Journal of Interconnection Networks
© World Scientific Publishing Company

Energy-Aware Scheduling for Acyclic Synchronous Data Flows on Multiprocessors

DAWEI LI and JIE WU

Department of Computer and Information Sciences, Temple University
Philadelphia, PA, 19122, United States of America
{*dawei.li, jiewu*}*@temple.edu*

Received 12 November 2013
Revised 8 March 2014

Synchronous Data Flow (SDF) is a useful computational model in image processing, computer vision, and DSP. Previously, throughput and buffer requirement analyses have been studied for SDFs. In this paper, we address energy-aware scheduling for acyclic SDF-s on multiprocessors. The multiprocessor considered here has the capability of Dynamic Voltage and Frequency Scaling (DVFS), which allows processors to operate at different power/energy levels to reduce the energy consumption. An acyclic SDF graph can first be transformed to an equivalent homogeneous SDF graph and then to a Directed Acyclic Graph (DAG) for one iteration of it. We propose pipeline scheduling to address two closely related problems. The first problem is minimizing energy consumption per iteration of the acyclic SDF given a throughput constraint; the second problem is maximizing throughput given an energy consumption constraint per iteration of the acyclic SDF. Since the space of valid total orders of the transformed DAG is exponential, and finding the optimal order is no easy task, we first derive a valid total order, which can be achieved via various strategies, based on the DAG. Given the derived order, we design two dynamic programming algorithms, which produce optimal pipeline scheduling (including pipeline stage partitioning and the frequency setting for each stage), for the two problems, respectively. We also compare the performances of using various strategies to derive a valid total order. Analyses, experiments, and simulations demonstrate the strength of our proposed pipeline scheduling and dynamic programming algorithms.

Keywords: Dynamic voltage and frequency scaling (DVFS); synchronous data flow (SDF); directed acyclic graph (DAG); pipeline scheduling; dynamic programming.

1. Introduction

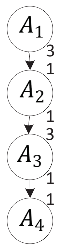
1.1. *Preliminaries on DVFS and SDF*

The main design goal of modern computational systems has been to improve computing capability. Recently, the high energy consumption in these systems has also become an important issue, because it not only results in high electricity bills, but it also increases the requirements for the cooling system and other system components. To facilitate energy-efficient design, the Dynamic Voltage and Frequency Scaling (DVFS) scheme is widely used [1] [2].

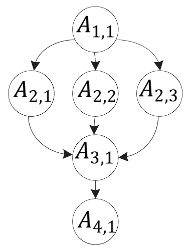
Over the past two decades, tremendous works have been done regarding energy-aware scheduling on DVFS-enabled platforms. Both circuit-level design and system scheduling have been studied in [3] and [4], respectively. It is impossible, and not necessary, to provide all of the existing research here; we refer the readers to comprehensive surveys in [5] and [6], where the task models that are covered are mainly traditional ones, namely, framed-based tasks, periodic tasks, sporadic tasks, and tasks with precedence constraints. The basic idea of the DVFS strategy is to reduce a processor's processing frequency, as long as tasks' predefined constraints are not violated. Since the power consumption of the processor is proportional to the cube of the processing frequency, while the overall execution time of a task is just inversely proportional to the processing frequency, DVFS provides the possibility of minimizing energy consumption, given a certain performance/timing requirement.

Synchronous Data Flow (SDF) [7] [8] is a useful computational model in image processing, computer vision and DSP, for example, signal transforming and MP3 audio coding and decoding, etc. An SDF can be represented by a directed graph, which is called an SDF graph, where nodes represent the SDF *actor*s, and *arc*s represent the dependence and data communications between the actors. From now on, we will use the application model and its corresponding graph interchangeably. For example, the SDF graph in Fig. 1(a) represents a simple SDF. A_1, A_2, A_3, and A_4 represent the SDF actors. The directed arcs represent the dependence and data communications between the actors. The numbers put at the beginning of an arc and at the end of an arc represent the number of data token(s) produced by the source of the arc, and the number of data consumed by the sink of the arc, respectively. For signal processing applications, we can consider that there are infinitely many raw signal data coming in at the actor A_1 and, after the processing by all of the actors, the results come out from the actor A_4.

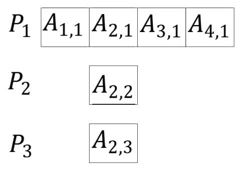
Most existing works on SDF focus on the throughput analysis on multiprocessors. Consider scheduling general SDFs on multiprocessor platforms; if the number of processors is unlimited, the formula of the optimal throughput of the SDF can be derived after transforming its SDF graph to its equivalent Homogeneous SDF (HSDF) graph (an HSDF is an SDF in which every actor consumes and produces only one data token from each of its inputs and outputs). To achieve this optimal throughput, it may be necessary to schedule several iterations of the transformed HSDF on the platform at one time. The number of iterations scheduled at one time is called the *unfolding factor*. The authors in [9] propose a technique for finding a minimal unfolding factor for multiprocessor implementation. If the number of processors is limited, the problem of finding the minimal optimal unfolding factor and deriving the corresponding schedule, which maximizes the throughput, is NP-hard [7]. The authors in [10] address allocating and scheduling an SDF on a multiprocessor platform, subject to a minimum throughput requirement. They develop a complete search algorithm, based on constraint-programming, without transforming the SDF into its equivalent HSDF. The same authors in [11] introduce acceleration techniques



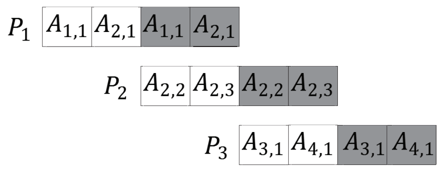
(a) An SDF.



(b) The transformed HSDF.



(c) List scheduling for one iteration.


P_1
$A_{1,1}$ $A_{2,1}$ $A_{1,1}$ $A_{2,1}$
P_2
$A_{2,2}$ $A_{2,3}$ $A_{2,2}$ $A_{2,3}$
P_3
$A_{3,1}$ $A_{4,1}$ $A_{3,1}$ $A_{4,1}$

(d) Pipeline scheme to schedule the graph, the grey task blocks represent the second iteration of the graph.

Fig. 1. A motivational example. P_i is used to denote processors; the same practice applies to other figures in this paper.

that significantly reduce the runtime of the constraint-programming by pruning the search space without compromising optimality.

Other works on SDFs consider buffer minimization under a throughput constraint [12]. The buffer comes from the fact that in an SDF, the number of data tokens produced by an actor may not be consumed immediately. A major concern in these works is to calculate the buffer size, formulate the optimization problem with a throughput constraint, and then solve it. Since our work in this paper has little to do with buffer requirements, we ignore the details here. In various works, the methods used to handle throughput are quite similar to traditional throughput analysis; however some of them present novel approaches to handle throughput, such as the pipeline scheduling in [13] and [14].

1.2. *Motivational Example*

In SDFs, an important metric is the throughput. A typical energy-related problem for SDFs lies in minimizing energy consumption per iteration of the SDF, given a throughput constraint. Here, we consider minimizing energy consumption per iteration, given a throughput constraint for an Acyclic SDF (ASDF, an SDF in whose corresponding graph there exists no loop) as shown in Fig. 1(a). An SDF graph can be transformed into its equivalent HSDF graph [15]. Fig. 1(b) shows the HSDF, where $A_{i,j}$ is the jth copy of the ith actor. In the HSDF, the number of token(s) produced and consumed on any arc is 1; thus, we omit the symbols on the arcs. Generally, an HSDF is further transformed into a DAG for the ease of scheduling. In this example, the transformed DAG is the same as the HSDF. Assume that the execution cycle(s) of any actor is C. The throughput constraint of the graph is set as F_c. The number of processors is 3.

To schedule a transformed DAG on a multiprocessor platform, maximizing the throughput is equivalent to minimizing the schedule length of one iteration of the DAG. To achieve a short schedule length, a widely used heuristic is a list scheduling scheme combined with the Largest Task First (LTF) strategy. It basically selects ready actors from the graph to be executed on free processors; after the completion of some task, the precedence relations should be updated. When there are more ready actors than free processors, it selects the largest actor(s) first. The scheduling derived by this scheme is shown in Fig. 1(c). To save the energy consumption, the most commonly used approach is to stretch the whole application, such that its throughput is exactly F_c [16]. Thus, the whole application finishes at time $1/F_c$; the execution frequency is $4C/(1/F_c) = 4CF_c$. Recall that the power consumption is the cube of execution frequency. The energy consumption of one iteration of the SDF is $E_1 = (4CF_c)^3(6C/(4CF_c)) = 96C^3F_c^2$.

There are several reasons for adopting a pipeline scheduling [13]. The first reason is the infinitely repeating nature of DSP applications. Instead of focusing on one iteration, we should consider the overall performance of a scheduling. The second reason is that there are no loops in an ASDF and its transformed HSDF. This fact makes a pipeline scheduling possible. The third reason is that, a pipeline scheduling, like a list scheduling, also does not need one processor to be able to execute all of the actors. The fourth reason is that, in a perfect pipeline scheduling, processors will not be idle even when precedence constraints exist. This fact is the main reason why a pipeline scheduling is efficient. Thus, a pipeline scheduling is friendly for practical design and implementation. Of course, pipeline scheduling has some disadvantages. For example, it requires the platform to keep several copies of the application simultaneously, which increases the buffer requirement. This aspect is out of the scope of our paper.

