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).

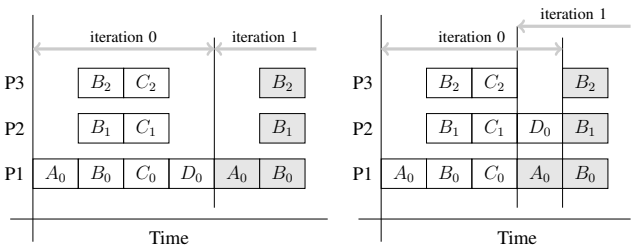
In real-time systems, for given limited set of processors the tasks should satisfy constraints on both throughput (*i.e.,* period) and latency (*i.e.,* response time, deadline). What makes the problem harder is the typical lack of support of task preemption in DSP multi-cores, which invalidates many 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 $\ell < \ell_{\max}$

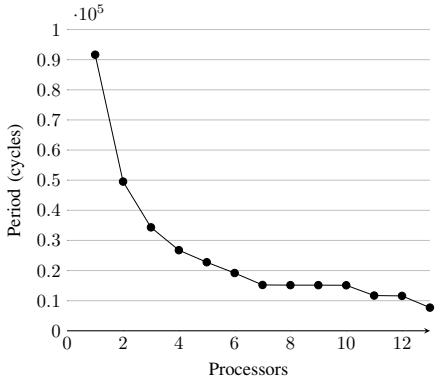
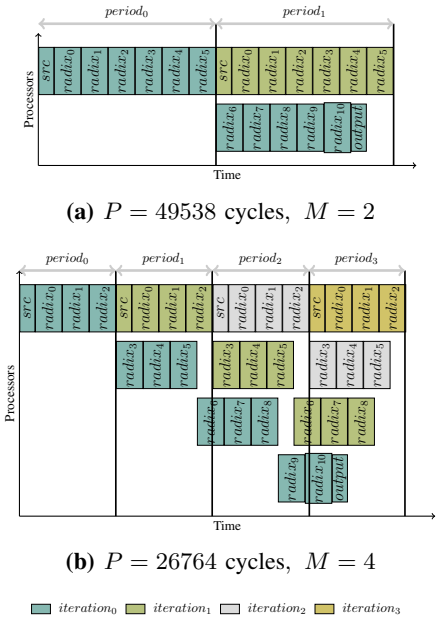


Fig. 2: Radix Sort : Processors used vs Period exploration



(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.

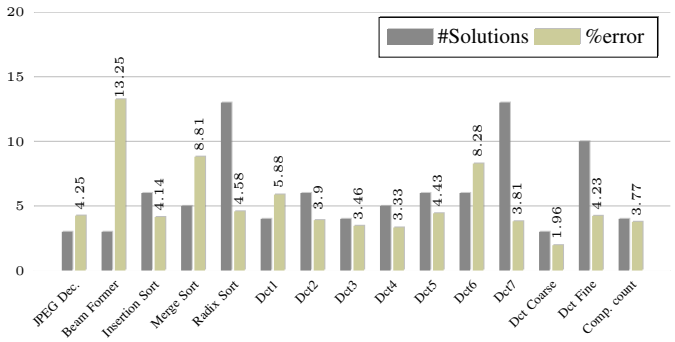


Fig. 4: Application benchmarks: maximum error for predicted vs. measured period

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.