Efficient Simulation of Critical Synchronous Dataflow Graphs

CHIA-JUI HSU, MING-YUNG KO, and SHUVRA S. BHATTACHARYYA
University of Maryland
and
SUREN RAMASUBBU and JOSÉ LUIS PINO
Agilent Technologies

System-level modeling, simulation, and synthesis using electronic design automation (EDA) tools are key steps in the design process for communication and signal processing systems, and the synchronous dataflow (SDF) model of computation is widely used in EDA tools for these purposes. Behavioral representations of modern wireless communication systems typically result in *critical SDF graphs*: These consist of hundreds of components (or more) and involve complex intercomponent connections with highly *multirate* relationships (i.e., with large variations in average rates of data transfer or component execution across different subsystems). Simulating such systems using conventional SDF scheduling techniques generally leads to unacceptable simulation time and memory requirements on modern workstations and high-end PCs. In this article, we present a novel *simulation-oriented scheduler* (SOS) that strategically integrates several techniques for graph decomposition and SDF scheduling to provide effective, joint minimization of time and memory requirements for simulating critical SDF graphs. We have implemented SOS in the *advanced design system* (ADS) from Agilent Technologies. Our results from this implementation demonstrate large improvements in simulating real-world, large-scale, and highly multirate wireless communication systems (e.g., 3GPP, Bluetooth, 802.16e, CDMA 2000, XM radio, EDGE, and Digital TV).

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms: Algorithms, Design

Additional Key Words and Phrases: Scheduling, simulation, synchronous dataflow

ACM Reference Format:

Hsu, C.-J., Ko, M.-Y., Bhattacharyya, S. S., Ramasubbu, S., and Pino, J. L. 2007. Efficient simulation of critical synchronous dataflow graphs. ACM Trans. Des. Autom. Electron. Syst. 12, 3, Article 21 (August 2007), 28 pages. DOI = 10.1145/1255456.1255458 http://doi.acm.org/10.1145/1255456.1255458

Authors' addresses: C.-J. Hsu, M.-Y. Ko, and S. S. Bhattacharyya, Department of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742; email: {jerryhsu,myko,ssb}@umd.edu; S. Ramasubbu and J. L. Pino, Agilent Technologies, Inc., Palo Alto, CA 94306; email: {suren_ramasubbu,jpino}@agilent.com.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2007 ACM 1084-4309/2007/08-ART21 $5.00 DOI 10.1145/1255456.1255458 http://doi.acm.org/10.1145/1255456.1255458

1. INTRODUCTION

Modeling systems using synchronous dataflow (SDF) [Lee and Messerschmitt 1987], or closely related models such as cyclo-static dataflow [Bilsen et al. 1996], is widespread in EDA tools for designing communication and signal processing systems. A variety of commercial and research-oriented tools incorporate SDF semantics, including ADS from Agilent [Pino and Kalbasi 1998], the Autocoding Toolset from MCCI [Robbins 2002], CoCentric System Studio from Synopsis [Buck and Vaidyanathan 2000], Compaan from Leiden University [Stefanov et al. 2004], Gedae from Gedae Inc., Grape from K. U. Leuven [Lauwereins et al. 1995], LabVIEW from National Instruments, PeaCE from Seoul National University [Sung et al. 1997], and Ptolemy II from U. C. Berkeley [Eker et al. 2003]. These tools provide graphical environments, libraries of functional modules, and capabilities of simulation and synthesis.

In the dataflow modeling paradigm, the computational behavior of a system is represented as a directed graph $G = (V, E)$. A vertex (*actor*) $v \in V$ represents a computational module or hierarchically nested subgraph. A directed edge $e \in E$ represents a FIFO buffer from its source actor $src(e)$ to its sink actor $snk(e)$, and imposes precedence constraints for proper scheduling of the dataflow graph. An edge e can have a nonnegative integer *delay* $del(e)$ associated with it. This delay value specifies the number of initial data values (*tokens*) that are buffered on the edge before the graph starts execution.

Dataflow graphs operate based on *data-driven* execution: An actor v can execute (*fire*) only when it has a sufficient number of data values (tokens) on all of its input edges $in(v)$. When firing, v consumes a certain number of tokens from its input edges, executes its computation, and produces a certain number of tokens on its output edges $out(v)$. In SDF, the number of tokens produced onto (consumed from) e by a firing of $src(e)$ ($snk(e)$) is restricted to be a constant positive integer that must be known at compile time; this integer is referred to as the *production rate* (*consumption rate*) of e and denoted as $prd(e)$ ($cns(e)$).

Before execution, a *schedule* (in our context, a single-processor schedule) of a dataflow graph is computed. Here, by a schedule, we mean a sequence of actor firings, or more generally, any static or dynamic sequencing mechanism for executing actors. An SDF graph $G = (V, E)$ has a valid schedule (is *consistent*) if it is free from deadlock and is sample rate consistent, that is, it has a *periodic schedule* that fires each actor at least once and produces no net change in the number of tokens on each edge [Lee and Messerschmitt 1987]. In more precise terms, G is *sample rate consistent* if there is a positive integer solution to the *balance equations*

$$\forall e \in E,\ prd(e) \times \boldsymbol{x}[src(e)] = cns(e) \times \boldsymbol{x}[snk(e)]. \quad (1)$$

When it exists, the minimum positive integer solution for the vector $\boldsymbol{x}$ is called the *repetitions vector* of G, and denoted by $\boldsymbol{q}_G$. For each actor v, $\boldsymbol{q}_G[v]$ is referred to as the *repetition count* of v. A *valid minimal periodic schedule* (which is abbreviated as *schedule* hereafter in this article) is then a sequence of actor firings in which each actor v is fired $\boldsymbol{q}_G[v]$ times, and the firing sequence obeys the data-driven properties imposed by the SDF graph.

Once a schedule is determined, the buffer sizes of dataflow edges can be computed either statically or dynamically for allocating memory space to the buffers that correspond to graph edges. Given a schedule S, we define the *buffer size* required for an edge e, namely, $buf(e)$, to be the maximum number of tokens simultaneously queued on e during an execution of S, and the *total buffer requirement* of an SDF graph $G = (V, E)$ to be the sum of buffer sizes of all edges.

$$buf(G) = \sum_{\forall e \in E} buf(e) \quad (2)$$

Generally, the design space of SDF schedules is highly complex, and the schedule has a large impact on the performance and memory requirements of an implementation [Bhattacharyya et al. 1996]. For synthesis of embedded hardware/software implementations, memory requirements (including memory requirements for buffers and for program code) are often of critical concern, while tolerance for compile time is relatively high [Marwedel and Goossens 1995], so high-complexity algorithms can often be used. On the other hand, for system simulation, simulation time (including time for scheduling and execution) is the primary metric, while memory usage (including memory for buffering and for the schedule) must only be managed so as to fit the available memory resources.

Scheduling in the former context (i.e., embedded hardware/software implementation) has been addressed extensively in the literature. In this article, we focus on the latter context (simulation), which is relatively unexplored in any explicit sense. Our target simulation platforms are single-processor machines including workstations and desktop PCs, which are widely used to host system-level simulation tools. The large-scale and highly multirate nature of today's wireless communication applications is our driving application motivation: For satisfactory simulation, the wireless communication domain requires SDF scheduling techniques that are explicitly and effectively geared towards simulation performance as the primary objective.

In particular, we present the *simulation-oriented scheduler* (SOS) for efficient simulation of large-scale, highly multirate synchronous dataflow graphs. The organization of the article is as follows: In Section 2, we discuss problems that arise from simulating modern wireless communication systems. We review related work in Section 3. We then introduce our SOS framework in Section 4. In Section 5, we present simulation results for various modern wireless communication designs. We conclude in Section 6 and provide a glossary at the end of the article.

2. PROBLEM DESCRIPTION

Real-world communication and signal processing systems involve complicated physical behaviors, and their behavioral representations may involve hundreds of coarse-grain components that are interconnected in complex topologies, and have heavily multirate characteristics. For example, simulating wireless communication systems involves complex encoder/decoder schemes, modulation/demodulation structures, communication channels, noise, and interference

signals. In transmitters, data is converted progressively across representation formats involving bits, symbols, frames, and RF signals. The corresponding conversions are then performed in reverse order at the receiver end. These transmitter-receiver interactions and the data conversions are often highly multirate. In addition, simulating communication channels may involve various bandwidths, noise, and multiple interference signals that may originate from different wireless standards. All of these considerations introduce heavily multirate characteristics across the overall system.

Modeling such communication and signal processing systems usually results in *critical SDF graphs*. By a critical SDF graph, we mean an SDF graph that has: *large scale* (consists of hundreds (or more) of actors and edges); *complex topology* (contains directed and undirected cycles across the graph components); and *heavily multirate behavior* (contains large variations in data transfer rates or component execution rates across graph edges). Here, we define multirate complexity as a measure of overall multirate behavior.

Definition 2.1 (*Multirate Complexity*). Given an SDF graph $G = (V, E)$, its *multirate complexity* (MC) is defined as an average of its repetitions vector components

$$MC(G) = \left(\sum_{\forall v \in V} \boldsymbol{q}_G[v] \right) / |V|, \tag{3}$$

where $|V|$ is the number of actors in G. In other words, it is an average number of firings per component in one iteration of a minimal periodic schedule.

A complex topology complicates the scheduling process because the properties of data-driven and deadlock-free execution must be ensured. However, large-scale and heavily multirate behavior cause the most serious problems due to the following three related characteristics:

(1) *High multirate complexity*. Multirate transitions in an SDF graph, namely, $\{e \in E \mid prd(e) \neq cns(e)\}$, generally lead to repetition counts that increase exponentially in the number of such transitions [Bhattacharyya et al. 1996]. Critical SDF graphs usually have extremely high multirate complexities, even up to the range of millions, as we show in Section 5. Such high multirate complexity seriously complicates the scheduling problem (i.e., sequencing large sets of firings for the same actors in addition to sequencing across actors), and has heavy impact on implementation metrics such as memory requirements, schedule length, and algorithm complexity.
(2) *Large number of firings*. Highly multirate behavior together with large graph scale generally makes the *number of firings* in a schedule (i.e., the sum of the repetitions vector components) increase exponentially in the number of multirate transitions, and also increase proportionally in graph size. In critical SDF graphs, schedules may have millions or even billions of firings, as we show in Section 5. As a result, any scheduling algorithm or schedule representation that works at the granularity of individual firings is unacceptable in our context.