Pipeline stage partitioning can be conducted based on a valid total order of the actor copies in the DAG. A valid total order of the DAG is $A_{1,1} \rightarrow A_{2,1} \rightarrow A_{2,2} \rightarrow A_{2,3} \rightarrow A_{3,1} \rightarrow A_{4,1}$. Based on this, the optimal stage partitioning can be derived, which is shown in Fig. 1(d). To meet the throughput constraint, each stage must finish within time $1/F_c$; thus, the execution frequencies for all of the three stages are: $2C/(1/F_c) = 2CF_c$. The energy consumption for one iteration of the SDF is $E_2 = (2CF_c)^3(6C/(2CF_c)) = 24C^3F_c^2$.

It can easily be seen that the energy consumption of list scheduling (E_1) is four times that of pipeline scheduling (E_2), and thus, pipeline scheduling can save a significant amount of energy. Similarly, pipeline scheduling can also have a far better performance for the problem of maximizing throughput, given an energy consumption constraint per iteration of an ASDF.

1.3. *Our Work and Contributions*

In this paper, we address energy-aware scheduling of Acyclic SDFs (ASDFs) on DVFS-enabled multiprocessor platforms. We first transform the ASDF into its equivalent HSDF, and then to a DAG. We derive a valid total order, which can be achieved via various strategies, and then we adopt a pipeline scheduling based on this order. Our main contributions can be outlined as follows:

- We propose adopting pipeline scheduling for general ASDFs with the energy-aware consideration. More specifically, we consider two closely related problems, namely, minimizing energy consumption per iteration of the ASDF given a throughput constraint, and maximizing throughput given an energy consumption constraint per iteration.
- For a given total order derived from the transformed DAG, we design two dynamic programming algorithms, which produce optimal scheduling (including both pipeline stage partitioning and the frequency setting for each stage), for the two problems, respectively. To derive a valid and good total order, we also adopt the list scheduling scheme combined with the LTF strategy.
- We compare our overall scheduling with other pipeline scheduling schemes and a non-pipeline scheduling scheme. Our pipeline scheduling achieves a near-optimal solution, namely, within 2% greater than the ideal minimal energy consumption; while the energy consumption of none-pipeline based scheduling is several times that of the optimal solution. Besides, the dynamic programming-based pipeline partitioning method reduces the energy consumption by up to 2%, compared to a simple partitioning method.

1.4. *Paper Organization*

The rest of the paper is organized as follows. Section 2 presents the problem settings and definitions. We address the first problem extensively, which is minimizing energy consumption per iteration given a throughput constraint for ASDFs in Section 3. The second problem, maximizing throughput given an energy consumption constraint per iteration, is addressed in Section 4. Some analytical analyses of our algorithm are presented in Section 5. Experiments and Simulations are provided in Section 6. The related research on energy-aware scheduling for SDFs is provided in Section 7. A brief conclusion is made in Section 8.

2. Problem Settings and Definitions

In our paper, we address energy-aware scheduling for Acyclic SDFs (ASDFs). We will present some basic concepts of SDFs and ASDFs in subsection 2.1. To adopt pipeline scheduling, a procedure is needed to transform an ASDF into a DAG; this procedure is described in subsection 2.2. The platform model is described in subsection 2.3. Subsection 2.4 provides the definitions for the two problems that we consider in this paper.

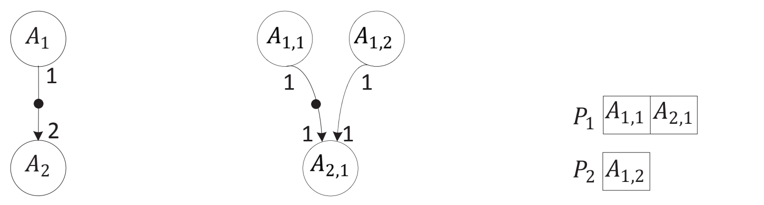
2.1. *Fundamentals of SDFs*

SDFs [7] [8] are widely used to model DSP and multimedia applications where data communications between functional actors are known *a priori*. An SDF can be represented by an SDF graph, $G_s = (V_s, E_s)$, where V_s represents the *actor* set of the SDF, and E_s represents the *arc* set of the SDF. $|V_s| = n$ is the total number of actors in the SDF. Vertex $v_i \in V_s$ represents SDF actor A_i, whose execution requirement is the number of execution cycles c_i. Each arc $e \in E_s$ can be described by a five-tuple $(src, snk, prd, cns, tok)$, where src and snk represent the source actor and sink actor of arc e, respectively. One unit of data is called a *data token*, or just *token* for short. prd and cns represent the number of data tokens produced to arc e by the source actor src and the number of tokens consumed by the sink actor snk, respectively. Initially, the arc e may have tok pre-filled tokens, called initial tokens. Though prd and cns values of each arc can be determined before execution, they can be different from each other. Thus, there might be data tokens that should be buffered on the system.

An SDF is called consistent if there exists a schedule such that no deadlocks will happen during the execution, and data buffers on each edge will not increase unboundedly. The first aspect, that no deadlocks will happen during execution, is affected by initial tokens on the arcs of the SDF. The second aspect, that data buffers on each edge will not increase unboundedly, is guaranteed if a non-null *repetition vector* can be calculated based on the graph G_s [7]. In this paper, we consider practical SDFs, so we assume all SDFs we are considering are consistent. A minimal integer repetition vector $q = [q_1, q_2, \cdots, q_n]$ can be calculated, where q_i represents the number of copies that actor A_i should execute in one iteration. The throughput of an SDF graph is the number of iterations that execute per unit time.

Also, a pseudo-polynomial time algorithm exists that transforms a consistent SDF, G_s into its equivalent HSDF, $G_h = (V_h, E_h)$, where an HSDF is an SDF in which every actor consumes and produces only one token from each of its inputs and outputs. V_h contains q_i copies of A_i, $\forall i = 1, 2, \cdots, n$. $|V_h| = N$ is the total number of actor copies in the transformed HSDF. The execution requirement of actor A_i in one iteration is $q_i c_i$. The total execution requirement of one iteration is $C_t = \sum_{i=1}^{n} q_i c_i$. The transformed HSDF can be further transformed into a DAG, for ease of scheduling.

Related concepts can be better described by a trivial example, which is shown in Fig. 2. Fig. 2(a) is a simple SDF. The top actor, A_1, is the starting actor. The small black circle on the arc represents the initial token of the arc; each actor's execution cycle is 1. The minimal repetition vector of the SDF is $q = [2, 1]$, which means, in one iteration of the SDF, actor A_1 should be executed two times, and actor A_2 should be executed once. Fig. 2(b) shows the equivalent HSDF of the SDF. $A_{i,j}$ represents the jth copy of the ith actor in the SDF. Assume that we have three processors to execute this SDF. By adopting a list scheduling combined with the largest task first strategy, we only need two processors, and the optimal throughput can be achieved



(a) SDF graph. (b) Transformed HSDF graph. (c) A list scheduling.

Fig. 2. A simple example to demonstrate some basic concepts.

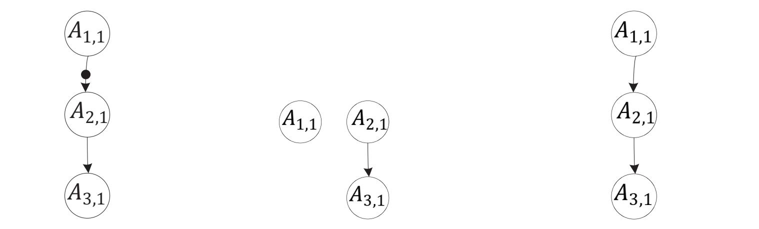
as 1/2 in the scheduling shown in Fig. 2(c). If we unfold the two iterations of the application, all of the three processors can be used, and a better throughput can be achieved as 2/3 in the scheduling shown in Fig. 2 (d). The number of iterations that are scheduled together is called the *unfolding factor*.

SDFs can be classified into two categories, namely, Cyclic SDFs (CSDFs) and Acyclic SDFs (ASDFs). If there exist loops in its corresponding graph, the SDF is a CSDF; otherwise, it is an ASDF. In this paper, we consider scheduling ASDFs.

2.2. *ASDF Preprocessing for Pipeline Scheduling*

Given an ASDF, $G_s = (V_s, E_s)$, it can be first transformed into its equivalent HSDF, $G_h = (V_h, E_h)$ [15]. Since no loops exist in a G_s, there will not be any loop in G_h. The transformed G_h can be further transformed into a DAG, G_d, simply by ignoring the initial tokens on all of the arcs. Actually, the transformation from an HSDF to a DAG can be conducted by removing arcs of the HSDF that contain initial token(s) [15]. The DAG achieved in this way is useful for multiprocessor scheduling that does not overlap multiple iterations of the SDF. However, pipeline scheduling may actually overlap multiple iterations, because a former stage of the next iteration may begin to execute before the execution of a latter stage of the current iteration. To implement pipeline scheduling, we adopt the transformation from an HSDF into a DAG as follows: first, eliminate initial tokens on all of the arcs in HSDF; then, transform the modified HSDF into DAG in the same way as in [15]. We denote the transformed DAG by $G_d = (V_d, E_d)$, where V_d contains q_i copies of actor A_i. $|V_d| = N$ is the total number of actor copies.

