

Buffer Minimization in Pipelined SDF Scheduling on Multi-Core Platforms

Yuankai Chen and Hai Zhou

Electrical Engineering and Computer Science, Northwestern University, U.S.A.

Abstract—With the increasing number of cores available on modern processors, it is imperative to solve the problem of mapping and scheduling a synchronous data flow graph onto a multi-core platform. Such a solution should not only meet the performance constraint, but also minimize resource usage. In this paper, we consider the pipeline scheduling problem for acyclic synchronous dataflow graph on a given number of cores to minimize the total buffer size while meeting the throughput constraint. We propose a two-level heuristic algorithm for this problem. The inner level finds the optimal buffer size for a given topological order of the input task graph; the outer level explores the space of topological order by applying perturbation to the topological order to improve buffer size. We compared our proposed algorithm to an enumeration algorithm which is able to generate optimal solution for small graphs, and a greedy algorithm which is able to run on large graphs. The experimental results show that our two-level heuristic algorithm achieves near-optimal solution compared to the enumeration algorithm, with only 0.8% increase in buffer size on average but with much shorter runtime, and achieves 38.8% less buffer usage on average, compared to the greedy algorithm.

I. INTRODUCTION

In order to meet the ever-increasing demands of computation performance, multi-core systems have gradually become the mainstream engine from high-end servers to personal computers in recent years. More cores running concurrently on a chip provides great potential for high-degree parallelism, but it also raises challenges to CAD designers: how to automatically schedule a specific application to a multi-core platform and to make best use of the parallelism from the limited resource.

Synchronous dataflow (SDF) graphs [1] are usually used to model stream-based multimedia applications, which can be found widely in TV sets, cellphones, etc. In an SDF, a node represents a computation task, and an edge represents a communication channel. When a task is executed, it takes data from its input channels and puts data to its output channels. When data is put in a channel, it may not be consumed immediately, instead it may accumulate, depending on the task scheduling, so buffers are needed to store this data. Because the total available storage space on a chip is limited and less buffers would be more energy-efficient, under the same performance constraint, schedules with fewer buffers are preferred.

Pipelining is a powerful technique to utilize parallelism of multi-core systems. A pipelined scheduling divides tasks into stages. All stages in pipelined schedule can be executed at the same time, but the greater the stage difference is for two tasks of a channel, the more buffers are needed. An advantage of pipelined scheduling over traditional self-timed scheduling, where a task can be executed whenever its inputs are ready, is that designers can set barrier at the end of each stage to ensure all cores are kept synchronized [2], and therefore it reduces buffer detection overhead.

The throughput of a pipelined schedule is determined by the longest stage. For general multimedia application, throughput is one of the most important measures of the performance of the system. One should not optimize the total buffer size at the cost of large sacrifice on performance. In this paper, for a given SDF graph, and given

number of processors, we consider the problem of pipeline scheduling for buffer size minimization, under the throughput constraint.

In order to schedule SDF graph onto multi-core platform, two problems need to be solved: processor mapping and stage assignment, where processor mapping is to map tasks to processors and stage assignment is to assign stages to each task.

Early work on pipeline scheduling was mainly focused on optimization of throughput. Bokhari [3] developed an efficient algorithm for mapping a chain-structured task graph onto a linear chain of processors to achieve optimal throughput. Hoang et al. [4] proposed list-scheduling based algorithm to optimize throughput for general graph. For performance constrained pipelining, the algorithm by Bakshi and Gajski [5] aims at minimizing hardware area under performance constraint. Shin et al. [6] proposed a genetic algorithm to minimize buffer size under throughput constraint. However, they assume the processor mapping is already known and also assume dynamic scheduling policy that allows task to be fired once its inputs are ready, which is different from the concept of pipelining we discuss in this paper.

Recently, several works have been published to explore the relation between buffer size and throughput. For a given buffer distribution, Stuijk et al. [7][8] used model checking method to simulate the state transition to determine the throughput of the system. Because the number of states grows exponentially with the size of the graph. Their method has exponential time complexity. For a given throughput, Wiggers et al. [9] proposed a linear-runtime algorithm to estimate the minimum buffer size needed to achieve the throughput. All of the above algorithms are based on multi-processor architecture, in which each processor has only one task and processors execute in self-timed manner. It is worth mentioning that Hu et al. [10] formulated stage assignment into a min-cost flow problem. We will use a similar technique as part of our algorithm. To the best of our knowledge, there is no existing work on buffer size minimization problem considering processor mapping and pipeline stage assignment simultaneously.

The contribution of this paper is: we formulate the buffer minimization problem under the throughput constraint, for the first time, with the consideration of both processor mapping and pipeline stage assignment. In addition, based on the proposed formulation, we design a two-level heuristic algorithm which is able to reduce the buffer size efficiently and effectively.

