Efficient Throughput Analysis of Synchronous Dataflow Graphs via Parametric Shortest Path

Zhengzheng Tian[1], Mingze Ma[2,*], Jian Hou[1]
[1]School of Computer Science and Technology, Zhejiang Sci-Tech University, Hangzhou, China
[2]School of Information Engineering, Wenzhou Business College, Wenzhou, China

***Abstract*—Synchronous Dataflow Graphs (SDFGs) are widely employed to model real-time embedded systems and streaming data processing, where throughput serves as a critical measure of computational efficiency. Parametric Shortest Path (PSP) algorithms offer an effective means of analyzing the optimal throughput of Homogeneous SDFGs (HSDFGs). However, applying PSP algorithms to general SDFGs typically requires a conversion to HSDFGs, which introduces additional overhead in graph transformation and may result in exponential growth in graph size. This paper proposes an extension to a traditional PSP algorithm, enabling direct throughput analysis of SDFGs without explicit conversion to HSDFGs. Furthermore, a graph size reduction technique is incorporated to further optimize the runtime of the proposed algorithm. Experimental results demonstrate that the proposed algorithm achieves, on average, a shorter runtime than three state-of-the-art algorithms. The advantage of the proposed algorithm scales with the size of the SDFG, achieving a speedup of up to 39.05x over the fastest of the three baseline algorithms.**

***Index Terms*—Synchronous Dataflow Graphs, Throughput Analysis, Parametric Shortest Path Algorithm**

I. Introduction

Synchronous Dataflow Graphs (SDFGs) [1] are widely adopted as computational models in real-time embedded systems and streaming data processing. Throughput is a significant performance metric for SDFGs, representing the number of iterations completed per unit time. During system design, throughput must be evaluated repeatedly across varying architectures and scheduling schemes. The speed of this analysis directly affects the efficiency of design-space exploration and optimization, making fast throughput analysis algorithms highly desirable.

In an SDFG, computational tasks are modeled as actors, and each execution of an actor produces and consumes data represented by tokens. These tokens are transferred between actors via first-in-first-out channels. Token production and consumption rates can differ across channels, which may cause an actor to execute multiple times within a single iteration.

An SDFG can be transformed into an equivalent Homogeneous SDFG (HSDFG), in which all token rates are uniform [2]. This transformation involves replicating actors according to their execution counts per iteration. Additionally, some channels may contain initial tokens at the start of each iteration. These initial tokens are essential for enabling deadlock-free execution in cyclic graphs, and the throughput of an SDFG is ultimately constrained by its critical cycles.

This work was supported by Zhejiang Provincial Natural Science Foundation of China under Grant LQ24F020010.

*This author is the corresponding author (mingze.ma@hotmail.com).

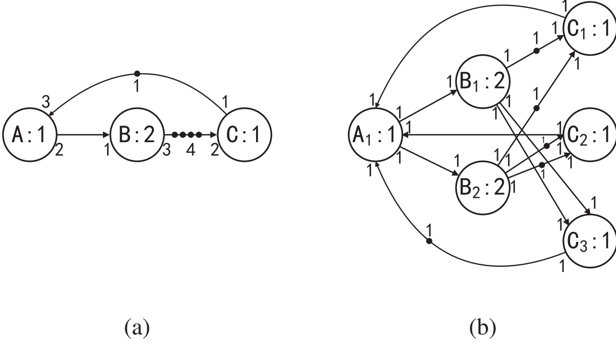
The optimal throughput of an HSDFG is the reciprocal of its Maximum Cycle Mean (MCM), defined as the highest ratio of actor execution time to the number of initial tokens across all cycles. Since the optimal throughput of an SDFG matches that of its equivalent HSDFG, analyzing the SDFG's throughput reduces to computing the MCM of its equivalent HSDFG.

The calculation of the MCM of an HSDFG is equivalent to a Parametric Shortest Path (PSP) problem. Many efficient algorithms have been proposed to solve the PSP problem [3]–[5]. [3] proposes an iterative refinement-based approach to approximate the minimum cycle mean. [4] introduces an approach that dynamically updates a shortest path tree with a priority queue to obtain the MCM. The time complexity of this algorithm is much lower than that of the algorithm proposed in [3]. The algorithm proposed by Young, Tarjant, and Orlin (YTO) [5] further enhances the approach proposed in [4] by employing a Fibonacci heap to optimize priority queue operations. The experimental evaluations in [6] show that YTO generally outperforms the other MCM analysis algorithms tested in [6]. However, an SDFG-to-HSDFG conversion is required if these algorithms are used to obtain the optimal throughput of SDFGs, which introduces substantial conversion overhead and may dramatically increase the graph size.

To mitigate the drawbacks introduced by the graph conversion, techniques have been developed to reduce the size of the converted HSDFG. The Linear Constraint Graph (LCG) method proposed in [7] constructs a reduced HSDFG that preserves optimal throughput by eliminating non-critical actors and channels. However, explicit SDFG-to-LCG conversion remains necessary.