The following example shows why our transformation can enable pipeline scheduling, while the original transformation method in [15] cannot. Fig. 3(a) shows a transformed HSDF, whose arc from $A_{1,1}$ to $A_{2,1}$ has one initial token. All execution cycles of the actors are equal to each other. The transformation adopted in [15] will produce a DAG in Fig. 3(b); based on this DAG, a valid total order can be $A_{2,1} \rightarrow A_{1,1} \rightarrow A_{3,1}$. Since we have three processors, each actor is mapped to one stage according to this order, which means that $A_{2,1}$ is mapped to the first stage, $A_{1,1}$, the second stage, and $A_{3,1}$, the third stage. However, it can be noticed that $A_{2,1}$ of the second iteration cannot begin to execute when $A_{1,1}$ of the first iteration has not finished due to the lack of data token(s). Thus, the DAG derived from this transformation is not valid for pipeline scheduling. Our transformation method will



(a) HSDF graph (b) Transformation in [15] (c) Our transformation

Fig. 3. Transform an HSDF graph to a DAG.

produce the DAG in Fig. 3(c). It is easy to verify that all valid total orders derived in this way are suitable for pipeline scheduling. Whether or not an arc has initial token(s) in our transformation method, the precedence constraint from the source of the arc to the sink of the arc still exists, and any total order derived from our transformed DAG will not encounter the case of lacking data token(s). Thus, our transformation method will produce the DAG, whose any valid topological sorting is valid for pipeline scheduling.

2.3. *Platform Model*

We consider a multiprocessor platform with m DVFS-enabled processors. We assume ideal processors whose frequency ranges are continuous on $[0, +\infty)$. The power consumption model that we consider in this paper is widely adopted by existing works [16] [17]. We assume that all of the processors have the same power consumption properties. Processors can operate in two modes: one is *run* mode, where the power consumption only consists of dynamic power $p = f^3$; the other one is *idle* mode, where the processor consumes no power. Additionally, we assume that when a processor has no task to execute, it transitions into idle mode immediately, without any overhead. The time it takes to execute one copy of actor A_i (with required execution cycles c_i) can be calculated as c_i/f. Thus, the energy consumption to execute one copy of actor A_i is $e = (c_i/f)f^3 = c_i f^2$.

Although data communications exist in the execution of SDFs, we assume tightly compact platforms, where the communication time is negligible compared to actors' execution times; communication time is not explicitly considered in this paper. These ideal assumptions help us to find the optimal scheduling, as will be shown later; in practical systems, the optimal scheduling (under the ideal assumptions) can be modified for practical problems, and still have good performances. For example, when communication times are non-negligible, a communication time can be incorporated as part of the corresponding actor's execution time; thus, our proposed algorithm is still applicable, after some modifications.

2.4. *Problem Definitions*

Given an ASDF $G_s = (V_s, E_s)$, we first transform it into its HSDF and then into a DAG, $G_d = (V_d, E_d)$. $|V_d| = N$ is the total number of actor copies of one iteration of the ASDF. To schedule it on a multiprocessor platform with m ($m \leq N$) DVFS-enabled processors, our goal is to first derive a valid total order, and then conduct pipeline stage partitioning and determine the frequency setting for each stage to solve the following two problems: 1) minimizing energy consumption per iteration given a throughput constraint and 2) maximizing throughput given an energy consumption constraint per iteration. We define the two problems in detail.

Minimizing Energy Consumption per Iteration Given a Throughput Constraint: this problem first requires deriving a valid total order based on the transformed DAG. Assume that the ordered sequence $V = (v_1, v_2, \cdots, v_N)$ is a valid total order. Then, it is required that we partition this ordered sequence of actor copies to $\bar{m}$ ($\bar{m} \leq m$) stages and determine the frequency setting for each stage, such that the throughput of the DAG is greater than or equal to F_c and the energy consumption per iteration of the DAG is minimized.

Maximizing Throughput Given an Energy Consumption Constraint per Iteration: this problem first requires deriving a valid total order based on the transformed DAG. Assume that the ordered sequence $V = (v_1, v_2, \cdots, v_N)$ is a valid total order. Then, it is required that we partition this ordered sequence of actor copies to $\bar{m}$ ($\bar{m} \leq m$) stages and determine the frequency setting for each stage, such that the energy consumption per iteration of the DAG is less than or equal to E_c and the throughput of the DAG is maximized.

Importation notations that are consistently used in this paper are listed in Table 1. Some of the notations will be made clear later in this paper.

Table 1. Notations used in this paper

Notation	Description
G_s,G_h,G_d	the original ASDF, the transformed HSDF, and the transformed DAG, respectively.
A_i	the ith actor of the ASDF.
$A_{i,j}$	the jth copy of A_i in the HSDF and DAG.
c_i	the execution requirement of one copy of A_i.
q_i	the number of copies of A_i.
V=(v_1,$\cdots$,v_N)	the total order derived from the DAG.
$c(v_i)$	the execution cycles of actor copy v_i.
N	the total # of actor copies of one iteration of the ASDF.
m	the total number of processors.
F_c, E_c	the throughput constraint and energy constraint for the first problem and second problem, respectively.
C_i,E_i,f_i	the execution cycles, energy consumption, and execution frequency, respectively, for the ith stage.
C_t	the total execution requirement of the ASDF.
$e(i, j)$	the minimal energy consumption when partitioning the first i elements of V into j stages.

3. Minimizing Energy Consumption per Iteration Given a Throughput Constraint

In this section, we will address the problem of minimizing energy consumption per iteration of an ASDF given a throughput constraint in detail. We will first analyze the problem and provide several important facts about a pipeline partitioning in subsection 3.1. For a given total order derived from the transformed DAG, a dynamic programming algorithm is presented to produce an optimal stage partitioning and frequency setting for each stage in subsection 3.2. The analysis of this dynamic programming algorithm is provided in subsection 3.3. In subsection 3.4, we describe the method that is used to derive a valid total order from the transformed DAG. Subsection 3.5 provides an example that illustrates the whole process of how we address this problem.

3.1. *Problem Analysis*

As has been mentioned, a pipeline scheduling strategy requires that a valid total order be derived from the transformed DAG, $G_d = (V_d, E_d)$. However, the space of valid total orders of the DAG is exponential in general; it is NP-hard to derive an optimal total order. In the following, we will first address the pipeline stage partitioning and the frequency setting problem when a valid total order of the DAG has been given. Assume that the given total order is represented by an ordered sequence, $V = (v_1, v_2, \cdots, v_N)$. V consists of q_i copies of A_i, $\forall i = 1, 2, \cdots, n$; $N = \sum_{i=1}^{n} q_i$ is the total number of actor copies. The execution cycles of each actor can be determined by recalling how we conduct the transformations and how we derive the valid total order. Without loss of generality, denote the execution cycles of v_i by $c(v_i)$.

Before stage partitioning and determining the frequency setting, we observe several important characteristics that will guide our solution. The following lemma determines the optimal frequency setting for a given pipeline stage partitioning.

Lemma 3.1. *If the number of execution cycles in the ith stage is C_i, $i = 1, 2, \cdots, \bar{m}$, the optimal frequency setting for the ith stage is $f_i = C_i F_c$.*

Proof. The execution time of the ith stage is C_i/f_i if executing at frequency f_i; to satisfy the throughput constraint, $C_i/f_i \leq 1/F_c$, and thus $f_i \geq C_i F_c$. Energy consumption of the ith stage is then: $E_i = (C_i/f_i)f_i^3 = C_i f_i^2 \geq C_i(C_i F_c)^2 = C_i^3 F_c^2$. Obviously, the minimal overall energy consumption requires that each E_i is minimized, which means $E_i = C_i^3 F_c^2$. Consequently, $f_i = C_i F_c$; besides, each stage has the same execution time $T_c = 1/F_c$. □

Another problem exists when addressing how many processors, or equivalently, how many pipeline stages should be used in the optimal pipeline scheduling. The following lemma tells us that the optimal pipeline scheduling should always use all of the m processors.

Lemma 3.2. *The optimal pipeline scheduling that minimizes energy consumption per iteration given a throughput constraint must use all of the m processors, or in other words, it must partition all of the actor copies to $\bar{m} = m$ stages.*

Proof. This characteristic can be proven by contradiction. Assume that one optimal pipeline scheduling, S_{opt}, uses $(m-1)$ processors and achieves the minimal energy consumption. Since $N \geq m$, there must be one processor with more than one actor. Without loss of generality, assume that the last stage (the $(m-1)$th stage) with execution requirement C_{m-1} has more than one actor. The optimal energy consumption of stage $(m-1)$ in this scheduling is $E_{m-1} = C_{m-1}^3 F_c^2$. Since, practically, there are m processors, we can construct a new schedule, S_{new}, which further splits the last stage into two stages. Assume that the resulting execution requirements of the last two stages, the $(m-1)$th and the mth stages, are C'_{m-1} and C'_m, respectively. $C'_{m-1} + C'_m = C_{m-1}$, $(C'_{m-1}, C'_m > 0)$. The optimal energy consumption of the last two stages in the new scheduling is $E'_{m-1} + E'_m = {C'_{m-1}}^3 F_c^2 + {C'_m}^3 F_c^2$. Since $E_{m-1} – E'_{m-1} – E'_m = (C_{m-1}^3 - ({C'_{m-1}}^3 + {C'_m}^3))F_c^2 = ({C'_{m-1}}^2 C'_m + C'_{m-1}{C'_m}^2)F_c^2 > 0$, the new schedule, which uses all of the m processors, achieves less energy consumption. Thus, S_{opt} is actually not optimal in terms of minimizing energy consumption, so the optimal pipeline scheduling must not use only $(m-1)$ processors. Further, we can also show that an optimal pipeline scheduling must not use less than $(m-1)$ processors. This completes the proof. □

Next, we will consider how to derive the optimal stage partitioning given a total order. Denote the minimal energy consumption to partition the first i actor copies of V, namely, $(v_1, v_2, \cdots, v_i)$ to j $(j \leq m)$ stages by $e(i, j)$. For $j = 1$, there is only one way to partition the first i elements into one stage, so the minimal energy consumption to partition actor copies $v_1, v_2, \cdots, v_i$ to one stage is $e(i, 1) = (\sum_{k=1}^{i} c(v_k))^3 F_c^2$. $e(i, j)$ is undefined when $i < j$, since each stage must have at least one element. For $j = 2$, there are $(i-1)$ way(s) to partition the first i actor copies: the first stage consists of the first $(i-1)$ actor copies, and the second stage consists of the ith actor copy; the first stage consists of the first $(i-2)$ actor copies, and the second stage consists of the $(i-1)$th and the ith actor copies; $\cdots$; the first stage consists of the first actor copy, and the second stage consists of the remaining $(i-1)$ actor copies.

