Static Optimal Scheduling for Synchronous Data Flow Graphs with Model Checking

Xue-Yang Zhu[1(✉)], Rongjie Yan[1], Yu-Lei Gu[1,2], Jian Zhang[1], Wenhui Zhang[1], and Guangquan Zhang[2]

[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[2] School of Computer Science and Technology, Soochow University, Suzhou, China
{zxy,yrj,guyl,zj,zwh}@ios.ac.cn, gqzhang@suda.edu.cn

Abstract. Synchronous data flow graphs (SDFGs) are widely used to model digital signal processing and streaming media applications. In this paper, we present exact methods for static optimal scheduling and mapping of SDFGs on a heterogenous multiprocessor platform. The optimization criteria we consider are throughput and energy consumption, taking into account the combination of various constraints such as auto-concurrency and buffer sizes. We present a concise and flexible (priced) timed automata semantics of system models, which include an SDFG and a multiprocessor platform, and formulate the optimization goals as temporal logic formulas. The optimization and scheduling problems are then transformed to model checking problems, which are solved by UPPAAL (CORA). Thanks to the exhaustive exploration nature of model checking and the facility of the tools, we obtain two pareto-optimal schedules, one with an optimal throughput and a best energy consumption and another with an optimal energy consumption and a best throughput. The approach is applied to two real applications with different parameters. The case studies show that our approach can deal with moderate models within reasonable execution time and reveal the impacts of different constraints on optimization goals.

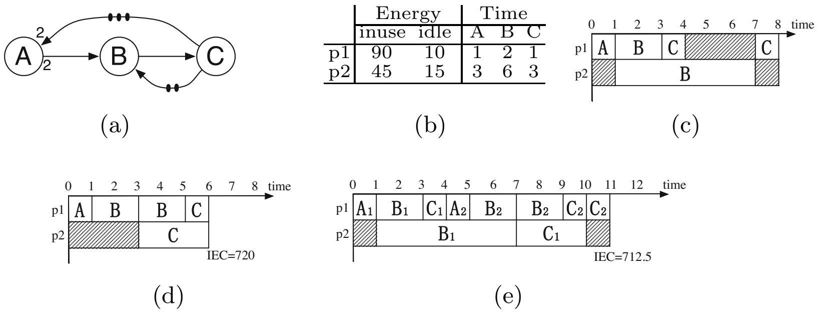
Keywords: Data Flow Graphs · Throughput · Energy Consumption · Multi-constraint · Timed Automata · UPPAAL

1 Introduction

Synchronous data flow graphs (SDFGs) [16] are widely used to represent DSP and streaming media applications, such as a spectrum analyzer [25] and an MPEG-4 decoder [23]. Such applications are usually operated on multiprocessor platforms and under real-time and resource constraints. In this paper, we are concerned with constructing efficient static (compile-time) schedules of SDFGs on a heterogeneous multiprocessor platform.

This work is partially supported by National Key Basic Research Program of China (973 program) (No. 2014CB340701) and the National Natural Science Foundation of China (Nos. 61472406, 61472474, 61272135, 61361136002 and U1435220).

Each node (also called actor) in an SDFG represents a computation and each edge models a FIFO channel; the sample rates of actors may differ. *Homogenous synchronous data flow graphs* (HSDFGs) are a special type of SDFGs, of which all sample rates of actors are set to 1. A *static schedule* arranges the actors of an SDFG to be executed repeatedly, also called a *periodic schedule*. Execution of all the actors for the required number of times is referred to as an *iteration*, which may include more than one execution, also called a *firing*, of an actor. Different actors may fire a different number of firings. Actor B in SDFG G_1, shown in Fig. 1(a), for example, fires twice in an iteration, while A fires once. The average computation time per iteration is called *iteration period* (IP). The IP is the reciprocal of the *throughput*. We use IP and throughput alternatively in the remainder of the paper. The *iteration energy consumption* (IEC) is the average energy consumption per iteration.



	Energy		Time		
	inuse	idle	A	B	C
p1	90	10	1	2	1
p2	45	15	3	6	3

Fig. 1. The system model $\mathcal{M}_1$ and its schedules. (a) The SDFG G_1; (b) the execution platform P_1 and the execution time of actors in G_1 on different processors; (c) an ASAP periodic schedule of G_1 with IP=8; (d) a periodic schedule of G_1 with IP=6; (e) an unfolding schedule of G_1 with IP=$\frac{11}{2}$. The sample rates in the SDFG are omitted when they are 1; black dots on edges represent initial tokens on the edges.

For homogeneous multiprocessor scheduling of SDFGs, an *as soon as possible* (ASAP) execution can be used to find schedules with minimal IP [24]. For heterogeneous multiprocessor scheduling, however, an ASAP schedule is not necessarily throughput-optimal. The ASAP schedule shown in Fig. 1(c), for example, arranges executions of actors of G_1 on a platform including two heterogeneous processors as shown in Fig. 1(b). It has an IP larger than the IP of another schedule shown in Fig. 1(d), which is not ASAP.

Scheduling f iterations as one *schedule cycle* may lead to more options for parallel execution and therefore may reduce the IP and the IEC of a schedule. This is *unfolding* scheduling [19] and f is called *unfolding factor*. See Fig. 1(e) for example. The IP of a periodic schedule of G_1 with unfolding factor 2 is $\frac{11}{2}$, smaller than that of the schedule shown in Fig. 1(d). The IEC is also improved.

In this paper, we present exact methods to schedule and map an SDFG on a heterogeneous multiprocessor platform. The schedules are pareto-optimal. That is, they are either throughput-optimal with a best energy consumption or energy consumption-optimal with a best throughput. Other kinds of constraints, e.g. buffer size constraints, are also considered and integrated into the framework of the proposed methods.

For a given platform and a given unfolding factor, even if we consider only one optimization criterion, e.g. throughput, the scheduling and mapping problem is already NP-complete [20]. For solving the multi-constraint and multi-criterion problems we are considering, we use model checking, which is widely acknowledged to be a powerful tool for such problems.

Actors of an SDFG can fire concurrently if the tokens and other required resources are available. For the analysis of the time and resource constraints, it is appropriate to model the behavior of SDFGs as networks of (priced) timed automata [3] [4], and we choose the real-time model checking tools UPPAAL (CORA) [15] [4] as the back-end solvers. Our contributions are as follows.