(3) *Large memory requirements*. Increases in multirate complexity lead to corresponding increases in the overall volume of data transfer and the length of actor firing sequences in an SDF graph. Simulation tools usually run on workstations and PCs, where memory resources are abundant. However, due to exponential growth in multirate complexity, algorithms for scheduling and buffer allocation that are not carefully designed may still run out of memory when simulating critical SDF graphs.

In this article, we present the simulation-oriented scheduler (SOS) for simulating critical SDF graphs in EDA tools. Our objectives include:

(1) minimizing simulation run-time;
(2) scaling efficiently across various graph sizes, topologies, and multirate complexities; and
(3) satisfying memory constraints.

Our simulation-oriented scheduler statically computes schedules and buffer sizes with emphasis on low complexity, static scheduling, and memory minimization. Static scheduling and static buffering allow tools to simulate systems and allocate buffers with low setup cost and low runtime overhead. Low-complexity algorithms scale efficiently across various kinds of SDF graphs and minimize scheduling runtime. In SOS, the memory requirements for storing schedules and buffering data are carefully kept under control to prevent out-of-memory problems, and alleviate virtual memory swapping behavior, which causes large runtime overhead.

3. RELATED WORK

Various scheduling algorithms and techniques have been developed for different applications of SDF graphs. For example, Bhattacharyya et al. [1996] has presented a heuristic for minimum buffer scheduling. A simpler variant of this algorithm has been used in both Gabriel [Lee et al. 1989] and Ptolemy [Buck et al. 1994] environments, and a similar algorithm is also given by Cubric and Panangaden [1993]. We refer to these demand-driven, minimum buffer scheduling heuristics as *classical SDF scheduling*. This form of scheduling is effective at reducing the total buffer requirements, but its time complexity and the lengths of its resulting schedules generally grow exponentially in the size of multirate SDF graphs.

Based on developments in Buck [1993], the *cluster-loop scheduler* has been developed in the Ptolemy design tool [Buck et al. 1994] as a fast heuristic, that is, with scheduling runtime as a primary criterion. This approach recursively encapsulates adjacent groups of actors into loops to enable possible execution rate matches and then clusters the adjacent groups. Multirate transitions, however, can prevent this method from completely clustering the whole graph. Since any unclustered parts of the graph are left to classical SDF scheduling, this can result in large runtimes and storage requirements for constructing the schedule. Our experiments in Section 5 demonstrate problems encountered with this method on critical graphs.

Ade et al. [1997] have developed methods to compute lower bounds on buffer requirements based on analysis of directed and undirected cycles in an SDF graph; Geilen et al. [2005] have also developed an approach to compute minimum buffer requirements based on model checking. Because the complexities of these approaches are not polynomially bounded in graph size, they are not acceptable for the purposes that we are addressing in this article. Patel and Shukla [2004] have developed several MoC (model of computation) kernels, including an SDF kernel, in SystemC for heterogeneous system modeling. The approach used in their SDF kernel is similar to classical SDF scheduling and not suitable for scheduling critical SDF graphs.

Oh et al. [2006] have developed the *dynamic loop-count single-appearance scheduling* technique. This approach generates a looped single-appearance schedule, that is, a compact schedule that employs looping constructs in such a way that each actor appears only once in the schedule. In this approach, the iteration counts of loops can be determined at runtime by evaluating statements that encapsulate states of edges. This algorithm is geared towards minimizing buffer requirements for software synthesis, and its complexity and runtime overheads are relatively high for simulating critical SDF graphs.

Various methods [Murthy and Bhattacharyya 2006] have been developed that build on those of Bhattacharyya et al. [1996] to construct single-appearance schedules in a way that aggressively employs lifetime analysis and related approaches to share memory space across multiple buffers. These methods are also geared more for synthesis of streamlined embedded software, and their relatively high complexity makes them not ideally suited for our primary concerns of simulation-time reduction and algorithm scalability in critical SDF graphs.

Bhattacharyya et al. [1996] and Ko et al. [2004] have developed algorithms for code and memory minimization for certain types of SDF graphs. These algorithms are implemented in Hsu et al. [2005] for synthesis of embedded DSP software. Some of these algorithms are integrated in the simulation-oriented scheduler. In fact, the SOS approach adapts and integrates a variety of previously developed techniques in novel ways that more efficiently address the novel constraint of simulation efficiency.

We have first presented the simulation-oriented scheduler in Hsu et al. [2006]. This article is an extended version. Whereas the developments of Hsu et al. [2006] focused primarily on the problem motivation, algorithm overview, experimental methodology, and results, in this article we present the algorithms in detail, as well as the underlying theory for our single-rate clustering and buffer-optimal two-actor scheduling techniques that are employed in the SOS scheduler. We also discuss the overall integration and present simulation results, including results on relevant new wireless communication designs.

4. SIMULATION-ORIENTED SCHEDULER

Our SOS approach integrates several existing and newly developed algorithms for graph decomposition and scheduling. Figure 1 illustrates the overall architecture. Among these techniques, LIAF, APGAN, and DPPO have been developed in Bhattacharyya et al. [1996], and the concept of recursive two-actor

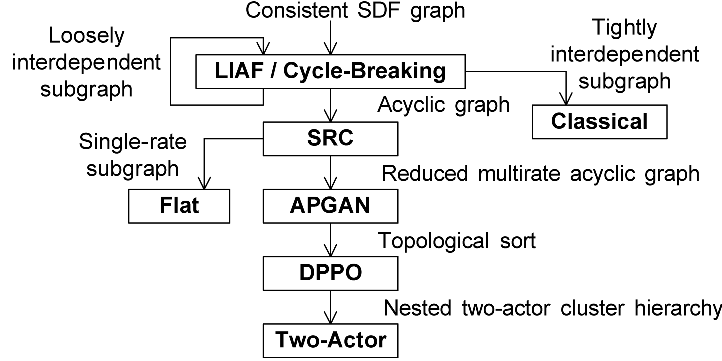

Fig. 1. Architecture of the simulation-oriented scheduler.

graph decomposition has been developed in Ko et al. [2004]. These techniques were originally designed for code and data memory minimization in software synthesis, and have not been applied with simulation of critical SDF graphs as an explicit concern. In SOS, we develop a novel integration of these methods, and incorporate into this integrated framework the following new techniques:

(1) *cycle breaking* to achieve fast execution in LIAF [Hsu and Bhattacharyya 2007];
(2) *single-rate clustering* (SRC) to alleviate the complexity of APGAN and DPPO; and
(3) *buffer-optimal two-actor scheduling* for handling nonzero delays on graph edges in addition to delayless two-actor graphs.

In this section, we present the novel integration as well as the algorithms and theory associated with the new techniques.

4.1 SDF Scheduling Preliminaries

To provide for more memory-efficient storage of schedules, *actor firing sequences* can be represented through looping constructs [Bhattacharyya et al. 1996]. For this purpose, a *schedule loop* $L = (n\ T_1 T_2 \cdots T_m)$ represents the successive repetition n times of the invocation sequence $T_1 T_2 \cdots T_m$, where each T_i is either an actor firing or a (nested) schedule loop. A *looped schedule* $S = L_1 L_2 \cdots L_N$ is an SDF schedule that is expressed in terms of the schedule loop notation. If every actor appears only once in S, S is called a *single-appearance schedule*, otherwise, S is called a *multiple-appearance schedule*. Every valid (looped) schedule has a *unique* actor firing sequence that can be derived by unrolling all of the loops in the schedule. For example, the schedule $S = a(3b)(2a(2b)a(3b))$ represents the firing sequence $abbbabbabbbabbabbb$. Hereafter in this article, we assume that an SDF schedule is represented in the looped schedule format.

SDF clustering is an important operation in SOS. Given a connected, consistent SDF graph $G = (V, E)$, clustering a connected subset $Z \subseteq V$ into a *supernode* α means:

(1) extracting a subgraph $G_\alpha = (Z, \{e \mid src(e) \in Z \text{ and } snk(e) \in Z\})$; and

(2) transforming G into a reduced form $G' = (V', E')$, where $V' = V - Z + \{\alpha\}$ and $E' = E - \{e \mid src(e) \in Z \text{ or } snk(e) \in Z\} + E^*$.

Here, E^* is a set of "modified" edges in G that originally connect actors in Z to actors outside of Z. More specifically, for every edge e that satisfies ($src(e) \in Z$ and $snk(e) \notin Z$), there is a modified version $e^* \in E^*$ such that $src(e^*) = \alpha$ and $prd(e^*) = prd(e) \times \boldsymbol{q}_{G_\alpha}(src(e))$, and similarly, for every e that satisfies ($src(e) \notin Z$ and $snk(e) \in Z$), there is a modified version $e^* \in E^*$ such that $snk(e^*) = \alpha$ and $cns(e^*) = cns(e) \times \boldsymbol{q}_{G_\alpha}(snk(e))$.

In the transformed graph G', execution of α corresponds to executing one iteration of a minimal periodic schedule for G_α. SDF clustering guides the scheduling process by transforming G into a reduced form G' and isolating a subgraph G_α of G such that G' and G_α can be treated separately, for example, by using different optimization techniques. SDF clustering [Bhattacharyya et al. 1996] guarantees that if we replace every supernode firing α in a schedule $S_{G'}$ for G' with a minimal periodic schedule S_{G_α} for G_α, then the result is a valid schedule for G.

4.2 LIAF Scheduling

The *loose interdependence algorithms framework* (LIAF) [Bhattacharyya et al. 1996] aims to decompose and break cycles in an SDF graph such that algorithms for scheduling or optimization that are subsequently applied can operated on acyclic graphs. Existence of cycles in the targeted subsystems prevents or greatly restricts application of many useful optimization techniques.

