Pareto Optimal Scheduling for Synchronous Data Flow Graphs on Heterogeneous Multiprocessor

Yu-Lei Gu*†, Xue-Yang Zhu*, Guangquan Zhang†, Yifan He‡
*State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
†School of Computer Science and Technology, Soochow University, Suzhou, China
‡School of Computer and Information Engineering, Xiamen University of Technology, Xiamen, China
{guyl, zxy}@ios.ac.cn, gqzhang@suda.edu.cn and y.he@xmut.edu.cn

***Abstract*—Streaming applications usually run on heterogeneous multiprocessor platforms and are required to have a high throughput, which in turn may increase the energy consumption. A trade-off between these two criteria is important for a system. *Synchronous data flow graphs* (SDFGs) are widely used to model streaming applications. In this paper, we propose a paralleled Pareto optimal scheduling method (PPOS) for SDFGs on heterogeneous multiprocessors. It deals with both time arrangement and processor allocation of computations. PPOS is an exact method to chart the Pareto space of energy consumption and throughput, and to find all Pareto optimal schedules of a system model. An approximation technique is presented to further increase the scalability of our methods. Our experiments are carried out on a practical multimedia application with different configurations and hundreds of synthesis graphs. The results show that the proposed methods are capable of dealing with large-scale models.**

***Keywords*—real-time; energy consumption; throughput; parallelization**

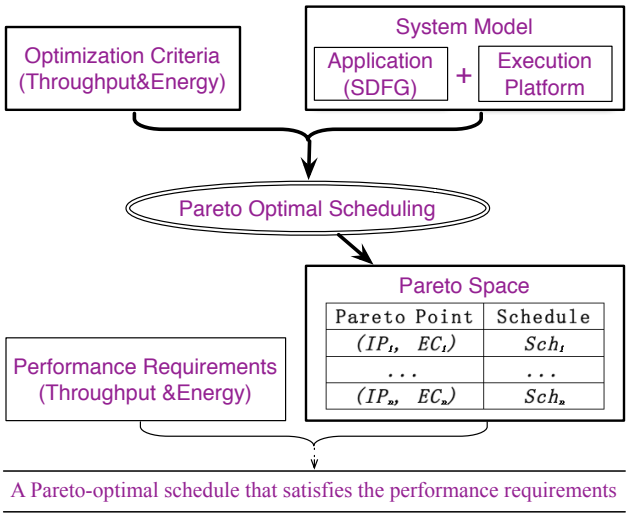


Fig. 1: Overview of the goal of this work.

I. Introduction

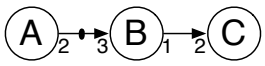
In embedded systems, energy efficiency is an essential issue. A high energy consumption leads to decreased mission duration and increased heat dissipation. An important class of applications in these systems are streaming applications, such as multimedia and digital signal processing applications. For these applications, high throughput is usually required to achieve smooth performance. Heterogeneous multiprocessor platforms are often used to improve the performance. However, a higher throughput is usually achieved at the cost of increased energy consumption. For a traditional development method, developers have to carefully tune the mapping of applications after they are implemented on the platforms to meet the performance and/or energy consumption requirements. When the failure to meet the requirement is due to the design of the application, the cost of fixing them would be high.

Synchronous data flow graphs (SDFGs) [1] are widely used to model streaming applications. They are further analyzed, optimized and scheduled according to different performance criteria [2], [3], [4]. An SDFG is a directed graph. Each node (also called actor) in an SDFG represents a computation or function 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. All sample rates of actors of an HSDFG are one. Execution of all the actors of an SDFG 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 of an SDFG may fire a different number of firings in an iteration, while each actor of an HSDFG fires once in an iteration. The number is decided by its repetition vector, which we will explain later. Practical streaming applications modeled with SDFGs include an MP3 playbacker [5], a satellite receiver [6], etc.

Streaming applications are usually nonterminating and repetitive. Static schedules are typically used to reduce run-time overhead because of real-time requirements and a strict resource budget. A *static schedule* arranges an iteration of the computations to be executed repeatedly in a fixed sequence.

In this paper, we focus on constructing throughput and energy efficient static schedules for SDFGs on heterogeneous multiprocessor platforms. An overview of the goal of this work is shown in Fig. 1. Applications are modeled in SDFGs as platform-independent models. We assume that those SDFGs are functionally correct. A system model is a platform-specific model, including an SDFG and a specific execution platform

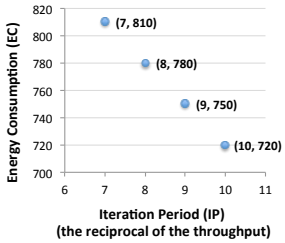
This work was supported in part by National Key Basic Research Program of China (No. 2014CB340701) and the National Natural Science Foundation of China (Nos. 61572478, 61472406 and 61472474).



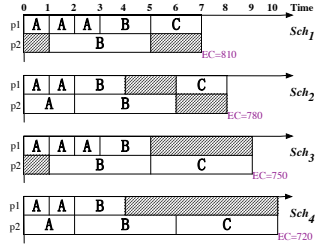
(a) The SDFG G_1

	Energy		Exec. Time		
	inuse	idle	A	B	C
p1	90	10	1	2	2
p2	30	20	2	4	4
Rep. Vector			3	2	1

(b) The execution platform P_1 and the execution time of actors in G_1 on different processors



(c) The Pareto space



(d) Pareto optimal schedules

Fig. 2: System model $\mathcal{M}_1$ and its Pareto space.

that the application uses. We analyze the system model to find schedules with their throughput and energy consumption being Pareto-optimal. We would like that an application executes with a higher throughput and less energy. These two goals, however, are usually in conflict with each other on heterogeneous multiprocessor platforms. Improving one may worsen the other. In a *Pareto point*, it is impossible to make one element better off without making the other worse off. These Pareto points form a Pareto space. Developers can choose a schedule from the Pareto space that satisfies the performance requirements to arrange the executions of their applications.

An example SDFG G_1 is shown in Fig. 2a. Actor A fires three times, while B fires twice and C once, in an iteration. The repetition vector of G_1 is shown in the last row of the table in Fig. 2b. A heterogeneous execution platform P_1 is shown in Fig. 2b. Different processors may have different unit energy consumption. For example, the unit energy consumption is 90 on $p1$ and 30 on $p2$ while they are in use. An actor may need different execution times on different processors. For example, Actor A of G_1 uses one unit of time on $p1$ and two units of time on $p2$. A system model consists of an SDFG and an execution platform. For example, G_1 and P_1 form system model $\mathcal{M}_1$. The computation time per iteration is called *iteration period* (IP), or *makespan*. The IP is the reciprocal of the *throughput*. We use IP and throughput alternatively in the remainder of the paper. The *energy consumption* (EC) is the total energy consumed during one iteration of execution. The Pareto space and corresponding schedules of $\mathcal{M}_1$, are shown in Fig. 2c and Fig. 2d, respectively.

In this paper, we chart Pareto space of throughput and energy consumption for a system model and find a schedule for each Pareto point. Our main contributions are as follows.

1) We define a procedure to construct a full space of schedules step by step.
2) We prune the space at each step and deliver a much smaller space of schedules, in which all the Pareto points are preserved. A Pareto optimal scheduling method (POS) is presented based on this.
3) We parallelize POS as PPOS to speed up. A fast approximation of PPOS, PPOS$_a$ is presented to improve the scalability further.
4) We implement PPOS and PPOS$_a$, and carry them out on a practical multimedia application with different configurations and a set of synthesis graphs to demonstrate their feasibility.