To partition the first i actor copies into j stages, we can classify all possible partitions into $(i-j+1)$ categories:

Category (1). Partition the first $(i-1)$ actor copies into $(j-1)$ stages; the ith actor copy is the only element in the jth stage.

Category (2). Partition the first $(i-2)$ actor copies into $(j-1)$ stages; the $(i-1)$th and the ith actor copies form the jth stage.

...

Category $(i-j+1)$. Partition the first $(j-1)$ actor copies into $(j-1)$ stages;

the jth, $(j+1)$th, $\cdots$, ith actor copies form the jth stage.

It is obvious that our goal is to find $e(N, m)$, where N is the total number of actor copies in V, and m is the number of processors, which is also the number of pipeline stages.

The following observation is the key foundation for our dynamic programming algorithm.

Lemma 3.3.

$$\begin{aligned} e(i,j) = \min\{ \quad & e(i-1, j-1) + (c(v_i))^3 F_c^2, \\ & e(i-2, j-1) + (c(v_{i-1}) + c(v_i))^3 F_c^2, \\ & \cdots, \\ & e(j-1, j-1) + (\textstyle\sum_{l=j}^{i} c(v_l))^3 F_c^2\}, \\ \forall i \geq j \geq 2, & \end{aligned} \tag{3.1}$$

or, in a more compact form:

$$e(i,j) = \min_{k=j-1,\cdots,i-1} \left\{ e(k, j-1) + (\sum_{l=k+1}^{i} c(v_l))^3 F_c^2 \right\}, \quad \forall i \geq j \geq 2. \tag{3.2}$$

Proof. Note that $\sum_{l=k+1}^{i} c(v_l))^3 F_c^2$ is the energy consumption of the last stage (the jth stage) which consists of actor copies v_{k+1}, v_{k+2}, $\cdots$, and v_i. $e(k, j-1)$ is the optimal energy consumption for partitioning the first k stages to $(j-1)$ stages. We can say that $e(i,j)$ is achieved by searching through all of the $(i-j+1)$ possible ways of partitioning the first i actor copies to j stages, and the minimal energy consumption among all of these possibilities is chosen as $e(i,j)$. Thus, $e(i,j)$ is the optimal energy consumption for partitioning the first i actor copies to j stages. This completes the proof. □

3.2. *Algorithm Description*

As has been stated, our goal is to find $e(N, m)$, which can be achieved according to Lemma 3.3. Another issue lies in retracing the partition that achieves the minimal energy consumption. To this end, we introduce a data structure to record the partition that achieves each $e(i,j)$. It is easy to notice that a partition is uniquely determined by the ending elements of all its stages, so we only need to record the j ending elements of the partition that achieves $e(i,j)$. We use the ordered set $r(i,j) = \{end_{i,1}, end_{i,2}, \cdots, end_{i,j}\}$ to record the partition that achieves the $e(i,j)$, where $end_{i,k}$, $(k = 1, 2, \cdots, j)$ is the index of the ending element $(v_{end_{i,k}})$ of the kth stage of a partitioning that achieves the minimal energy consumption $e(i,j)$. Thus, the stage partition that achieves $e(N, m)$ can be retraced by looking up $r(N, m)$.

Algorithm 1 Dynamic Programming to Minimize Energy Consumption per Iteration Given a Throughput Constraint

Input: A valid total order, $V = (v_1, v_2, \cdots, v_N)$, of the transformed DAG, the execution requirements of v_i being $c(v_i)$, $\forall i = 1, 2, \cdots, N$, the number of processors, m ($m \leq N$);

Output: m pipeline stages and the frequency setting f_j for each stage j, $\forall j = 1, 2, \cdots, m$;

```
1:  for $i := 1$ to $N$ do
2:      Initialize $e(i,1) = (\sum_{k=1}^{i} c(v_k))^3 F_c^2$;
3:      Initialize $r(i,1) = \{i\}$;
4:  end for
5:  for $j := 2$ to $m$ do
6:      for $i := j$ to $N$ do
7:          $e(i,j) = \min_{k=j-1,\cdots,i-1}\{e(k,j-1) + (\sum_{l=k+1}^{i} c(v_l))^3 F_c^2\}$;
8:          Record the $k^*$ that minimizes $e(i,j)$;
9:          $r(i,j) = r(k^*, j-1) \bigcup \{i\}$;
10:     end for
11: end for
12: $end_{N,0} = 0$;
13: for $j := 1$ to $m$ do
14:     $f_j = \left(\sum_{k=end_{N,j-1}+1}^{end_{N,j}} c(v_k)\right) F_c$;
15: end for
```

Based on the three lemmas above, a dynamic programming algorithm can be designed to achieve the minimal energy consumption per iteration of an ASDF for a given total order, and it is outlined in Algorithm 1.

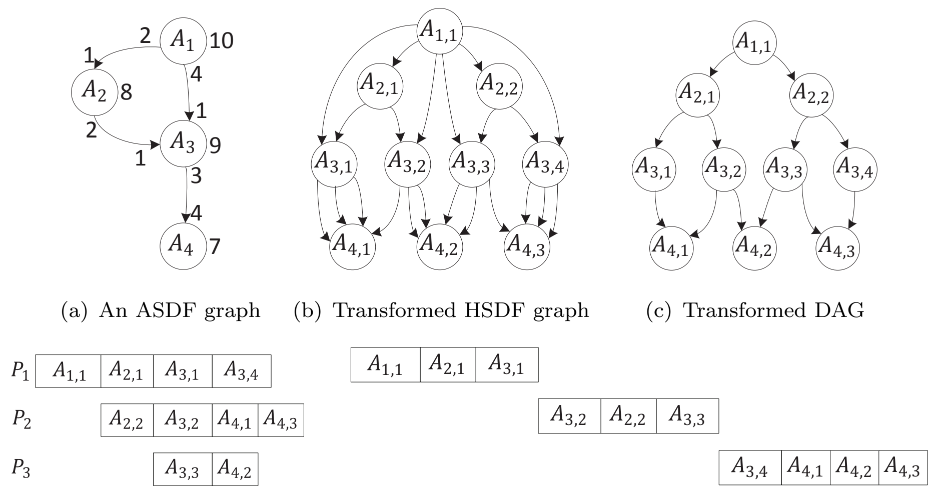
3.3. *Algorithm Analysis*

3.3.1. *Complexity of Algorithm 1*

The majority of the execution time of Algorithm 1 results from the overlapping loops from line 4 to line 8. The outer loop runs $O(m)$ times; each of the inner loops runs $O(N)$ times. The worst case time of calculating $e(i,j)$ is $O(N)$. The complexity of Algorithm 1 is $O(mN^2)$. Notice that our scheduling is an offline approach; besides, the SDF applications usually have a infinitely repeating characteristics, such as the video coding and decoding, and the image processing, etc. Thus, the long execution times of the applications will definitely compensate our algorithm's complexity.

3.3.2. *Optimality of Algorithm 1, Given a Valid Total Order*

Theorem 3.1. Algorithm 1 produces an optimal stage partitioning and optimal frequency setting for each stage that minimizes energy consumption while satisfying the throughput constraint, given a valid total order.



(a) An ASDF graph (b) Transformed HSDF graph (c) Transformed DAG

(d) A list scheduling (e) Our proposed pipeline scheduling

Fig. 4. An illustrative example.

Proof. This theorem is guaranteed by Lemma 3.1, Lemma 3.2, and Lemma 3.3 □

3.4. *Deriving a Valid Total Order From the Transformed DAG*

In the above analysis and solution, we have assumed that a valid total order is given. Here, we will discuss how to construct a valid total order. For a DAG, any list scheduling scheme can derive a valid total order. Since the space of all valid total orders may be exponential, finding the optimal total order is no easy task. We notice that Algorithm 1 attempts to arrive at a stage partitioning where the execution cycles of each stage are well "balanced." Intuitively, if, in a valid total order, large actors gather together and small actors gather together, it is hard for a stage partitioning to be well balanced.

In our overall scheme, we use a list scheduling combined with the Largest Task First (LTF) strategy to construct a valid total order. Applying the LTF strategy here can perturb the execution cycles of actor copies in the derived order, and thus, results in a more balanced stage partitioning. We will show via experiments that various other strategies, such as Smallest Task First (STF) and randomly picking strategy, can also arrive at balanced partitioning.

3.5. *An Illustrative Example*

We will provide an example to illustrate our overall solution in this subsection. Consider the ASDF graph shown in Fig. 4(a). Each arc's number of produced tokens and consumed tokens is put at the source port and the source port of the arc. Actors' execution cycles are put beside the actor node. Fig. 4(b) is the transformed HSDF. By removing redundant arcs and initial tokens (if there are any), the HSDF can be transformed into a DAG, which is shown in Fig. 4(c). According to our list scheduling

Table 2. $e_{i,j}$ values given by our dynamic programming algorithm.

