mapping and scheduling decisions (manually chosen by the designer) as an input, maps all SDF actors to tiles and all SDF edges to communication resources, and configures the scheduling and arbitration strategies of resources, resulting into an annotated parallel Hardware/Software model (called Model of Structure (MoS)). The MoS would then be used for further refinement steps (such as implementation on a virtual-or on a real hardware platform) which is not considered in this work.

In order to be able to verify that the timing of all mapped SDFGs stay within specified bounds, we must keep track of all possible timing delays including delays caused by communication interferences in the multi-core platform. To achieve this, a Model of Performance (MoP) is extracted from the synthesis process. The MoP is a network of TA representing all actor WCETs, communication delays, scheduling and communication resource access protocols of the platform [3]. Pre-defined TA templates are configured and instantiated in the UPPAAL framework, taking into account the mapping, timing and platform configuration. After converting the timing requirements into UPPAAL CTL queries, performance analysis (e.g. end-to-end deadline) is done using the UPPAAL model-checker.

For the purpose of automating these last steps, as highlighted in Fig. 1, we developed the SDF2TA editor using the Eclipse Modeling Framework (EMF) [1] Ecore-model, where the developer can provide all needed parameters (SDFGs, mapping, hardware constraints) and the equivalent UPPAAL system is generated automatically.

Fig. 2 shows the work flow of the SDF2TA tool. First,

[1] http://www.eclipse.org/modeling/emf/

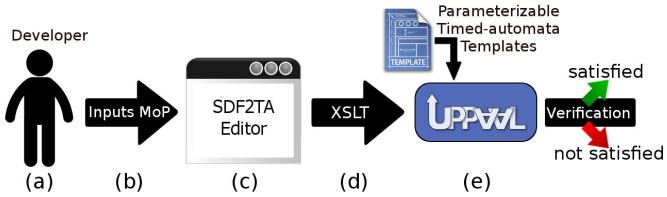


Figure 2: Work flow of the SDF2TA tool

the developer needs to provide all relevant timing properties as an input to the SDF2TA editor. The SDF2TA editor is already provided with an Ecore-model (which is an XML like format) of the supported MoP and can validate if the input conforms with the model definition. If the input validation step is successful, the developer can now choose some property to be checked on the given MoP. Properties which can be checked are either *timing properties* such as the period, end-to-end deadline or *liveness properties* such as checking if repetition vector of an SDFG is valid or if specific states are reached (for e.g. is finishing of sink actors always guaranteed to be after finishing source actors for all possibilities) during execution. Now, the SDF2TA generates an XML equivalent representation of the input model and processes it by applying pre-defined XSLT (Extensible Stylesheet Language Transformations) rules to transform it into an equivalent UPPAAL XML representation. This step includes the parametrization and instantiation of our pre-defined UPPAAL Timed-Automata templates, which represent different components of the MoP. At the end, the generated UPPAAL model is checked against CTL (Computation Tree Logic) queries using the verifyta[2] tool. The verification results are provided to the developer through the SDF2TA editor.

SDF2TA editor (see Fig. 3) provides a user-friendly GUI, supports tool-tips and validates input models in order to generate a correct timed-automata model. Furthermore, the usage of Ecore as the common modeling language for capturing the MoP makes SDF2TA maintainable. If any extensions or changes should be done on the MoP, the developer only needs to change the Ecore-model and with minimal additional effort, modified the SDF2TA editor is generated. In addition, our SDF2TA editor allows to import SDFGs from the well known SDF^3 tool [14], and is able to run SDF^3 in the background to perform different analysis of the SDFGs created in SDF2TA, such as finding and adopting values of repetition vector, buffer sizes, etc.

III. Real-Time Analysis Results

Our approach was first presented and evaluated in [3] with a focus on reducing overestimates in the Worst Case Period (WCP) of an SDF in comparison to a an existing analytical approach [11]. In this work, the authors calculate the worst-case waiting times for a non-preemptive SDF actor scheduling using a FCFS (First-Come-First-Serve) strategy, by assuming that all other competing actors, mapped to the same resource, always run before the waiting actor. We use the system, shown in Fig. 5, which consists of two SDFGs mapped to a 4-tile shared bus platform, configured with the same parameters as used in [11]. Afterward, we calculated the WCP using our model-checking-based approach (MC WCP) and the analytical approach from [11] (Pess. WCP) for different mappings of the SDFGs (see Fig. 4). We define improvement as $((Pess.WCP/MC.WCP)-1)\times 100$ which describes how far could the results be tightened through our approach (in percent) compared to the analytical approach from [11].

We have also started to evaluate our approach with regard to scalability in terms of tiles and actors. Our current restrictions

[2]This is the stand-alone UPPAAL timed-automata verification tool.