The remainder of this paper is organized as follows. The related work is discussed in the next section. The input models and the problems addressed are formulated in Section III. Our main contributions are illustrated in Section IV. Section V provides an experimental evaluation. Section VI concludes.

II. Related work

Most scheduling methods reported in the literature fall under design-time mapping [2]. The optimization goals of the mapping includes timing, energy consumption, etc. For a given platform, even only one optimization criterion is considered, e.g. throughput, the scheduling and mapping problem is already NP-complete. Researchers working on this problem usually target the solutions for particular application domain and specific optimization goals. And there are more heuristic methods than exact ones. We propose both exact methods and heuristic methods in this paper.

Scheduling SDFGs according to different optimization goals have been studied extensively [1], [7], [8], on both single processor platforms and mutilprocessor platforms. Many have been done on HSDFGs, while much less on the general SDFGs. Theoretically, it is always possible to convert an SDFG to its equivalent HSDFG [9] and then use the available methods for HSDFGs. However, converting an SDFG to an HSDFG is very time-consuming, especially when SDFGs scale up. The size of the HSDFG can be exponentially larger than the original SDFG in extreme cases [10]. Our methods in this paper work directly on SDFGs without converting them to any other kind of graphs.

Dynamic Frequency Voltage Scaling (DFVS) is a technique to reduce the power consumption of embedded systems. It dynamically scales the supply voltage and operational frequency of system components during run-time in accordance with the temporal performance requirement of the application. [11] proposes an energy efficient design exploration flow for streaming applications with guaranteed throughput. It slows down the frequency of the processor to reduce the energy consumption at the cost of the larger execution time of the application. D. Zhu et al. [12] focus on reducing the energy consumption based on the idea of slack sharing among processors. It reclaimed the time unused by a task to reduce the execution speed of future tasks. A Genetic Algorithm based approach is introduced in [13]. It uses DFVS to reduce the energy consumption on heterogeneous architectures to meet a deadline constraint. The DFVS technique is used when the mapping of tasks on each processor is given, while the mapping itself is what our methods try to find.

Authors of [14] present a heuristic method for HSDFGs on heterogeneous platforms to minimize the total cost while

the time constraint is satisfied. Energy consumption can be seen as a specific kind of cost. A mapping methodology is proposed in [15] to reduce power consumption by decreasing the energy consumption in communication while guaranteeing the required performance. A method to reduce the energy consumption under time constraint is presented in [16]. In the design-time scheduling phase, it uses a genetic algorithm to find an energy consumption and time Pareto-optimal set represented by a Pareto Curve.

There are exact methods that consider time and energy consumption constraints using model checking techniques, which explore state spaces of problems exhaustively. Authors of [17] use time automata as a common semantic model to represent embedded systems to guarantee the worst case response time of every actor. Authors of [18] use the concept of Voltage and Frequency Islands and encode the optimization problem as a query over priced timed automata to save the energy consumption of the system under time constraint. The methods in [19] find two Pareto optimal schedules using the same input models as our methods. None of them focus on exploring full Pareto space and find all Pareto optimal schedules considering both energy consumption and throughput, although it is possible to use those methods iteratively to find the Pareto space. We revise the method in [19] to produce Pareto space of system models and compare its results with our methods to evaluate the exactness of the proposed methods in Section V.

III. System Model and Problem Fomulation

An *execution platform* P is a set of heterogeneous processors. A computation may require different amounts of running time when it is executed on different processors. The energy consumption per unit time for each processor p is defined by $uEC(p)$ and $iEC(p)$. $uEC(p)$ indicates the energy consumption when p is used for some tasks, and $iEC(p)$ indicates the energy consumption when p is idle. The unit energy consumption of processor p_1 shown in Fig. 2b is 90 when it's in use and 10 when it's idle, for example.

An SDFG is a finite directed graph $G = (V, E)$. V is the set of actors, modeling the functional elements of the system; E is the set of directed edges, modeling interconnections between functional elements. Each edge e is weighted with three properties, $d(e)$, $prd(e)$ and $cns(e)$. Property $d(e)$ is the number of initial tokens on e, $prd(e)$ is the number of tokens produced onto e by each firing of the source actor 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 $OutE(\alpha)$. For a given execution platform P, each actor α is weighted with computation times $t(\alpha, p)$, for all $p \in P$. Normally, $t(\alpha, p)$ is a nonnegative integer. If $prd(e) = cns(e) = 1$ for each $e \in E$, G is an HSDFG.

A simple SDFG G_1 is depicted in Fig. 2a. Actor A needs 1 unit of time to finish on p_1 and 2 units of time on p_2, respectively. The production rate and consumption rate of edge $\langle A, B\rangle$ are 2 and 3, respectively, and there is one initial token on $\langle A, B\rangle$. A firing of actor A will produce 2 tokens on $\langle A, B\rangle$ and actor B can fire only when there are at least 3 tokens on $\langle A, B\rangle$. The actor without any incoming edge is free to fire at any time point, e.g. actor A.

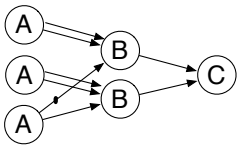


Fig. 3: The equivalent HSDFG of G_1.

SDFG $G = (V, E)$ is *sample rate consistent* [1] if and only if there exists a positive integer vector $q(V)$ satisfying *balance equation*s, $q(src(e)) \times prd(e) = q(snk(e)) \times cns(e)$ for all $e \in E$. The smallest such q is called the *repetition vector*. We use q to represent the repetition vector directly. The repetition vector of G_1 is $q = [3, 2, 1]$, for example. A sample rate consistent SDFG is *deadlock-free* if there is no zero-delay cycle in its equivalent HSDFG. Only sample rate consistent and deadlock-free SDFGs are meaningful in practice. Therefore we consider only such SDFGs.

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

A simple system model $\mathcal{M}_1 = (G_1, P_1)$ is shown in Fig. 2a and Fig. 2b.

An *iteration* is a firing sequence in which each actor α occurs exactly $q(\alpha)$ times. An iteration of G_1 (Fig. 2a), for example, includes $three$ firings of actor A, two firings of B and one of C. A sample rate consistent SDFG can always be converted to an equivalent HSDFG, which captures the data dependencies among firings of actors in the SDFG in an iteration [9]. Fig. 3 is the equivalent HSDFG of G_1, for example. After execution of an iteration, the number of tokens in the channels are equal to their initial state values.

Definition 2. A *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 an *iteration period* 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, and $S(\alpha, i).st \leq S(\alpha, i + 1).st$;
2) $S(\alpha, i).pa$ is the processor assigned to (α, i), which is available at the moment $S(\alpha, i).st$;
3) $S(\alpha, i + q(\alpha)).st = S(\alpha, i).st + T$;
4) $S(\alpha, i + q(\alpha)).pa = S(\alpha, i).pa$.

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

The *Iteration period* (IP) of schedule S is the computation time of an iteration. For example, the IPs of schedules Sch_1 and Sch_4 in Fig. 2d are 7 and 10, respectively.