The rest of this paper is organized as follow: In Section II, we introduce basic concepts and assumptions used in this paper. Problem formulation is given formally in Section III. In Section IV, we present our two-level heuristic algorithm. Experimental results are shown in Section V. We conclude the paper in the last section.

II. PRELIMINARY AND ASSUMPTIONS

A. Synchronous Dataflow Graphs

In a synchronous dataflow (SDF) graph $G = (V, E)$, each node represents a computation task, and it is called an actor. Each edge is a communication channel, in which the actor at the head produces data

and the actor at the tail consumes data. When an actor fires (performs computation), it consumes a certain amount of tokens (data items) from each of its input channels, and when the computation finishes, it produces a certain amount of tokens on each of its output channels.

The production rate and the consumption rate of a channel may be different. In order for the SDF graph to be executed with bounded memory, a repetition vector $\vec{q}$ is computed, where each element $q(u)$ for actor u is a positive integer, representing the minimum number of times actor u needs to be executed in a complete scheduling cycle [1]. In this paper, we assume $q(u)$ consecutive executions of actor u as the basic schedulable unit. The execution time $t(u)$ of actor u is the time of executing the actor $q(u)$ times. With this assumption, we have equal production and consumption rates for each channel. We use a single rate $r(e) \in \mathbb{Z}_0^+$ for channel e to represent both of its production and consumption rates. An example of SDF graph is shown in Fig.1 (a).

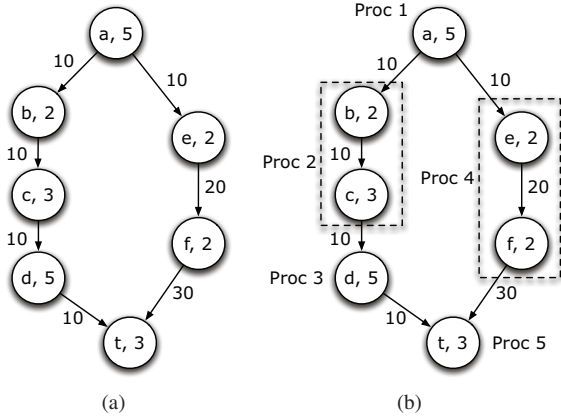


Fig. 1. (a) An Example of SDF graph. Numbers in nodes are execution times. Numbers on edges are channel rates. (b) A possible processor mapping

B. Multi-Core Pipeline Scheduling and Buffer Minimization

Multi-core pipeline scheduling consists of two tasks: stage assignment and processor mapping. Stage assignment assigns processors to different stages. Let u be the producer and v be the consumer for a channel (u, v) . If they are assigned to different processors $p(u)$ and $p(v)$ respectively, inter-processor communication is needed for by u to pass data to v . In a pipelined schedule, $p(u)$ must be assigned to an earlier stage than that of $p(v)$. To avoid cyclic data dependencies among processors (Fig. 2), we require that the set of actors assigned to each processor is convex [11], where convexity is defined as follows:

Definition 1: A set of actors S is convex if there exists no path from a node u in S to another node v in S that involves a node w that is not in S .

In Fig. 2 (a), nodes a and c do not form a convex set. Therefore they cannot be assigned to the same processor because otherwise it would result in a cyclic data dependency as is shown in Fig. 2 (b).

For a channel (u, v) , suppose $p(u)$ is assigned to stage $s_{p(u)}$, $p(v)$ is assigned to stage $s_{p(v)}$ and the rate of the channel is $r(u, v)$. Then the number of buffers needed for this channel can be expressed as:

$$(s_{p(v)} - s_{p(u)}) \times r(u, v) \quad (1)$$

The equation above indicates that the stage assignment directly affects the total inter-processor buffer size. For a given processor mapping (Fig. 1 (b)), if processor Proc4 is assigned to stage 2 (Fig. 3 (a)), the total number of buffers is $10+10+10+10+30 \times 2=100$; while if Proc 4 is assigned to stage 4 (Fig. 3 (b)), the total number of buffers is $10+10+10+10 \times 2+30=80$, with 20 buffers reduced.