Given a connected, consistent SDF graph $G = (V, E)$, LIAF starts by clustering all *strongly connected components*[1] $Z_1, Z_2, \ldots, Z_N$ into supernodes $\alpha_1, \alpha_2, \ldots, \alpha_N$, and this results in an acyclic graph G_a [Cormen et al. 2001]. For each strongly connected subgraph $G_i = (Z_i, E_i)$, LIAF tries to break cycles by properly removing edges that have "sufficient" delays. An edge $e_i \in E_i$ can be removed in this sense if it has enough initial tokens to satisfy the consumption requirements of its sink actor for a complete iteration of G_i—that is, if $del(e_i) \geq cns(e_i) \times \boldsymbol{q}_{G_i}(snk(e_i))$—so that scheduling without considering e_i does not deadlock G_i. Such an edge e_i is called an *interiteration edge* in our context.

Now suppose that ${G_i}^*$ denotes the graph that results from removing all interiteration edges from the strongly connected subgraph G_i. Then, G_i is said to be *loosely interdependent* if ${G_i}^*$ is not strongly connected, and G_i is said to be *tightly interdependent* if G_i^* is strongly connected. If G_i is found to be loosely interdependent, then LIAF is applied recursively to the modified version G_i^* of G_i.

Careful decomposition of strongly connected SDF graphs into hierarchies of acyclic graphs—a process that is referred to as *subindependence partitioning* or *cycle-breaking*—is a central part of the LIAF framework. LIAF does not specify the exact algorithm to break cycles, but rather specifies the constraints that such an algorithm must satisfy [Bhattacharyya et al. 2000, 1996]. For use in SOS, we have developed a low-complexity cycle-breaking algorithm for properly

[1] A strongly connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $Z \subseteq V$ such that for every pair of vertices u and v in Z, there is a path from u to v and a path from v to u.

removing interiteration edges such that decomposing and breaking cycles in LIAF can be implemented in time that is linear in the number of actors and edges in the input SDF graph. We have also developed a technique to compute buffer sizes for the removed interiteration edges. For details, we refer the reader to Hsu and Bhattacharyya [2007].

In our application of LIAF in SOS, tightly interdependent subgraphs are scheduled by classical SDF scheduling, which is discussed in more detail in Section 4.3, and the acyclic graphs that emerge from the LIAF decomposition process are further processed by the techniques developed in Sections 4.4 through 4.8.

4.3 Classical SDF Scheduling

As described in Section 3, classical SDF scheduling is a demand-driven, minimum buffer scheduling heuristic. By simulating demand-driven dataflow behavior (i.e., by deferring execution of an actor until output data from it is needed by other actors), we can compute a buffer-efficient actor firing sequence and the associated buffer sizes. The complexity of classical SDF scheduling is not polynomially bounded in size of the input graph, and we use it only as a backup process for scheduling tightly interdependent subgraphs from LIAF. Fortunately, this does not cause any major limitation in SOS because tightly interdependent subgraphs arise very rarely in practice [Bhattacharyya et al. 1996]. For example, we have tested SOS on a suite of 126 wireless network designs and 267 wireless communication designs, and among all of these designs, no tightly interdependent subgraphs were found.

4.4 Single-Rate Clustering

Intuitively, a single-rate subsystem in an SDF graph is a subsystem in which all actors execute at the same average rate. In practical communication and signal processing systems, single-rate subsystems arise commonly, even within designs that are heavily multirate at a global level. In precise terms, an SDF graph is a single-rate graph if for every edge e, we have $prd(e) = cns(e)$. Since clustering single-rate subsystems does not increase production and consumption rates at the interface of the resulting supernodes, we have developed the single-rate clustering (SRC) technique to further decompose an acyclic graph into a reduced (smaller) multirate version along with several single-rate subgraphs. Due to their simple structure, single-rate subgraphs can be scheduled and optimized effectively by the accompanying *flat scheduling* (Section 4.5) algorithm in a very fast manner. Furthermore, the reduced multirate graph, which is scheduled using the more intensive techniques described in Sections 4.6 through 4.8, takes less time to schedule due to its significantly smaller size, that is, since each single-rate subsystem is abstracted as a single actor (supernode).

Definition 4.1 (*Single-Rate Clustering*). Given a connected, consistent, acyclic SDF graph $G = (V, E)$, the *single-rate clustering* (SRC) technique

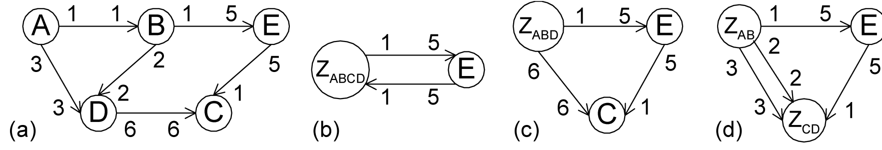

Fig. 2. Single-rate clustering examples.

clusters disjoint subsets $R_1, R_2, \ldots, R_N \subseteq V$ such that:

(1) In the subgraph $G_i = (R_i, E_i)$, we have that $\forall e_i \in E_i = \{e \mid src(e) \in R_i$ and $snk(e) \in R_i\}$, $prd(e_i) = cns(e_i)$;
(2) the clustering of R_i does not introduce any cycles into the clustered version of G;
(3) R_i satisfies $|R_i| > 1$ (i.e., R_i contains at least two actors); and
(4) each R_i contains a maximal set of actors that satisfy all of the three aforementioned conditions.

Such R_is are defined as *single-rate subsets*; and such G_is are defined as *single-rate subgraphs*.

The targeting of "maximal" clusters in the fourth condition is important in order to effectively reduce the size of the clustered graph, and to maximize the extent of the overall system that can be handled with the streamlined techniques available for single-rate graphs.

Simply clustering a set of actors that is connected through single-rate edges may introduce cycles in the clustered graph. Figure 2 illustrates how this simple strategy fails. Nodes A, B, C, and D in Figure 2(a) are connected by single-rate edges, and clustering them will result in a cyclic graph as shown in Figure 2(b). In contrast, Figures 2(c) and 2(d) present two acyclic SDF graphs after valid single-rate clustering. The following theorem provides a precise condition for the introduction of a cycle by a clustering operation.

THEOREM 4.2 (CYCLE-FREE CLUSTERING). *Given a connected, acyclic SDF graph $G = (V, E)$, clustering a connected subset $R \subseteq V$ introduces a cycle in the clustered version of G if and only if there is a path $v_1 \to v_2 \to \cdots \to v_n$ ($n \geq 3$) in G such that $v_1 \in R$, $v_n \in R$, and $v_2, \ldots, v_{n-1} \in \{V - R\}$. Clustering R is cycle free if and only if no such path exists.*

PROOF. Without loss of generality, suppose that we cluster R into a supernode α, and this results in a subgraph G_α and the clustered version G'. Based on SDF clustering, as discussed in Section 4.1, we have:

(1) for every edge $e \in \{e \mid src(e) \in R$ and $snk(e) \notin R\}$, it becomes an output edge $e' = (\alpha, snk(e))$ of α in G', and every output edge of α comes from this transformation; and
(2) for every edge $e \in \{e \mid src(e) \notin R$ and $snk(e) \in R\}$, it becomes an input edge $e' = (src(e), \alpha)$ of α in G', and every input edge of α comes from this transformation.

```
SRC(G ≡ (V, E)) /*The input G is a connected acyclic SDF graph*/
 1   G′ ≡ (V′, E′) ← G
 2   i ← 1
 3   for the next actor v ∈ V′
 4       R_i ← SRS(G′, v)
 5       if R_i ≠ ∅    i ← i + 1    end
 6   end
 7   N ← i − 1
 8   for i from 1 to N    CLUSTER(G, R_i)    end

SRS(G′, v)
 9   R ← {v}
10   for the next edge a ∈ {in(v) + out(v)}
11       if src(a) = v and x ← snk(a) is not in any subset
12           A ← {e ∈ in(x) | src(e) = v and prd(e) = cns(e)}
13           B ← {e ∈ in(x) | src(e) = v and prd(e) ≠ cns(e)}
14           C ← {e ∈ in(x) | src(e) ≠ v}
15           if B = ∅ and IS-CYCLE-FREE(G′, v, x)
16               for each e ∈ out(x)    src(e) ← v    end
17               for each e ∈ C    snk(e) ← v    end
18               R ← R + {x}, E′ ← E′ − A, V′ ← V′ − {x}
19           end
20       else if snk(a) = v and x ← src(a) is not in any subset
21           A ← {e ∈ out(x) | snk(e) = v and prd(e) = cns(e)}
22           B ← {e ∈ out(x) | snk(e) = v and prd(e) ≠ cns(e)}
23           C ← {e ∈ out(x) | snk(e) ≠ v}
24           if B = ∅ and IS-CYCLE-FREE(G′, x, v)
25               for each e ∈ in(x)    snk(e) ← v    end
26               for each e ∈ C    src(e) ← v    end
27               R ← R + {x}, E′ ← E′ − A, V′ ← V′ − {x}
28           end
29       end
30   end
31   if R = {v}    return ∅    else    return R    end

IS-CYCLE-FREE(G′, y, z)
32   E_yz ← {e | src(e) = y and snk(e) = z}
33   if {out(y) − E_yz} = ∅ or {in(z) − E_yz} = ∅    return TRUE
34   else    return ! IS-REACHABLE((V′, {E′ − E_yz}), y, z)
35   end
```

Fig. 3. Single-rate clustering (SRC) algorithm.