$e_{i,j}$	$j = 1$	$j = 2$	$j = 3$
$i = 1$	1000	–	–
$i = 2$	5832	1512	–
$i = 3$	19683	5913	2241
$i = 4$	46656	11664	6642
$i = 5$	85184	23408	10826
$i = 6$	148877	37259	16577
$i = 7$	238328	62558	29240
$i = 8$	328509	82593	39033
$i = 9$	438976	110656	49426
$i = 10$	571787	144503	64259

combined with the LTF strategy, a valid total order is $A_{1,1} \rightarrow A_{2,1} \rightarrow A_{3,1} \rightarrow A_{3,2} \rightarrow A_{2,2} \rightarrow A_{3,3} \rightarrow A_{3,4} \rightarrow A_{4,1} \rightarrow A_{4,2} \rightarrow A_{4,3}$.

Assume that the throughput constraint is F_c. Algorithm 1 gives the ordered set $r_{10,3} = \{3, 6, 10\}$, which indicates that the first 3 actor copies, $A_{1,1}$, $A_{2,1}$, and $A_{3,1}$, are partitioned to the first stage; actor copies 4 to 6, $A_{3,1}$, $A_{2,2}$, and $A_{3,3}$, are partitioned to the second stage; actor copies 7 to 10, $A_{3,4}$, $A_{4,1}$, $A_{4,2}$, and $A_{4,3}$, are partitioned to the third stage. The pipeline stage partitioning is shown in Fig. 4(e). The frequencies for the three stages are set as $f_1 = 27F_c$, $f_2 = 26F_c$, and $f_3 = 30F_c$, respectively. Our dynamic programming algorithm can determine the minimal energy consumption as $e_{10,3} = 64259F_c^2$, as shown in Table 2.

For comparison, a list scheduling that schedules one iteration of the ASDF is shown in Fig. 4(d). After mapping the actors, the schedule length (in terms of the number of execution cycles) is 41. Thus, the execution frequency is set as $41F_c$. The energy consumption of this scheduling is $41^3 * (83/41)F_c^2 = 139523F_c^2$. Comparing this list scheduling with our pipeline scheduling, we can see that our approach saves a significant amount of energy.

4. Maximizing Throughput given an Energy Consumption Constraint per Iteration

In this section, we address the problem of maximizing throughput given an energy consumption constraint per iteration for an ASDF. The solution for this problem is quite similar to that of the first one. So, we will just briefly present the main steps and approaches.

4.1. *Problem Analysis*

We will provide some characteristics of the optimal stage partitioning and scheduling to achieve the maximal throughput in the following.

Lemma 4.1. *The execution times of all of the $\bar{m}$ stages of the optimal scheduling, which achieves the maximal throughput, must be equal to each other, i.e., $T_1 = T_2 = \cdots = T_{\bar{m}}$.*

Proof. Prove by contradiction. Assume that in the optimal scheduling, T_i's are not equal. Specifically, several stages have the largest execution time, and several stages have the second largest execution time. Without loss of generality, we assume that the last x stages have the largest execution time T_{max} and the next last y stages have the second largest execution time T_{next}. $T_1, T_2, \cdots, T_{\bar{m}-x-y} < T_{\bar{m}-x-y+1} = T_{\bar{m}-x-y+2} = \cdots = T_{\bar{m}-x} (= T_{next}) < T_{\bar{m}-x+1} = \cdots = T_{\bar{m}} (= T_{max})$, where $1 \leq x, y \leq \bar{m}, x + y \leq \bar{m}$. Thus, the maximal throughput achieved by this scheduling is $1/T_{max}$. The energy consumption of the last $(x + y)$ stages is:

$$\begin{aligned} & E_{\bar{m}-x-y+1} + E_{\bar{m}-x-y+2} + \cdots + E_{\bar{m}} = \\ & \frac{\sum_{i=\bar{m}-x-y+1}^{\bar{m}-x} C_i^3}{T_{next}^2} + \frac{\sum_{i=\bar{m}-x+1}^{\bar{m}} C_i^3}{T_{max}^2}. \end{aligned} \tag{4.1}$$

Let $h(t) =$

$$\frac{\sum_{i=\bar{m}-x-y+1}^{\bar{m}} C_i^3}{t^2} - \frac{\sum_{i=\bar{m}-x-y+1}^{\bar{m}-x} C_i^3}{T_{next}^2} - \frac{\sum_{i=\bar{m}-x+1}^{\bar{m}} C_i^3}{T_{max}^2}, \tag{4.2}$$

which is obviously a differentiable and monotonically decreasing function in $[T_{next}, T_{max}]$. $g(T_{next}) > 0$, $g(T_{max}) < 0$. According to the Mean Value Theorem [18], there must exist a number $T_e \in (T_{next}, T_{max})$ such that $h(T_e) = 0$; in other words:

$$\frac{\sum_{i=\bar{m}-x-y+1}^{\bar{m}} C_i^3}{T_e^2} = \frac{\sum_{i=\bar{m}-x-y+1}^{\bar{m}-x} C_i^3}{T_{next}^2} + \frac{\sum_{i=\bar{m}-x+1}^{\bar{m}} C_i^3}{T_{max}^2}, \tag{4.3}$$

which means we can set the frequency such that the energy consumption of the last $(x + y)$ stages remains unchanged, while they have the same execution time T_e, $T_{next} < T_e < T_{max}$. Since all stages except the last $(x + y)$ stages remain unchanged, we can achieve a better throughput $1/T_e > 1/T_{max}$, while the overall energy consumption remains unchanged. Thus, the original scheduling is not the optimal scheduling in terms of maximizing throughput. The proof is completed. □

The following lemma shows that an optimal pipeline scheduling must also use all of the m processors, or in other words, partitioning the actor copies to m stages.

Lemma 4.2. *The optimal scheduling that achieves the maximal throughput must use all of the m processors, i.e., it must partition all of the actor copies into $\bar{m} = m$ stages.*

Proof. This fact can also be proven by contradiction. Assume that one optimal scheduling has $(m - 1)$ stages and all of the stages have the same execution time, which is a fact that follows from Lemma 4.1. Since $N > m$, there must be a stage that consists of more than one actor. Without loss of generality, assume that the last stage, i.e., the $(m - 1)$th stage, has more than one actor. Its execution time is T_{m-1}. Since there is one more processor, we can further partition the execution time into two stages, the $(m - 1)$th and mth stages. The execution times of the last two stages are T'_{m-1} and T'_m, respectively. Since we only partition the last stage of the original scheduling, the throughput of the new scheduling will not increase. Besides, we can notice that the energy consumption is the same as the original scheduling. According to the proof of Lemma 4.1, the new scheduling cannot be an optimal scheduling, for the execution times of all of its stages are not equal. More specifically, the throughput of the new schedule can be further improved without increasing the overall energy consumption. Thus, the optimal scheduling cannot only have $(m-1)$ stages. It is also easy to show that an optimal scheduling cannot have less than $(m-1)$ stages. Conclusion: the optimal scheduling must have m stages.□

Next, we consider how to construct a stage partition given a valid total order of the transformed DAG. Assume that the energy consumption is upper bounded by E_c and that the number of execution cycles of each stage is C_i. Since execution times of all stages are equal in the optimal scheduling, denoted by T, the problem can be formulated as the following optimization problem:

$$\max \qquad 1/T \tag{4.4}$$

$$s.t. \sum_{i=1}^{m} C_i^3/T^2 \le E_c. \tag{4.5}$$

Obviously, in the optimal scheduling, $\sum C_i^3/T^2 = E_c$, which means $T = \sqrt{\sum_{i=1}^{m} C_i^3/E_c}$. Thus, maximizing throughput is equivalent to minimizing $\sum_{i=1}^{m} C_i^3$.

Thus, given a valid total order of all the actors and overall energy constraint E_c, to find a scheduling that achieves the maximal throughput is equivalent to first finding a partition that achieves the minimal $\sum_{i=1}^{m} C_i^3$, and then, stretching each stage such that they all have the same execution time and the overall energy consumption is exactly E_c. Denote the minimal $\sum_{l=1}^{j} C_l^3$ value to partition the first i actors to j stages as $cube(i,j)$. $cube(i,j)$ also has the characteristics that are similar to $e_{i,j}$:

Lemma 4.3.

$$\begin{aligned} cube(i,j) = \min\{ \quad & cube(i-1,j-1) + (c(v_i))^3, \\ & cube(i-2,j-1) + (c(v_{i-1}) + c(v_i))^3, \\ & \cdots, \\ & cube(j-1,j-1) + (\textstyle\sum_{k=j}^{i} c(v_j))^3\}, \\ \forall i \ge j \ge 2, & \end{aligned} \tag{4.6}$$

or, in a more compact form:

$$cube(i,j) = \min_{k=j-1,\cdots,i-1} \{cube(k,j-1) + (\sum_{l=k+1}^{i} c(v_l))^3\}, \quad \forall i \ge j \ge 2. \tag{4.7}$$

Proof. This fact is similar to Lemma 3.3, and can be proven in a way similar to that of Lemma 3.3. □