1. We present a concise (priced) timed automata (TA) semantics of system models, which include an SDFG and a multiprocessor platform. Various constraints can be integrated flexibly.
2. Based on the semantics, we present two novel exact methods: one for finding static schedules with an optimal throughput and a best energy consumption, and the other for finding static schedules with an optimal energy consumption and a best throughput for SDFGs on heterogenous multiprocessor platforms. Optimal solutions under various constraints are guaranteed.
3. We implement the methods and apply it to two real applications. Although state explosion is inevitable as the models become larger (for checking NP-complete problems), the experimental results show that our methods can deal with moderate models within reasonable time and reveal the impacts of different constraints on optimization goals.

The remainder of this paper is organized as follows. We introduce related work in Section 2. The input models and the problems addressed are formulated in Section 3 and (priced) timed automata are introduced in Section 4. Our main contributions are illustrated in Sections 5, 6 and 7. Section 8 provides case studies. Section 9 concludes and discusses future work.

2 Related Work

Scheduling SDFGs according to different optimization goals have been studied extensively [16], [13], [21], and there are also many studies on real-time schedulability analysis using model checking [11] [8] [1] [17] [5]. Here we review those works most related to our methods, which solve scheduling problems of SDFGs via model checking.

Using model checking to schedule SDFGs according to a particular optimization goal was first presented by Geilen et al. [9]. They focus on buffer minimization problem on a single processor with model checker SPIN [14]. [10] and [12] solve the same problem with NuSMV [6] and SPIN, respectively.

The closest works to our methods are [7] and [18]. Both use UPPAAL as a solver to analyze or schedule SDFGs on a heterogeneous platform. The main differences between them and our methods are summarized as follows.

1. The problems addressed are different. [7] analyzes the schedulability for a given timing constraint, [18] schedules an SDFG to achieve a minimal makespan (i.e. the IP of 1-schedule in this paper), while we consider multiple optimization goals and constraints.
2. The input models are different. In [7], actors of SDFGs are binding to some core and edges to memories, while in our methods, no binding are considered. On the contrary, we try to find bindings according to the optimization goals. In [18], besides data dependencies between actors, task parallelism is explicitly denoted by split and join nodes. In our methods, only data dependencies available in the models, task parallelism needs to be explored to decide whether two tasks can be executed concurrently.
3. The transformations are different. [7] transforms each actor to a TA and each processor to an NTA. In [18], each possible allocation is represented by a TA and each possible communication is also represented by a TA. In our methods, we combine the behavior of actors on processors. The conciseness makes our methods easy to be extended to deal with additional constraints.

3 Model Description and Problem Formulation

An *execution platform* P is a set of heterogeneous processors. A computation may require different amounts of running time if it is executed on different processors. The energy consumption for each processor p is defined by $uEC(p)$ and $iEC(p)$, indicating the energy consumption per unit time when p is used and when p is idle, respectively.

A *synchronous dataflow graph* (SDFG) is a finite directed graph $G = \langle V, E\rangle$, where V is the set of actors, modeling the computations of the system; E is the set of directed edges, modeling interconnections between computations. Each edge e is weighted with three properties, $d(e)$, $prd(e)$ and $cns(e)$, where $d(e)$ is the number of initial tokens on e, $prd(e)$ is the number of tokens produced onto e by each firing of the source of e, and $cns(e)$ is the number of tokens consumed from e by each firing of the sink actor of e. These numbers are also called the *delay*, *production rate* and *consumption rate*, respectively. The source actor and sink actor of e are denoted by $src(e)$ and $snk(e)$, respectively. The set of incoming edges to actor α is denoted by $InE(\alpha)$, and the set of outgoing edges from α by $OutE(\alpha)$. If $prd(e) = cns(e) = 1$ for each $e \in E$, G is a *homogeneous SDFG* (HSDFG).

If execution platform P is considered, each actor α is weighted with computation times $t(\alpha, p)$, for all $p \in P$. Normally, $t(\alpha, p)$ is a positive integer. For technical reason, we also allow $t(\alpha, p)$ to be 0 or -1. The former is used for some dummy actors; the latter is used when α is not allowed to run on p.

An SDFG G is *sample rate consistent* [16] if and only if there exists a positive integer vector $q(V)$ satisfying *balance equations*, $q(src(e)) \times prd(e) = q(snk(e)) \times$

$cns(e)$ for all $e \in E$. The smallest such q is called the *repetition vector*. We use q to represent the repetition vector directly. For example, a balance equation can be constructed for each edge of G_1 in Fig. 1 (a). By solving the equations, we have G_1's repetition vector $q = [1, 2, 2]$. An *iteration* is a firing sequence in which each actor α occurs exactly $q(\alpha)$ times. Only sample rate consistent and deadlock-free SDFGs are meaningful in practice. We consider only such SDFGs, which can be verified efficiently [16].

Definition 1 (System model). *A* system model *includes an SDFG G and its execution platform P, denoted by $\mathcal{M} = (G, P)$.*

A *static schedule* arranges computations of an algorithm to be executed repeatedly. An *unfolding schedule* of system model $\mathcal{M} = (G, P)$ is a static schedule arranging f consecutive iterations of G running on P. The number f is called *unfolding factor* and the f iterations form a *schedule cycle.*

Definition 2 (f-schedule). *An* f-schedule *of system model $\mathcal{M} = (G, P)$ is a function $S : V \times \mathbb{N} \to \mathbb{N} \times P$, where $\mathbb{N}$ is the set of non-negative integers, defining the time arrangement and the processor allocation of firings of actors in G. Schedule S with a cycle period (CP) T is defined as follows. For the i^{th} firing of actor α, denoted by (α, i), $i \in [1, \infty)$:*

1. *$S(\alpha, i).st$ is (α, i)'s start time, when there are sufficient tokens on each $e \in InE(\alpha)$ for a firing of α;*
2. *$S(\alpha, i).pa$ is the processor assigned to (α, i), which is available at the moment $S(\alpha, i).st$;*
3. *$S(\alpha, i + f \cdot q(\alpha)).st = S(\alpha, i).st + T$;*
4. *$S(\alpha, i + f \cdot q(\alpha)).pa = S(\alpha, i).pa$*

Such a schedule can be represented by the first f iterations and period T. It is the part of the schedule defined by $S(\alpha, i)$ with $1 \le i \le f \cdot q(\alpha)$ for all α. From now on, we only consider the finite part of f-schedules.

The *iteration period* (IP) of S is the average computation time of an iteration, that is, $IP = \frac{T}{f}$.