Therefore, a path $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n$ in G, where $v_1 \in R$, $v_n \in R$, and $v_2 \cdots \in \{V - R\}$, becomes a cycle $\alpha \rightarrow v_2 \rightarrow \cdots \rightarrow \alpha$ in G'. In addition, a cycle containing α in G' can only come from such a path in G. □

We have developed the SRC algorithm as presented in Figure 3. Given a connected, acyclic SDF graph $G = (V, E)$, we first duplicate G into $G' = (V', E')$ to prevent us from modifying G before actually clustering the single-rate subsets. Given G' and an actor v, the subroutine SRS (*s*ingle-*r*ate *s*ubset) returns a single-rate subset that contains v, or returns ∅ if no such single-rate subset exists for v. In lines 2–7, all single-rate subsets $R_1, R_2, \ldots, R_N$ are computed, and particularly, the "next actor" in line 3 refers to the next actor that has

not yet been visited in the *remaining* V' after each call of SRS. In line 8, we cluster $R_1, R_2, \ldots, R_N$ in G by repeatedly calling the SDF clustering operation CLUSTER.

SRS iteratively determines whether an adjacent actor x of v belongs to the single-rate subset R in a breadth-first fashion. An adjacent, nonclustered successor $x = snk(a)$ of v in line 11 can be included in R if

(1) every edge connecting v to x is single rate (i.e., $B = \varnothing$ in line 15); and
(2) clustering v and x does not introduce a cycle (i.e., IS-CYCLE-FREE(G',v,x) returns TRUE).

If both criteria are satisfied, x is included in R (line 18), and G' is transformed to mimic the effect of clustering x in lines 16–18. After that, the actor v in G' represents the R-clustered supernode. On the other hand, for an adjacent predecessor $x = src(a)$ of v, similar operations are performed in lines 20–29. Note that after each iteration, v's incident edges $\{in(v) + out(v)\}$ in line 10 may have been changed because the online topology transformation removes and inserts incident edges of v, and particularly, the "next" edge in line 10 refers to the next edge that has not yet been visited.

IS-CYCLE-FREE determines whether clustering a source node y and a sink node z of an edge in G' is cycle free based on Theorem 4.2 (i.e., by checking whether there is a path from y to z through other actors). If all output edges of y connect to z or all input edges of z connect from y, we can immediately determine that no such path exists (line 33). Otherwise, in line 34, we test to ensure that z is *not* reachable from y when all edges connecting y to z are removed.

PROPERTY 4.3. *The set R returned by SRS in the SRC algorithm is a single-rate subset.*

PROOF. The set R is a single-rate subset if it satisfies the conditions in Definition 4.1. Because an adjacent actor x of v (in lines 11 and 20) can be included in R if every edge connected between x and v is single rate and clustering x and v is cycle free, and since v represents the up-to-date R-cluster, conditions 1 and 2 in Definition 4.1 are satisfied. Condition 3 is simply checked by line 31. Condition 4 can be satisfied if at the end of the iterations, every surrounding edge of R has been searched for determining whether the adjacent actor x belongs to the single-rate subset. This is true because SRS iterates over every incident edge of v in a breadth-first way and updates v's incident edges in each iteration. □

Before discussing the complexity of the SRC algorithm and the complexity of the algorithms in the following sections, we make the assumption that every actor has a constant (limited) number of input and output edges, that is, $|V|$ and $|E|$ are within a similar range. This is a reasonable assumption because actors in simulation tools are usually predefined, and practical SDF graphs in communications and signal processing domains are sparse in their topology [Bhattacharyya et al. 1996].

PROPERTY 4.4. *The complexity of the SRC algorithm is* $O(|E|^2)$*, where* E *denotes the set of edges in the input graph.*

PROOF. By the combination of the for-loop in line 3 and the for-loop in line 10, an edge can be visited by line 10 once (if clustered), twice (if not clustered), or no times (if there are parallel edges between two nodes). Therefore, the total number of edges examined in line 10 is $O(|E|)$. With efficient data structures and the assumption that every actor has a limited number of incident edges, operations (lines 11–29) within the for-loop in line 10 require constant time, except for IS-CYCLE-FREE, which takes $O(|E'| + |V'|)$ time. As a result, the running time to compute all single-rate subsets (lines 3–6) is $O(|E|^2)$. In the last step, the complexity to cluster $R_1, R_2, \ldots, R_N$ in line 8 is bounded by $O(|V| + |E|)$. Therefore, the complexity of the SRC algorithm is $O(|E|^2)$. □

4.5 Flat Scheduling

Given a consistent, acyclic SDF graph $G = (V, E)$, a valid single appearance schedule S can be easily derived by *flat scheduling*. Flat scheduling simply computes a topological sort $v_1\, v_2 \cdots v_{|V|}$ of G, and iterates each actor v_i $\boldsymbol{q}_G[v_i]$ times in succession. More precisely, the looped schedule constructed from the topological sort in flat scheduling is $S = (\boldsymbol{q}_G[v_1]\, v_1)\, (\boldsymbol{q}_G[v_2]\, v_2) \cdots (\boldsymbol{q}_G[v_{|V|}]\, v_{|V|})$. The complexity of flat scheduling is $O(|V| + |E|)$; the topological sort can be performed in linear time [Cormen et al. 2001], and the repetitions vector can also be computed in linear time [Bhattacharyya et al. 1996]. However, in general, the memory requirements of flat schedules can become very large in multirate systems [Bhattacharyya et al. 1996].

We apply flat scheduling only to single-rate subgraphs, which do not suffer from the memory penalties that are often associated with flat scheduling in general SDF graphs. This is because in a single-rate subgraph, each actor only fires once within a minimal periodic schedule. Thus, the buffer size of each single-rate edge e can be set to $buf(e) = prd(e) + del(e)$, which is the minimum achievable size whenever $del(e) < prd(e)$.

4.6 APGAN Scheduling

In general, computing buffer-optimal topological sorts in SDF graphs is NP-hard [Murthy et al. 1997]. The *acyclic pairwise grouping of adjacent nodes* (APGAN) [Bhattacharyya et al. 1996] technique is an adaptable (to various cost functions) heuristic to generate topological sorts. Given a consistent, acyclic SDF graph $G = (V, E)$, APGAN iteratively selects and clusters adjacent pairs of actors until the top-level clustered graph consists of a single supernode. The clustering process is guided by a cost function $f(e)$ that estimates the impact of clustering of actors $src(e)$ and $snk(e)$ into a supernode. In each iteration, APGAN clusters an adjacent pair $\{src(e), snk(e)\}$ in the current version of G such that:

(1) clustering this pair does not introduce cycles in the clustered graph; and
(2) the applied cost function f is maximized for e over all edges e^* for which $\{src(e^*), snk(e^*)\}$ can be clustered without introducing cycles.

Once the clustering process is complete, a topological sort is obtained through depth-first traversal of the resulting cluster hierarchy.

In our incorporation of APGAN in SOS, we use the following as the cost function f: $\gcd(\boldsymbol{q}_G[src(e)], \boldsymbol{q}_G[snk(e)])$, where gcd represents the *greatest common divisor* operator. This cost function has been found to direct APGAN towards solutions that are efficient in terms of buffering requirements [Bhattacharyya et al. 1996].

The complexity of APGAN is $O(|V|^2|E|)$ [Bhattacharyya et al. 1996]. At first, this appears relatively high in relation to the objective of low complexity in this article. However, due to the design of our SOS framework which applies the LIAF and SRC decomposition techniques, $|V|$ and $|E|$ are typically much smaller in the instances of APGAN that result during operation of SOS compared to the numbers of actors and edges in the overall SDF graph.

4.7 DPPO Scheduling

Given a topological sort $L = v_1\, v_2 \cdots v_{|V|}$ of an acyclic SDF graph $G = (V, E)$, the *dynamic programming postoptimization* (DPPO) technique [Bhattacharyya et al. 1996] constructs a memory-efficient hierarchy of nested two-actor clusters. This hierarchy is constructed in a bottom-up fashion by starting with each two-actor subsequence $v_i\, v_{i+1}$ in the topological sort and progressively optimizing the decomposition of longer subsequences. Each l-actor subsequence $L_{i,j} = v_i\, v_{i+1} \cdots v_{j=i+l-1}$ is "split" into two shorter "left" ($L_{i,k} = v_i\, v_{i+1} \cdots v_k$) and "right" ($L_{k+1,j} = v_{k+1}\, v_{k+2} \cdots v_j$) subsequences. In particular, the split position k is chosen to minimize the buffer requirement of $L_{i,j}$, that is, the sum of buffer requirements associated with the left and right subsequences plus the buffer requirements for the set $E_{i,j,k} = \{e \mid src(e) \in \{v_i, v_{i+1}, \ldots, v_k\}$ and $snk(e) \in \{v_{k+1}, v_{k+2}, \ldots, v_j\}\}$ of edges that cross from left to right. Through dynamic programming, where the outer loop l iterates from 2 to $|V|$, the middle loop i iterates from 1 to $|V| - l + 1$, and the inner loop k iterates from i to $i + l - 2$, the best split position and minimal buffer requirement for every subsequence can be derived. A memory-efficient hierarchy is then built in a top-down fashion by starting from the topological sort $L_{1,|V|}$, and recursively clustering the left $L_{i,k}$ and right $L_{k+1,j}$ subsequences through the best split position k of $L_{i,j}$.

In SOS, we have developed an adapted DPPO where each split k of $L_{i,j}$ is interpreted as a two-actor SDF graph $G_{i,j,k} = (\{\alpha_{i,k}, \alpha_{k+1,j}\}, E'_{i,j,k})$, where the left $L_{i,k}$ and right $L_{k+1,j}$ subsequences make up the two hierarchical actors $\alpha_{i,k}$ and $\alpha_{k+1,j}$, and every edge $e' \in E'_{i,j,k}$ is a transformation from the corresponding edge $e \in E_{i,j,k}$ such that $prd(e') = prd(e) \times \boldsymbol{q}_G[src(e)]/\gcd(\boldsymbol{q}_G[v_i], \boldsymbol{q}_G[v_{i+1}], \ldots, \boldsymbol{q}_G[v_k])$ and $cns(e') = cns(e) \times \boldsymbol{q}_G[snk(e)]/\gcd(\boldsymbol{q}_G[v_{k+1}], \boldsymbol{q}_G[v_{k+2}], \ldots, \boldsymbol{q}_G[v_j])$ based on SDF clustering concepts. This two-actor graph is further optimized, after DPPO, by the buffer-optimal two-actor scheduling algorithm discussed in Section 4.8. Furthermore, the optimal buffer requirements for $E'_{i,j,k}$ can be computed in constant time (based on Theorem 4.13, which is developed to follow) without actually computing a two-actor schedule for each split.