Algorithm 2 Dynamic Programming to Maximize Throughput Given an Energy Consumption Constraint per Iteration

```
Input: A valid total order, $V = (v_1, v_2, \cdots, v_N)$, of the DAG for the transformed
    HSDF graph, the execution requirements of $v_i$ being $c(v_i)$, $\forall i = 1, 2, \cdots, N$, the
    number of processors, $m$ ($m \leq N$);
Output: $m$ pipeline stages and the frequency setting $f_j$ for each stage $j$, $\forall j =
    1, 2, \cdots, m$;
for $i := 1$ to $N$ do
    Initialize $cube(i, 1) = (\sum_{k=1}^{i} c(v_k))^3$;
    Initialize $r(i, 1) = \{i\}$;
end for
for $j := 2$ to $m$ do
    for $i := j$ to $N$ do
        $cube(i, j) = \min_{k=j-1,\cdots,i-1}\{cube(k, j-1) + (\sum_{l=k+1}^{i} c(v_l))^3\}$;
        Record the $k^*$ that minimizes $cube(i, j)$;
        $r(i, j) = r(k^*, j-1) \bigcup \{i\}$;
    end for
end for
$end_{N,0} = 0$;
for $j := 1$ to $m$ do
    $C_j = \sum_{k=end_{N,j-1}+1}^{end_{N,j}} c(v_k)$;
end for
$T = \sqrt{\sum_{j=1}^{m} C_j^3 / E_c}$;
for $j := 1$ to $m$ do
    $f_j = C_j / T$;
end for
```

4.2. *Algorithm*

It can also be noticed that, given a valid total order, the optimal partition can be achieved similarly to Algorithm 1. The main modification is just to replace $e(i, j)$ with $cube(i, j)$. To avoid redundancy, we just give the algorithm to achieve the optimal scheduling for the second problem. We also use the data structure $r(i, j)$ for retracing the optimal stage partitioning that achieves the maximal throughput.

Theorem 4.1. Algorithm 2 produces the optimal scheduling to achieve the maximal throughput, given a valid total order.

Proof. It is guaranteed by Lemma 4.1, Lemma 4.2, and Lemma 4.3. □

4.3. *The Illustrative Example*

We also adopt a list scheduling combined with the LTF strategy to derive a valid total order. For the same example in Fig. 4(a), our solution to the second problem will derive a same stage partitioning as that of the first problem.

Given the energy consumption constraint per iteration, E_c, the frequency for each stage is set as $f_j = C_j/\sqrt{\sum_{j=1}^{m} C_j^3/E_c}$, and the maximal throughput is $\sqrt{\sum_{j=1}^{m} C_j^3/E_c}$.

5. Performance Analysis

We will present some analyses to verify the strength of our proposed energy-aware scheduling methods. For both of the two problems, we first analyze the ideal optimal solutions and then discuss two special cases where our method can achieve the practical optimal pipeline scheduling.

5.1. *The Ideal Optimal Solutions*

For the practical problem, finding the optimal solution is no easy task for both of the two problems. However, there exists a lower bound for any scheduling. The optimal energy consumption, given a throughput constraint F_c, is achieved by ignoring any precedence constraints and allowing each actor copy to be arbitrarily split. The optimal energy consumption corresponds to the case where workloads on all of the processors are exactly balanced, and they are stretched to finish exactly at $1/F_c$. The optimal frequency is $f_{opt} = \frac{C_t}{m/F_c}$, where C_t is the total execution cycles of all the actor copies. The ideal minimal energy consumption can be calculated as:

$$E_{opt} = (C_t/f_{opt})f_{opt}^3 = C_t^3 F_c^2/m^2. \tag{5.1}$$

Similarly, the optimal throughput, given an energy consumption constraint per iteration E_c, is also achieved in this case, and the ideally maximal throughput can be calculated as:

$$F_{opt} = \sqrt{m^2 E_c/C_t^3}. \tag{5.2}$$

In our simulations, which are presented in the next section, we will normalize our results by these ideal optimal solutions.

5.2. *A Special Case: Uniform Graphs*

We denote a *uniform* ASDF as a graph in which all of the actors have the same execution cycles. Obviously, all of the actor copies in the transformed DAG also have the same execution cycles. Consequently, in any valid total order of the DAG, the distribution of actor copy's execution cycles are the same. Thus, the energy consumption of the optimal pipeline scheduling will be the same for any valid total order. Further, we notice that the dynamic scheduling algorithm attempts to derive a more balanced partitioning. Assume that each actor copy's number of execution cycles is $C = C_t/N$. To achieve the most balanced partitioning, we first allocate $\lfloor N/m \rfloor$ copies to each stage, and then, we allocate one extra actor copy to each of the $(N - m\lfloor N/m \rfloor)$ stages. In the end, $(N - m\lfloor N/m \rfloor)$ stages have $(\lfloor N/m \rfloor + 1)$

actor copies, and $(m - (N - m\lfloor N/m \rfloor))$ stages have $\lfloor N/m \rfloor$ actor copies. For the first problem, assume that the throughput constraint is F_c. Therefore, the optimal energy consumption of this case is:

$$\begin{aligned} E'_{opt} &= \frac{(N-m\lfloor\frac{N}{m}\rfloor)((\lfloor\frac{N}{m}\rfloor+1)CF_c)^3+(m-N+m\lfloor\frac{N}{m}\rfloor)(\lfloor\frac{N}{m}\rfloor CF_c)^3}{F_c} \\ &= \frac{[(N-m\lfloor\frac{N}{m}\rfloor)(\lfloor\frac{N}{m}\rfloor+1)^3+(m-N+m\lfloor\frac{N}{m}\rfloor)\lfloor\frac{N}{m}\rfloor^3]C_t^3F_c^2}{N^3}. \end{aligned} \tag{5.3}$$

We define the Normalized Energy Consumption (NEC) of a schedule as the energy consumption of the schedule divided by E_{opt}. The practical optimal NEC of this case can be calculated as:

$$R_1 = \frac{E'_{opt}}{E_{opt}} = \frac{(N - m\lfloor\frac{N}{m}\rfloor)(\lfloor\frac{N}{m}\rfloor + 1)^3 + (m - N + m\lfloor\frac{N}{m}\rfloor)(\lfloor\frac{N}{m}\rfloor)^3}{\frac{N^3}{m^2}}. \tag{5.4}$$

For the second problem, assume that the energy consumption constraint is E_c. Thus, the optimal throughput of this case can be calculated as:

$$F'_{opt} = \sqrt{\frac{\frac{N^3E_c}{C_t^3}}{(N-m\lfloor\frac{N}{m}\rfloor)(\lfloor\frac{N}{m}\rfloor + 1)^3 + (m - N + m\lfloor\frac{N}{m}\rfloor)\lfloor\frac{N}{m}\rfloor^3}}. \tag{5.5}$$

We define the Normalized Throughput (NT) of a schedule as the throughput of the schedule divided by F_{opt}. The practical optimal NT of this case can be calculated as:

$$R_2 = \frac{F'_{opt}}{F_{opt}} = \sqrt{\frac{\frac{N^3}{m^2}}{(N-m\lfloor\frac{N}{m}\rfloor)(\lfloor\frac{N}{m}\rfloor + 1)^3 + (m - N + m\lfloor\frac{N}{m}\rfloor)\lfloor\frac{N}{m}\rfloor^3}} = \sqrt{\frac{1}{R_1}}. \tag{5.6}$$

Experiments show that our pipeline scheduling and dynamic programming algorithms also achieves the optimal NEC and NT for the two problems, respectively. Thus, it shows the optimality of our dynamic programming algorithms for this special case.

5.3. *A Special Case: Transformed DAG is Linear*

If the transformed DAG is a linear graph, there is only one valid total order that can be derived from the transformed DAG. Notice that, previously in this paper, we have proven that our dynamic programming algorithms achieve the optimal stage partitioning for a given total order. Thus, our dynamic programming algorithms achieve the optimal pipeline scheduling for this special case.

6. Simulations

We use the toolbox SDF^3 [19] to randomly generate ASDF graphs; after that, our methods are applied to these ASDFs. Experimental results verify the strength of our approaches.

6.1. *Simulation Design*

We design two groups of comparisons to verify two aspects of our scheduling method. The first comparison verifies the strength of our pipeline scheduling against a list scheduling. The second comparison verifies the optimality of the dynamic programming given a valid total order, and compares various schemes to derive a good valid total order from the transformed DAG.

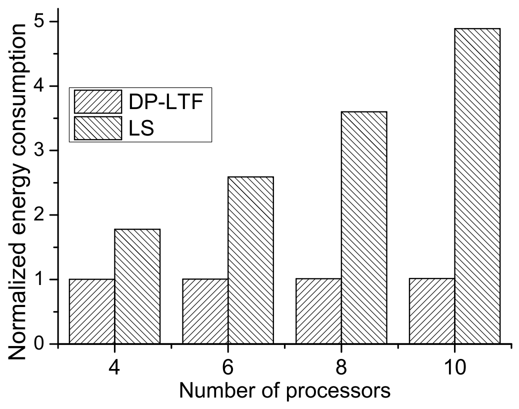
6.1.1. *First Comparison Group*

In the first comparison, we compare our pipeline scheduling against a list scheduling. A List Scheduling (LS) is a non-pipeline based scheduling, which is commonly used to minimize the schedule length of a graph (thus, maximizing throughput). Basically, at each scheduling point (when a processor becomes free), it selects the largest ready task/actor first. Thus, an actor copy to processor mapping can be achieved; we use the pre-power method in [16] to determine the optimal power supply for the graph.