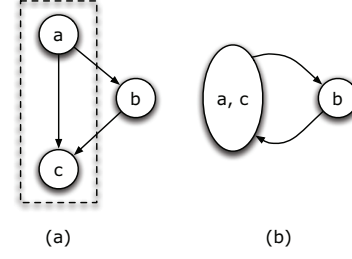


Fig. 2. (a) An example of non-convex mapping (b) Cyclic data dependency

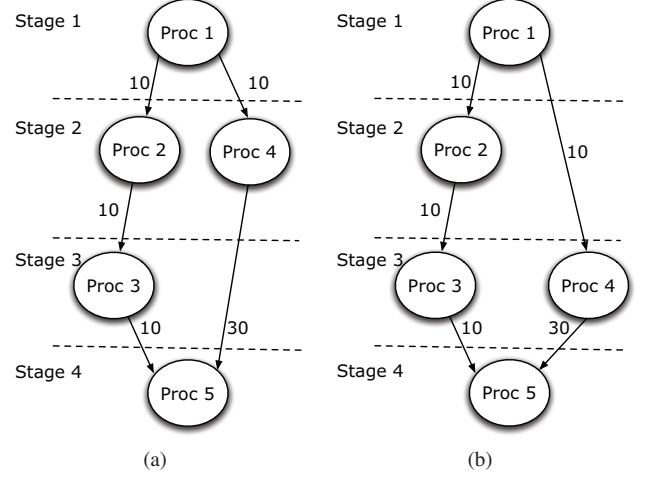


Fig. 3. Stage assignment directly affects buffer size.

Processor mapping assigns actors to a given number of processors. In this paper, we assume processors are homogeneous, i.e., the execution time of an actor is the same on all processors. Since processor mapping will decide the data dependencies among processors, it also has an intricate impact on buffer size. Moreover, processor mapping also affects the performance of the system in other aspects, such as throughput, which is considered as one of the most important measures of the system efficiency.

III. PROBLEM FORMULATION

As is discussed in last section, both processor mapping and stage assignment will affect the total inter-processor buffer size. Therefore, in this work, we propose to minimize total inter-processor buffer size with consideration of both processor mapping and stage assignment under throughput constraint.

Multi-Core Scheduling for Buffer Size Minimization (MCS-BM) Problem: Given an acyclic SDF graph $G = (V, E)$, functions $t(\cdot) : V \rightarrow \mathbb{Z}_0^+$, $r(\cdot) : E \rightarrow \mathbb{Z}_0^+$ representing the execution time of each node (in number of clock cycles) and rate of production and consumption of each edge; and given a set of P homogeneous processors, V_p , the problem asks to find:

- 1) A 0-1 integer a_{up} for each actor and processor pair $(u, p) \in V \times V_p$, which equals 1 if actor u is assigned to processor p , otherwise equals 0;
- 2) A stage number s_p for each processor $p \in V_p$.

subject to the following constraints:

$$\sum_{p \in V_p} a_{up} = 1 \quad \forall u \in V \quad (2)$$

$$\sum_{u \in V} a_{up} t(u) \leq T \quad \forall p \in V_p \quad (3)$$

$$s_{p(v)} - s_{p(u)} \geq 1 \quad \forall (u, v) \in E \wedge p(u) \neq p(v) \quad (4)$$

Constraint Eq. (2) requires that each actor is assigned to one and only one processor; constraint Eq. (3) is the throughput constraint, where T is the limit of stage makespan (the reciprocal of the throughput). The longest stage cannot exceed T clock cycles. We assume that the inter-processor delay δ is much smaller than T , so we do not consider inter-processor delay in our model. For the cases where inter-processor delay is not negligible, by adjusting stage makespan to $T + \max(\delta)$, our solution is still applicable, where $\max(\delta)$ is the longest delay among all inter-processor channels. In constraint Eq. (4), $p(\cdot)$ is a function: $V \rightarrow V_p$, returning the processor that actor is mapped to. Eq. (4) is the data dependency constraint, requiring the processor containing the producer of a channel needs to be assigned to a stage earlier than the processor containing the consumer of that channel.

We define buffer size function $b(\cdot): E \rightarrow \mathbb{Z}_0^+$, representing the smallest buffer size needed for each channel. $b(u, v)$ on channel (u, v) is given by the following equation:

$$b(u, v) = \begin{cases} 0 & \text{if } p(u) = p(v) \\ (s_{p(v)} - s_{p(u)}) \times r(u, v) & \text{if } p(u) \neq p(v) \end{cases} \quad (5)$$

The objective is to minimize the total inter-processor buffer size, that is:

$$\min \sum_{(u, v) \in E} b(u, v) \quad (6)$$

IV. ALGORITHM

The structure of this section is organized as follows: we first study the stage assignment problem in which we suppose the processor mapping is given. We can find optimal solution for stage assignment by applying min-cost flow algorithm. Then we study MCS-BM problem and explore its dynamic programming structure. In the last two subsections we introduce our proposed two-level heuristic algorithm. At inner level, we reduce the acyclic graph into topological order such that buffer minimization problem can be solved by dynamic programming efficiently. At outer level, we apply perturbation to the topological order. New topological order is sent to dynamic programming for a new round of iteration. This process is repeated until no improvement on the result is obtained during a certain steps of iterations.