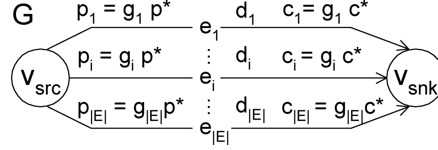

Fig. 4. Consistent, acyclic, two-actor SDF graph.

DPPO can be performed in $O(|V|^3)$ time [Bhattacharyya et al. 1996]. The complexity of our adapted DPPO is also $O(|V|^3)$ because the cost function, that is, the optimal buffer requirement of each two-actor graph $G_{i,j,k}$, can be computed in constant time. Also, as with the techniques discussed in the previous section, the value of $|V|$ for instances of our adapted DPPO technique is relatively small due to the preceding stages of LIAF- and SRC-based decomposition.

4.8 Buffer-Optimal Two-Actor Scheduling

The concept of recursive two-actor scheduling for a nested two-actor SDF hierarchy was originally explored in Ko et al. [2004]. For *delayless* SDF graphs, the resulting schedules are proven to be buffer optimal at each (two-actor) level of the cluster hierarchy. These schedules are also polynomially bounded in graph size. However, the algorithm in Ko et al. [2004] does not optimally handle the scheduling flexibility provided by edge delays, and therefore, does not always achieve minimum buffer sizes in the presence of delays. We have developed a new buffer-optimal two-actor scheduling algorithm that computes a buffer-optimal schedule for a *general* (with or without delays), consistent, acyclic, two-actor SDF graph. This algorithm is applied in SOS to schedule each two-actor subgraph in the DPPO hierarchy. An overall schedule is then constructed by recursively traversing the hierarchy and replacing every supernode firing by the corresponding two-actor subschedule. In this subsection, we present definitions, analysis, and an overall algorithm that are associated with our generalized, two-actor scheduling approach.

PROPERTY 4.5. *A consistent, acyclic, two-actor SDF graph* $G = (\{v_{src}, v_{snk}\}, E)$ *has a general form as shown in Figure* 4*, where for each* $e_i \in E$, $src(e_i) = v_{src}$, $snk(e_i) = v_{snk}$, $p_i = prd(e_i)$, $c_i = cns(e_i)$, $d_i = del(e_i)$, $g_i = \gcd(p_i, c_i)$, $p^* = p_i/g_i$, *and* $c^* = c_i/g_i$. *For consistency, the coprime positive integers* p^* *and* c^* *must satisfy* $p_i/c_i = p^*/c^*$ *for every* $e_i \in E$.

Definition 4.6 (*Primitive Two-Actor SDF Graph*). Given a consistent, acyclic, two-actor SDF graph $G = (\{v_{src}, v_{snk}\}, E)$ as described in Property 4.5, its *primitive form* is defined as a two-actor, single-edge SDF graph $G^* = (\{v_{src}, v_{snk}\}, \{e^*\})$ as shown in Figure 5, where $src(e^*) = v_{src}$, $snk(e^*) = v_{snk}$, $prd(e^*) = p^*$, $cns(e^*) = c^*$, $\gcd(p^*, c^*) = 1$, and $del(e^*) = d^* = \min_{e_i \in E}(\lfloor d_i/g_i \rfloor)$. The values p^*, c^*, and d^* are defined as the *primitive production rate*, *primitive consumption rate*, and *primitive delay* of G, respectively. An edge e_i that satisfies $\lfloor d_i/g_i \rfloor = d^*$ is called a *maximally constrained edge* of G.

Here, we also define some notations that are important to our development of two-actor scheduling. Suppose that we are given a consistent SDF graph

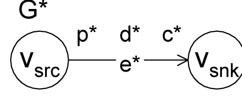

Fig. 5. Primitive two-actor SDF graph.

$G = (V, E)$ and a valid minimal periodic schedule S for G. By a *firing index* for S, we mean a nonnegative integer that is less than or equal to the sum Q_G of repetitions vector components for G (i.e., $Q_G = \sum_{v \in V} \boldsymbol{q}_G[v]$). In the context of S, a firing index value of k represents the kth actor execution within a given iteration (minimal period) of the execution pattern derived from repeated executions of S.

Now, let $\sigma(S, k)$ denote the actor associated with firing index k for the schedule S; let $\tau(S, v, k)$ denote the firing count of actor v up to firing index k (i.e., the number of times that v is executed in a given schedule iteration up to the point in the firing sequence corresponding to k); and let

$$tok_G(S, e, k) = \tau(S, src(e), k) \times prd(e) - \tau(S, snk(e), k) \times cns(e) + del(e) \qquad (4)$$

denote the number of tokens queued on edge $e \in E$ immediately after the actor firing associated with firing index k in any given schedule iteration. Firing index 0 represents the initial state: for $k = 0$, $\sigma(S, 0)$ is defined to be $\varnothing$ (the "null actor"), $\tau(S, v, 0)$ is defined to be 0, and $tok_G(S, e, 0)$ is defined as $del(e)$. Note that from the properties of periodic schedules, the values of σ, τ, and tok_G are uniquely determined by k, and not dependent on the schedule iteration [Bhattacharyya et al. 1996]. The repeated execution of S leads to an infinite sequence $x_1, x_2, \ldots$ of actor executions, where each x_i corresponds to firing index $((i - 1) \bmod Q_G) + 1$.

For example, suppose that we have an SDF graph $G = (\{a, b\}, \{e = (a, b)\})$, where $prd(e) = 7$, $cns(e) = 5$, and $del(e) = 0$. Suppose also that we have the schedule $S = (1(2ab)(1a(2b)))(1(1ab)(1a(2b)))$. Then we can unroll S into a firing sequence $ababab bababb$, where $\sigma(S, 1) = a$, $\sigma(S, 6) = b$, $\tau(S, a, 6) = 3$, $\tau(S, b, 6) = 3$, $tok_G(S, e, 0) = 0$, and $tok_G(S, e, 2) = 2$.

The following lemma is useful in simplifying scheduling and analysis for acyclic, two-actor SDF graphs.

LEMMA 4.7. *A schedule S is a valid minimal periodic schedule for a consistent, acyclic, two-actor SDF graph G if and only if S is a valid minimal periodic schedule for the primitive form G^* of G.*

PROOF. Without loss of generality, suppose G is in a general form as shown in Figure 4, and suppose Figure 5 represents its primitive form G^*. First, we prove the *only-if* direction. S is a valid minimal periodic schedule for G if and only if:

(1) S fires v_{src} $\boldsymbol{q}_G[v_{src}]$ times and fires v_{snk} $\boldsymbol{q}_G[v_{snk}]$ times; and
(2) S is deadlock free.

Because $\boldsymbol{q}_G[v_{src}] = c^* = \boldsymbol{q}_{G^*}[v_{src}]$ and $\boldsymbol{q}_G[v_{snk}] = p^* = \boldsymbol{q}_{G^*}[v_{snk}]$, we know that S fires v_{src} $\boldsymbol{q}_{G^*}[v_{src}]$ times and fires v_{snk} $\boldsymbol{q}_{G^*}[v_{snk}]$ times in G^*—(A). Furthermore,

S is deadlock free for G if and only if the number of tokens queued on every edge e_i is greater than or equal to c_i before every firing of v_{snk}. In other words, for every $k_{snk} \in \{k \mid \sigma(S,k) = v_{snk}\}$, $tok_G(S, e_i, k_{snk} - 1) \geq c_i$ is true for every e_i. Through Eq. (5), where $\tau(S, v_{src}, k_{snk} - 1)$ is denoted as a, and $\tau(S, v_{snk}, k_{snk} - 1)$ is denoted as b, we can derive that $tok_{G^*}(S, e^*, k_{snk} - 1) \geq c^*$ for every k_{snk}, that is, the number of tokens queued on e^* before every firing of v_{snk} is greater than or equal to c^* in G^*, so S is deadlock free for G^*—(B). Based on (A) and (B), the only-if direction is proved.

$$\begin{aligned} &\forall i \;\; tok_G(S, e_i, k_{snk} - 1) = p_i \times a - c_i \times b + d_i \geq c_i \\ \Leftrightarrow\, &\forall i \;\; p^* \times g_i \times a - c^* \times g_i \times b + \lfloor d_i / g_i \rfloor \times g_i + d_i \bmod g_i \geq c^* \times g_i \\ \Leftrightarrow\, &\forall i \;\; p^* \times a - c^* \times b + \lfloor d_i / g_i \rfloor \geq c^* \\ \Leftrightarrow\, &p^* \times a - c^* \times b + d^* = tok_{G^*}(S, e^*, k_{snk} - 1) \geq c^* \end{aligned} \tag{5}$$

The *if* direction can be proved in a similar manner by applying the same derivations in reverse order and based on the reverse direction in Eq. (5). We omit the details for brevity. □

Definition 4.8 (*SASAP Schedule*). A sink-as-soon-as-possible (SASAP) schedule S for a consistent, acyclic, two-actor SDF graph $G = (\{v_{src}, v_{snk}\}, E)$ is defined as a valid minimal periodic schedule such that:

(1) S fires v_{src} $\boldsymbol{q}_G[v_{src}]$ times and fires v_{snk} $\boldsymbol{q}_G[v_{snk}]$ times;
(2) for every firing index $k_{snk} \in \{k \mid \sigma(S,k) = v_{snk}\}$, we have that $tok_G(S, e_i, k_{snk} - 1) \geq c_i$ for every $e_i \in E$, and $\tau(S, v_{snk}, k_{snk} - 1) < \boldsymbol{q}_G[v_{snk}]$; and
(3) for every firing index $k_{src} \in \{k \mid \sigma(S,k) = v_{src}\}$, either there exists an edge $e_i \in E$ such that $tok_G(S, e_i, k_{src} - 1) < c_i$ or such that $\tau(S, v_{snk}, k_{src} - 1) = \boldsymbol{q}_G[v_{snk}]$.