The energy consumption of f-schedule S can be computed as follows. For conciseness, we omit parameters S and f when it is clear in context. Denote the set of all firings assigned on processor p by $AonP(p)$.

$$AonP(p) \equiv_{def} \{(\alpha, i) | S(\alpha, i).pa = p \wedge i \in [1, f \cdot q(\alpha)] \wedge \alpha \in V\}.$$

The total time p occupied in S is

$$occT(p) = \sum_{(\alpha,i) \in AonP(p)} t((\alpha, i), p), \text{ where } t((\alpha, i), p) = t(\alpha, p). \tag{1}$$

Then the energy consumption of S is

$$EC = \sum_{p \in P} occT(p) \cdot uEC(p) + [T - occT(p)] \cdot iEC(p). \tag{2}$$

The *iteration energy consumption* (IEC) of S is the average energy consumption per iteration, that is, $IEC = \frac{EC}{f}$.

Given a system model $\mathcal{M} = (G, P)$ and an unfolding factor f, suppose the set of all f-schedules of $\mathcal{M}$ is $\mathbf{S}$, the problems we address are:

1. how to find an f-schedule S_{optP} such that

$$\begin{aligned} IP(S_{optP}) &= \min\{IP(S) | S \in \mathbf{S}\}, \text{ and} \\ IEC(S_{optP}) &= \min\{IEC(S) | S \in \mathbf{S} \wedge IP(S) = IP(S_{optP})\} \end{aligned}$$

2. how to find an f-schedule S_{optE} such that

$$\begin{aligned} IEC(S_{optE}) &= \min\{IEC(S) | S \in \mathbf{S}\}, \text{ and} \\ IP(S_{optE}) &= \min\{IP(S) | S \in \mathbf{S} \wedge IEC(S) = IEC(S_{optE})\} \end{aligned}$$

4 Introduction to Timed Automata

In this section we recap the concepts of syntax and semantics of timed automata (TA) [3] and its extension with cost [4]. Let X be a set of clocks, $\mathcal{V}$ be a set of bounded integer variables. We use $C(X, \mathcal{V})$ and $U(X, \mathcal{V})$, respectively, to denote the set of linear constraints and the set of updates over clocks and integer variables, where updates on clocks are restricted to resetting clock variables to zero.

A TA is a tuple $(L, X, \mathcal{V}, \mathcal{E}, Inv, l_0)$, where L is a set of locations, $\mathcal{E} \subseteq L \times C(X, \mathcal{V}) \times U(X, \mathcal{V}) \times L$ is a set of edges, $Inv : L \to C(X, \mathcal{V})$ assigns invariants to locations, and l_0 is the initial location. A network of n timed automata (NTA) is a tuple of timed automata $A_1 || \cdots || A_n$ over X, $\mathcal{V}$. A clock valuation γ for a set X is a mapping from X to $\mathbb{R}^+$, where $\mathbb{R}^+$ is the set of non-negative real numbers. A variable valuation u is a function from $\mathcal{V}$ to $\mathbb{Z}$, where $\mathbb{Z}$ is the set of integers. A pair of valuation (γ, u) satisfies a constraint ϕ over X and $\mathcal{V}$, denoted by $(\gamma, u) \models \phi$, if and only if ϕ evaluates to *true* with the valuations γ and u. Let $\gamma_0(x) = 0$ for all $x \in X$. For $\delta \in \mathbb{R}^+$, $\gamma + \delta$ denotes the clock valuation that maps every clock x to the value $\gamma(x) + \delta$. For an update $\eta(Y, \mathcal{V}')$ over a pair of (γ, u), where $Y \subseteq X$ and $\mathcal{V}' \subseteq \mathcal{V}$, $(\gamma, u)[\eta(Y, \mathcal{V}')]$ denotes the clock valuation that maps all clocks in Y to zero and agrees with γ for all clocks in $X \setminus Y$, and the variable valuation that maps all integer variables in $\mathcal{V}'$ according to the update expression in η and agrees with u in $\mathcal{V} \setminus \mathcal{V}'$.

Definition 3 (Semantics of timed automata). *The semantics of a timed automaton $A = (L, X, \mathcal{V}, \mathcal{E}, Inv, l_0)$ is a timed transition system $\mathcal{T} = \langle \mathcal{S}, s_0, \to \rangle$ where $\mathcal{S} \subseteq L \times \mathbb{R}^+ \times \mathbb{Z}$ is the set of states, $s_0 = (l_0, \gamma_0, u_0)$ is the initial state and $\to$ is the transition relation such that*

- $(l, \gamma, u) \xrightarrow{\delta} (l, \gamma + \delta, u)$ *if* $\forall \delta' : 0 \leq \delta' \leq \delta \Rightarrow (\gamma + \delta', u) \models Inv(l)$ *where* $\delta \in \mathbb{R}^+$, *and*
- $(l, \gamma, u) \to (l', \gamma', u')$ *if there exists* $e = (l, g, \eta(Y, \mathcal{V}'), l') \in \mathcal{E}$ *such that* $(\gamma, u) \models g$, $(\gamma', u') = (\gamma, u)[\eta(Y, \mathcal{V}')]$, *and* $(\gamma', u') \models Inv(l')$.

The former is called delay transition and the latter is called discrete transition.

The trace of a timed automaton is a finite or infinite sequence $(l_0, \gamma_0, u_0) \rightarrow (l_1, \gamma_1, u_1) \rightarrow \ldots$, where $\rightarrow$ is either a delay transition or a discrete transition. For an NTA, the discrete transitions are executed interleavingly.

Priced timed automata (PTA) [4] is an extension of TA to allow the accumulation of costs during behaviour. The extension from timed automata is $A_c = (L, X, \mathcal{V}, \mathcal{E}, Inv, l_0, \mathcal{P})$, where $\mathcal{P} : L \cup \mathcal{E} \rightarrow \mathbb{N}$ assigns cost rates and costs to locations and edges, resp. The semantics of priced timed automata is similar to the version without price, except that the cost in a delay transition is in direct proportion to the time elapsed, and the cost in a discrete transition is the cost of the edge. For a network of PTAs, which is defined similarly to an NTA, we use vectors of locations and the cost rate of a vector of locations is the sum of cost rates in the locations of the vector. For a finite trace of a PTA, the cost is the sum of the costs for all discrete and delay transitions.

5 A Timed Automata Semantics of System Models

The behavior of an SDFG consists of a sequence of *firings*. We use updates $\mathit{sFiring}(\alpha)$ and $\mathit{eFiring}(\alpha)$ to encode the start and the end of a firing of α, and use $\mathit{readyS}(\alpha)$ to describe the enabling condition of $\mathit{sFiring}(\alpha)$. Additionally, we introduce sets of variables $tn(E) = \{tn(e) | e \in E\}$ and $\mathit{numF}(V) = \{\mathit{numF}(v) | v \in V\}$, to record the current number of tokens on edges in E and the firing times of actors in V, respectively. Testing and updating the value of $\mathit{numF}(V)$ are not really a part of the behavior of SDFGs, which are used to facilitate the construction of an f-schedule.