Many approaches aim to compute throughput directly on SDFGs to circumvent the drawbacks of the graph conversion. The State-Space Exploration (SSE) method proposed in [8] models SDFG execution as state transitions and simulates the system until a periodic pattern emerges. The periodic execution period then determines the optimal throughput of an SDFG. While SSE is efficient for small graphs, its runtime can grow exponentially with graph size. The Kiter algorithm proposed in [9], [10] requires each actor to execute at a fixed period to reduce the complexity of the throughput analysis problem. It constructs a partial constraint graph to obtain the minimum iteration period under this constraint. The optimal throughput of an SDFG can be obtained by replicating each actor and iteratively increasing the number of replicas. However, if the actors in an SDFG require extensive replication to achieve

978-3-9826741-1-7/DATE26/© 2026 EDAA



(a) (b)

Fig. 1: (a) An example SDFG. (b) The equivalent HSDFG of (a).

optimal throughput, this algorithm becomes relatively time-consuming.

Building on the YTO algorithm, this paper proposes a novel PSP algorithm for SDFGs, termed PSP-SDFG, which achieves optimal throughput without requiring explicit graph conversion. The PSP-SDFG approach yields exact optimal throughput and demonstrates significantly faster runtime than existing methods. The key contributions of this paper are as follows.

- This paper extends the PSP algorithm from [5] to analyze SDFGs directly, eliminating the overhead of converting to HSDFGs.
- This paper incorporates a graph size reduction technique in [7] to minimize the size of the shortest path tree used in the proposed algorithm, which improves computational efficiency.
- This paper validates the proposed PSP-SDFG algorithm by experimental comparison with three state-of-the-art approaches [7]–[9].

The remainder of this paper is organized as follows. Section II introduces background concepts. Section III details the proposed algorithm. Section IV presents the experimental results, and Section V concludes the paper.

II. Preliminaries

A. Synchronous Dataflow Graph

Definition 1 (SDFG): An SDFG is a directed graph $G = (V, E)$ where V and E are finite sets of actors and channels, respectively. Each actor $v \in V$ is defined as $v = (t, E_{in}, E_{out})$, where $t(v) \in \mathbb{N}^0$ denotes the execution time of the actor, and $E_{in}(v)$ and $E_{out}(v)$ are the sets of input and output channels, respectively. Each channel $e = \langle u, v \rangle \in E$ is represented as $e = (u, v, prd, cns, d)$, where actors u and v are the source and destination actors of the channel, respectively, $prd(e)$ and $cns(e)$ specify the number of tokens produced and consumed per execution of u and v, respectively, and $d(e)$ denotes the number of initial tokens on the channel.

Fig. 1a illustrates an example SDFG, where each node corresponds to an actor, and each directed edge represents a channel. As indicated by the number inside each node, the execution time of actors A, B, and C is 1, 2, and 1 time units, respectively. The token production and consumption rates are labeled at the ends of each channel, and the initial tokens are shown as black dots. For example, on channel $\langle B, C \rangle$, there are 4 initial tokens, and each execution of actor C consumes 2 tokens while each execution of actor B produces 3 tokens.

Definition 2 (Repetition Vector): The repetition vector q of an SDFG is a positive integer vector that specifies the minimum number of execution required for each actor in one iteration. It is derived by solving the balance equations:

$$prd(e) \cdot q_u = cns(e) \cdot q_v, \forall e = \langle u, v \rangle \in E, \tag{1}$$

where q_u and q_v denote the execution counts of actors u and v in an iteration, respectively.

For the example SDFG in Fig. 1a, the repetition vector is $\{1, 2, 3\}$ for actors A, B, and C.

An SDFG can execute periodically only if it is sample-rate consistent and deadlock-free. Sample-rate consistency is ensured by the existence of a valid repetition vector, since the token distribution on an SDFG remains invariant after all actors execute according to their repetition counts in an iteration. Since SDFGs may contain cycles, insufficient initial tokens can lead to deadlocks due to cyclic dependencies. While these properties can be efficiently verified using the methods described in [2], this paper considers only SDFGs that are sample-rate consistent and deadlock-free.

A graph is strongly connected if there exists a bidirectional path between every pair of actors, i.e., for any two actors u and v in a graph, there are paths from u to v and from v to u. A non-strongly-connected SDFG can be decomposed into Strongly Connected Components (SCCs), and the overall throughput of the SDFG is determined by the minimum throughput among all its SCCs. Therefore, this paper focuses on throughput analysis for strongly connected graphs only.

Definition 3 (HSDFG): An HSDFG is an SDFG graph $G_H = (V_H, E_H)$ in which $prd(e) = cns(e) = 1, \forall e \in E_H$.