If an actor firing subsequence (subschedule) S' satisfies items (2) and (3), we say that S' fires v_{snk} as-soon-as-possible (ASAP).

Intuitively, an SASAP schedule can be viewed as a specific form of demand-driven schedule for periodic scheduling of acyclic, two-actor SDF graphs. An SASAP schedule defers execution of the source actor in an acyclic, two-actor configuration until the sink actor does not have enough input data to execute. This form of scheduling leads to minimum buffer schedules, as we state in the following property, because tokens are produced by the source actor only when necessary.

PROPERTY 4.9. *An SASAP schedule for a consistent, acyclic, two-actor SDF graph G is a minimum buffer schedule for G.*

The following lemma relates SASAP schedules and primitive forms.

LEMMA 4.10. *A schedule S is an SASAP schedule for a consistent, acyclic, two-actor SDF graph G if and only if S is an SASAP schedule for the primitive form G^* of G.*

PROOF. The validity and minimal periodic property of S in both directions is proved in Lemma 4.7. Here, we prove the SASAP property. Again, without loss

of generality, suppose G is in a general form as shown in Figure 4, and suppose Figure 5 represents its primitive form G^*. We first prove the only-if direction. S is an SASAP schedule for G if and only if:

(1) For every $k_{snk} \in \{k \mid \sigma(S,k) = v_{snk}\}$, $tok_G(S, e_i, k_{snk} - 1) \geq c_i$ for every e_i and $\tau(S, v_{snk}, k_{snk} - 1) < \boldsymbol{q}_G[v_{snk}]$; and

(2) for every $k_{src} \in \{k \mid \sigma(S,k) = v_{src}\}$, there exists at least one e_i where either $tok_G(S, e_i, k_{src} - 1) < c_i$ or $\tau(S, v_{snk}, k_{src} - 1) = \boldsymbol{q}_G[v_{snk}]$.

Through Eq. (5) and based on $\boldsymbol{q}_G[v_{snk}] = p^* = \boldsymbol{q}_{G^*}[v_{snk}]$, we can derive that for every $k_{snk} \in \{k \mid \sigma(S,k) = v_{snk}\}$, $tok_{G^*}(S, e^*, k_{snk} - 1) \geq c^*$ and $\tau(S, v_{snk}, k_{snk} - 1) < \boldsymbol{q}_{G^*}[v_{snk}]$. Furthermore, through Eq. (6), where $\tau(S, v_{src}, k_{src} - 1)$ is denoted as a, and $\tau(S, v_{snk}, k_{src} - 1)$ is denoted as b, we can derive that for every $k_{src} \in \{k \mid \sigma(S,k) = v_{src}\}$, either $tok_{G^*}(S, e^*, k_{src} - 1) < c^*$ or $\tau(S, v_{snk}, k_{src} - 1) = \boldsymbol{q}_{G^*}[v_{snk}]$. Therefore, if S is an SASAP schedule for G, then S is an SASAP schedule for G^*.

$$
\begin{aligned}
& \exists i \;\; tok_G(S, e_i, k_{src} - 1) = p_i \times a - c_i \times b + d_i < c_i \\
\Leftrightarrow\; & \exists i \;\; p^* \times g_i \times a - c^* \times g_i \times b + \lfloor d_i / g_i \rfloor \times g_i + d_i \bmod g_i < c^* \times g_i \\
\Leftrightarrow\; & \exists i \;\; p^* \times a - c^* \times b + \lfloor d_i / g_i \rfloor \leq c^* - 1 \\
\Leftrightarrow\; & p^* \times a - c^* \times b + d^* \leq c^* - 1 \\
\Leftrightarrow\; & p^* \times a - c^* \times b + d^* = tok_{G^*}(S, e^*, k_{src} - 1) < c^*
\end{aligned}
\tag{6}
$$

As with Lemma 4.7, the if direction can be proved in a similar manner by applying the same derivations in reverse order and based on the reverse directions in both Eqs. (5) and (6). □

The following corollary follows from Property 4.9 and Lemma 4.10.

COROLLARY 4.11. *A minimum buffer schedule for a consistent, acyclic, two-actor SDF graph can be obtained by computing an SASAP schedule for its primitive form.*

The following property follows from Eq. (4) and Definition 4.6 and relates the buffer activity in an acyclic, two-actor SDF graph to that of its primitive form.

PROPERTY 4.12. *Suppose S is a valid schedule for a consistent, acyclic, two-actor SDF graph $G = (\{v_{src}, v_{snk}\}, E)$ and for the primitive form $G^* = (\{v_{src}, v_{snk}\}, \{e^*\})$ of G. Then, for every edge $e_i \in E$, and for every firing index k, $tok_G(S, e_i, k) = tok_{G^*}(S, e^*, k) \times g_i + d_i - d^* \times g_i$, where $d_i = del(e_i)$, $g_i = \gcd(prd(e_i), cns(e_i))$, and $d^* = \min_{e_i \in E}(\lfloor d_i / g_i \rfloor)$.*

THEOREM 4.13. *For a consistent, acyclic, two-actor SDF graph $G = (\{v_{src}, v_{snk}\}, E)$, the minimum buffer requirement for an edge $e_i \in E$ is $p_i + c_i - g_i + d_i - d^* \times g_i$ if $0 \leq d^* \leq p^* + c^* - 1$, and is d_i otherwise. Here, $p_i = prd(e_i)$, $c_i = cns(e_i)$, $d_i = del(e_i)$, $g_i = \gcd(p_i, c_i)$, $p^* = p_i / g_i$, $c^* = c_i / g_i$, and $d^* = \min_{e_i \in E}(\lfloor d_i / g_i \rfloor)$.*

PROOF. For a consistent, single-edge, two-actor SDF graph $(\{v_{src}, v_{snk}\}, \{e = (v_{src}, v_{snk})\})$, Bhattacharyya et al. [1996] have proved that the minimum buffer requirement for e is $p + c - g + d \bmod g$ if $0 \leq d \leq p + c - g$, and is d otherwise, where $p = prd(e)$, $c = cns(e)$, $d = del(e)$, and $g = \gcd(p, c)$. As a result, the

minimum buffer requirement for e^* is $p^* + c^* - 1$ if $0 \le d^* \le p^* + c^* - 1$, and is d^* otherwise. From Lemma 4.10, Properties 4.9 and 4.12, and the minimum buffer requirement for e^*, the proof is complete. □

Theorem 4.13 presents a constant-time minimum buffer computation for any consistent, acyclic, two-actor SDF graph, and is used in our adapted form of DPPO to compute buffer requirements for each nested two-actor subgraph $G_{i,j,k} = (\{\alpha_{i,k}, \alpha_{k+1,j}\}, E'_{i,j,k})$, as described in Section 4.7.

In order to build an overall schedule from a nested two-actor DPPO hierarchy, we compute an optimal buffer schedule for each two-actor subgraph. Based on Corollary 4.11, we have developed the BOTAS (*b*uffer-*o*ptimal *t*wo-*a*ctor *s*cheduling) algorithm. This algorithm is shown in Figure 6. The BOTAS algorithm computes a minimum buffer schedule for a consistent, acyclic, two-actor SDF graph $G = (\{v_{src}, v_{snk}\}, E)$ by constructing an SASAP schedule for its primitive form.

In Figure 6, we first compute the p^*, c^*, and d^* of G to construct the primitive form $G^* = (\{v_{src}, v_{snk}\}, \{e^*\})$. Then, in lines 6–19, we compute two sequences of scheduling components $A_1, A_2, \ldots, A_I$ and $B_1, B_2, \ldots, B_I$, where I denotes the iteration i that ends the while-loop. Table I illustrates how to compute the sets of scheduling components for $p^* = 7$ and $c^* = 5$. For convenient schedule loop representation in Figure 6, we define the expression $(k \times L)$ for a positive integer k and a schedule loop $L = (n\ T_1 T_2 \cdots T_m)$ as a new schedule loop with the same loop body $T_1 T_2 \cdots T_m$ and the new iteration count $k \times n$, namely, $(k \times L) = (k \times n\ T_1 T_2 \cdots T_m)$.

From the results of this computation, we construct an SASAP schedule S for G^*. If the initial token population $d^* = 0$, S can be immediately built from A_I and B_I by line 22. Otherwise, in lines 25–38, we first use the scheduling components A_i and B_i from $i = 1$ to I to consume initial tokens until either the token population d on e^* is 0 or we exhaust execution of v_{snk}. Then, in lines 39–49, we use the scheduling components from $i = I$ to 1 to make up the remaining firings of v_{src} and v_{snk}, and bring the token population back to its initial state d^*. Table I illustrates how to compute SASAP schedules for $d^* = 0$, 6, and 12.

From the BOTAS algorithm, we can directly derive the following properties.

Property 4.14. *In the BOTAS algorithm, each A_i produces p_i tokens and fires v_{snk} ASAP and each B_i consumes c_i tokens and fires v_{snk} ASAP whenever there are c_i tokens.*

Property 4.15. *In the BOTAS algorithm, for every $i \in \{1, 2, \ldots, I-1\}$, $p_{i+1} + c_{i+1} = \min(p_i, c_i)$.*

The following property is derived from the BOTAS algorithm, Euclid's algorithm for computation of greatest common divisors, and mathematical induction.