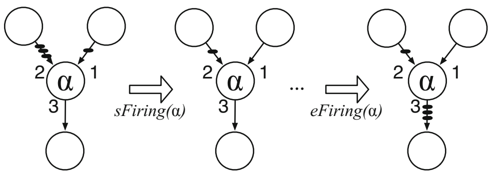


Fig. 2. The effect of *sFiring* and *eFiring*

Guard $\mathit{readyS}(\alpha)$ tests if there are sufficient tokens on the incoming edges of actor α to enable a firing. If the firing number of α reaches $f \cdot q(\alpha)$, no new firing of α is allowed, because α has finished its firings in f iterations.

$$\mathit{readyS}(\alpha) \equiv_{def} \forall e \in \mathit{InE}(\alpha) : tn(e) \geq \mathit{cns}(e) \wedge \mathit{numF}(\alpha) < f \cdot q(\alpha).$$

When a firing of α starts, it reduces the number of tokens of its incoming edges according to the consumption rates.

$$\mathit{sFiring}(\alpha) \equiv_{def} \forall e \in \mathit{InE}(\alpha) : tn'(e) = tn(e) - \mathit{cns}(e) \wedge \mathit{numF}'(\alpha) = \mathit{numF}(\alpha) + 1,$$

where x' refers to the value of x in the new state. For conciseness, we omit the elements of states if their values remain unchanged.

If a firing of α runs on processor p, it will finish after $t(\alpha, p)$ units of time. And update $eFiring(\alpha)$ increases tokens of α 's outgoing edges according to their production rates.

$$eFiring(\alpha) \equiv_{def} \quad \forall e \in OutE(\alpha) : tn'(e) = tn(e) + prd(e)$$

The effects of *sFiring* and *eFiring* are demonstrated in Fig. 2.

At a first glance, it seems natural to model each actor as a TA with status *idle* and *firing*, and each processors as a TA with status *idle* and *running* and then to model the allocation as synchronization between these TAs to form an NTA. Having a closer look, however, we observe that once an actor is firing, it must be running on some processor. Hence, we can represent the behavior of the system model only by the behavior of processors.

The behavior of actor α running on processor p can be modeled in a TA $ta_p(\alpha)$; and the behavior of p can be modeled by $ta_p(\alpha)$ with non-deterministically selecting actor α from V.

Definition 4 (TA of the behavior of processors). *A TA of the behavior of processor p is $ta_p = \exists \alpha \in V : ta_p(\alpha)$, and $ta_p(\alpha) = (L, X, \mathcal{V}, \mathcal{E}, Inv, l_0)$, where $L = \{idle, running\}$, $X = \{x\}$, $\mathcal{V} = tn(E) \cup numF(V)$, $l_0 = idle$, $Inv = \{running : x \leq t(a, p)\}$, and $\mathcal{E} = \{ir, ri\}$, where $ir = (idle, readyS, \{sFiring(\alpha), x := 0\}, running)$, and $ri = (running, x == t(\alpha, p), eFiring(\alpha), idle)$.*

The locations of ta_p indicate the status of processor p. That is, $ta_p.idle$ means p is idle and therefore is available for a firing of actors to run, and $ta_p.running$ means p is occupied by some firing. The graphical representation of ta_p is shown in Fig. 3. When the guard $readyS(\alpha)$ is satisfied, the transition from location *idle* to *running* is enabled. Once the transition is triggered, updates on clock $x := 0$ and other integer variables in $sFiring(\alpha)$ are executed. The invariant $x \leq t(\alpha, p)$ of location *running* restricts the allowed maximal delay.

Actors of SDFG G can fire in parallel only if they are ready and there are available processors. Subsequently, system model $\mathcal{M}$ can be modeled in an NTA $nta_{\mathcal{M}}$, which has $|P|$ concurrent processes and a global clock, where $|P|$ is the size of P. The global clock is used to measure the execution time of the system.

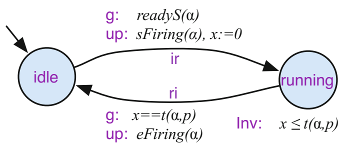


Fig. 3. The timed automaton ta_p.

Definition 5 (NTA of the behavior of system models). *The behavior of system model $\mathcal{M} = (G, P)$ is an NTA $nta_{\mathcal{M}} = ||_{p \in P} ta_p$ with a global clock glbClk.*

The above-mentioned semantics are the standard timed automata description, which can be translated into the input of UPPAAL straightforwardly. Quantification $\exists \alpha \in V$ can be implemented by the 'Selections' feature of UPPAAL.

The above defined ta_p and $nta_{\mathcal{M}}$ implicatively include f as a parameter. We omit it in the notations for conciseness. The semantics we present is much more concise than those in related works. For example, [7] transforms a system model to an NTA with more than $|V| + 3|P|$ TAs, and [18] more than $|V| \cdot |P| + |E|$ TAs, while our methods use $|P|$ TAs. This provides our methods the flexibility to deal with various constraints as shown in Section 7.

Algorithm 1. Sch($\mathcal{M}$,σ)

Input: A trace σ of $nta_{\mathcal{M}}$
Output: An f-schedule of $\mathcal{M}$, S

```
1. for all $e \in \mathcal{E}_\sigma$ do
2.     if $\exists \alpha \in V : e == p.sf(\alpha)$ then
3.         $S(\alpha, s_{p,\alpha}.numF(\alpha)).st = s_{p,\alpha}.glbClk$
4.         $S(\alpha, s_{p,\alpha}.numF(\alpha)).pa = p$
5.     end if
6. end for
7. return $S$
```

6 Static Optimal Scheduling and Mapping

6.1 Traces and Schedules

An f-schedule of $\mathcal{M}$ can be constructed from a trace of $nta_{\mathcal{M}}$ as follows.

Let $p.sf(\alpha)$ and $p.ef(\alpha)$ be discrete transitions, representing the transition caused by update $sFiring(\alpha)$ of edge ir of ta_p and the transition caused by $eFiring(\alpha)$ of edge ri. The use of $numF(\alpha) < f \cdot q(\alpha)$ as a guard in $readyS(\alpha)$ will force $nta_{\mathcal{M}}$ to be deadlocked after the firings of f-iterations of G are finished. Therefore a trace of $nta_{\mathcal{M}}$ includes finitely many discrete transitions.

Hence we consider only the finite part of a trace that includes all finite discrete transitions. Denote the set of transitions of trace σ as $\mathcal{E}_\sigma$ and the state caused by $p.sf(\alpha)$ as $s_{p,\alpha}$.