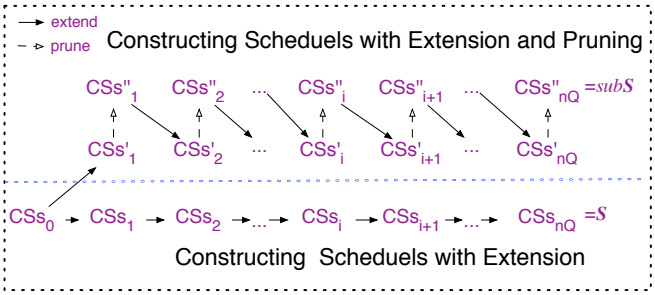


Fig. 4: The framework of POS

For conciseness, we omit parameter S when it is clear in context. Denote the set of all firings assigned on processor p by $AonP(p)$.

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

The total time p occupied in S is

$$ocT(p) = \sum_{(\alpha,i) \in AonP(p)} t((\alpha, i), p), \quad (1)$$

where $t((\alpha, i), p) = t(\alpha, p)$.

The *energy consumption* (EC) of schedule S is the sum of energy used by all processors in an iteration. For each processor, it includes the energy consumed both while it's idle and while it's in use.

$$EC = \sum_{p \in P} ocT(p) \cdot uEC(p) + [IP - ocT(p)] \cdot iEC(p). \quad (2)$$

For schedule Sch_2 in Fig. 2d, for example,

$$EC = (2 \times 10 + 6 \times 90) + (2 \times 20 + 6 \times 30) = 780.$$

A *Pareto point* is a tuple (EC, IP) of S. It is impossible to make one (IP or EC) better off without making the other worse off. Fig. 2c shows the Pareto space of the system model $\mathcal{M}_1$, including *four* Pareto points. Schedule S is a *Pareto optimal schedule* when its (EC, IP) is a Pareto point. There may be more than one Pareto optimal schedules corresponding to a Pareto point. A schedule of a Pareto point means anyone of them. Fig. 2d shows Pareto optimal schedules of corresponding Pareto points in Fig. 2c.

Given system model $\mathcal{M} = (G, P)$, suppose the set of all schedules of $\mathcal{M}$ is $\mathbf{S}$, the problem we address is to find schedules of all Pareto points in $\mathbf{S}$, denoted by $Pareto\mathbf{S}$. For each Pareto point, $Pareto\mathbf{S}$ includes one schedule of it. Set $\mathbf{S}$ does not include those schedules that for some moments all processors are idle. By removing all idle periods from such schedules, better schedules can always be found.

IV. Pareto Optimal Scheduling

A. Basic Ideas of Our Methods

It is straightforward that if we can find set $\mathbf{S}$ that includes all schedules of system model $\mathcal{M}$, all Pareto optimal schedules can be found by search in $\mathbf{S}$. In Section IV-C, we first illustrate how $\mathbf{S}$ is constructed step by step with *extension*. During the construction, the partial schedules are called *constructing schedules*, which are defined in Section IV-B. This construction procedure is very time and space consuming, however. The complexities are caused by the procedure of constructing $\mathbf{S}$ on one hand, and by the procedure of search $Pareto\mathbf{S}$ from $\mathbf{S}$ on the other hand. If we can find a subset of $\mathbf{S}$ with much smaller size and with $Pareto\mathbf{S}$ still included, we can find $Pareto\mathbf{S}$ much more efficiently.

We describe in Section IV-D how to prune the space during the construction procedure and therefore reduce the number of final schedules, collected in $sub\mathbf{S}$. We prove that $sub\mathbf{S} \subseteq \mathbf{S}$ and that $Pareto\mathbf{S} \subseteq sub\mathbf{S}$.

The framework of Pareto optimal scheduling method (POS) is shown in Fig. 4, in which CSs, CSs' and CSs'' are sets of constructing schedules.

POS is further parallelized in Section IV-E as PPOS to speed up. A fast approximation of PPOS, PPOS_a is presented in Section IV-F to improve the scalability further. The only difference between constructing schedules with firing mapping and actor mapping is in extension procedure. Section IV-C illustrates the extension for firing mapping. Actor mapping only limits the mapping of firings of an actor to a same processor. This constraint is introduced in Section IV-G.

B. Constructing Schedules

Definition 3. A *constructing schedule* (CS) of system model $\mathcal{M} = (G, P)$ is a function $cS : V \times \mathbb{N} \to \mathbb{N} \times P$ that satisfies conditions 1) and 2) of Definition 2 when S is replace with cS and $1 \leq i \leq k$ for all $\alpha \in V$ and $k \in [1, q(\alpha)]$.

For example, cs_1^1, cs_1^2 and cs_2^2 shown in Fig. 5 are constructing schedules of system model $\mathcal{M}_1$ (in Fig. 2). For technical reason, let $cS(\alpha, 0).st = -1$ for all $\alpha \in V$.

To develop our method, we first define some useful attributions of CS. A CS cs is labeled with vectors $sA(V)$, $tsE(E)$, $ocT(P)$, $tP(P)$ and $tA(V)$, a set eA, and integers SL and SE. All of them can be computed by the information of cs.

The number of already scheduled firings of actor α is recorded with a positive integer $sA(\alpha)$. For example, in Fig. 5, $cs_1^1.sA(A) = 1$ and $cs_1^2.sA(A) = 2$. The $sA(\alpha)^{th}$ firing of α is the last firing of α arranged by cs. For conciseness, when mention $cs(\alpha, sA(\alpha)).st$ and $cs(\alpha, sA(\alpha)).pa$, we use st and pa, respectively.

Each token tk is tagged with a *time stamp* to indicate the time when it is produced. Tokens on edge e is expressed by a queue of their time stamp, $tsE(e)$. All initial tokens have a time stamp 0. For example, in Fig. 5, cs_1^1 arranges A of G_1 executes once on $p1$ at time point 0, the two tokens it produces will have a time stamp 1, then we have $cs_1^1.tsE(\langle A, B\rangle) = \{0, 1, 1\}$, including a time stamp of the initial token on edge $\langle A, B\rangle$. Assume that there is always a dummy token with time stamp -1 on each edge. That is, $tsE(e)[0] = -1$ for all $e \in E$. Operations on $tsE(e)$ that affect its size (denoted by $|tsE(e)|$) do not count the dummy token.

When there are sufficient tokens on the incoming edges of actor α, it is enabled for a firing. If the firing number of α reaches $q(\alpha)$, no new firing of α is allowed, because α has finished its firings in one iteration. The enabling condition of α is denoted by $en(\alpha)$.

$$en(\alpha) \equiv_{def} \forall e \in InE(\alpha) : |tsE(e)| \geq cns(e) \wedge sA(\alpha) < q(\alpha).$$

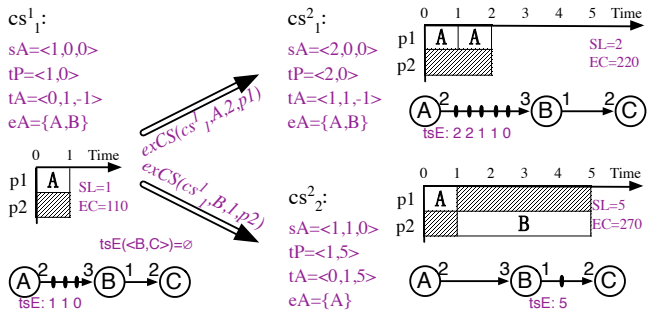


Fig. 5: Extension of a constructing schedule.