Property 4.16. *In the BOTAS algorithm, for every $i \in \{1, 2, \ldots, I\}$, $\gcd(p_i, c_i) = 1$.*

Proof. Initially, $\gcd(p_1, c_1) = \gcd(p^*, c^*) = 1$. Suppose in an iteration $i > 1$, $\gcd(p_i, c_i) = 1$. By Eq. (7) when $p_i > c_i$, and by Eq. (8) when $p_i < c_i$, we can

BOTAS($G \equiv (\{v_{src}, v_{snk}\}, E)$) /*The input G is a consistent acyclic two-actor SDF graph*/
1 $p^* \leftarrow prd(e_1)/\gcd(prd(e_1), cns(e_1))$
2 $c^* \leftarrow cns(e_1)/\gcd(prd(e_1), cns(e_1))$
3 $d^* \leftarrow \min_{e_i \in E}(\lfloor del(e_i)/\gcd(prd(e_i), cns(e_i))\rfloor)$
4 $A_1 \leftarrow v_{src}$, $B_1 \leftarrow v_{snk}$, $p_1 \leftarrow p^*$, $c_1 \leftarrow c^*$, $m_{A_1} \leftarrow 1$, $n_{A_1} \leftarrow 0$, $m_{B_1} \leftarrow 0$, $n_{B_1} \leftarrow 1$
5 $i \leftarrow 1$
6 **while !**($p_i \bmod c_i = 0$ **or** $c_i \bmod p_i = 0$)
7 **if** $p_i > c_i$
8 $A_{i+1} \leftarrow (1\ A_i\ (\lfloor p_i/c_i \rfloor \times B_i))$, $p_{i+1} \leftarrow p_i \bmod c_i$
9 $m_{A_{i+1}} \leftarrow m_{A_i} + \lfloor p_i/c_i \rfloor \times m_{B_i}$, $n_{A_{i+1}} \leftarrow n_{A_i} + \lfloor p_i/c_i \rfloor \times n_{B_i}$
10 $B_{i+1} \leftarrow (1\ A_i\ (\lceil p_i/c_i \rceil \times B_i))$, $c_{i+1} \leftarrow c_i - p_i \bmod c_i$
11 $m_{B_{i+1}} \leftarrow m_{A_i} + \lceil p_i/c_i \rceil \times m_{B_i}$, $n_{B_{i+1}} \leftarrow n_{A_i} + \lceil p_i/c_i \rceil \times n_{B_i}$
12 **else**
13 $A_{i+1} \leftarrow (1\ (\lceil c_i/p_i \rceil \times A_i)\ B_i)$, $p_{i+1} \leftarrow p_i - c_i \bmod p_i$
14 $m_{A_{i+1}} \leftarrow \lceil c_i/p_i \rceil \times m_{A_i} + m_{B_i}$, $n_{A_{i+1}} \leftarrow \lceil c_i/p_i \rceil \times n_{A_i} + n_{B_i}$
15 $B_{i+1} \leftarrow (1\ (\lfloor c_i/p_i \rfloor \times A_i)\ B_i)$, $c_{i+1} \leftarrow c_i \bmod p_i$
16 $m_{B_{i+1}} \leftarrow \lfloor c_i/p_i \rfloor \times m_{A_i} + m_{B_i}$, $n_{B_{i+1}} \leftarrow \lfloor c_i/p_i \rfloor \times n_{A_i} + n_{B_i}$
17 **end**
18 $i \leftarrow i + 1$
19 **end**
20 $I \leftarrow i$
21 **if** $d^* = 0$
22 **if** $p_I > c_I$ $S \leftarrow A_I\ (p_I/c_I \times B_I)$ **else** $S \leftarrow (c_I/p_I \times A_I)\ B_I$ **end**
23 **return** S
24 **else**
25 $S \leftarrow \varnothing$, $d \leftarrow d^*$, $m \leftarrow c^*$, $n \leftarrow p^*$
26 **for** i from 1 to I
27 **if** $d \geq c_i$
28 **if** $(x \leftarrow \lfloor d/c_i \rfloor) \times n_{B_i} > n$ **break** **end**
29 $S \leftarrow S\ (x \times B_i)$, $d \leftarrow d \bmod c_i$, $m \leftarrow m - x \times m_{B_i}$, $n \leftarrow n - x \times n_{B_i}$
30 **if** $d = 0$ **break** **end**
31 **end**
32 **if** $d \geq p_i$
33 **if** $(x \leftarrow \lceil (c_i - d)/p_i \rceil) \times n_{A_i} + n_{B_i} > n$ **break** **end**
34 $S \leftarrow S\ (x \times A_i)\ B_i$, $d \leftarrow d + x \times p_i - c_i$
35 $m \leftarrow m - x \times m_{A_i} - m_{B_i}$, $n \leftarrow n - x \times n_{A_i} - n_{B_i}$
36 **if** $d = 0$ **break** **end**
37 **end**
38 **end**
39 **for** i from I to 1
40 $x \leftarrow \min(\lfloor m/m_{A_i} \rfloor, \lfloor n/n_{A_i} \rfloor, \lceil (c_i - d)/p_i \rceil)$
41 **if** $x \geq 1$
42 $S \leftarrow S\ (x \times A_i)$, $d \leftarrow d + x \times p_i$, $m \leftarrow m - x \times m_{A_i}$, $n \leftarrow n - x \times n_{A_i}$
43 **end**
44 $x \leftarrow \min(\lfloor m/m_{B_i} \rfloor, \lfloor n/n_{B_i} \rfloor, \lfloor d/c_i \rfloor)$
45 **if** $x \geq 1$
46 $S \leftarrow S\ (x \times B_i)$, $d \leftarrow d - x \times c_i$, $m \leftarrow m - x \times m_{B_i}$, $n \leftarrow n - x \times n_{B_i}$
47 **end**
48 **end**
49 $S \leftarrow S\ (m \times A_1)$
50 **return** S
51 **end**

Fig. 6. Buffer-optimal two-actor scheduling (BOTAS) algorithm.

Table I. Demonstration of Buffer-Optimal Two-Actor Scheduling

Computing scheduling components for $p^* = 7$ $c^* = 5$					
lines	i	A_i	B_i	p_i	c_i
4-20	1	a	b	7	5
	2	$(1\ A_1\ B_1)$	$(1\ A_1\ (2\ B_1))$	2	3
	3	$(1\ (2\ A_2)\ B_2)$	$(1\ A_2\ B_2)$	1	1

Two-actor scheduling for $d^* = 0$	
lines	S
22-23	$A_3B_3 = (1(2ab)(1a(2b)))(1(1ab)(1a(2b)))$

Two-actor scheduling for $d^* = 6$		
lines	i	S
25-38	1	$B_1 = b$
	3	$S\ B_3 = b(1(1ab)(1a(2b)))$
39-48	2	$S\ (2A_2) = b(1(1ab)(1a(2b)))(2ab)$
	1	$S\ A_1B_1 = b(1(1ab)(1a(2b)))(2ab)ab$

Two-actor scheduling for $d^* = 12$		
lines	i	S
25-38	1	$(2B_1) = (2b)$
	2	$S\ A_2B_2 = (2b)(1ab)(1a(2b))$
39-48	2	$S\ A_2 = (2b)(1ab)(1a(2b))(1ab)$
	1	$S\ A_1B_1 = (2b)(1ab)(1a(2b))(1ab)ab$
49	N/A	$S\ A_1 = (2b)(1ab)(1a(2b))(1ab)aba$

For $p^* = 7$, $c^* = 5$, and $d^* = 0, 6, 12$, where $a = v_{src}$ and $b = v_{snk}$.

derive that $\gcd(p_{i+1}, c_{i+1}) = 1$.

$$\begin{aligned}\gcd(p_i, c_i) = 1 &\Rightarrow \gcd(c_i, p_i \mod c_i) = 1 \text{ and } \gcd(c_i, c_i - p_i \mod c_i) = 1 \\ &\Rightarrow \gcd(p_{i+1}, c_{i+1}) = 1\end{aligned} \tag{7}$$

$$\begin{aligned}\gcd(c_i, p_i) = 1 &\Rightarrow \gcd(p_i, c_i \mod p_i) = 1 \text{ and } \gcd(p_i, p_i - c_i \mod p_i) = 1 \\ &\Rightarrow \gcd(c_{i+1}, p_{i+1}) = 1\end{aligned} \tag{8}$$

By mathematical induction, $\forall i \in \{1, 2, \ldots, I\}$, $\gcd(p_i, c_i) = 1$. □

Directly from Property 4.16, we can derive the following termination property.

PROPERTY 4.17. *In the BOTAS algorithm, the while-loop in line* 6 *terminates when either* $p_i = 1$ *or* $c_i = 1$.

The following lemma determines a bound on the number of iterations I of the while-loop in line 6. In practical cases, I is usually much smaller than the bound.

LEMMA 4.18. *In the BOTAS algorithm, the iteration number I that terminates the while-loop in line* 6 *is bounded by* $\log_2 \min(p^*, c^*)$.

PROOF. From Property 4.15, it follows that $\min(p_{i+1}, c_{i+1}) \leq \min(p_i, c_i)/2$. Because the while-loop ends when $p_i = 1$ or $c_i = 1$ (Property 4.17), it takes at most $\log_2 \min(p_1, c_1)$ iterations to achieve $p_i = 1$ or $c_i = 1$. Therefore, the iteration $i = I$ that terminates the while-loop is bounded by $\log_2 \min(p^*, c^*)$. □

Finally, we establish the correctness, optimality, and complexity of the BOTAS algorithm in Theorem 4.19, Property 4.20, and Property 4.21.

THEOREM 4.19. *In the BOTAS algorithm, suppose there are no initial tokens on edge e^* ($d^* = 0$), and define the schedule $S = A_I\ (p_I/c_I \times B_I)$ if $p_I > c_I$, and $S = (c_I/p_I \times A_I)B_I$, otherwise (line 22). Then S is an SASAP schedule for G^*.*

PROOF. From Property 4.14, it follows that S fires v_{snk} ASAP. We then show that S fires v_{src} c^* times and v_{snk} p^* times to prove that S is an SASAP schedule.