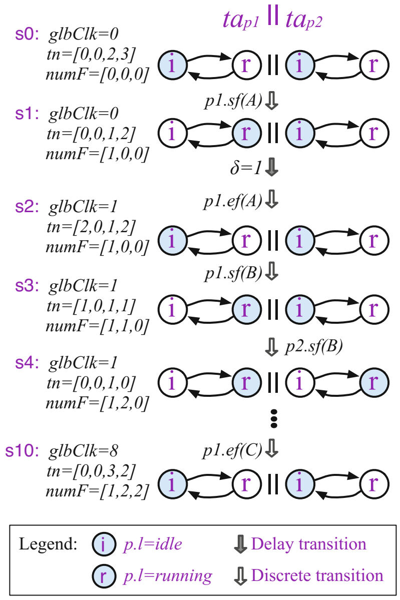


Fig. 4. A part of a trace of system model $\mathcal{M}_1$ shown in Fig. 1, where circles in blue show the current location.

Theorem 1. *In a trace σ of $nta_{\mathcal{M}}$, for each actor α:*

1. *$\nexists s_{p,\alpha}$ such that $s_{p,\alpha}.numF(\alpha) > f \cdot q(\alpha)$;*
2. *$\forall i \in [1, f \cdot q(\alpha)]$, there is a unique $s_{p,\alpha}$ such that $s_{p,\alpha}.numF(\alpha) = i$;*
3. *when $p.sf(\alpha)$ occurs, there are sufficient tokens on each $e \in InE(\alpha)$ for one firing of α and processor p is available.*

Proof. 1) is guaranteed by $readyS(\alpha)$; 2) is guaranteed by $sFiring(\alpha)$; according to the definition of ta_p, only when $ta_p.idle$ and $readyS(\alpha)$ are satisfied, p may select α to fire and therefore 3) is guaranteed.

Algorithm 1 presents the procedure of finding an f-schedule from a trace. Its correctness is ensured by Theorem 1. The schedule in Fig. 1(c), for example, is a 1-schedule of system model $\mathcal{M}_1$. It can be found in a trace of $nta_{\mathcal{M}_1}$, part of which is shown in Fig. 4.

6.2 Throughput-Optimal Solution

We denote the f-schedule derived by trace σ as S_σ. The cycle period of S_σ is the time when the last firing terminates, that is:

$$CP(S_\sigma) = \max\{s_{p,\alpha}.glbClk + t(\alpha, p) | s_{p,\alpha} \in \sigma\}.$$

Suppose the set of traces of $nta_{\mathcal{M}}$ is Σ, the optimal IP of f-schedules of $\mathcal{M}$ is

$$optIP(\mathcal{M}) = \min\left\{\frac{CP(S_\sigma)}{f}\middle|\sigma \in \Sigma\right\}$$

For given model $\mathcal{M}$ and unfolding factor f, $nta_{\mathcal{M}}$ will be deadlocked after the firings of f-iterations of G terminate. This property can be formalized by a CTL (Computation Tree Logic) formula **EF** *deadlock*. CTL formula $\mathbf{EF}\phi$ is true when ϕ is eventually true at some states of some traces of $nta_{\mathcal{M}}$, denoted by $nta_{\mathcal{M}} \models \mathbf{EF}\phi$.

Algorithm 2. optPSch($\mathcal{M}$)

```
Input: $\mathcal{M}$
Output: An $f$-schedule $S_{optP}$ of $\mathcal{M}$
 1. $S_{optIP} = Sch(\mathcal{M}, trace(nta_{\mathcal{M}}, \mathbf{EF}\ deadlock))$
 2. $ec = EC(S_{optIP})$
 3. $S_{optP} = S_{optIP}$
 4. repeat
 5.    $\phi = \mathbf{EF}\ deadlock \wedge con(ec - 1)$
 6.    $S_{IP} = Sch(\mathcal{M}, trace(nta_{\mathcal{M}'}, \phi))$
 7.    if $IP == optIP$ then
 8.       $ec = EC(S_{IP})$
 9.       $S_{optP} = S_{IP}$
10.    end if
11. until $IP > optIP$
12. return $S_{optP}$
```

A binary search can be used to find the minimal t that makes **EF** (*deadlock* $\wedge$ $glbClk \leq t$) true; then the minimal t is $f \cdot optIP$. By the returned trace, we find a throughput-optimal f-schedule. Even better, we can ask UPPAAL to check **EF** *deadlock* and to return a *fastest* trace, which is a trace with the shortest accumulated time delay. The latter way returns the same results as the binary search but only checks the property once. In the following discussion, we always apply UPPAAL to return a fastest trace, implemented by function $trace(nta_{\mathcal{M}}, \psi)$. From the trace returned by $trace(nta_{\mathcal{M}}, \mathbf{EF}\ deadlock)$, we obtain a throughput-optimal f-schedule of $\mathcal{M}$, denoted by S_{optIP}, i.e.,

$$S_{optIP} = Sch(\mathcal{M}, trace(nta_{\mathcal{M}}, \mathbf{EF}\ deadlock)).$$

The energy consumption of the schedule, $EC(S_{optIP})$, can be computed according to Eqn. (2).

To find an f-schedule with *optIP* and a best energy consumption, we need to add a constraint on energy consumption. Therefore, we add an update $occT(p) = occT(p) + t(\alpha, p)$ to edge ri in ta_p, and the subsequent model is $nta_{\mathcal{M}'}$. When *deadlock* occurs, *glbClk* is the CP of the schedule. Then according to Eqn. (2), the property that the energy consumption at time *glbClk* is no more than a given ec is defined as

$$con(ec) \equiv_{def} glbClk \leq \frac{ec - \sum_{p \in P} occT(p) \cdot [uEC(p) - iEC(p)]}{\sum_{p \in P} iEC(p)}$$

With $con(ec)$ as the additional constraint, we decrease ec gradually to check whether we can reach a smaller energy consumption with *optIP*. The details on computing an f-schedule S_{optP} with *optIP* and a best energy consumption are explained in Algorithm 2.

6.3 Energy-Optimal Solution

Decreasing ec in Algorithm 2 until ϕ is not satisfied, we can obtain an f-schedule with an optimal energy consumption and a best throughput. We can answer our second problem formulated in Section 3 by this way. The experiments we performed reveal that this method is inefficient, however. A more efficient way is to integrate the use of PTA.