If α has no incoming edges, the first condition is always $true$ and therefore whether it is enabled is decided by $sA(\alpha)$. All enabled actors are collected in set eA.

$$eA = \{\alpha | \alpha \in V \wedge en(\alpha) = true\}. \tag{3}$$

The total time while p is occupied in cs is captured by $ocT(p)$, which is computed by Eqn. (1) when S is replaced with cS. The end time of the last scheduled firing on p is recorded with $tP(p)$. Next firing on p can not be arranged before $tP(p)$.

$$tP(p) = \max_{\alpha \in V \wedge pa = p} st + t(\alpha, p). \tag{4}$$

The time stamps of those tokens already on the incoming edges of actor α set a bound of start time of next firing of α. See Fig. 5 for example. In CS cs_2^2, there is one token with time stamp 5 on edge $\langle B, C\rangle$. Then next firing of actor C can not be scheduled before time point 5. $tA(\alpha)$ is used to express the possible start time of the $(sA(\alpha)+1)^{th}$ firing of actor α.

$$tA(\alpha) = \begin{cases} -1, & \text{if } sA(\alpha) = q(\alpha); \\ \max\{tsB(\alpha), st\}, & \text{otherwise,} \end{cases} \tag{5}$$

where

$$tsB(\alpha) = \begin{cases} 0, & \text{if } InE(\alpha) = \emptyset; \\ \max_{e \in InE(\alpha) \wedge 0 \le i \le nT} tsE(e)[i], & \text{otherwise,} \end{cases}$$

and $nT = \min(cns(e), |tsE(e)|)$ is the number of tokens that affects the next firing of $snk(e)$.

The schedule length SL is the time when all scheduled firings finished. That is,

$$SL = \max_{p \in P}(tP(p)). \tag{6}$$

The energy consumption SE can be computed by Eqn. (2), in which IP is replaced with SL.

When $sA(\alpha) = q(\alpha)$ for all $\alpha \in V$, cs is a schedule and SL and SE are its IP and EC, respectively. Above defined attributions of a constructing schedule are summarized in Table I.

C. Extension

We call a CS cs an *empty* CS if $cs.sA(\alpha) = 0$ and $cs(\alpha, 0) = -1$ for all $\alpha \in V$. Schedules of a system model are constructed step by step, beginning with a set including an empty CS. At each step, we get a set of extensions of current set of CSs.

TABLE I: Attributions of a constructing schedule

Name	Eqn.	Meaning
$sA(V)$	-	$sA(\alpha)$ is the number of already scheduled firings of α.
$tA(V)$	(5)	$tA(\alpha)$ is the possible start time of the $(sA(\alpha)+1)^{th}$ firing of α.
eA	(3)	The set of enabled actors.
$tsE(E)$	-	$tsE(e)$ is a queue of time stamps of edge e.
$ocT(P)$	(1)	$ocT(p)$ is the total time while processor p is occupied.
$tP(P)$	(4)	$tP(p)$ is the end time of the last scheduled firing on p.
SL	(6)	Current schedule length.
SE	(2)	Current energy consumption.

Definition 4. A CS cs' is an *extension* of CS cs on the i^{th} firing of actor α and processor p, denoted by $cs' = exCS(cs, \alpha, i, p)$, if Eqns. (7) to (12) are satisfied. For conciseness, we omit the elements of attributions if their values remain unchanged.

$$\alpha \in cs.eA \wedge cs.sA(\alpha) = i - 1 \tag{7}$$

$$cs'(\alpha, i).st = \max\{cs.tA(\alpha), cs.tP(p)\} \tag{8}$$

$$cs'(\alpha, i).pa = p \tag{9}$$

$$\forall e \in InE(\alpha) : |cs'.tsE(e)| = |cs.tsE(e)| - cns(e) \\ \wedge\, cs'.tsE(e) = cs.tsE(e) - \{\text{first } cns(e) \text{ elements}\} \tag{10}$$

$$\forall e \in OutE(\alpha), k \in [1, prd(e)] : \\ |cs'.tsE(e)| = |cs.tsE(e)| + prd(e) \\ \wedge cs'.tsE(e)[|cs.tsE(e)| + k] = cs'.tP(p) \tag{11}$$

$$cs'.sA(\alpha) = i \tag{12}$$

The i^{th} firing of actor α can only be scheduled after its previous firing has been scheduled and α is enabled. Eqn. (7) is the precondition to guarantee a valid extension.

If α is enabled, the i^{th} firing can neither start earlier than the time stamp of the latest tokens that make α enabled, nor can it start earlier than the start time of its previous firing, i.e.,$cs'(\alpha, i-1).st (= cs(\alpha, i-1).st)$. If the firing is arrange on processor p, its start time cannot also be earlier than the available time of p, $tP(p)$. Therefore, according to Def. 3 and Eqns. (5), (8) and (9), the extension cs' is also a CS.

Theorem 5. An extension of a CS is also a CS.

After a firing of an actor is scheduled, the tokens on its incoming edges and outgoing edges are changed. The firing consumes $cns(e)$ tokens from each incoming edge e and produces $prd(e)$ tokens on each outgoing edge e with their produce time as time stamps. Eqns. (10) and (11) formalize the changes. Also, the number of already scheduled firings of the actor is increased by one (Eqn. (12)). They are postconditions of a valid extension.

Other attributions of $exCS(cs, \alpha, i, p)$ can be obtained by the equations introduced in Section IV-B. Fig. 5 shows effects of an extension, for example. Lemma 6 can be derived from the definition straightforwardly.

Lemma 6. If $cs' = exCS(cs, \alpha, i, p)$, then $cs'.sA(\alpha) = cs.sA(\alpha) + 1$ and $cs'.ocT(p) = cs.ocT(p) + t(\alpha, p)$.

Let $L_{cs} = \sum_{\alpha \in V} sA(\alpha)$ denotes the number of firings already arranged. For cs', an extension of cs, $L_{cs'} = L_{cs} + 1$ always holds. Let $L_{CSs} = i$, if $\forall cs \in CSs : L_{cs} = i$. A CS cs has $|eA| \cdot |P|$ extensions, which include arrangements combining all enabled actors and all processors. Algorithm 1 shows how to extend a set of constructing schedules. It is obvious that $L_{CSs'} = L_{CSs} + 1$.

Algorithm 1 Extend(CSs)

Require: A set of constructing schedules CSs of system model $\mathcal{M} = (G, P)$
Ensure: A set of constructing schedules CSs', including extensions of all constructing schedules in CSs

```
1: CSs' = ∅
2: for all cs in CSs do
3:   for all α ∈ cs.eA do
4:     for all p ∈ P do
5:       CSs' ← exCS(cs, α, cs.sA(α) + 1, p)
6:     end for
7:   end for
8: end for
9: return CSs'
```

In a schedule of system model $\mathcal{M} = (G, P)$, nQ firings are arranged, where $nQ = \sum_{\alpha \in V} q(\alpha)$. The space of schedules of $\mathcal{M}$, $\mathbf{S}$, can be constructed as follows.

Initially, the set of constructing schedules includes an empty CS cs^0. That is, $CSs_0 = \{cs^0\}$ and $L_{CSs_0} = 0$. By computing $CSs_{i+1} = Extend(CSs_i)$ iteratively until CSs_{nQ} and $L_{CSs_{nQ}} = nQ$, we get a set that includes all schedules of the system model. The construction procedure includes nQ steps, each of which considers all possible extensions. Therefore, we have $CSs_{nQ} = \mathbf{S}$. The construction procedure is shown in the lower part of Fig. 4.