All SDFGs can be transformed into their equivalent HSDFGs by replicating actors according to their repetition vectors and adding channels based on token dependencies between the replicated actors [2]. For example, the repetition vector of the SDFG in Fig. 1a is $\{1, 2, 3\}$ for actors A, B and C. Thus, in its equivalent HSDFG as shown in Fig. 1b, the actors A, B and C are replicated into one, two, and three actors, respectively, representing multiple execution instances of these actors within a single iteration. The eleven channels in Fig. 1b reflect the token dependencies between these instances. The number of actors and channels in the HSDFG in Fig. 1b is approximately 2 and 3.67 times greater than in the original SDFG, respectively.

Definition 4 (Maximum Cycle Mean): In an HSDFG, the cycle mean of a cycle is defined as the total execution time of all actors in the cycle divided by the total number of tokens on its channels. The MCM is the highest cycle mean among all cycles in the graph. Formally:

$$MCM = \max_{\forall C \in G_H} \frac{\sum_{v \in C} t(v)}{\sum_{e \in C} d(e)}, \tag{2}$$

where C denotes a cycle in the HSDFG G_H.

The optimal throughput of an HSDFG is given by $\lambda_H = \frac{1}{MCM}$ [2]. Since the throughput of an SDFG equals that of its equivalent HSDFG, it follows that $\lambda = \lambda_H$. For example, in the HSDFG shown in Fig. 1b, one of the critical cycles is $A_1 \rightarrow B_2 \rightarrow C_2 \rightarrow A_1$, with a cycle mean of 4/1. Thus, the optimal throughput of both the HSDFG and its corresponding SDFG in Fig. 1a is 0.25.

B. Linear Constraint Graph

To obtain the optimal throughput of an SDFG more efficiently, this paper adopts a simplified HSDFG model known as the LCG and its subgraph, as proposed in [7]. An LCG resembles an HSDFG in that actors are replicated according to the repetition vector, while only channels that may constitute critical cycles are retained.

Definition 5 (LCG [7]): An LCG $G_{LCG} = (V_{LCG}, E_{LCG})$ of an SDFG $G = (V, E)$ is a reduced HSDFG. Each actor $v \in V$ is expanded into q_v actors v_i $(i = 1, 2, \ldots, q_v)$ in V_{LCG}, where v_i represents the i-th execution of the actor v in one iteration. Each channel $e = \langle u, v\rangle \in E$ is expanded into q_v channels $e_j = \langle\ u_k,\ v_j\ \rangle$ in E_{LCG}, where $j = 1, 2, \ldots, q_v$ and k is determined by

$$k = \left(\left\lceil \frac{j \cdot cns(e) - d(e)}{prd(e)} \right\rceil - 1\right) \bmod q_u + 1. \tag{3}$$

The initial token count on each channel e_j is computed as

$$d(e_j) = -\left\lfloor \frac{\left\lceil \frac{j \cdot cns(e) - d(e)}{prd(e)} \right\rceil - 1}{q_u} \right\rfloor. \tag{4}$$

Fig. 2a shows the LCG derived from the SDFG in Fig. 1a. Similar to the HSDFG in Fig. 1b, actors are replicated based on the repetition vector in the LCG. However, the LCG has significantly fewer channels than the HSDFG (6 compared to 11). Meanwhile, the critical cycle $A_1 \rightarrow B_2 \rightarrow C_2 \rightarrow A_1$ is preserved in the LCG, resulting in an optimal throughput of 0.25, which is identical to that of the original SDFG.

A Critical Induced Subgraph (CIS) of an LCG consists of an arbitrary actor in the LCG and all the actors that can reach the actor in the LCG, along with their connecting channels. As proven in [7], a CIS of an LCG preserves the critical cycles of the LCG. Fig. 2b shows a CIS of the LCG in Fig. 2a, which consists of A_1 and all the paths in the LCG that can reach A_1. In this CIS, only the critical cycle $A_1 \rightarrow B_2 \rightarrow C_2 \rightarrow A_1$ is retained. Compared to the HSDFG in Fig. 1b, the CIS reduces the number of actors by 50% (from 6 to 3) and channels by 72.73% (from 11 to 3).

III. ALGORITHM

A. Overview

Traditionally, PSP algorithms are applicable only to HSDFGs. In this section, a novel optimal throughput analysis algorithm, PSP-SDFG, is presented, which extends the PSP-based YTO algorithm [5] to analyze SDFGs directly. The original YTO algorithm constructs a parameterized shortest path tree by introducing a virtual root node connected to all