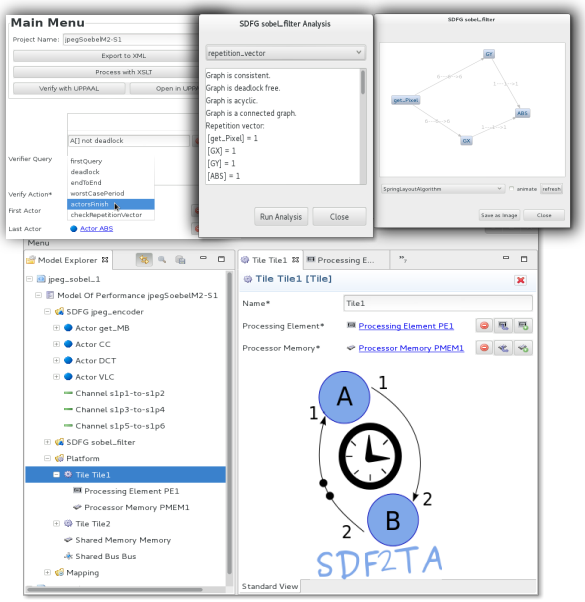


Figure 3: SDF2TA GUI

on the system model made it possible to analyze up to 36 actors on 4 tiles and up to 96 actors on 2 tiles without running into state-space induced complexity problems of the model-checker. In future work, we intend to further increase the actual number of actors by applying an actor partitioning and fusion algorithm [1] before applying model checking.

In [4] we demonstrated the applicability of our analysis method to an industrial automotive case-study of a multi-phase electric motor control algorithm running on a commercially available state-of-the-art multi-core platform (Aurix TC275 [7]). Furthermore, we have successfully used our approach to obtain upper and lower bounds on the execution times of two different communication implementation styles (see Fig. 6). Excessive simulation has been used to analyze the over-approximation of our approach, resulting in an acceptable range between 37% up to 63%.

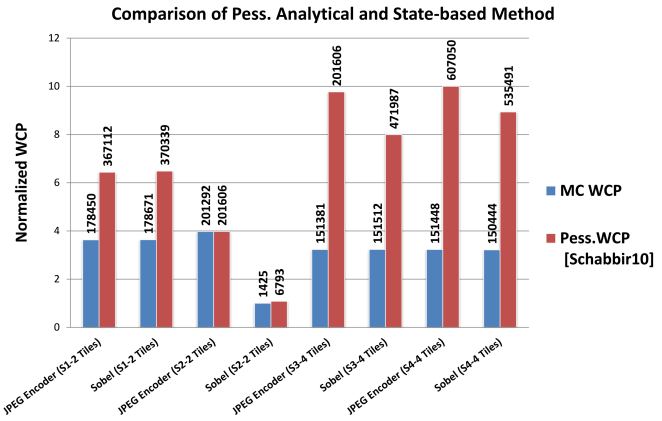
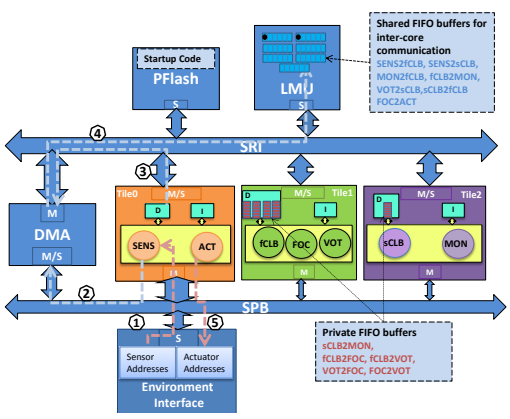
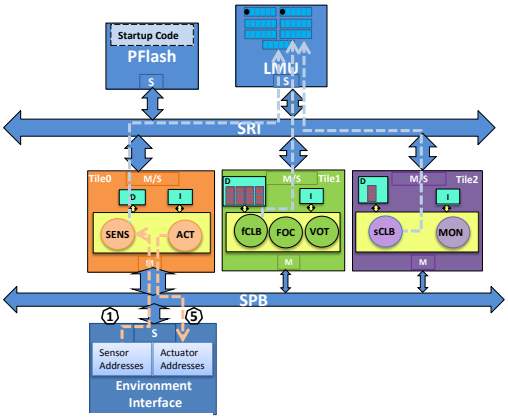