Pareto space $Pareto\mathbf{S}$ can always be found in $\mathbf{S}$. The size of $\mathbf{S}$ is usually large, however. In the worst case, an extension of a CS includes $(|V| \cdot |P|)$ CSs and $\mathbf{S}$ includes $(|V| \cdot |P|)^{nQ}$ schedules. In the following section, we illustrate how to prune the CS space at each step and therefore reduce the size of the final set of schedules, in which $Pareto\mathbf{S}$ is still included.

D. Pruning

The space at each step is pruned according to a domination relation as defined below.

Definition 7. Constructing schedules cs_1 **dominate** cs_2, denoted by $cs_1 \gg_d cs_2$, if Eqns. (13) to (16) are satisfied.

$$cs_1.sA = cs_2.sA \tag{13}$$
$$cs_1.ocT \preceq cs_2.ocT \tag{14}$$
$$cs_1.tP \preceq cs_2.tP \tag{15}$$
$$cs_1.tA \preceq cs_2.tA \tag{16}$$

where, $X \preceq Y$ means that $X[i] \leq Y[i]$ for each i.

Two CSs are compared only when they have arranged the same numbers of firings of all actors (Eqn. (13)). As above analysis, the length of a schedule is decided by its tP and tA and the energy consumption of a schedule is decided by its ocT and tP. According to Def. 7 and Eqns. (6) and (2), SL and SE of the dominated CS are never better than SL and SE of the dominator.

Theorem 8. If $cs_1 \gg_d cs_2$, then $cs_1.SL \leq cs_2.SL$ and $cs_1.SE \leq cs_2.SE$.

Below we prove that these better properties of the dominator are preserved by extension.

Theorem 9. If $cs_1 \gg_d cs_2$, then $\forall \alpha \in V, p \in P$: $exCS(cs_1, \alpha, i, p) \gg_d exCS(cs_2, \alpha, i, p)$.

Proof: Let $cs'_k = exCS(cs_k, \alpha, i, p), k \in \{1, 2\}$. We need to prove that cs'_1 and cs'_2 satisfy all four conditions in Def. 7. By Lemma 6, Eqns. (13) and (14) hold. Let $i = cs'_1.sA(\alpha) = cs'_2.sA(\alpha)$.

$$\begin{aligned} & cs_1 \gg_d cs_2 \\ & \Rightarrow cs_1.tP(p) \leq cs_2.tP(p) \wedge cs_1.tA(\alpha) \leq cs_2.tA(\alpha) \\ & \Rightarrow cs'_1(\alpha, i).st \leq cs'_2(\alpha, i).st \ \text{//by Eqn. (8)} \\ & \Rightarrow cs'_1.tP(p) \leq cs'_2.tP(p) \ \text{//by Eqn. (4)} \end{aligned} \tag{17}$$

Extension does not affect $tP(p')$ with $p' \neq p$. Hence, Eqn. (15) holds.

By Eqn. (17), we have $cs'_1.tA(\alpha) \leq cs'_2.tA(\alpha)$. For each $e \in OutE(\alpha)$, cs'_1 add $prd(e)$ tokens on e with time stamp $cs'_1.tP(p)$, hence the maximal time stamp of the first $cns(e)$ tokens of e may only be increased to $cs'_1.tP(p)$. It is the same case for cs'_2. As $cs_1.tA \preceq cs_2.tA$ and we have already proven that $cs'_1.tP \preceq cs'_2.tP$, we have $cs'_1.tA(v) \leq cs'_2.tA(v)$ for $v = snk(e)$. Extension does not change $tA(u)$ for other actor $u \in V$. Hence, Eqn. (16) holds. ■

A procedure that prunes a set of CSs with domination is shown in Algorithm 2. The domination relation is partially ordered. It is possible that two CSs do not dominate each other. Algorithm 2 returns CSs'' as the pruned set of CSs'. Line 1 initializes CSs'' to be an empty set. Lines 2-15 check each CS cs' in CSs' to decide whether it should be added into CSs''. If cs' is not dominated by any CS in CSs'', it's added to CSs'' (Lines 13). Those CSs in CSs'' dominated by cs are removed from CSs'' (Line 9).

Inserted pruning at each step, the procedure of construction of Pareto optimal schedules (POS) is illustrated in the upper part of Fig. 4 and sketched in Algorithm 3.

The initial set is the same as CSs_0, including empty constructing schedule cs_0 (Line 2). Each step includes extension and pruning (Lines 3-6). By Theorems 9 and 8, it holds that $CSs''_i \subseteq CSs'_i$ and all the potential Pareto points are preserved in CSs''_{nQ}. Therefore, we have $Pareto\mathbf{S} \subseteq CSs''_{nQ} \subseteq \mathbf{S}$.

The worst complexity of POS is the same as that of the procedure including only extension. However, in most cases it reduces the space dramatically and therefore finds $Pareto\mathbf{S}$ speedily. For example, the number of schedules of system model $\mathcal{M}_1$ in Fig. 2 would be 6^6 in the worst case. However, POS delivers only 13 schedules, in which all the 4 Pareto optimal schedules are included. The number of constructing schedules of $\mathcal{M}_1$ at each step is shown in Table II, in which $|CS_i|$ is the size when constructing schedules with extension only and $|CS''_i|$ is the size of CSs at each step of POS.

Below we further speed up POS by parallelizing it.

Algorithm 2 Prune(CSs')

Require: A set of constructing schedules CSs' of system model $\mathcal{M} = (G, P)$ with $L_{CSs'} = i$
Ensure: A set of constructing schedules $CSs'' \subseteq CSs'$ with $L_{CSs''} = i$

```
1: CSs'' = ∅
2: for all cs' ∈ CSs' do
3:    isDom=false
4:    for all cs'' ∈ CSs'' do
5:       if cs'' ≫_d cs' then
6:          isDom=true
7:          break
8:       else if cs' ≫_d cs'' then
9:          remove cs'' from CSs''
10:      end if
11:   end for
12:   if isDom=false then
13:      CSs'' ← cs'
14:   end if
15: end for
```

Algorithm 3 POS($\mathcal{M}$)

Require: System Model $\mathcal{M} = (G, P)$
Ensure: $Pareto\mathbf{S}$ of $\mathcal{M}$

```
1: cs_0=empty CS
2: CSs''_0 = {cs_0}
3: for i = 0; i < nQ; i++ do
4:    CSs'_{i+1} = Extend(CSs''_i)
5:    CSs''_{i+1} = Prune(CSs'_{i+1})
6: end for
7: ParetoS={Pareto optimal schedules in CSs''_{nQ}}
8: return ParetoS
```

TABLE II: The number of constructing schedules of $\mathcal{M}_1$ at each step

i	1	2	3	4	5	6
$\lvert CS_i \rvert$	2	8	24	48	96	192
$\lvert CS''_i \rvert$	2	7	13	14	9	13

E. Parallelization

Recall the definitions of extension and domination. At each step of POS (Algorithm 3), each extension does not affect each other, while pruning operations are possible only for those CSs with the same sA. This observation reveals the possibility that, if sets CSs'_i and CSs''_i are divided properly into some subsets according to sA, at each step of POS, extension and pruning may be conducted on those subsets in parallel. Below we illustrate the parallelized POS algorithm, denoted by PPOS.

Consider a set of constructing schedules, CSs. It is divided as subsets in terms of sA of its elements. All CSs in CSs with $sA = key$ is put into a subset, $CSs[key]$. That is,

$$CSs[key] = \{cs | cs \in CSs \wedge cs.sA = key\}.$$

Let $\mathbf{K}$ be the set of keys of CSs' at each step. As an intermediate variable, each subset of CSs', $CSs'[key], key \in \mathbf{K}$, is stored with a FIFO queue, denoted by $bQ[key]$.