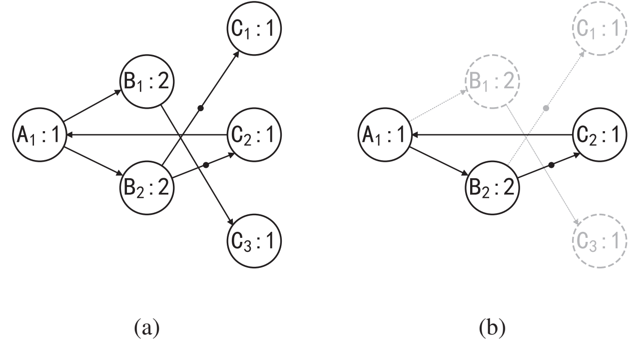


Fig. 2: (a) The LCG derived from the SDFG in Fig. 1a. (b) A CIS of the LCG in (a).

the nodes in a graph. This step is necessary for non-connected graphs to unify disconnected components into a single tree. However, because the target graphs in this paper are connected, this virtual root node is unnecessary. Instead, PSP-SDFG treats each actor as the root of a tree and construct a Parametric Shortest Path Forest (PSPF).

To reduce the computational complexity of the proposed algorithm, PSP-SDFG integrates the graph size reduction technique LCG and its CIS. PSP-SDFG constructs the PSPF for the CIS of the LCG of an SDFG directly from the SDFG. An ordered sequence, denoted as $treeNodes$, is used to store all the tree nodes in the PSPF. Each tree node corresponds to an actor in the LCG. The replicated actors in the LCG that correspond to a single actor in the original SDFG are stored contiguously in $treeNodes$. For example, the sequence of the tree nodes in $treeNodes$ for the SDFG in Fig. 1a is $\{A_1, B_1, B_2, C_1, C_2, C_3\}$. Although $treeNodes$ has the same size as the number of actors in an LCG, only the tree nodes corresponding to the actors of a CIS of the LCG are actually accessed during the throughput analysis, as are the tree edges connecting these nodes. Consequently, the runtime of the proposed approach is determined mainly by the scale of the LCGs CIS rather than the LCG itself.

Each tree node in $treeNodes$ is denoted as $n = (tokens, time, v_{SDFG}, E_{in}, E_{out})$, where $tokens(n)$ is the total number of tokens along the path from n to its root in the PSPF, $time(n)$ is the cumulative execution time along that path, $v_{SDFG}(n)$ is the corresponding actor of the tree node in the original SDFG, and $E_{in}(n)$ and $E_{out}(n)$ are the sets of input and output tree edges of node n, respectively.

Tree edges connecting the tree nodes in the PSPF correspond to the channels in the CIS. These edges may be activated or deactivated during PSPF updates. A tree node may have multiple incoming tree edges, yet at most one of these edges can be activated at the same time. Only the activated tree edges are viewed as part of the PSPF. Therefore, if a tree edge is deactivated, the edge is removed from the PSPF. Likewise, if a tree edge is activated, the edge is added to the PSPF. Each directed tree edge $\epsilon = \langle n, m\rangle$ is represented as $(n, m, d, t, key, inTree)$, where n and m are the source and

destination nodes of the edge, respectively, $d(\epsilon)$ is the initial token count on the corresponding CIS channel of the edge, $t(\epsilon)$ is the execution time of the source actor of the corresponding CIS channel, $key(\epsilon)$ denotes the weight of the edge, and the Boolean flag $inTree(\epsilon)$ indicates whether the edge is activated.

In addition, an auxiliary array, $startPos$, is used, where each element $startPos_u$ in the array records the index of the first node in $treeNodes$ corresponding to the actor u in the original SDFG. For example, for the actors A, B, and C in the SDFG in Fig. 1a, if $treeNodes = \{A_1, B_1, B_2, C_1, C_2, C_3\}$, then $startPos_A$, $startPos_B$, and $startPos_C$ are 0, 1, and 3, respectively.

B. PSP-SDFG

To identify a CIS of the LCG on the original SDFG, PSP-SDFG first locates an actor that participates in a cycle of the LCG using a variant of depth-first search. Next, PSP-SDFG initializes all the nodes in $treeNodes$ that correspond to the CIS starting from this actor via a breadth-first search. These nodes, along with the edges connecting them, are used to construct the PSPF. Finally, PSP-SDFG applies the PSP update procedure from the original YTO algorithm to iteratively refine the PSPF until the optimal throughput is achieved.

The proposed PSP-SDFG algorithm is presented in Algorithm 1. Given an input SDFG $G = (V, E)$, the algorithm computes its optimal throughput λ. In lines 1-6, the repetition vector q is obtained for the input SDFG. Then, the auxiliary array $startPos$ is initialized to record the indexes of the first nodes in $treeNodes$ for the actors of the input SDFG. In line 8, $cycleIndex$, denoting the index of the tree node corresponding to an actor that belongs to a cycle in the LCG, is identified by the procedure elaborated in Algorithm 2. After that, in line 9, all tree nodes and edges in the PSPF are initialized by the procedure presented in Algorithm 3. At this point, all tree nodes and edges corresponding to a CIS of the LCG of the input SDFG have been initialized. Each tree node initially forms a singleton tree with itself as the root, since all incoming edges of each tree nodes are inactivated.