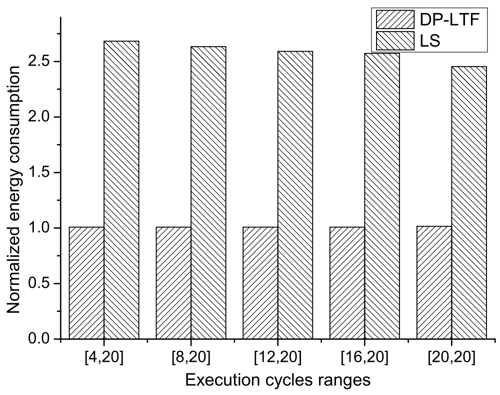
6.1.2. *Second Comparison Group*

We also compare our pipeline scheduling against several other pipeline scheduling schemes. We denote DP-LTF, DP-LCP, DP-STF, and DP-RDM as the methods to derive a valid total order by using the LTF strategy, the Longest Critical Path (LCP) first strategy, the Smallest Task First (STF) strategy, and the randomly selecting strategy (RDM) before adopting the dynamic programming, respectively. The critical path of an actor is defined as the longest path, from the current actor to one of the finishing actors. The DP-LCP scheme selects the actor with the longest critical path first when deriving a valid total order. To demonstrate the optimality of the dynamic programming partitioning scheme, we design a naive mean partitioning method, which is described as follows.

Naive mean partitioning: This method basically constructs a partitioning by referring to the ideal optimal solution. It considers the tasks one by one, according to a given total order. For the first stage, whenever considering whether the next task should be added to this stage, compare the accumulated cycles of the first stage with the ideal optimal solution. Assume that the existing number of cycles in this stage is C_e, and that the number of execution cycles of the next actor copy is c_{next}. The ideal optimal number of execution cycles for each stage is $C_{opt,1} = C_t/m$. If $|C_e - C_{opt} > |C_e + c_{next} - C_{opt}|$, then include the next actor copy in the first stage; otherwise, complete the construction for the first stage. Denote that the execution cycles of the first i completed stages are $C_1, C_2, \cdots, C_i$, $1 \leq i \leq m-1$. When constructing the $(i+1)$th stage, the optimal number of cycles is set to be the remaining execution cycles divided by the remaining number of processors: $C_{opt,i+1} = (C_t - \sum_{k=1}^{i} C_k)/(m-i)$. We approximate this optimal number of cycles in a similar way to that of the first stage. This method repeats the processes until it finishes partitioning all of the m pipeline stages. In the comparison, we also adopt the



(a) Normalized energy consumption for different numbers of processors.



(b) Normalized energy consumption for different execution cycles ranges.

Fig. 5. NEC comparison between list scheduling and pipeline scheduling.

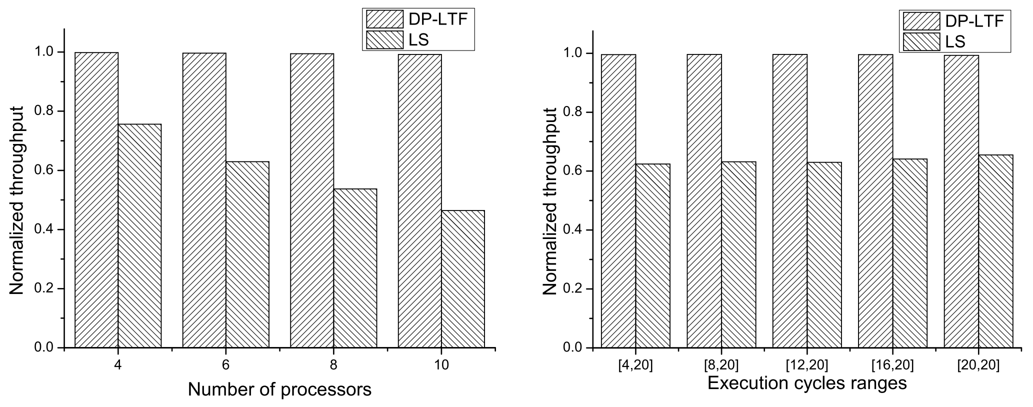
LTF strategy to derive a valid total order for the naive mean partitioning method, so we denote this scheme as NMP-LTF.

In our simulations, we let actors have integer numbers of execution cycles. For an actor with C execution cycles, if we allow it to be split into C unit sub-actors, and adopt dynamic programming to these sub-actors of all of the original actors, the derived scheduling will serve as lower bound that is a tighter than the ideal optimal solution. We denote this method as DP-UNIT, and also include it in the comparison. We normalize the results of all of the methods according to the ideal optimal solutions.

6.2. *Simulation Settings and Results*

As has been mentioned, the two problems considered in this paper are actually closely related. This fact has also been revealed by the relationship between R_1 and R_2 in Section 5.2. Thus, we will discuss the results for the first problem in detail, and briefly provide the results for the second problem.

For the first comparison group, given a total number of actor copies, there are two key factors that influence the NEC; they are the number of processors and the execution cycles ranges of the actors. We evaluate the two methods in three settings. For each setting, we use the SDF^3 tool to randomly generate 1,000 ASDF graphs, and we calculate the average NEC of DP-LTF and LS schemes of the 1,000 cases, respectively. In Setting I, given the execution cycles range of $[12, 20]$, we evaluate the performance of the two methods for different numbers of processors. The NEC for processor numbers 4, 6, 8, and 10 is presented in Fig. 5(a). In setting II, given a fixed number of processors $m = 6$, we evaluate the performance of these three methods for different execution cycles ranges of the actor copies. The NEC for execution cycles ranges $[4, 20]$, $[8, 20]$, $[12, 20]$, $[16, 20]$ and $[20, 20]$, is presented in Fig. 5(b). In setting III, we evaluate the performance of these three methods for various combinations of execution cycles range and the number of processors. The NEC in various cases are



(a) Normalized throughput for different numbers of processors.

(b) Normalized throughput for different execution cycles ranges.

Fig. 6. NT comparison between list scheduling and pipeline scheduling.

Table 3. NEC for various execution cycles range and number of processor combinations.

NEC	$m = 4$		$m = 6$		$m = 8$		$m = 10$	
	DP-LTF	LS	DP-LTF	LS	DP-LTF	LS	DP-LTF	LS
$[4, 20]$	1.00399	1.80251	1.00903	2.6834	1.01657	3.79357	1.02607	5.19984
$[8, 20]$	1.00344	1.79028	1.00788	2.63404	1.01353	3.70107	1.02087	5.02865
$[12, 20]$	1.00312	1.7787	1.00775	2.59117	1.01161	3.59962	1.01644	4.88953
$[16, 20]$	1.00248	1.7618	1.00849	2.57252	1.0055	3.57627	1.00608	4.85156
$[20, 20]$	1	1.69885	1.01476	2.4532	1	3.40141	1	4.61794

provided in Table 3. In all of these settings, we set the total number of actor copies in the transformed DAG to be 40.

For the second comparison group, we notice that, when the variance of the actors' execution requirements is small, all these methods produce similar results. Thus, to distinguish them, we set the number of execution cycles range to be $[1, 20]$, which is the greatest variance in our setting, and just vary the number of processors from 2 to 16, with an increase step of 2. The result for this comparison is provided in Fig. 7(a).

6.3. *Simulation Analyses*

In Fig. 5(a) and Fig. 5(b), the NEC of our proposed DP-LTF is just about 1.02, which means that the energy consumption achieved by the two approaches is only about 2% greater than the ideal optimal solution. However, the energy consumption achieved by the LS is from 78% to 389% greater than the ideal optimal solution. As the number of processors increases, the advantage of our proposed DP-LTF over LS becomes obvious. In Table 3, the NEC of various combinations of execution cycles range and number of processors for all of the three methods is provided. We can see that DP-LTF has stable and good performance under various situations, while the performance of LS decreases when the execution cycles range becomes large and when the number of processors increases. This demonstrates the superiority of a pipeline scheduling over the traditional list scheduling. This fact is also intuitive;

Table 4. NT for various execution cycles range and number of processor combinations.

NT	$m = 4$		$m = 6$		$m = 8$		$m = 10$	
	DP	LS	DP	LS	DP	LS	DP	LS
$[4, 20]$	0.99801	0.75279	0.9956	0.62436	0.9919	0.52960	0.98786	0.45664
$[8, 20]$	0.99839	0.75561	0.99609	0.63155	0.99319	0.53882	0.98964	0.46405
$[12, 20]$	0.99847	0.75621	0.99625	0.62955	0.99417	0.53733	0.99167	0.46437
$[16, 20]$	0.99879	0.76279	0.99570	0.64095	0.99748	0.54800	0.99723	0.47384
$[20, 20]$	1	0.77746	0.99269	0.65490	1	0.56012	1	0.48405

because of the precedence constraints, a list scheduling has a large possibility of leaving processors idle. Consequently, to meet the throughput constraint, a high frequency should be chosen, thus increasing the energy consumption.

In Fig. 7(a), comparing the NEC of the DP-LTF and NMP-LTF, we can see that the DP-LTF always achieves a better solution. When the number of processors increases, the advantage of DP-LTF over NMP-LTF is more obvious. This verifies the optimality of our proposed dynamic programming algorithm for a given total order. As we can see, for different methods to derive a valid total order, if dynamic programming is adopted for pipeline stage partitioning, the NEC values do not differ much from each other, especially when the number of processors is small. However the DP-LCP method requires calculating the LCP of each actor before scheduling, while DP-LTF and DP-STF only need to check the ready actors' execution requirements. Thus, we prefer DP-LTF, DP-STF, and DP-RDM for their overall good and stable performance and the simplicity of implementation.

The results for the second problem, namely, maximizing throughput given energy consumption constraint per iteration of the ASDF, are presented in Fig. 6, Fig. 7(b), and Table 4, respectively. All of these results also demonstrate the strength of pipeline scheduling and dynamic programming for pipeline stage partitioning. We omit the detailed discussions on these results.