By adding cost $iEC(p)$ and $uEC(p)$ to locations *idle* and *running* of ta_p, respectively, we obtain a priced timed automaton pta_p for processor p. Consequently, we use $npta_{\mathcal{M}} = ||_{p \in P} pta_p$ with a global clock *glbClk* to describe system model $\mathcal{M}$. With this formalization, by applying UPPAAL CORA to check $npta_{\mathcal{M}} \models \mathbf{EF}$ *deadlock*, we obtain an energy consumption-optimal f-schedule of $\mathcal{M}$ with *optEC*, denoted by S_{optEC}. Taking $con(optEC)$ as the additional constraint, we can apply UPPAAL to check $nta_{\mathcal{M}} \models \mathbf{EF}\ (deadlock \wedge con(optEC))$, and obtain an f-schedule S_{optE} with an optimal energy consumption and a best throughput.

7 Dealing with More Constraints

In this section, we show how various kinds of constraints can be integrated into our methods. We first introduce the general framework of our methods, then discuss the details of the three kinds of constraints, auto-concurrency constraints, buffer size constraints and processor constraints.

The effects of constraints on the behavior of an SDFG are summarized in Table 1. The first column lists the corresponding names of *readyS*, *sFiring* and *eFiring* for constraint *con*. The second column includes guard and updates we defined before. The 3-5 columns give the extra guard and updates for different constraints, auto-concurrency (ac), buffer size (bs) and both of them, respectively. Combining any of them with the second column forms the corresponding $readyS_{con}$, $sFiring_{con}$ and $eFiring_{con}$. For example, the enable condition of starting firing for an auto-concurrency constraint is represented as:

$$readyS_{ac} \equiv_{def} readyS \wedge hasF.$$

Table 1. Constrained Behavior of actor α

Constrained Behavior of α	NO Con.	Constraints (con)		
		auto-conc. (ac)	buffer size (bs)	both
$readyS_{con}$	$readyS$	$hasF$	$sufB$	$hasF \wedge sufB$
$sFiring_{con}$	$sFiring$	$addF$	$claB$	$addF \wedge claB$
$eFiring_{con}$	$eFiring$	$delF$	$relB$	$delF \wedge relB$

Replacing $readyS$, $sFiring$ and $eFiring$ in ta_p and pta_p defined in Section 5 with $readyS_{con}$, $sFiring_{con}$ and $eFiring_{con}$, respectively, we get NTA and NPTA of a system model with constraint con. The ways to find f-schedules S_{optP} and S_{optE} are the same as the system without these constraints.

7.1 Auto-concurrency constraints

When there are no limitation on auto-concurrency, at the same time, there can be unlimited number of concurrent firings of the same actor. Suppose the number of auto-concurrent actors is limited to $conN$. At each moment, only $conN$ firings allowed for each actor. We use a set $conC(V)$ to control the number of concurrent firings of each actor $\alpha \in V$. The extra condition for $readyS$, updates for $sFiring$ and $eFiring$ are formulated as $hasF(\alpha)$, $addF(\alpha)$ and $delF(\alpha)$, respectively.

$$\begin{aligned} hasF(\alpha) &\equiv_{def} \quad conC(\alpha) \leq conN \\ addF(\alpha) &\equiv_{def} \quad conC'(\alpha) = conC(\alpha) + 1 \\ delF(\alpha) &\equiv_{def} \quad conC'(\alpha) = conC(\alpha) - 1 \end{aligned}$$

Non-auto-concurrency, which can be used to model stateful actor [18], is a special case, which can be specified by $conN = 1$. Our method can also be used in a generalized case in which there is a constraint for each actor. For the generalized case, a set $conN(V)$ is used and above $conN$ are replace by $conN(\alpha)$.

7.2 Buffer size constraints

In practice, the storage space of a system must be bounded. The storage used by edges may be shared or separate. Firstly, we consider a relatively conservative separate buffer storage abstraction. That is, when an actor starts firing, it claims the space of the tokens it will produce, and it releases the space of the tokens it consumes only when the firing ends. A set $tnb(E)$ is added to capture the buffer space used by each $e \in E$.

Suppose a schedule is constrained by a set $B(E)$, which limits the buffer usage of each edge, an enabled firing can not start when there is no sufficient space on its outgoing edges. The extra condition for $readyS$ is formulated as $sufB(\alpha)$. When an actor starts a firing, it claims the required space on its outgoing edges. The update is formulated as $claB(\alpha)$. Only when a firing ends, it releases the space of its incoming edges. The update is formulated as $relB(\alpha)$.

$$\begin{aligned} sufB(\alpha) &\equiv_{def} \quad \forall e \in OutE(\alpha) : prd(e) \le B(e) - tnb(e) \\ claB(\alpha) &\equiv_{def} \quad \forall e \in OutE(\alpha) : tnb'(e) = tnb(e) + prd(e) \\ relB(\alpha) &\equiv_{def} \quad \forall e \in InE(\alpha) : tnb'(e) = tnb(e) - cns(e) \end{aligned}$$

A separate storage with other abstraction is even easier to be integrated. For example, suppose an actor releases the space of its incoming edges when it starts a firing and claims and occupies the space of its outgoing edges only when it ends a firing, we do not need the extra set $tnb(E)$ and updates $claB$ and $relB$. In $sufB(\alpha)$, $tnb(e)$ is simply replaced by $tn(e)$.

A shared memory usage can be easily integrated in the framework by modifying $sufB(\alpha)$ as $\sum_{e \in OutE(\alpha)} prd(e) \le sM - \sum_{e \in E} tnb(e)$, where sM is the bound of the shared memory.

7.3 Constraints on processors

The situation that an actor is not allowed to be allocated on some processors can be modeled by adding extra condition $t(\alpha, p) \ge 0$ to the enable condition of starting firing. That is, $readyS(\alpha) \wedge t(\alpha, p) \ge 0$. The constraint that actor α is not allowed to run on processor p can be represented by $t(\alpha, p) = -1$.

The constraint that a processor has a higher priority than another can be modeled by the 'Priorities' feature of UPPAAL.

8 Case Studies

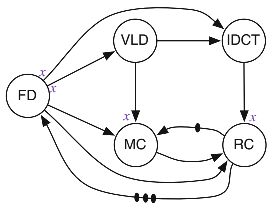
We have implemented the translation from system models with different constraints to input models of UPPAAL and UPPAAL CORA and the procedure to extract f-schedules from the returned traces. The approach has been applied to two practical applications with different parameters, running on a 2.90GHz CPU with 24M Cache and 384GB RAM. If not marked specially, the units of execution time and memory in performance evaluation are in second (s) and megabyte (MB), respectively.

The execution platforms for all SDFGs includes two types of processors, PT1 with $uEC = 90\text{W}$ and $iEC = 10\text{W}$ and PT2 with $uEC = 30\text{W}$ and $iEC = 20\text{W}$. PT1 is faster than PT2. We consider 2 processors, including one PT1 processor and one PT2 processor, and 4 processors, including two PT1 processors and two PT2 processors. We use the first buffer storage abstraction described in Section 7.2. The units of time and energy consumption used in system models are in picosecond and nanojoule, respectively.