A min-heap H is created in line 10 to assist in updating the activated incoming tree edge of each tree node. Then, for each node, the incoming edge with the minimum key value is inserted into H in line 11. The iterative process in lines 12-29 follows the structure of the traditional YTO algorithm. In line 13, the edge with the smallest key ϵ_{min} is extracted from the heap H. In lines 14-17, if adding ϵ_{min} to the PSPF would create a cycle, the loop terminates and the key value is returned as the optimal throughput. In line 18, ϵ_{min} is added to the PSPF, replacing any existing edge pointing to the same destination node. In lines 19-27, the values of $tokens$ and $time$ of the destination node of ϵ_{min}, along with all the nodes in its subtree, are updated. The key values of the incoming and outgoing edges of these nodes are also updated. Then, in Line 28, the heap H is updated according to the key values of the incoming edges of these nodes. Through the iterations of the while loop, the individual trees in the PSPF are progressively divided and merged until a cycle is detected, resulting in the optimal throughput λ.

Algorithm 1 PSP-SDFG

Require: A strongly-connected SDFG $G = (V, E)$.
Ensure: The optimal throughput λ of G.
1: Obtain the repetition vector q for the SDFG G
2: $pos \leftarrow 0$
3: **for** each actor v in V **do**
4: $\quad startPos_v \leftarrow pos$
5: $\quad pos \leftarrow pos + q_v$
6: **end for**
7: Create $treeNodes$ and set v_{SDFG} for each nodes in $treeNodes$
8: $cycleIndex \leftarrow$ findCycleIndex$(G, treeNodes, q, startPos)$
9: Call $initTreeNodes(G, treeNodes, q, startPos, cycleIndex)$
10: Create a min-heap H for tree edges
11: Find the incoming edge with the minimum key for each node and insert the edge into the min-heap H
12: **while** true **do**
13: $\quad$ Take the top element of the H as $\epsilon_{min} = \langle n, m \rangle$
14: $\quad$ **if** $n = m$ **or** m is an ancestor of n in the current tree **then**
15: $\quad\quad \lambda \leftarrow key(\epsilon_{min})$
16: $\quad\quad$ **break**
17: $\quad$ **end if**
18: $\quad$ Remove the tree edge pointing to m from the PSPF and add ϵ_{min} to the PSPF.
19: $\quad$ Update the $tokens$ and $time$ for the node m and all the nodes in its subtree.
20: $\quad$ **for** each m' in (m and its subtree) **do**
21: $\quad\quad$ **for all** $\epsilon' = \langle n', m' \rangle \in E_{in}(m')$ **do**
22: $\quad\quad\quad key(\epsilon') \leftarrow \frac{tokens(n')+d(\epsilon')-tokens(m')}{time(n')+t(\epsilon')-time(m')}$ if $time(n') + t(\epsilon') > time(m')$ else $+\infty$
23: $\quad\quad$ **end for**
24: $\quad\quad$ **for all** $\epsilon' = \langle m', n' \rangle \in E_{out}(m')$ **do**
25: $\quad\quad\quad key(\epsilon') \leftarrow \frac{tokens(m')+d(\epsilon')-tokens(n')}{time(m')+t(\epsilon')-time(n')}$ if $time(m') + t(\epsilon') > time(n')$ else $+\infty$
26: $\quad\quad$ **end for**
27: $\quad$ **end for**
28: $\quad$ Get the incoming edges which are not in the PSPF with the minimum key for m and the nodes in its subtree, and add these edges to H and remove the original edges with the same destination nodes from H.
29: **end while**
30: return λ

C. Finding an Actor in a Cycle of an LCG

A CIS of an LCG can be obtained by a traversal from any actor in the LCG according to [7]. However, there may exist actors not in any cycle in the LCG. If such an actor is used as the starting actor to find a CIS, a path consists of noncritical actors and channels will become part of the obtained CIS. For example, if a CIS is constructed starting from the actor B_1 in the LCG in Fig. 2a, a noncritical actor B_1 and a noncritical channel $\langle A_1, B_1 \rangle$ will be included in the obtained CIS. Therefore, Algorithm 2 finds an actor in a cycle of the LCG as the starting actor for finding a CIS to avoid such unwanted inclusions. According to [7], each actor in an LCG must have at least one input channel, and a cycle in an LCG must be found starting from any actor in the LCG. Based on these properties, Algorithm 2 identifies a node corresponding to an actor in a cycle in the LCG by performing a variant of depth-first traversal over predecessor nodes. It leverages the construction rule for LCG channels in Eq. (3) to trace back from a destination actor in the LCG to its source. Algorithm 2 begins