Figure 4: Worst Case Period (WCP) analysis results





nsfer inter-core communication via DMA

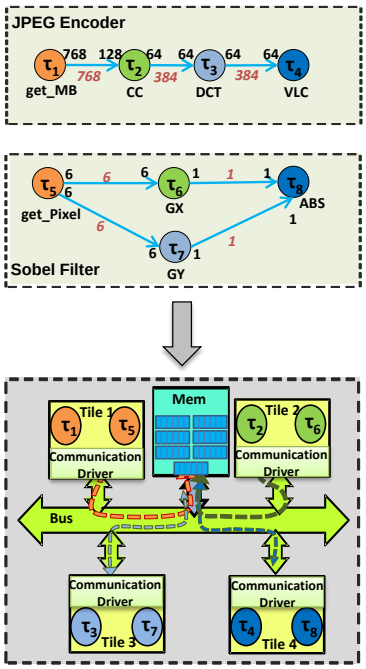


Figure 5: Mapping of JPEG encoder and Sobel filter on a 4-tiles platform

IV. Conclusion

In this article, we have summarized our previously published model-checking-based performance analysis method for the validation of multiple hard real-time SDFGs mapped to multi-core platforms with shared communication resources. Our method shows a significant reduction in the worst-case period time prediction, compared to an existing analysis method. Furthermore, we have demonstrated the applicability of our approach to a multi-phase electric motor control algorithm mapped to Infineon's TriCore-based Aurix multi-core platform, using DMA-based burst-transfer and non DMA single-beat inter-core communication.

Acknowledgement

This work has been partially supported by the *MotorBrain* ENIAC project (13N11480) and the *EMC*^{2} collaborative ARTEMIS project (01IS14002R), both funded by the German Federal Ministry of Research and Education (BMBF). Special thanks to Christof Schlaak who helped developing the SDF2TA tool.

References

[1] Bhattacharyya S, Murthy P, Lee E (1997) APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations. Design Automation for Embedded Systems 2(1):33–60

[2] Boström P, Grönblom R, Huotari T, Wiik J (2010) An Approach to Contract-Based Verification of Simulink Models. No. 985 in TUCS Technical Reports, Turku Centre for Computer Science

[3] Fakih M, Grüttner K, Fränzle M, Rettberg A (2013) Towards performance analysis of SDFGs mapped to shared–bus architectures using model–checking. In: Proceedings of the Conference on Design, Automation and Test in Europe, European Design and Automation Association, Leuven, Belgium, DATE '13

[4] Fakih M, Grüttner K, Fränzle M, Rettberg A (2014) Multicore performance analysis of a multi–phase electrical motor controller. In: Proceedings of the Embedded Real Time Software and Systems Congress (ERTS^2) 2014

[5] Ferdinand C, Heckmann R (2004) ait: Worst-case execution time prediction by static program analysis. In: Building the Information Society, Springer, p 377–383

[6] Ghamarian A (2008) Timing Analysis of Synchronous Data Flow Graphs. PhD thesis, Eindhoven University of Technology

[7] Infineon Technologies (2013) AURIX – Safety joins Performance. http://www.infineon.com/

[8] Kumar A (2009) Analysis, Design and Management of Multimedia Multiprocessor Systems. PhD thesis, Ph. D. thesis, Eindhoven University of Technology

[9] Li X, Liang Y, Mitra T, Roychoudhury A (2007) Chronos: A timing analyzer for embedded software. Science of Computer Programming 69:56–67

[10] Moonen A (2009) Predictable Embedded Multiprocessor Architecture for Streaming Applications. PhD thesis, Eindhoven University of Technology

[11] Shabbir A, Kumar A, Stuijk S, Mesman B, Corporaal H (2010) CA-MPSoC: An Automated Design Flow for Predictable Multi-processor Architectures for Multiple Applications. Journal of Systems Architecture 56(7):265–277

[12] Sriram S, Bhattacharyya SS (2000) Embedded Multiprocessors: Scheduling and Synchronization, 1st edn. CRC Press

[13] Stuijk S (2007) Predictable Mapping of Streaming Applications on Multiprocessors. PhD thesis, Faculty of Electrical Engineering, Eindhoven University of Technology, The Netherlands

[14] Stuijk S, Geilen M, Basten T (2006) SDF^ 3: SDF for free. In: Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on, p 276–278

Science Reports **Department of Mathematics and Computer Science**
Technische Universiteit Eindhoven

If you want to receive reports, send an email to: wsinsan@tue.nl (we cannot guarantee the availability of the requested reports).