8.1 MPEG-4 Decoder

The first case is an MPEG-4 decoder [23] with different parameters. The MPEG-4 decoder supports various kinds of frames. It is modeled as a Scenario-aware dataflow (SADF) model in [23]. Each scenario in an SADF model is actually an SDFG. We consider three scenarios, P30, P70 and P99. The system models of the MPEG-4 decoder are shown in Fig. 5. The parameterized SDFG is shown in Fig. 5 (a), the value of x corresponding to Px. The repetition vector and the sum of its elements (nQ) of each Px and the execution times of actors on different processors are shown in Fig. 5 (b). This case is used to evaluate our methods when different parameters are considered: the sum of the repetition vector, the unfolding factor, the number of processors, and the buffer size constraints. Auto-concurrency are not allowed in all models.

To evaluate the impact of the buffer size constraints, we consider two cases: a model with a low buffer size bound and a high bound. The low bound is computed according to the method described in [2] to guarantee deadlock-freeness of an SDFG. The high bound is a minimal buffer size requirement to guarantee throughput-optimal of an SDFG when it is scheduled in an infinite number of homogeneous processors [22]. The sum of buffer size bounds of all edges of Px are shown in the last two columns of Fig. 5 (b).



(a)

frame	x	Repetition Vector					nQ	Buffer Bound	
		FD	VLD	IDCT	MC	RC		Low	High
P30	30	1	30	30	1	1	63	128	149
P70	70	1	70	70	1	1	143	288	309
P99	99	1	99	99	1	1	201	404	425
The Execution Times Of Actors On Different Processors									
PT1	-	0	1	1	9	15	-	-	-
PT2	-	0	3	2	18	25	-	-	-

(b)

Fig. 5. System models of the MPEG-4 decoder. (a) Its SDFG; (b) the repetition vector of each Px, the sums of the vectors, the considered bound of buffer size, and the execution times of actors on different processors.

We show the experimental results for the MPEG-4 decoder in Table 2, in which the parameters are shown in the first two rows and the first two columns. The others are the results. The first column is the unfolding factor f. We consider 1-schedule and 2-schedule of models. The second column is the number of processors $\#P$. The other 6 columns are the results for SDFG Px under a low buffer size bound and a high buffer size bound. The results include three parts. The first part shows the optimal iteration period (optIP) and the best iteration energy consumption under optIP (bestIEC). The second part is the optimal iteration energy consumption (optIEC) and the best IP under optIEC (bestIP). The third part shows the execution times and memory consumptions of the procedure finding optIP.

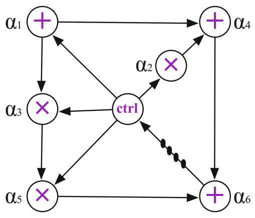
Table 2. Experimental results for MPEG-4 Decoder

info		Low Bound			High Bound		
		P30	P70	P99	P30	P70	P99
f	$\#P$	optIP/bestIEC					
1	2	83/9.2	163/18.0	221/24.3	82/7.4	162/13.8	220/18.4
	4	83/11.6	163/N	221/N	54/N	94/N	123/N
2	2	83/9.2	163/18.0	221/24.3	74/7.0	154/13.4	212/18.0
	4	83/N	163/N	221/N	48/N	88/N	117/N
		optIEC/bestIP					
1	2	7.4/131	15.0/251	20.5/338	6.6/102	13.0/182	17.6/240
	4	11.3/93	22.5/N	30.6/N	9.5/64	18.3/N	24.7/N
2	2	7.4/131	15.0/251	20.5/338	6.5/89.5	12.9/169.5	17.6/227.5
	4	11.3/N	22.5/N	30.6/N	8.6/N	17.4/N	23.8/N
		Execution Time (s)/Memory Consumed (MB) of optIP					
1	2	0.0/4.7	0.0/4.8	0.0/4.9	0.0/4.8	0.0/5.0	0.0/5.2
	4	0.1/5.6	0.1/6.8	0.2/7.7	0.2/7.1	0.5/10.6	0.6/13.7
2	2	0.0/4.9	0.0/5.2	0.1/5.5	0.1/5.6	0.1/6.3	0.2/7.0
	4	0.3/11.9	0.8/18.8	0.9/26.8	2.8/34.3	3.9/54.0	4.3/70.8

* N: not finished after 3 hours or running out of memory.

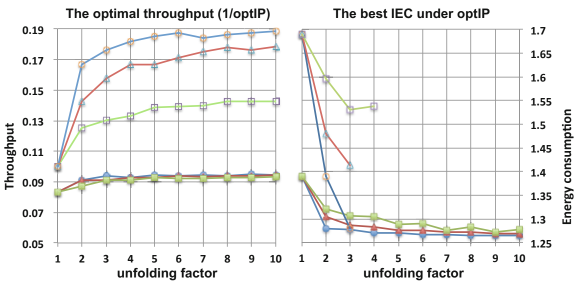
When a low buffer size bound is used, the increase of unfolding factor and number of processors have no improvement on the four values we have evaluated. Therefore, small unfolding factor and a few of processors are good enough for an optimal schedule of Px with a low buffer size constraint. A high bound provides more room for the improvement of iteration period and energy consumption at the cost of longer execution time and larger memory consumption.

When two processors are considered, our methods perform well on all cases. When four processors, 2-schedule and IEC are considered, state explosion occurs and hence our methods perform poorly. Another reason is that we only find 32bit version of UPPAAL CORA, which uses no more than 4GB memory. Besides the number of processors, the nQ seems affecting the performance of our methods mostly. Note that nQ is also the number of actors of the equivalent HSDFG of an SDFG, or the number of jobs in a task graph [1]. It is an important factor affecting the performance of almost all algorithms on SDFGs.

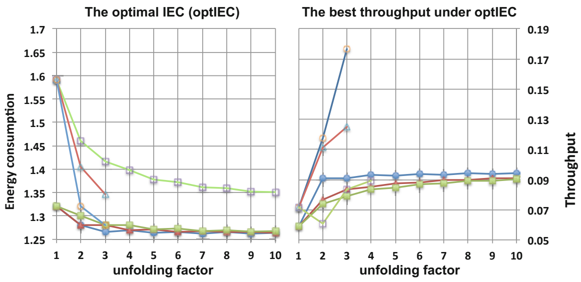


proType	Time (ps)		
	+	×	ctrl
PT1	2	3	0
PT2	5	7	0

(a)



(b)



(c)

Legend: ● No bound, 2 Processors; ○ No bound, 4 Processors; ▲ Non auto-con., 2 Processors; △ Non auto-con., 4 Processors; ■ Bound, 2 Processors; □ Bound, 4 Processors

Fig. 6. (a) System model of the computation example; (b) the optimal throughput and the best energy consumption under the optimal throughput; (c) the optimal energy consumption and the best throughput under the optimal energy consumption.