Algorithm 2 findCycleIndex

```
1: function FINDCYCLEINDEX(G, treeNodes, q, startPos)
2:     curIndex ← 0
3:     Mark all the nodes in treeNodes as unvisited
4:     while true do
5:         v_c ← v_SDFG(treeNodes[curIndex])
6:         Get an arbitrary edge e = (u, v_c) ∈ E_in(v_c)
7:         j ← curIndex − startPos_{v_c} + 1
8:         i ← (⌈(j×cns(e)−d(e))/prd(e)⌉ − 1) mod q_u
9:         preCycleIndex ← startPos_u + i − 1
10:        if treeNodes[preCycleIndex] has been visited then
11:            return preCycleIndex
12:        else
13:            Set treeNodes[preCycleIndex] as visited
14:            curIndex ← preCycleIndex
15:        end if
16:    end while
17: end function
```

Algorithm 3 initTreeNodes

```
1: function INITTREENODES(G,treeNodes,q,startPos,cycleIndex)
2:     Initialize tokens and time for each nodes in treeNodes as
       0
3:     Initialize a queue queue and push cycleIndex to it
4:     Mark all the nodes in treeNodes as unvisited
5:     while queue is not empty do
6:         curIndex ← queue.pop()
7:         n ← treeNodes[curIndex]
8:         if n is not visited then
9:             Mark n as visited
10:            v_c ← v_SDFG(treeNodes[curIndex])
11:            for all e = ⟨u, v_c⟩ ∈ E_in(v_c) do
12:                j ← curIndex − startPos_{v_c} + 1
13:                temp = ⌈(j×cns(e)−d(e))/prd(e)⌉
14:                i ← (temp − 1) mod q_u + 1
15:                m ← treeNodes[startPos_u + i − 1]
16:                Let ε_{i,j} = ⟨m, n⟩
17:                d(ε_{i,j}) ← −⌊(temp−1)/q_u⌋
18:                t(ε_{i,j}) ← t(u)
19:                key(ε_{i,j}) ← d(ε_{i,j})/t(ε_{i,j}) if t(ε_{i,j}) > 0 else +∞
20:                inTree(ε_{i,j}) ← 0
21:                Add ε_{i,j} to both E_in(n) and E_out(m)
22:                if m has never enqueued then
23:                    push the index of m to queue
24:                end if
25:            end for
26:        end if
27:    end while
28: end function
```

at the first node in $treeNodes$ (line 2) and iteratively explores the predecessors (lines 5-15). When a previously visited node is encountered (Line 10), a cycle is detected and its index is returned. Otherwise, traversal continues.

D. Initializing PSPF

Algorithm 3 identifies a CIS in the LCG through a breadth-first search and initializes all the tree nodes and edges in the PSPF based on their corresponding actors and channels in the CIS. The traversal begins from the actor corresponding to the node found in Algorithm 2 and proceeds toward the source actors of its input channels in the LCG. The index of the node corresponding to the source actor of an incoming channel of the current actor is obtained in lines 12-14, according to Eq. (3). A tree edge connecting the nodes m (i.e. the i-th node of the source actor) and n (i.e. the j-th node of the current actor) is denoted as $\epsilon_{i,j}$. The initial token count $d(\epsilon_{i,j})$ is calculated in line 17 using Eq. (4). Then, $t(\epsilon_{i,j})$ is set as the execution time of the source actor in line 18. After that, the weight of $\epsilon_{i,j}$ is initialized according to $d(\epsilon_{i,j})$ and $t(\epsilon_{i,j})$ in line 19. All tree edges are initialized as inactivated by setting their $inTree$ attributes to 0 in line 20. Finally, $\epsilon_{i,j}$ is added to both the outgoing edges' set of node m and the incoming edges' set of node n in line 21.

According to the original YTO algorithm in [5], the time complexity of the while loop in Algorithm 1 is $O(|V_{LCG}||E_{LCG}| + |V_{LCG}|^2 log|V_{LCG}|)$ for the LCG of an SDFG, where the first term accounts for edge updates and subtree traversals, and the second term corresponds to heap operations during shortest path updates. The time complexity of Algorithm 2 is $O(|V_{LCG}|)$ since it may traverse all the actors in the LCG to locate a node in a cycle in the worst case. The time complexity of Algorithm 3 is $O(|V_{LCG}| + |E_{LCG}|)$ since it performs a breadth-first search on the LCG. Overall, the time complexity of PSP-SDFG is $O(|V_{LCG}||E_{LCG}| + |V_{LCG}|^2 log|V_{LCG}|)$.

IV. Experimental Evaluation

TABLE I: Graph information of synthetic SDFGs.