A. Solving Stage Assignment Problem by Min-Cost Flow Algorithm

Suppose the processor mapping is given, stage assignment is to assign a stage number to each processor, such that the total inter-processor buffer size is minimized under the constraint of Eq. (4). This problem is a special case of generalized formulation in [10]. In this subsection, we briefly introduce how we obtain the min-cost flow formulation.

We construct a data flow graph $G_p = (V_p, E_p)$ on the set of processors. A directed edge (p_i, p_j) is added to E_p if there exists channel in G across p_i and p_j and the producer is in p_i . The following Eq. (7) depicts constraint Eq. (4) in G_p :

$$s_{p_j} - s_{p_i} \geq 1 \quad \forall (p_i, p_j) \in E_p \quad (7)$$

We define the edge rate $r(p_i, p_j)$ between processors p_i and p_j as the sum of the rates of channels from p_i to p_j :

$$r(p_i, p_j) = \sum_{(u, v) \in E \wedge p(u)=p_i \wedge p(v)=p_j} r(u, v) \quad (8)$$

A “demand” for each processor $d(\cdot): V_p \rightarrow \mathbb{N}$ is defined as the net number of tokens that each processor outputs in a stage, i.e.,

$$d(p_i) = \sum_{(p_i, p_j) \in E_p} r(p_i, p_j) - \sum_{(p_j, p_i) \in E_p} r(p_j, p_i) \quad (9)$$

With some straightforward mathematical manipulation, objective function Eq. (6) can be transformed in terms of demands and stage numbers:

$$\max \sum_{p_i \in V_p} d(p_i) s_{p_i} \quad (10)$$

The objective function Eq. (10) under constraint Eq. (7) form a dual problem of min-cost flow problem [12]. The primal formulation can be written as Eq. (11)-(13).

$$\min \sum_{(p_i, p_j) \in E_p} -x_{p_i p_j} \quad (11)$$

$$s.t. \quad \sum_{(p_i, p_j) \in E_p} x_{p_i p_j} - \sum_{(p_j, p_i) \in E_p} x_{p_j p_i} = d(p_i) \quad (12)$$

$$x_{p_i p_j} \geq 0 \quad (13)$$

In this min-cost flow formulation, all edges are uncapacitated and all edge costs are -1. Because we require that the set of actors on each processor is convex, G_p is also acyclic. Therefore this min-cost flow problem has finite solution and potentials of nodes are well defined. After solving min-cost flow problem, we obtain the potentials of nodes $pt(\cdot): V_p \rightarrow \mathbb{N}$ in the flow residual graph [12]. The stage number of each processor is the negative of its potential:

$$s_p = -pt(p) \quad \forall p \in V_p \quad (14)$$

B. Dynamic Programming Structure

We start considering processor mapping and stage assignment simultaneously. In this subsection, we introduce some definitions and explore the dynamic programming structure of MCS-BM problem.

Definition 2: A pair of non-empty sets of nodes $(S, V - S)$ is a **partition** on an acyclic directed graph $G = (V, E)$ if and only if for any edge (u, v) , if $v \in S$, $u \in S$.

This definition requires that the graph is partitioned in topological order, i.e., no forward edge from $V - S$ to S exists.

Definition 3: A **staged partition** $(S, V - S)$ with respect to stage number $\bar{s}$ is a partition that satisfies the two following conditions:

- 1) $\forall u \in S, s_{p(u)} \leq \bar{s}$
- 2) $\forall v \in V - S, s_{p(v)} > \bar{s}$

Definition 4: We define the **cost** of a staged partition $c(S_1, S_2)$ to be the sum of the rates of crossing edges:

$$c(S_1, S_2) = \sum_{(u, v) \in E \wedge u \in S_1 \wedge v \in S_2} r(u, v) \quad (15)$$

Suppose (S_1, S_2) is a staged partition with respect to $\bar{s}$, for any crossing edge e , $r(e)$ tokens need to be passed from stage $\bar{s}$ to $\bar{s} + 1$, which requires the same size of buffers to store these tokens. $c(S_1, S_2)$ reflects the buffer requirement at the partition boundary between S_1 and S_2 .

Given a feasible scheduling, suppose the maximum stage is m , we can obtain a series of sets of nodes $G_1, G_2, \dots, G_m$ by grouping the actors in the same stage together, i.e.,

$$G_i = \{u | u \in V \wedge s_{p(u)} = i\} \quad (16)$$

Let $S^i = G_1 \cup G_2 \cup \dots, G_i$. By definition, $(S^i, V - S^i)$ is a staged partition with respect to stage i .

Using the above definitions, it is easy to express the total buffer size of a given feasible scheduling solution in terms of costs of staged partitions:

$$\sum_{i=1}^{m-1} c(S^i, V - S^i) \quad (17)$$

Given a set of actors S , let $G(S)$ be the subgraph of G that only contains all of the actors in S , and all of the edges between actors in S . Let $\pi(S)$ be the minimum number of processors that are needed to schedule $G(S)$ in a single stage under the makespan constraint. In cases where $G(S)$ cannot be scheduled into a single stage with any number of processors, $\pi(S)$ is infinity.