8.2 Computation Example

The second case study is mainly used to measure the impact of the unfolding factor. We consider a computation example, which is described in a task graph in [5]. Its system model is shown in Fig. 6(a). Actor *ctrl* connecting with original source and sink actors is added to limit the total latency. We have computed the results of unfolding factor from 1 to 10, and taken into account different combinations of values of three parameters: with and without a buffer bound, with and without auto-concurrency, 2 processors and 4 processors.

The experimental results are illustrated in Fig. 6 (b) and (c). The throughput and energy consumption of schedules are improved by increasing unfolding factor; the degree of improvement decreasing accordingly. The buffer size bound and auto-concurrency constraints have larger impact on the cases with 4 processors than that with 2 processors. Some lines stop at the point that unfolding factor reaches 4 or 5, because the corresponding procedures for larger unfolding factors run out of memory.

9 Conclusion

In this paper, we have presented exact methods for scheduling SDFGs on heterogenous multiprocessor platforms considering both throughput and energy consumption. Various parameters, including unfolding factors, constraints on auto-concurrency, buffer sizes and processors, can be integrated into the methods. Our experimental results show that our methods can deal with moderate scale models within reasonable execution time, and can find how different parameters impact on the results of different optimization goals.

We have used model checking as backend technique to solve the scheduling problems. While enjoying the benefits it provides, we encountered state explosion inevitably. As a future work, we will explore further the features of the considered models to reduce the state space. On the one hand, we will try to provide more domain insight when encoding the considered problems to model checking problems; on the other hand, we may tailor model checking techniques to deal with specialized tasks, instead of using a model checker directly. We have not considered the communications between processors based on the assumption that its cost is much smaller than the execution times of actors. In practical designs, the cost may be large in some situations. Then the communication needs to be taken into account. This can be integrated into our approach straightforwardly by modeling communications as actors that use special processors which model the connections between processors. But this method enlarges the scale of system models accordingly. A more efficient way to deal with communications is also an interesting topic for our further study.

References

1. Abdeddaïm, Y., Asarin, E., Maler, O., et al.: Scheduling with timed automata. Theor. Comput. Sci. 354(2), 272–300 (2006)
2. Adé, M., Lauwereins, R., Peperstraete, J.: Data memory minimisation for synchronous data flow graphs emulated on DSP-FPGA targets. In: Proc. of the 34th Ann. Design Automation Conf (DAC), pp. 64–69 (1997)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
4. Behrmann, G., Larsen, K.G., Rasmussen, J.I.: Priced timed automata: Algorithms and applications. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2004. LNCS, vol. 3657, pp. 162–182. Springer, Heidelberg (2005)
5. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N.: Quantitative analysis of real-time systems using priced timed automata. Comm. of the ACM 54(9), 78–87 (2011)

6. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model checker. International Journal on Software Tools for Technology Transfer 2(4), 410–425 (2000)
7. Fakih, M., Grüttner, K., Fränzle, M., Rettberg, A.: Towards performance analysis of SDFGs mapped to shared-bus architectures using model-checking. In: Proc. of the Conference on Design, Automation and Test in Europe, pp. 1167–1172 (2013)
8. Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: Schedulability analysis of fixed-priority systems using timed automata. Theor. Comput. Sci. 354(2), 301 (2006)
9. Geilen, M., Basten, T., Stuijk, S.: Minimising buffer requirements of synchronous dataflow graphs with model checking. In: Proc. of the 42nd Annu. Design Automation Conf. (DAC) (2005)
10. Gu, Z., Yuan, M., Guan, N., Lv, M., He, X., Deng, Q., Yu, G.: Static scheduling and software synthesis for dataflow graphs with symbolic model-checking. In: Proc. of 28th International Real-Time Systems Symposium (RTSS), pp. 353–364 (2007)
11. Harbour, M.G., Klein, M.H., Lehoczky, J.P.: Timing analysis for fixed-priority scheduling of hard real-time systems. IEEE Trans. on Soft. Eng. 20(1), 13–28 (1994)
12. Hartel, P.H., Ruys, T.C., Geilen, M.C.: Scheduling optimisations for SPIN to minimise buffer requirements in synchronous data flow. In: Proc of the International Conference on Formal Methods in Computer-Aided Design, p. 21 (2008)
13. Hirzel, M., Soulé, R., Schneider, S., Gedik, B., Grimm, R.: A catalog of stream processing optimizations. ACM Comput. Surv. 46(4), 46:1–46:34 (2014)
14. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
15. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer (STTT) 1(1), 134–152 (1997)
16. Lee, E., Messerschmitt, D.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. 36(1), 24–35 (1987)
17. Madsen, J., Hansen, M.R., Knudsen, K.S., Nielsen, J.E., Brekling, A.W.: System-level verification of multi-core embedded systems using timed-automata. In: Proc. of the 17th World Congress International Federation of Automatic Control, Seoul, Korea, pp. 9302–9307 (2008)
18. Malik, A., Gregg, D.: Orchestrating stream graphs using model checking. ACM Trans. Archit. Code Optim. 10(3), 19:1–19:25 (2013)
19. Parhi, K.K., Messerschmitt, D.G.: Static rate-optimal scheduling of iterative dataflow programs via optimum unfolding. IEEE Trans. Comput. 40(2), 178–195 (1991)
20. Singh, A.K., Shafique, M., Kumar, A., Henkel, J.: Mapping on multi/many-core systems: Survey of current and emerging trends. In: Proc. of the 50th Ann. Design Automation Conf. (DAC), p. 1 (2013)
21. Sriram, S., Bhattacharyya, S.S.: Embedded multiprocessors: scheduling and synchronization. CRC Press (2009)
22. Stuijk, S., Geilen, M., Basten, T.: Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. IEEE Trans. Comput. 57(10), 1331–1345 (2008)
23. Theelen, B., Katoen, J.P., Wu, H.: Model checking of scenario-aware dataflow with CADP. In: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 653–658 (2012)

24. Zhu, X.-Y., Geilen, M., Basten, T., Stuijk, S.: Static rate-optimal scheduling of multirate DSP algorithms via retiming and unfolding. In: Proc. of the 18th Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 109–118 (2012)
25. Zivojnovic, V., Ritz, S., Meyr, H.: Optimizing DSP programs using the multirate retiming transformation. Proc. EUSIPCO Signal Process. VII, Theories Applicat. (1994)