Group	Avg. $\|V\|$	Avg. $\|E\|$	Avg. $\|D\|$	Avg. SRV
Group1	9.04	22.13	318.81	87.95
Group2	29.46	77.47	1754.72	291.33
Group3	48.71	128.23	4053.07	640.06
Group4	99.05	268.23	7363.63	986.15
Group5	198.81	548.71	10082.79	1490.3
Group6	498.13	1376.63	26732.8	4992.85

The proposed PSP-SDFG algorithm was implemented in SDF3 [11], a well-known analysis tool for SDFGs. Three state-of-the-art approaches were adopted as baselines, including SSE [8], Kiter [9], and the traditional YTO algorithm [5] combined with the SDFG-to-LCG conversion approach [7] (LCG+YTO). The evaluation involved testing on 600 synthetic SDFGs generated by SDF3, as well as 12 real-world application SDFGs from [12]. The synthetic SDFGs were generated with a configuration in which the average degree of actors was 3, the average token production/consumption rates were 3, and the sum of the repetition vector was set to a value between 7.5 and 14 times the number of actors. The synthetic SDFGs were divided into six groups. Each group contains SDFGs with about 10, 30, 50, 100, 200, or 500 actors. The graph information of the synthetic SDFGs is shown in Table I, where $|D|$ is the number of initial tokens in an SDFG, and SRV refers to the sum of the elements in the repetition vector of an SDFG.

Table II presents the experimental results of SSE, LCG+YTO, Kiter, and PSP-SDFG for the real-world applications in terms of throughput and runtime. According to Table II, for all realistic applications, all four approaches achieve the

TABLE II: Comparing SSE, LCG+YTO, Kiter, and PSP-SDFG on realistic applications in terms of throughput and runtime.

Application	$\|V\|$	$\|E\|$	$\|D\|$	SRV	Throughput				Runtime(ms)			
					SSE	LCG+YTO	Kiter	PSP-SDFG	SSE	LCG+YTO	Kiter	PSP-SDFG
Vocoder	116	150	1	406	4.646	4.646	4.646	4.646	8.426	2.172	1.681	0.089
TDE	31	31	1	5029	1.421	1.421	1.421	1.421	0.739	73.051	11.944	0.077
Serpent	122	131	1	6357	1.675	1.675	1.675	1.675	12.501	134.401	29.441	0.228
MPEGdecoder	25	29	1	2179	2.103	2.103	2.103	2.103	0.337	17.554	5.348	0.054
FMRadio	45	56	3	49	9.560×10^{-4}	9.560×10^{-4}	9.560×10^{-4}	9.560×10^{-4}	0.258	0.329	0.233	0.034
FilterBank	87	102	1	514	2.888	2.888	2.888	2.888	2.310	3.434	2.584	0.115
DES	55	63	1	993	7.387	7.387	7.387	7.387	1.386	6.412	2.577	0.055
FFT	19	19	1	1408	8.071	8.071	8.071	8.071	0.183	10.410	2.631	0.033
DCT	10	10	1	1060	8.219	8.219	8.219	8.219	0.143	5.727	2.118	0.043
ChannelVocoder	57	73	1	1837	9.368	9.368	9.368	9.368	0.596	13.435	7.609	0.125
BitonicSort	42	49	3	66	5.190×10^{-3}	5.190×10^{-3}	5.190×10^{-3}	5.190×10^{-3}	0.311	0.269	0.383	0.036
BeamFormer	58	72	2	108	1.362×10^{-4}	1.362×10^{-4}	1.362×10^{-4}	1.362×10^{-4}	0.485	0.418	1.121	0.037
Geometric Mean									0.780	4.999	2.586	0.064

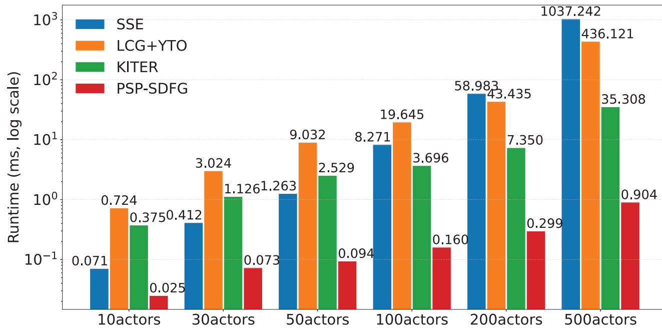


Fig. 3: Average runtime comparison for SSE, LCG+YTO, Kiter, and PSP-SDFG on six groups of synthetic SDFGs.

same throughput, and the proposed PSP-SDFG achieves the minimum runtime among them. The proposed algorithm is about 12.19 times faster than SSE, the second-fastest approach, in the geometric mean of the runtime. The runtime of the state-transition-based SSE algorithm is mainly influenced by the number of actors in an SDFG and less affected by the SRV. Therefore, for SDFGs with relatively few actors and/or a large ratio of SRV and $|V|$, such as TDE, Serpent, MPEGdecoder, FilterBank, DES, FFT, DCT, and ChannelVocoder, the runtime of SSE is shorter than that of the other two baseline algorithms. In contrast, the time complexity of LCG+YTO and Kiter is determined by the SRV, resulting in a relatively shorter runtime when the SRV is smaller, as observed in cases like Vocoder, FMRadio, BitonicSort, and BeamFormer. Although the runtime of PSP-SDFG is also influenced by SRV, in the less favorable cases in Table II, PSP-SDFG remains at least 3.34 times faster than SSE.