Definition 5: For a staged partition $(S, V - S)$, we define $f(S, \lambda)$ as the minimum buffer size of scheduling $G(S)$ onto λ processors. Notes that this buffer size includes part of buffers on channels from S to $V - S$. For example, for a staged partition with respect to stage 4, and a channel from stage 2 to stage 5, the buffers on that channel from stage 2 to stage 4 are counted in $f(S, \lambda)$.

Lemma 1: Let $(S, V - S)$ be a staged partition, $f(S, \lambda)$ is independent of schedule in $V - S$.

Proof: By Eq. (17), the total buffer size of S is the sum of the costs of every staged partition in S . Suppose $(S_1, S - S_1)$ is a staged partition in S , the cost of this partition is

$$c(S_1, V - S_1) = c(S_1, (S - S_1) \cup (V - S)) \quad (18)$$

where we can see the nodes from $V - S$ appear in the cost function collectively, by which it means any cost of staged partition in S is independent of schedule in $V - S$, so is $f(S, \lambda)$. ■

Theorem 1: If $\pi(S) \leq \lambda$, we set $f(S, \lambda) = 0$, otherwise $f(S, \lambda)$ can be computed by the following recursive function:

$$f(S, \lambda) = \min_{(S_1, S_2) \in \Omega} f(S_1, \lambda - \pi(S_2)) + c(S_1, V - S_1) \quad (19)$$

where Ω represents the set of all possible partitions on S . The boundary condition of this recursive function is:

$$f(S, \lambda) = \infty, \quad \text{if } \lambda \leq 0 \quad (20)$$

The minimum buffer size of MCS-BM problem is given by $f(V, P)$.

Proof: Case 1: If $\pi(S) \leq \lambda$, S can be scheduled into a single stage with λ processors. In this case, no buffer is needed, so $f(S, \lambda) = 0$;

Case 2: If $\pi(S) > \lambda$, at least two stages are needed. Therefore there exists a partition of S , (S_1, S_2) , where S_2 is the last stage of S . First, it is easy to prove that $f(S, \lambda)$ is non-increasing with respect to λ . In Eq. (19), for a fixed partition, i.e. with a fixed cost $c(S_1, V - S_1)$, $\lambda - \pi(S_2)$ leaves greatest possible number of processors to S_1 , which guarantees $f(S, \lambda)$ is minimum under a fixed partition. Second, by Lemma 1, $f(S_1, \lambda - \pi(S_2))$ is independent of schedule in $V - S_1$. Eq. (19) examines all possible partitions of S , which guarantees the optimal partition cannot be missed. ■

C. Dynamic Programming for Given Topological Order

Though Eq. (19) provides a way to compute optimal buffer size, the number of possible partitions grows exponentially along with the size of the graph, which becomes infeasible in terms of runtime and storage requirement when the graph is large. At the inner level of the two-level heuristic algorithm, we assume the actors are lined up in a given topological order, and each stage spans over some consecutive actors in the line. We can solve buffer minimization problem efficiently with this assumption. Our overall algorithm is repeatedly to perturb topological order and to find optimal solution for each generated topological order.

All actors are sorted in vector $\vec{A}$ in topological order. We use $A(i)$ to represent the actor at the i th position in $\vec{A}$, and $S(i, j)$ to represent the set of nodes $A(i), A(i+1), \dots, A(j)$. Stage partitioning is to partition $\vec{A}$ into segments, each of which is a stage.

To adapt Eq. (19) to this context, a few changes in notations are first made for convenience as the problem is reduced to one dimensional

domain. We use $f(i, \lambda)$ to represent the minimum buffer size of scheduling actors of $S(1, i)$ onto λ processors, under the condition that $(S(1, i), S(i+1, |V|))$ is a staged partition. $\pi(i, j)$ denotes the least number of processors needed to schedule actors $S(i, j)$ into a single stage. $c(j)$ is the cost of partition $(S(1, j), S(j+1, |V|))$, which can be calculated by summing up the rates of channels whose heads are before or at j and tails are after j . Eq. (19) can be rewritten as follows:

$$f(i, \lambda) = \begin{cases} 0 & \text{if } \pi(1, i) \leq \lambda \\ \min_{j < i} f(j, \lambda - \pi(j+1, i)) + c(j) & \text{otherwise} \end{cases} \quad (21)$$

To find out $\pi(i, j)$, we first construct subgraph $G(S(i, j))$. Because these nodes are to be assigned to a single stage, it requires that no channel is allowed to cross two processors in this stage. This inspires us to find out all the connected components in $G(S(i, j))$. For two nodes u and v if there exists a path in $G(S(i, j))$ between them, they are in the same connected components.