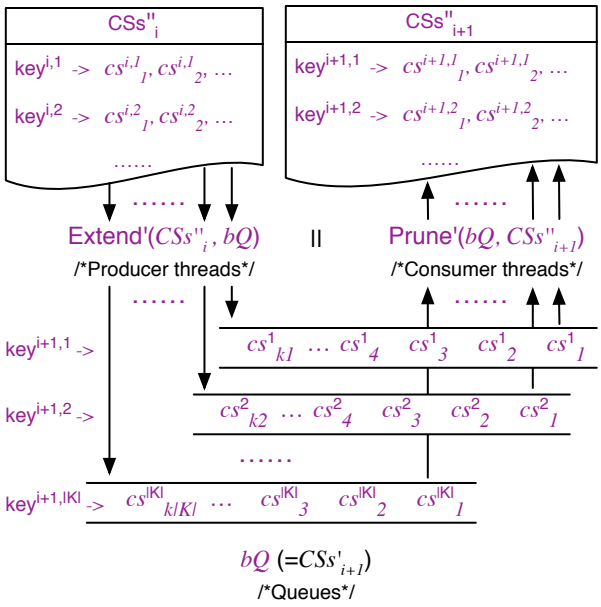


Fig. 6: Demostration of a step of PPOS

Only those extensions that produce constructing schedules with $sA = key$ may operate on queue $bQ[key]$. Let Extend$'(CSs, bQ[key])$ be a variant of Algorithm 1, in which CSs' is replaced with $bQ[key]$ and after Line 3, a guard to check whether $cs.sA(\alpha)+1 = key[\alpha]$ and $cs.sA(\beta) = key[\beta]$ for any other actor β is added. For all key in $\mathbf{K}$, the $|\mathbf{K}|$ threads of Extend$'(CSs, bQ[key])$ can work in parallel without data race in bQ.

The constructing schedules are pruned according to domination relation among them. Only CSs with the same sA may dominate each other. Let Prune$'(bQ[key], CSs''[key])$ be a variant of Algorithm 2, in which CSs' is replaced with $bQ[key]$ and CSs'' is replaced with $CSs''[key]$. At each step, the $|\mathbf{K}|$ threads of Prune$'(bQ[key], CSs''[key])$ for all key in $\mathbf{K}$ can work in parallel without data race in bQ and CSs''.

Therefore, PPOS method allows at most $|\mathbf{K}_{i+1}|$ producers for extension and $|\mathbf{K}_{i+1}|$ consumers for pruning to work in parallel at i^{th} step. It is outlined in Algorithm 4, where

$$\text{Extend}'(CSs''_i, bQ) = \|_{key \in \mathbf{K}_{i+1}} \text{Extend}'(CSs''_i, bQ[key])$$
$$\text{Prune}'(bQ, CSs''_{i+1}) = \|_{key \in \mathbf{K}_{i+1}} \text{Prune}'(bQ[key], CSs''_{i+1}[key])$$

Algorithm 4 PPOS($\mathcal{M}$)

```
1-3: The same as Lines 1-3 in Algorithm 3.
4: bQ = ∅
5: Extend'(CSs''_i, bQ) || Prune'(bQ, CSs''_{i+1})
6-8: The same as Lines 6-8 in Algorithm 3.
```

A step of PPOS is demonstrated in Fig. 6. Usually at each step, extension procedure is much faster than pruning procedure, we allocate more threads for consumers and less for producers in implementation.

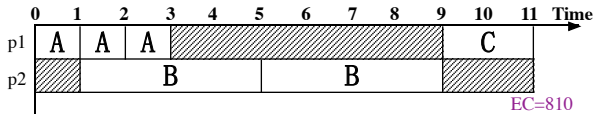


Fig. 7: A schedule of $\mathcal{M}_1$ with actor mapping

F. Approximation

A fast approximation of PPOS, PPOS$_a$, is obtained by pruning the space according to a quasi-domination relation, which compares only the schedule length SL and energy consumption SE of two constructing schedules.

Definition 10. Constructing schedules cs_1 **quasi-dominate** cs_2, if Eqns. (13), (18) and (19) are satisfied.

$$cs_1.SL \leq cs_2.SL \tag{18}$$
$$cs_1.SE \leq cs_2.SE \tag{19}$$

Pruning according to a quasi-domination relation may remove some potentially better schedules and therefore lead to a suboptimal set of schedules at the final step. However, our experimental results in Section V show that results returned by PPOS$_a$ are close to results returned by PPOS.

G. Actor mapping

For some applications, all executions of a computation are required to run on the same processor. In this case, a schedule with actor mapping is needed. Actor mapping maps all firings of an actor to a same processor. That is,

$$\forall \alpha \in V, s(\alpha,1).pa = s(\alpha,2).pa = ... = s(\alpha,i).pa = ...$$

A schedule of $\mathcal{M}_1$ with actor mapping is demonstrated in Fig. 7. At time point 3, the second firing of actor B is enabled and processor $p1$ is available. Since the first firing of B has been arranged on $p2$, which is being occupied, the second firing of B cannot start at this moment.

The only difference between constructing schedules with firing mapping and actor mapping is in the extension procedure, which is outlined in Algorithm 5. When the first firing of actor α is being scheduled (Line 4), it can be arranged on any available processor p with proper start time (Lines 5 to 7). Then $cs(\alpha,1).pa = p$. The later firings of α can only be extended on $cs(\alpha,1).pa$ (Line 9).

From now on, prefixes *fm* and *am* are used for PPOS with firing mapping and actor mapping, respectively. That is, fmPPOS and fmPPOS$_a$ return Pareto space of schedules with firing mapping; amPPOS and amPPOS$_a$ return that of schedules with actor mapping.

V. Experimental Evaluation

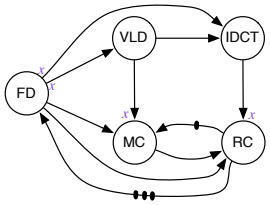
A. Experimental Setup

The proposed fmPPOS, fmPPOS$_a$, amPPOS and amPPOS$_a$ algorithms are implemented [20] and applied on two sets of system models. They run on a 2.4GHz server with 160 cores, 32MB Cache and 256GB RAM. If not marked specially, the units of execution time in performance evaluation are in second (s). The timeout limit is set to 30 minutes.