Fig. 3 shows the runtime of the four approaches for the synthetic graphs. The results show that the proposed algorithm achieves the shortest average runtime across all the tested SDFG groups. The proposed algorithm achieves an average of 2.84-39.05 times' speedup over the second-fastest algorithm across all the tested SDFG groups. The speed advantage of the proposed algorithm over the other three approaches grows progressively larger as the number of actors increases. For SDFGs with no more than 50 actors, SSE is the second-fastest approach. When the number of actors reaches about 100, the Kiter algorithm becomes the second-fastest approach. For graphs with about 200 actors, SSE demonstrates the worst runtime among all evaluated approaches. This matches the phenomenon observed in the testing on the realistic SDFGs. Since the time complexity of SSE is exponential with the size of the SDFG, the scalability of this approach is worse than the other three approaches, which have polynomial time complexity. For the LCG+YTO approach, the graph conversion process accounts for a significant proportion of the total runtime. As the graph size increases, this conversion overhead becomes increasingly prominent, since the time complexity of the conversion is proportional to the size of the LCG. By avoiding explicit SDFG-to-LCG conversion, the proposed algorithm eliminates the associated conversion overhead. Furthermore, PSP-SDFG only accesses the tree nodes and edges corresponding to the actors and channels within the CIS of an LCG. For the synthetic SDFGs tested in the experiments, the number of actors and channels in their associated LCGs is, on average, 7.5 to 13.1 times greater than that in the CISs of the corresponding LCGs. Consequently, the runtime of LCG+YTO is significantly longer than that of PSP-SDFG.

V. Conclusion And Future Work

This paper proposes a novel PSP algorithm, called PSP-SDFG, to achieve the optimal throughput for SDFGs. The traditional YTO algorithm is extended to construct a PSPF directly from an SDFG. The LCG model is integrated to restrict the range of traversal on an SDFG, reducing the size of the PSPF in the proposed algorithm. The experimental results validate the effectiveness of the proposed algorithm by comparing it with three state-of-the-art approaches. For realistic applications, the proposed algorithm is on average about 12.19 times faster than the second-fastest approach among all the approaches tested in the experiment. For synthetic SDFGs, the proposed algorithm achieves the minimum average runtime across all the tested SDFG groups among the four approaches, reaching up to 39.05 times' speedup than the fastest of the three competing algorithms. Future directions may include extending the proposed approach to cyclo-static dataflow models, incorporating scheduling into the throughput analysis process, etc.

References

[1] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[2] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, 2nd ed. CRC Press, 2009.

[3] R. A. Howard, *Dynamic Programming and Markov Processes*. The M.I.T. Press, 1960.

[4] R. M. Karp and J. B. Orlin, “Parametric shortest path algorithms with an application to cyclic staffing,” *Discrete Applied Mathematics*, vol. 3, no. 1, pp. 37–45, 1981.

[5] N. E. Young, R. E. Tarjant, and J. B. Orlin, “Faster parametric shortest path and minimum-balance algorithms,” *Networks*, vol. 21, no. 2, pp. 205–221, 1991.

[6] A. Dasdan, “Experimental analysis of the fastest optimum cycle ratio and mean algorithms,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 9, no. 4, pp. 385–418, 2004.

[7] R. de Groote, J. Kuper, H. Broersma, and G. J. Smit, “Max-plus algebraic throughput analysis of synchronous dataflow graphs,” in *Proceedings of Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2012.

[8] A. H. Ghamarian, M. C. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. Moonen, and M. J. Bekooij, “Throughput analysis of synchronous data flow graphs,” in *Proceedings of International Conference on Application of Concurrency to System Design (ACSD)*, 2006.

[9] B. Bodin and A. M. Kordon, “Evaluation of the exact throughput of a synchronous dataflow graph,” *Journal of Signal Processing Systems*, vol. 93, no. 9, pp. 1007–1026, 2021. [Online]. Available: https://github.com/bbodin/kiter

[10] B. Bodin, A. Munier-Kordon, and B. D. De Dinechin, “Optimal and fast throughput evaluation of CSDF,” in *Proceedings of Design Automation Conference (DAC)*, 2016, pp. 1–6.

[11] S. Stuijk, M. Geilen, and T. Basten, “SDF3: SDF for free,” in *Proceedings of International Conference on Application of Concurrency to System Design (ACSD)*, 2006. [Online]. Available: http://www.es.ele.tue.nl/sdf3

[12] M. Ma, J. Hou, D. Xiang, W. Lin, and Z. Ding, “Efficient pipelining of synchronous dataflow graphs via graph conversion,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 43, no. 6, pp. 1704–1714, 2024.