Suppose the connected components are $C_1, C_2, \dots, C_m$. Each connected component C_i has a execution time $t(C_i)$, which is the sum of execution times of the nodes in C_i . The problem becomes: given a set of connected components, what is the least number of processors needed to accommodate all of them. This is equivalent to the bin packing problem that asks for the least number of bins to accommodate all items. Bin packing problem is a well-known NP-hard problem. Here we use first-fit decrease algorithm to get an approximate solution.

When computing $f(i, \lambda)$, we can use an array $g(i, \lambda)$ to record where the minimum value of $f(i, \lambda)$ is obtained. After $f(|V|, P)$ is obtained, we can trace back along $g(i, \lambda)$ to find out the optimal stage partitioning. For each stage, we can call bin packing algorithm again to obtain the processor mapping solution. The overall dynamic programming algorithm is summarized in Algorithm 1.

The time complexity of Algorithm 1 can be analyzed as follows: To initialize each $c(i), i = 1..|V|$, each edge needs to be visited, which takes $O(|E|)$ time. The initialization of vector c takes $O(|V||E|)$ time. In the initialization of each $\pi(i, j), 1 \leq i \leq j \leq |V|$, to find connected components can be realized with breadth first search, which takes $O(|E|)$ time in the worst case. The greedy bin packing algorithm takes $O(|V|)$ time in the worst case. It takes $O(|E| + |V|)$ time to initialize each $\pi(i, j)$, therefore $O(|V|^2(|E| + |V|))$ time to initialize the whole table π . The size of the table f is $|V|P$. To compute each element in f takes $O(|V|)$ time based on Eq. (21). The time to compute the whole f is $O(|V|^2P)$. The total runtime of Algorithm 1 is:

$$O(|V||E|) + O(|V|^2(|E| + |V|)) + O(|V|^2P) = O(|V|^2|E|)$$

The above equality holds due to the fact that $|V| < |E|$ and $P < |V|$.

D. Topological Order Perturbation

At the outer level, we perturb the topological order after current topological order is solved by Algorithm 1. In this subsection, we present two heuristic techniques to perturb topological order based upon the solution obtained by inner level.

Even though by running Algorithm 1 we can obtain an optimal stage partition for current topological order, the buffer size may be further reduced by exploring the topological order space. First we call min-cost flow algorithm to re-assign stages to each processor. Note that min-cost flow algorithm could only further reduce the buffer size, or keep even. After re-assigning stages, we rearrange actors by their stage numbers in ascending order. Nodes in the same stage are sorted by their topological order.

Algorithm 1 DP Algorithm

Input: SDF graph $G=(V, E)$, a topological order of actors $\vec{A}$ and number of processors P .

Output: processor mapping solution $\{a_{up}|u \in V, p \in V_p\}$

```

1: initialize  $c(i), i = 1..|V|$ 
2: for  $i = 1$  to  $|V|$  do
3:   for  $j = i$  to  $|V|$  do
4:     find connected components of  $S(i, j)$ 
5:     initialize  $\pi(i, j)$  by calling bin packing algorithm
6:   end for
7: end for
8: for  $i = 1$  to  $|V|$  do
9:   for  $\lambda = 1$  to  $P$  do
10:    compute  $f(i, \lambda)$  by Eq. (21), record  $g(i, \lambda)$  accordingly
11:   end for
12: end for
13: trace back from  $g(|V|, P)$  to obtain stage partition
14: for each stage, call bin packing algorithm again to obtain
    $\{a_{up}|u \in V, p \in V_p\}$ 
15: return  $\{a_{up}|u \in V, p \in V_p\}$ 

```

The other technique is to switch nodes between two stages. For each node u , we calculate a priority number $pri(u)$ by:

$$pri(u) = \sum_{(v,u) \in E} b(v, u) - \sum_{(u,v) \in E} b(u, v) \quad (22)$$

Higher priority number indicates the node is preferred to be placed to early stage. The key idea of this technique is to perturb the topological order by switching nodes at the boundary of two stages. For each stage s , we define the set of input nodes I_s as the nodes in stage s whose input edges are all from previous stages, and the set of output nodes O_s whose output edges are all going to afterward stages. We re-arrange the nodes in the same stage by first placing the input nodes, last placing the output nodes while not violating topological order. If a node is an input node and an output node at the same time, we classify it to input node if its priority number is positive, otherwise to output node. After this re-arrangement, input nodes in stage s become adjacent to the output nodes in stage $s - 1$. Now we switch the order of boundary nodes in O_{s-1} and I_s . For each stage s , we sort nodes in O_{s-1} and I_s in descending order of priority number. while not violating topological order.