Algorithm 5 Extend$_{am}(CSs)$

Require: A set of constructing schedules CSs of system model $\mathcal{M} = (G, P)$ with $L_{CSs} = i$
Ensure: A set of constructing schedules CSs', including extensions of all CSs in CSs, with $L'_{CSs} = i + 1$

```
1:  CSs' = ∅
2:  for all cs in CSs do
3:    for all α ∈ cs.eA do
4:      if cs.sA(α) = 0 then
5:        for all p ∈ P do
6:          CSs' ← exCS(cs, α, 1, p)
7:        end for
8:      else
9:        CSs' ← exCS(cs, α, cs.sA(α) + 1, cs(α, 1).pa)
10:     end if
11:   end for
12: end for
13: return CSs'
```



(a) SDFG of the MP4 decoder

Sce.	x	Rep. Vector					nQ
		FD	VLD	IDCT	MC	RC	
P5	5	1	5	5	1	1	13
P10	10	1	10	10	1	1	23
P30	30	1	30	30	1	1	63

(b) The repetition vector of each Px and the sums of its elements

	Energy		Exec. Time				
	inuse	idle	FD	VLD	IDCT	MC	RC
p1	90	10	0	1	1	1	1
p2	30	20	0	2	2	2	2

(c) The execution platform P_1 and the execution time of actors on different processors

Fig. 8: System models of the MPEG-4 decoder

The first set is an MPEG-4 decoder with different parameters. The MPEG-4 decoder supports various kinds of frames. It is modeled as a scenario-aware dataflow (SADF) model in [21]. Each scenario in an SADF model is actually an SDFG. Three scenarios are considered, namely P5, P10, and P30. For each scenario Px, the sum of elements of its repetition vectors is $3 + 2x$. The problem scale grows when x goes larger. The SDFGs of the MPEG-4 decoder and their repetition vectors are shown in Fig. 8a and Fig. 8b, respectively. The execution platform P_1 and the execution time of actors on different processors are shown in Fig. 8c. The second set consists of 180 synthetic strongly connected SDFGs generated with SDF3 [22], mimicking real DSP applications. The number of actors in an SDFG (nA) and the sum of the elements in the repetition vector (nQ) have significant impact on the

performance of various methods. We use six groups with different values of nA and nQ. Each group includes 30 graphs with the same values of nA and nQ. For example, a group named $a5q20$ includes 30 SDFGs with $nA = 5$ and $nQ = 20$. They all use execution platform P_1 as shown in Fig. 8c. The execution time of their actors on processor $p1$ are assigned randomly and each execution time on $p2$ is double of that on $p1$. A tested system model consists of an above-mentioned SDFG and execution platform P_1, which may include two, four or eight processors with one, two or four processors per type, respectively.

The total number of threads is set to the maximum value allowed by the experimental platform, which is 160 in our experiments. Since extension is usually much faster than pruning at each step, we allocate 90 percent of cores to pruning threads (Consumers) and 10 percent to extension threads (Producers).

B. Experimental Results

We revise the method in [19] (denoted by MC) to produce Pareto space of system models and compare its results with that of fmPPOS and fmPPOS$_a$ to evaluate the accuracy of our methods. The results for the MPEG-4 decoder are shown in Table III, which include two parts. The first part shows results for firing mapping, including the Pareto spaces of different scenarios of the MPEG-4 decoder with different parameters and the execution time of MC, fmPPOS and fmPPOSa. Similarly, the second part shows results for actor mapping. Since MC cannot deal with actor mapping, its results are not shown. The first column, labeled with $\#Pro$, is the number of processors in the tested system models.

TABLE III: Experimental results for MPEG-4 decoder

Results for Firing Mapping			
Pareto Space (EC,IP)			
$\#Pro$	P5	P10	P30
2	(990,9)(960,10)	(1790,15)(1760,16)	(5020,42)(4990,43), (4960,44)
4	(1020,5)	(1820,9)	(5080,22)(5020,23)
Execution Time (s) MC [19]/fmPPOS/fmPPOS$_a$			
2	0.1/0.1/0.1	0.2/0.2/0.2	6.4/1.2/1.0
4	13.5/0.3/0.1	19m/11.4/0.2	N/N/0.7
Results for Actor Mapping			
Pareto Space (EC,IP)			
2	(1230,11)(1020,12) (990,13)(960,14)	(2330,21)1820,22) ((1790,23)(1760,24)	(6730,61)(5020,62) (4990,63),(4960,64)
4	(1380,7)(1320,12) (1080,13)	(2480,12)(2420,22) (1880,23)	(6880,32)(6820,62) (5080,63)
Execution Time (s) amPPOS/amPPOS$_a$			
2	0.1/0.1	0.1/0.6	1.4/0.4
4	0.1/0.1	0.2/0.4	0.4/0.4

N - Timeout.

The model checking method in [19] can deal with various constraints such as buffer size, which may limit the space that need to be explored. Our experiments here use no constraints. For all scenarios and parameters of MPEG-4 decoder that [19] can finish, fmPPOS and fmPPOS$_a$ return exact Pareto spaces as [19] does. The Pareto Points of scenario P30 and four processors are returned by fmPPOS$_a$, because neither MC nor fmPPOS finishes in 30 minutes. Our methods outperform MC method more when the models grow larger. For example, for the system model with P10 and four processors, MC takes 19 minutes, while fmPPOS takes 11.4 seconds and fmPPOS$_a$ only 0.2 second. The three times are almost the same for the system model with P5 and two processors.

When all firings of an actor are limited on one processor, the EC and IP of a schedule are generally larger than that of a schedule with firing mapping. This is also demonstrated by the Pareto space returned by amPPOS in the second part of Table III. Algorithm amPPOS is much faster than fmPPOS because the space of each extension step is limited by actor mapping. The number of processors of system models seems to have important impacts on exact methods MC, fmPPOS and amPPOS, but not for the approximate methods. For the case when the exact methods fail to meet the timeout limit, the approximate methods do well, e.g. the case with P30 and four processors.

To further evaluate the proposed methods, we carry out them on the set of synthesis models. The results of relative small cases are shown in Table IV and the results of large cases are shown in Table V.

The experiments on the relative small cases are mainly used to evaluated the performance of exact methods fmPPOS and amPPOS and measure the accuracy of the approximate methods. The second, third, fifth and sixth columns of Table IV show the execution time of different methods. Each point includes the average, maximal and minimal values (AVG/MAX/MIN) of graphs in the same group. For those groups that PPOS or PPOS$_a$ can not finish all cases within timeout period, their values are that of those finished. For all the cases with 5 actors ($a5q20$ and $a5q50$), fmPPOS finishes in 11.3 seconds and amPPOS finishes in 1.4 seconds. For the larger cases ($a10q50$ and $a10q100$), fmPPOS fails to meet the timeout limit on 7 cases and amPPOS fails on 1 case. In most cases, the approximate methods finished in several seconds.

TABLE IV: Results for synthesis models with two processors

	fmPPOS	fmPPOS$_a$	NADRS	amPPOS	amPPOS$_a$
	Exe. Time (AVG/MAX/MIN)		of fmPPOSa	Exe. Time (AVG/MAX/MIN)	
$a5q20$	0.2/1.1/0.1	0.1/0.3/0.1	1.0%	0.1/0.4/0.1	0.1/0.4/0.0
$a5q50$	1.9/11.3/0.1	0.3/0.7/0.1	0.6%	0.4/1.4/0.1	0.5/1.7/0.1
$a10q50$	37.2/8.2m/0.3[b]	0.7/7.3/0.1	0.8%	2.6/30.0/0.1[a]	0.7/2.5/0.1
$a10q100$	4.6m/23.7m/1.1[b]	7.7/2.6m/0.2	0.5%	12.7/3.2m/0.3[a]	1.7/16.9/0.2

[a] Timeout on 1 case.
[b] Timeout on 7 cases.

Similar to [23], we use the Average Distance to Reference Set (ADRS) to measure the accuracy of our approximation methods. The Pareto optimal sets returned by proposed exact methods are used to be reference sets. We calculate Euclidean distance for each point in the approximation set to the closest Pareto-optimal point and take average over all of these distances as ADRS [24]. The ADRS is normalized by the

ratio of ADRS to the average distance of reference set for each system model to a zero point. The normalized ADRS is called NADRS. NADRS of fmPPOS$_a$ is calculated for each SDFG in group $a5q20$, $a5q50$, $a10q50$ and $a10q100$, with two processors. The NADRS of each group of models is shown in the fourth column of Table IV. The experimental results show that the results obtained by our approximation methods are very close to the exact Pareto optimal sets.