In this series appeared (from 2012):

12/01	S. Cranen	Model checking the FlexRay startup phase
12/02	U. Khadim and P.J.L. Cuijpers	Appendix C / G of the paper: Repairing Time-Determinism in the Process Algebra for Hybrid Systems ACP
12/03	M.M.H.P. van den Heuvel, P.J.L. Cuijpers, J.J. Lukkien and N.W. Fisher	Revised budget allocations for fixed-priority-scheduled periodic resources
12/04	Ammar Osaiweran, Tom Fransen, Jan Friso Groote and Bart van Rijnsoever	Experience Report on Designing and Developing Control Components using Formal Methods
12/05	Sjoerd Cranen, Jeroen J.A. Keiren and Tim A.C. Willemse	A cure for stuttering parity games
12/06	A.P. van der Meer	CIF MSOS type system
12/07	Dirk Fahland and Robert Prüfer	Data and Abstraction for Scenario-Based Modeling with Petri Nets
12/08	Luc Engelen and Anton Wijs	Checking Property Preservation of Refining Transformations for Model-Driven Development
12/09	M.M.H.P. van den Heuvel, M. Behnam, R.J. Bril, J.J. Lukkien and T. Nolte	Opaque analysis for resource-sharing components in hierarchical real-time systems - extended version –
12/10	Milosh Stolikj, Pieter J. L. Cuijpers and Johan J. Lukkien	Efficient reprogramming of sensor networks using incremental updates and data compression
12/11	John Businge, Alexander Serebrenik and Mark van den Brand	Survival of Eclipse Third-party Plug-ins
12/12	Jeroen J.A. Keiren and Martijn D. Klabbers	Modelling and verifying IEEE Std 11073-20601 session setup using mCRL2
12/13	Ammar Osaiweran, Jan Friso Groote, Mathijs Schuts, Jozef Hooman and Bart van Rijnsoever	Evaluating the Effect of Formal Techniques in Industry
12/14	Ammar Osaiweran, Mathijs Schuts, and Jozef Hooman	Incorporating Formal Techniques into Industrial Practice
13/01	S. Cranen, M.W. Gazda, J.W. Wesselink and T.A.C. Willemse	Abstraction in Parameterised Boolean Equation Systems
13/02	Neda Noroozi, Mohammad Reza Mousavi and Tim A.C. Willemse	Decomposability in Formal Conformance Testing
13/03	D. Bera, K.M. van Hee and N. Sidorova	Discrete Timed Petri nets
13/04	A. Kota Gopalakrishna, T. Ozcelebi, A. Liotta and J.J. Lukkien	Relevance as a Metric for Evaluating Machine Learning Algorithms
13/05	T. Ozcelebi, A. Weffers-Albu and J.J. Lukkien	Proceedings of the 2012 Workshop on Ambient Intelligence Infrastructures (WAmIi)
13/06	Lotfi ben Othmane, Pelin Angin, Harold Weffers and Bharat Bhargava	Extending the Agile Development Process to Develop Acceptably Secure Software
13/07	R.H. Mak	Resource-aware Life Cycle Models for Service-oriented Applications managed by a Component Framework
13/08	Mark van den Brand and Jan Friso Groote	Software Engineering: Redundancy is Key
13/09	P.J.L. Cuijpers	Prefix Orders as a General Model of Dynamics

14/01	Jan Friso Groote, Remco van der Hofstad and Matthias Raffelsieper	On the Random Structure of Behavioural Transition Systems
14/02	Maurice H. ter Beek and Erik P. de Vink	Using mCRL2 for the analysis of software product lines
14/03	Frank Peeters, Ion Barosan, Tao Yue and Alexander Serebrenik	A Modeling Environment Supporting the Co-evolution of User Requirements and Design
14/04	Jan Friso Groote and Hans Zantema	A probabilistic analysis of the Game of the Goose
14/05	Hrishikesh Salunkhe, Orlando Moreira and Kees van Berkel	Buffer Allocation for Real-Time Streaming on a Multi-Processor without Back-Pressure
14/06	D. Bera, K.M. van Hee and H. Nijmeijer	Relationship between Simulink and Petri nets
14/07	Reinder J. Bril and Jinkyu Lee	CRTS 2014 - Proceedings of the 7th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems
14/08	Fatih Turkmen, Jerry den Hartog, Silvio Ranise and Nicola Zannone	Analysis of XACML Policies with SMT
14/09	Ana-Maria Şutîi, Tom Verhoeff and M.G.J. van den Brand	Ontologies in domain specific languages – A systematic literature review
14/10	M. Stolikj, T.M.M. Meyfroyt, P.J.L. Cuijpers and J.J. Lukkien	Improving the Performance of Trickle-Based Data Dissemination in Low-Power Networks
15/01	Önder Babur, Tom Verhoeff and Mark van den Brand	Multiphysics and Multiscale Software Frameworks: An Annotated Bibliography
15/02	Various	Proceedings of the First International Workshop on Investigating Dataflow In Embedded computing Architectures (IDEA 2015)