A summary of overall two-level heuristic algorithm is described in Algorithm 2.

V. EXPERIMENTAL RESULTS

We implemented our algorithm in C++ using GCC 4.3.2. All experiments are conducted on a Linux machine with 2.8GHz Intel Pentium Dual-core and 2GB RAM. SDF3 [13] is used to generate random DAG cases for our experiments. Execution times and degrees of nodes are generated with parameters listed in Table I. Rates of each edge are assigned with random integers from 1 to 100. Stage makespan constraint T is estimated by:

$$T = \frac{T_{avg.} \times |V| \times 120\%}{P} \quad (23)$$

where $T_{avg.}$ is the average execution time of tasks, and 120% is chosen to leave proper amount of space to assign tasks, making the scheduling not too easy nor too hard.

Since we are the first to address pipeline scheduling for buffer minimization, there is no existing algorithm that we can use for comparison. Therefore we compared our proposed two-level heuristic algorithm to two simple algorithms. First we compared to an optimal

Algorithm 2 Two Level Heuristic Algorithm

Input: SDF graph $G=(V, E)$, number of processors P .

Output: processor mapping $\{a_{up}|u \in V, p \in V_p\}$, stage assignment $\{s_p|p \in V_p\}$

```

1: treat each node as a single processor, call min-cost flow algorithm to assign stages
2: get an initial topology order by sorting nodes in ascending order of stage number
3: minBufferSize  $\leftarrow \infty$ 
4: repeat
5:   call Algorithm 1
6:   call min-cost flow algorithm to re-assign stages
7:   sort nodes in ascending order of new stage number
8:   if currentBufferSize  $<$  minBufferSize then
9:     minBufferSize  $\leftarrow$  currentBufferSize
10:    update  $\{a_{up}|u \in V, p \in V_p\}$  and  $\{s_p|p \in V_p\}$ 
11:   end if
12:   compute priority number  $pri(u)$  for each node  $u$ 
13:   for each stage  $s$  do
14:     re-arrange the nodes in  $s$  by first placing the input nodes, last placing the output nodes, while not violating topological order
15:   sort nodes in  $O_{s-1}$  and  $I_s$  in descending order of priority number, while not violating topological order
16:   end for
17: until minBufferSize has not been reduced in recent  $N$  iterations
18: return  $\{a_{up}|u \in V, p \in V_p\}$  and  $\{s_p|p \in V_p\}$ 

```

TABLE I
PARAMETERS FOR GENERATION OF DAGS

	Avg.	Min	Max	Var
Execution Time	10	2	15	8
Degree of Nodes	3	1	5	1

enumeration algorithm that enumerates all topological orders and applies DP algorithm to each topological order. As the number of topological orders grows near-factorially with the graph size, it is not a practical algorithm for large SDF graphs. Therefore we generated four test cases for this comparison: they are all small graphs with 20 nodes, the number of available processors is four and the stage makespan constraint is 60 clock cycles. In the enumeration algorithm, we also replace the greedy bin packing algorithm with a dynamic programming algorithm that is able to generate optimal solution for small cases [14]. Therefore this enumeration algorithm gives optimal solution for given SDF graph. Table II shows the performance of both algorithms, where $\#B$ stands for the number of buffers and $\#Iter$ stands for the number of iterations run by our proposed algorithm. We can see that the buffer size of our proposed algorithm is very close to the optimal one, with only 0.8% increase on average. For two of the test cases, our proposed algorithm achieved the optimal solution. The runtime of our proposed algorithm is much shorter, compared to the enumeration algorithm.

Second, we implemented a greedy algorithm with polynomial runtime complexity. We use it as a baseline for test cases with larger graph size. The idea of the greedy algorithm is described as follows: First, we define that if edge (u, v) is contractable, it must satisfies the following two conditions:

- 1) The difference of stages of u and v must be exactly one. If not, contracting u and v will introduce negative cycle in graph G .
- 2) The sum of execution times of u and v must not be greater than

TABLE II
EXPERIMENTAL RESULTS OF COMPARING TO ENUMERATION ALGORITHM

Tests	Enumeration Alg.		Proposed Alg.			$\frac{\#B_p - \#B_e}{\#B_e} (\%)$
	$\#B_e$	Runtime(sec.)	$\#B_p$	$\#Iter$	Runtime(sec.)	
T20-1	643	315	653	19	< 1	1.5
T20-2	836	181	852	12	< 1	1.9
T20-3	899	268	899	14	< 1	0.0
T20-4	607	2035	607	14	< 1	0.0
Average						0.8

TABLE III
EXPERIMENTAL RESULTS OF COMPARING TO GREEDY ALGORITHM

Tests	Statistics				Greedy Alg.		Proposed Alg.		$\frac{\#B_g - \#B_p}{\#B_g} (\%)$
	$ V $	$ E $	P	T	$\#B_g$	$\#B_p$	$\#Iter$		
T50	50	106	8	75	4974	3642	12		26.8
T100	100	192	8	150	7648	5524	17		27.8
T150	150	300	16	112	24903	12736	49		48.9
T200	200	418	16	150	33765	16210	23		52.0
Average									38.8

the stage makespan T . Otherwise they cannot be scheduled onto the same processor.