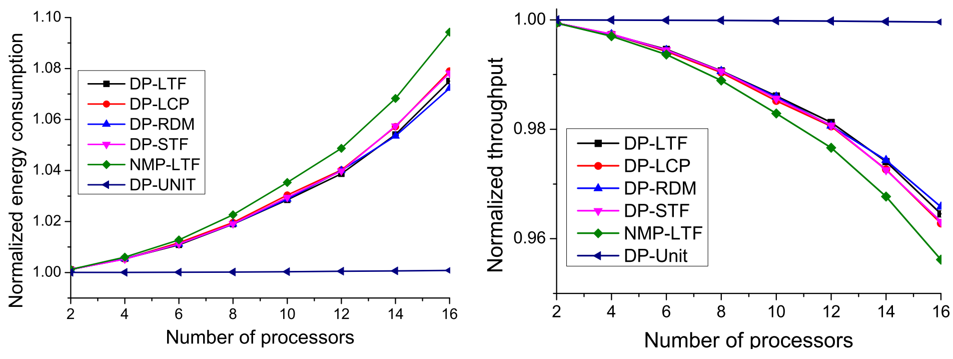
6.4. *Additional Remarks*

In our work, we assume that a processor's dynamic power consumption is $p = f^3$; our approach is not limited by this assumption. Actually, our proposed pipeline scheduling and dynamic programming algorithms work well for the general assumption that the power consumption is $p = f^\alpha$, where α can be any real number greater than or equal to 2.

In this paper, we only address energy-aware scheduling, and have not discussed the buffer requirement of our approach. To consider the buffer requirement, some rules and guidance can be applied to deriving a valid total order, such that the buffer requirement can be minimized.

7. Related Works

Various works consider energy-aware issues for SDFs. The authors in [20] implement an energy-efficient real-time CPU scheduler for multimedia signal processing applications. Their system verifies the applicability of DVFS in SDFs. [21] also deals with



(a) NEC comparison among different pipeline scheduling schemes.

(b) NT comparison among different pipeline scheduling schemes.

Fig. 7. NEC and NT comparisons.

energy minimization for scheduling SDFs on DVFS-enabled platforms. However, this work focuses on the special case where an SDF has several frequency domains, and the actors of the SDF need to be executed according to their input and output data rates, as well as to their frequency domains. The authors of [22] address scheduling (H)SDFs on DVFS-enabled multiprocessors. It takes both static and dynamic power consumption into consideration. To minimize the energy consumption given a throughput constraint, the problem is formulated as a convex optimization problem, which can be solved by a generic convex optimization solver; consequently, the optimal frequency setting for each actor can be determined. The overall scheduling only considers one iteration of the SDF. Although the optimal solution is derived, it still may not be optimal since it does not consider the possibility of scheduling several iterations of an SDF at one time, and due to the precedence constraints among actors, some processors may still be left idle, and cannot be fully utilized.

SDFs belong to the broad category of streaming/workflow applications. A survey [23] by Anne Benoit, *et al*, summarizes existing works in this field, where various application models, platform models, and performance metrics have been considered. However, the main metrics are period and latency, and few of them consider energy consumption. Specifically, [24] considers energy consumption as one metric, but it only addresses the linear chain application model. Our work in this paper addresses general ASDFs. [25] studied the problem of mapping streaming applications onto a 2-dimensional tiled CMP architecture, with the objective of minimizing the energy consumption using DVFS, while maintaining a given throughput. In this paper, we consider energy-aware scheduling for general ASDFs. We propose pipeline scheduling for energy-awareness. Pipeline scheduling will not leave processors idle, even though the precedence constraints among actors exist; thus, it can achieve good performances for energy-aware scheduling.

8. Conclusion

We address the problems of minimizing energy consumption given a throughput constraint, and maximizing throughput under an energy consumption constraint per iteration for ASDFs. We propose the pipeline scheduling scheme to schedule the DAG. After transforming the ASDF to a DAG, we derive a valid total order, which can be achieved via various strategies, from the transformed DAG. Given the derived total order, we design two dynamic programming algorithms to produce optimal pipeline stage partitioning, and the frequency setting, for each stage for the two problems, respectively. We analyze our approaches in detail, and various experiments verify the strength of our approaches.

Acknowledgement

This research was supported in part by NSF grants ECCS 1231461, ECCS 1128209, CNS 1065444, and CCF 1028167.

References

1. M. Horowitz, T. Indermaur, and R. Gonzalez, “Low-power digital design,” in *Low Power Electronics. Digest of Technical Papers, IEEE Symposium*, Oct. 1994, pp. 8 – 11.
2. A. Chandrakasan, A. Burstein, and R. W. Brodersen, “A low power chipset for portable multimedia applications,” in *Solid-State Circuits Conference, Digest of Technical Papers, IEEE International*, Feb. 1994, pp. 82 – 83.
3. T. D. Burd and R. W. Brodersen, “Energy efficient cmos microprocessor design,” in *Proceedings of the 28th Hawaii International Conference on System Sciences*, vol. 1, Jan. 1995, pp. 288 – 297.
4. F. Yao, A. Demers, and S. Shenker, “A scheduling model for reduced cpu energy,” in *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, Oct. 1995, pp. 374 – 382.
5. J.-J. Chen and C.-F. Kuo, “Energy-efficient scheduling for real-time systems on dynamic voltage scaling (dvs) platforms,” in *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug. 2007, pp. 28 – 38.
6. D. Li and J. Wu, *Energy-aware Scheduling on Multiprocessor Platforms*, ser. Springerbriefs in Computer Science. Springer, Oct. 2012.
7. E. A. Lee and D. G. Messerschmitt, “Static scheduling of synchronous data flow programs for digital signal processing,” *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24 – 35, Jan. 1987.
8. ——, “Synchronous data flow,” *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235 – 1245, Sep. 1987.
9. D.-J. Wang and Y. H. Hu, “Fully static multiprocessor array realizability criteria for real-time recurrent dsp applications,” *IEEE Transactions on Signal Processing*, vol. 42, no. 5, pp. 1288 – 1292, May 1994.
10. A. Bonfietti, M. Lombardi, M. Milano, and L. Benini, “Throughput constraint for synchronous data flow graphs,” in *Proceedings of the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 2009, pp. 26 – 40.
11. A. Bonfietti, L. Benini, M. Lombardi, and M. Milano, “An efficient and complete approach for throughput-maximal sdf allocation and scheduling on multi-core platforms,”

in *Proceedings of the Conference on Design, Automation and Test in Europe*, Mar. 2010, pp. 897 – 902.

12. T.-H. Shin, H. Oh, and S. Ha, "Minimizing buffer requirements for throughput constrained parallel execution of synchronous dataflow graph," in *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, Jan. 2011, pp. 165 – 170.
13. Y. Chen and H. Zhou, "Buffer minimization in pipelined sdf scheduling on multi-core platforms," in *Proceedings of the 17th Asia and South Pacific Design Automation Conference*, Jan. 30 - Feb. 2 2012, pp. 127 – 132.
14. M. Lattuada and F. Ferrandi, "Modeling pipelined application with synchronous data flow graphs," in *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, July 2013, pp. 49 – 55.
15. S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessor: Scheduling and Synchronization*, 2000.
16. K. Li, "Scheduling precedence constrained tasks with reduced processor energy on multiprocessor computers," *Computers, IEEE Transactions on*, vol. 61, no. 12, pp. 1668 – 1681, Dec. 2012.
17. R. Xu, C. Xi, R. Melhem, and D. Moss, "Practical pace for embedded systems," in *Proceedings of the 4th ACM international conference on Embedded software*, 2004, pp. 54 – 63.
18. R. C. Wrede and M. Spiegel, *Schaum's Outline of Advanced Calculus, Second Edition.* The McGraw-Hill Companies, Inc., Feb. 2002.
19. S. Stuijk, M. C. W. Geilen, and T. Basten, "SDF^3: SDF For Free," in *Proceedings of the 6th International Conference Application of Concurrency to System Design*, Jun. 2006, pp. 276 – 278.
20. W. Yuan and K. Nahrstedt, "Energy-efficient cpu scheduling for multimedia applications," *ACM Trans. Comput. Syst.*, vol. 24, no. 3, pp. 292 – 331, Aug. 2006.
21. B. Knerr, M. Holzer, and M. Rupp, "Task sheduling for power optimisation of multi frequency synchronous data flow graphs," in *Proceedings of the 18th Symposium on Integrated Circuits and System Design*, 2005, pp. 50 – 55.
22. A. Nelson, O. Moreira, A. Molnos, S. Stuijk, B. T. Nguyen, and K. Goossens, "Power minimisation for real-time dataflow applications," in *Proceedings of the 14th Euromicro Conference on Digital System Design*, Aug. 31 - Sep. 2 2011.
23. A. Benoit, U. V. Catalyrek, Y. Robert, E. Saule, and et al., "A survey of pipelined workflow scheduling: Models and algorithms," Sep. 2011.
24. A. Benoit, P. R.-Goud, and Y. Robert, "Performance and energy optimization of concurrent pipelined applications," in *IEEE International Symposium on Parallel Distributed Processing*, Apr. 2010, pp. 1 – 12.
25. A. Benoit, P. R.-Goud, Y. Robert, and R. Melhem, "Energy-aware mappings of series-parallel workflows onto chip multiprocessors," in *International Conference on Parallel Processing*, Sep. 2011, pp. 472 – 481.