The experiments on the large cases are mainly used to evaluate the performance of proposed approximate methods. Table V shows that except for the extreme large cases ($a10q1000$), fmPPOSa and amPPOSa are efficient on most cases. For example, having a close look at the execution time of amPPOS$_a$ on group $a20q100$ with $\#Pro = 4$, we find there is one case taking 29.2 minutes, two cases taking more than 1 minute, one case 14 seconds, and 8 cases less than 1 second. The results also show that, dislike the exact methods, the number of processors of a system model has not important impact on the performance of the approximate methods.

TABLE V: Execution time (s) for large scale system models

	$\#Pro = 4$		$\#Pro = 8$	
	fmPPOS$_a$	amPPOS$_a$	fmPPOS$_a$	amPPOS$_a$
	AVG/MAX/MIN	AVG/MAX/MIN	AVG/MAX/MIN	AVG/MAX/MIN
$a10q50$	0.7/5.5/0.1	0.8/3.4/0.1	0.7/4.3/0.1	0.7/2.3/0.1
$a10q100$	5.6/1.8m/0.2	1.7/11.5/0.4	5.3/1.4m/0.1	1.5/12.5/0.4
$a20q100$	44.2/14.6m/0.5[a]	1.1m/29.2m/0.4	1.5m/29.4m/0.5	12.2/2.2m/0.4[a]
$a10q1000$	4.2m/21.4m/19.9[b]	40.6/4.6m/0.9[a]	3.1m/15m/4.2[b]	70.8/12.2m/1.5[a]

[a] Timeout on 1 to 2 cases.
[b] Timeout on 9 to 13 cases.

VI. Conclusions

In this paper, we have presented a parallelized Pareto optimal scheduling method (PPOS) for scheduling SDFGs on heterogeneous multiprocessor platforms. The optimization criteria are throughput and energy consumption. PPOS is an exact method that can find all exact Pareto optimal schedules, with firing mapping or with actor mapping. An approximation variant of PPOS, PPOS$_a$, has also been presented to deal with larger system models. The exactness and efficiency of the exact methods are further confirmed by the experimental results. The approximate methods return results close to the exact ones.

The design of complex embedded systems usually needs to take into account various resource constraints. Besides processors and energy, we will extend our methods to deal with more resource constraints in the future work.

References

[1] E. Lee and D. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput*, vol. 36, no. 1, pp. 24–35, 1987.

[2] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: Survey of current and emerging trends," in *Proc. of the 50th Ann. Design Automation Conf. (DAC)*, 2013, pp. 1–10.

[3] S. Stuijk, M. Geilen, and T. Basten, "Throughput -buffering trade-off exploration for cyclo-static and synchronous dataflow graphs," *IEEE Trans. Comput*, vol. 57, no. 10, pp. 1331–1345, 2008.

[4] X. Y. Zhu, M. Geilen, T. Basten, and S. Stuijk, "Multiconstraint static scheduling of synchronous dataflow graphs via retiming and unfolding," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 6, pp. 905–918, 2016.

[5] M. H. Wiggers, M. J. Bekooij, and G. J. Smit, "Efficient computation of buffer capacities for cyclo-static dataflow graphs," in *Proc. of the 44th Design Automation Conference (DAC)*. IEEE, 2007, pp. 658–663.

[6] S. Ritz, M. Willems, and H. Meyr, "Scheduling for optimum data memory compaction in block diagram oriented software synthesis," in *Proc. of the 1995 Acoustics, Speech, and Signal Processing Conf.* IEEE, 1995, pp. 2651–2654.

[7] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software synthesis from dataflow graphs*. Springer, 1996, vol. 360.

[8] R. Govindarajan, G. R. Gao, and P. Desai, "Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks," *The Journal of VLSI Signal Processing*, vol. 31, no. 3, pp. 207–229, 2002.

[9] S. Sriram and S. S. Bhattacharyya, *Embedded multiprocessors: scheduling and synchronization*. CRC Press, 2009.

[10] V. Zivojnovic and R. Schoenen, "On retiming of multirate DSP algorithms," in *Proc. of the Acoustics, Speech, and Signal Processing, 1996.* IEEE Computer Society, 1996, pp. 3310–3313.

[11] J. Zhu, I. Sander, and A. Jantsch, "Energy efficient streaming applications with guaranteed throughput on mpsocs," in *Proc. of the 8th ACM int. conf. on Embedded software*. ACM, 2008, pp. 119–128.

[12] D. Zhu, R. Melhem, and B. R. Childers, "Scheduling with dynamic voltage/speed adjustment using slack reclamation in multiprocessor real-time systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 14, no. 7, pp. 686–700, 2003.

[13] D. Wu, B. Al-Hashimi, and P. Else, "Scheduling and mapping of conditional task graph for the synthesis of low power embedded systems," *Computers and Digital Techniques*, vol. 150, no. 5, pp. 262–273, 2003.

[14] Z. Shao, Q. Zhuge, C. Xue, and E. H.-M. Sha, "Efficient assignment and scheduling for heterogeneous DSP systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 16, no. 6, pp. 516 – 525, 2005.

[15] J. Hu and R. Marculescu, "Energy- and performance-aware mapping for regular noc architectures," *Computer-Aided Design of Integrated Circuits and Systems*, vol. 4, no. 24, pp. 551–562, 2005.

[16] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest, and R. Lauwereins, "Energy-aware runtime scheduling for embedded-multiprocessor socs," *IEEE Design and Test of Computers*, 2001.

[17] M. Fakih, K. Grüttner, M. Fränzle, and A. Rettberg, "State-based real-time analysis of SDF applications on MPSoCs with shared communication resources," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 486–509, 2015.

[18] W. Ahmad, P. K. Holzenspies, M. Stoelinga, and J. Van De Pol, "Green computing: power optimisation of VFI-based real-time multiprocessor dataflow applications," in *Proc. of Digital System Design (DSD)*. IEEE, 2015, pp. 271–275.

[19] X.-Y. Zhu, R. Yan, Y.-L. Gu, J. Zhang, W. Zhang, and G. Zhang, "Static optimal scheduling for synchronous data flow graphs with model checking," in *Proc. of FM 2015: Formal Methods*, ser. LNCS, vol. 9109. Springer, 2015, pp. 551–569.

[20] [Online]. Available: http://lcs.ios.ac.cn/∼zxy/tools/idfos.html

[21] B. Theelen, J.-P. Katoen, and H. Wu, "Model checking of scenario-aware dataflow with CADP," in *Proc. of Design, Automation and Test in Europe*, 2012, pp. 653–658.

[22] S. Stuijk, M. Geilen, and T. Basten, "Sdf3: Sdf for free," in *the 6th Int. Conf. on Application of Concurrency to System Design*. IEEE, 2006, pp. 276–278.

[23] Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal, "Iteration-based trade-off analysis of resource-aware sdf," in *2011 14th Euromicro Conference on*. Digital System Design (DSD), 2011, pp. 567–574.

[24] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. G. da Fonseca, "Performance assessment of multiobjective optimizers: an analysis and review," *IEEE Transactions on Evolutionary Computation*, vol. 7, no. 2, pp. 117–132, April 2003.