At every iteration, we first apply min-cost flow algorithm to determine the stages of each remaining node. Then we find the contractable edge e with the max rate and contract this edge. The execution time of the joint node is the sum of the execution times of the two original nodes. This process is repeated until no more edges can be contracted. If the number of remaining nodes are still greater than the number of available processors P , we call bin packing algorithm to combine nodes in the same stage together under the makespan constraint. If the resulted number of bins is greater than P , the algorithm claims no solution is found; otherwise it outputs the buffer size.

We generated another four test cases with larger graph size to compare our proposed algorithm with the greedy algorithm. The statistics of test cases and performance of algorithms are shown in Table III. It can be observed that our proposed algorithm achieves 38.8% less buffer usage on average compared to the greedy algorithm. Our proposed algorithm also reaches solution very fast on large graphs. It terminates within no more than 50 iterations on all test cases. For the largest test case T200, it finishes in 5.4 seconds.

VI. CONCLUSION

In this paper, we formulated the pipeline scheduling problem on acyclic synchronous dataflow graph and fixed number of processors to minimize the total buffer size while meeting the throughput constraint and designed an efficient two-level heuristic algorithm for the formulated problem to effectively improve buffer size. We compared our proposed algorithm to an enumeration algorithm which is able to generate optimal solution for small graphs, and a greedy algorithm which is able to run on large graphs. The experimental results show that our two-level heuristic algorithm achieves near-optimal solution compared to the enumeration algorithm, with only 0.8% increase in buffer size on average but with much shorter runtime, and compared to the greedy algorithm, our algorithm achieves 38.8% less buffer usage on average.

VII. ACKNOWLEDGEMENT

This work is supported by the NSF under CCF-0811270 and CCF-1115550.

REFERENCES

- [1] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow graph," *IEEE Trans. on Computers*, vol. 36, pp. 24–35, 1987.
- [2] J. Park and W. J. Dally, "Buffer-space efficient and deadlock-free scheduling of stream applications on multi-core architectures," *ACM Symposium on Parallelism in Algorithms and Architectures*, June 2010.
- [3] S. H. Bokhari, "Partitioning problems in parallel, pipeline, and distributed computing," *IEEE Trans. on Computer*, vol. 37, pp. 48–57, January 1988.
- [4] P. Hoang and J. M. Rabaey, "Scheduling of DSP programs onto multi-processors for maximum throughput," *IEEE Trans. on Signal Processing*, pp. 2225–2235, June 1993.
- [5] S. Bakshi and D. D. Gajski, "Partitioning and pipelining for performance-constrained hardware/software systems," *IEEE Trans. on VLSI Systems*, vol. 7, no. 4, pp. 419–432, December 1999.
- [6] T. Shin, H. Oh, and S. Ha, "Minimizing buffer requirements for throughput constrained parallel execution of synchronous dataflow graph," *IEEE Proc. ASPDAC*, pp. 165–170, 2011.
- [7] M. Geilen, T. Basten, and S. Stuijk, "Minimizing buffer requirements of synchronous dataflow graphs with model checking," *IEEE Proc. DAC*, pp. 819–824, 2005.
- [8] S. Stuijk, M. Geilen, and T. Basten, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graph," *IEEE Proc. DAC*, pp. 899–904, 2006.
- [9] M. Wiggers, M. Bekooij, P. Jansen, and G. Smit, "Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure," *CODES+ISSS*, pp. 10–15, 2006.
- [10] X. Hu, S. Bass, and R. Harber, "Minimizing the number of delay buffers in the synchronization of pipelined systems," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 12, pp. 1441–1449, December 1994.
- [11] J. Cong, G. Han, and W. Jiang, "Synthesis of an application-specific soft multiprocessor system," *Proceedings of International Symposium on Field Programmable Gate Arrays*, pp. 99–107, February 2007.
- [12] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, "Network flows," *Prentice Hall*, 1993.
- [13] S. Stuijk, M. Geilen, and T. Basten, "SDF3: SDF for free," *International Conference on Application of Concurrency to System Design*, pp. 276–278, June 2006.
- [14] R. M. Karp, "Dynamic programming meets the principle of inclusion and exclusion," *Operations Research Letters*, vol. 1, no. 2, pp. 49–51, April 1982.