For $p_I > c_I$ and $p_{I-1} > c_{I-1}$, because $p_I = p_{I-1} \bmod c_{I-1}$, $c_I = c_{I-1} - p_{I-1} \bmod c_{I-1} = 1$, $\lceil p_{I-1}/c_{I-1} \rceil = \lfloor p_{I-1}/c_{I-1} \rfloor + 1$, $S = A_I\ (p_I/c_I\ B_I)$, $A_I = (1\ A_{I-1}\ (\lfloor p_{I-1}/c_{I-1} \rfloor\ B_{I-1}))$, and $B_I = (1\ A_{I-1}\ (\lceil p_{I-1}/c_{I-1} \rceil\ B_{I-1}))$, we can derive that S fires A_{I-1}

$$1 + p_I/c_I = 1 + c_{I-1} - 1 = c_{I-1} \text{ times,}$$

and S fires B_{I-1}

$$\begin{aligned}&\lfloor p_{I-1}/c_{I-1} \rfloor + p_I/c_I \times \lceil p_{I-1}/c_{I-1} \rceil \\ &\quad = \lfloor p_{I-1}/c_{I-1} \rfloor + (c_{I-1} - 1) \times \lfloor p_{I-1}/c_{I-1} \rfloor + p_{I-1} \bmod c_{I-1} = p_{I-1} \text{ times.}\end{aligned}$$

By using a similar approach, we can derive that S fires A_{I-1} c_{I-1} times and fires B_{I-1} p_{I-1} times for the following cases: (a) $p_I > c_I$ and $p_{I-1} < c_{I-1}$; (b) $p_I \le c_I$ and $p_{I-1} > c_{I-1}$; and (c) $p_I \le c_I$ and $p_{I-1} < c_{I-1}$.

Now, suppose S fires A_i c_i times and fires B_i p_i times for some $i \in \{2, 3, \ldots, I - 1\}$. Then, if $p_{i-1} > c_{i-1}$, we can derive that S fires A_{i-1}

$$(c_{i-1} - p_{i-1} \bmod c_{i-1}) + (p_{i-1} \bmod c_{i-1}) = c_{i-1} \text{ times,}$$

and S fires B_{i-1}

$$\begin{aligned}&(c_{i-1} - p_{i-1} \bmod c_{i-1}) \times \lfloor p_{i-1}/c_{i-1} \rfloor + (p_{i-1} \bmod c_{i-1}) \times \lceil p_{i-1}/c_{i-1} \rceil \\ &\quad = c_{i-1} \times \lfloor p_{i-1}/c_{i-1} \rfloor + p_{i-1} \bmod c_{i-1} = p_{i-1} \text{ times}\end{aligned}$$

because $A_i = (1\ A_{i-1}\ (\lfloor p_{i-1}/c_{i-1} \rfloor\ B_{i-1}))$, $B_i = (1\ A_{i-1}\ (\lceil p_{i-1}/c_{i-1} \rceil\ B_{i-1}))$, $p_i = p_{i-1} \bmod c_{i-1}$, and $c_i = c_{i-1} - p_{i-1} \bmod c_{i-1}$.

In a similar way, when $p_{i-1} < c_{i-1}$, we can derive that S fires A_{i-1} c_{i-1} times and fires B_{i-1} p_{i-1} times.

As a result, if S fires A_i c_i times and fires B_i p_i times for some $i \in \{2, 3, \ldots, I - 1\}$, then S fires A_{i-1} c_{i-1} times and fires B_{i-1} p_{i-1} times. Because we have proved that S fires A_{I-1} c_{I-1} times and fires B_{I-1} p_{I-1} times, we can conclude by mathematical induction that S fires A_i c_i times and fires B_i p_i times for every $i \in \{1, 2, \ldots, I - 1\}$. Taking $i = 1$, we have that S fires v_{src} c^* times and fires v_{snk} p^* times. □

Using demand-driven analysis on the state of $e*$ (i.e., d in BOTAS) and the number of remaining firings of v_{src} and v_{snk} (i.e., m and n in BOTAS), the following result can be shown. This result, together with Theorem 4.19, establishes the correctness of the BOTAS algorithm.

PROPERTY 4.20. *Suppose that there are d^* initial tokens on edge e^* ($d^* > 0$). Then, the BOTAS algorithm in lines 25–49 constructs an SASAP schedule for G^*.*

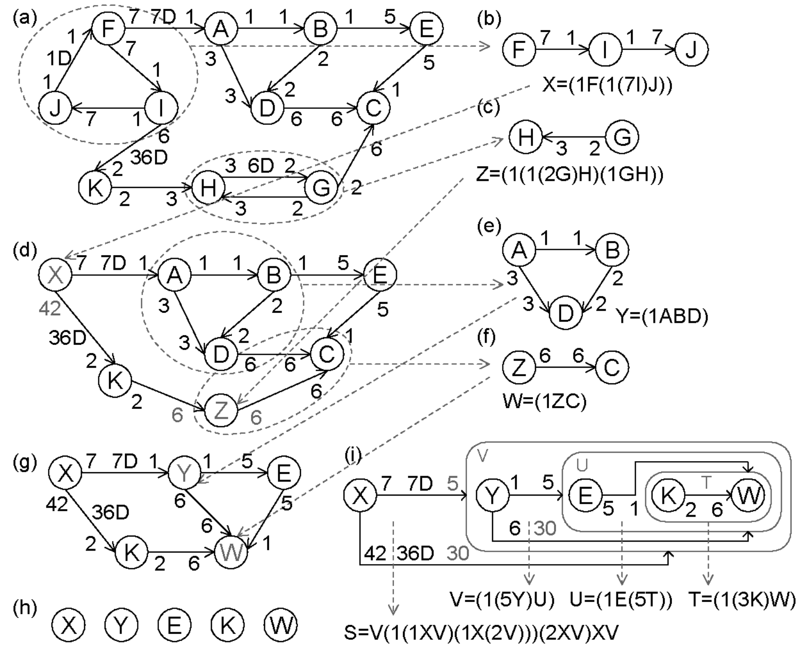

Fig. 7. SOS scheduling example.

PROPERTY 4.21. *The complexity of the BOTAS algorithm is* $O(\log_2 \min(p^*, c^*))$.

PROOF. From Lemma 4.18, the iteration I that ends the while-loop in line 6 is bounded by $\log_2 \min(p^*, c^*)$. The first for loop (lines 26–38) and the second for loop (lines 39–48) are both bounded by I. All other operations can be implemented in constant time. As a result, the complexity of the BOTAS algorithm is $O(\log_2 \min(p^*, c^*))$. □

4.9 Overall Integration

The overall integration of component algorithms in SOS is illustrated in Figure 1. A major contribution of this work is the selection, adaptation, and integration of these algorithms—along with development of associated theory and analysis—into a complete simulation environment for the novel constraints associated with simulating critical SDF graphs. In fact, the complexity involved in the overall SOS approach is dominated by the complexity of scheduling the subgraphs that it isolates in its top-down process of LIAF- and SRC-based decomposition. For this reason, we are able to apply the intensive APGAN, DPPO, and buffer-optimal two-actor scheduling algorithms in SOS without major degradation in simulation performance. This is beneficial because these intensive techniques provide significant reductions in the total buffer requirement.

Figure 7 presents an example to illustrate SOS. Given a connected, consistent SDF graph (e.g., Figure 7(a)), SOS first applies LIAF (Section 4.2) to decompose all strongly connected components in order to derive an acyclic SDF graph (as

Table II. Characteristics of Wireless Communication Designs

Design #	Description	Number of Actors	Number of Edges (single-/multirate)	Multirate Complexity
1	3GPP Uplink Source	82	133 (101/32)	1.86 E6
2	3GPP Downlink Source	179	236 (194/42)	1.10 E6
3	Bluetooth Packets	104	107 (97/10)	808
4	802.16e Source Constellation	71	73 (49/24)	9.95 E6
5	CDMA2000 Digital Distortion	707	855 (805/50)	3.83 E6
6	XM Radio	269	293 (245/48)	5.43 E6
7	Edge Signal Source	186	222 (192/30)	36.36 E6
8	Digital TV	114	126 (74/52)	1.37 E6
9	WiMax Downlink Source	368	389 (276/113)	73191

illustrated in Figure 7(d)) and break cycles for strongly connected subgraphs (as illustrated in Figures 7(b) and 7(c)). If a subgraph is loosely interdependent, LIAF is applied recursively to derive a schedule for the subgraph (e.g., $(1F(1(7I)J))$ for Figure 7(b) and $(1(1(2G)H)(1GH))$ for Figure 7(c)).

For the acyclic graph, SOS applies SRC (Section 4.4) to isolate single-rate subgraphs, and to reduce the acyclic graph into a smaller multirate version. This is illustrated in Figure 7(g). For single-rate subgraphs (e.g., Figures 7(e) and 7(f)), SOS efficiently computes schedules (e.g., $(1ABD)$ for Figure 7(e) and $(1ZC)$ for Figure 7(f)) by the flat scheduling approach (Section 4.5).

After SRC, SOS uses APGAN (Section 4.6) to obtain a buffer-efficient topological sort (e.g., Figure 7(h)) for the multirate, acyclic graph. Then, from the topological sort, SOS applies DPPO (Section 4.7) to construct a buffer-efficient two-actor hierarchy. This is illustrated in Figure 7(i). Finally, SOS computes a buffer-optimal schedule for each two-actor subgraph based on the two-actor scheduling algorithm (Section 4.8), for example, $(1(3K)W)$ for the two-actor subgraph $\{K, W\}$, $(1E(5T))$ for the subgraph $\{E, T\}$, $(1(5Y)U)$ for the subgraph $\{Y, U\}$, and $V(1(1XV)(1X(2V)))(2XV)XV$ for the top-level two-actor graph $\{X, V\}$. An overall schedule is then obtained by traversing the constructed hierarchies and replacing supernodes with the corresponding subschedules.

5. SIMULATION RESULTS

We have implemented and integrated the simulation-oriented scheduler in Agilent ADS [Pino and Kalbasi 1998]. Here, we demonstrate our simulation-oriented scheduler by scheduling and simulating state-of-the-art wireless communication systems in ADS. However, the design of SOS is not specific to ADS, and the techniques presented in this article can be generally implemented in any simulation tool that incorporates SDF semantics.

The experimental platform is a PC with a 1GHz CPU and 1GB of memory. In our experiments, we include 9 wireless communication designs from Agilent Technologies in the following standards: 3GPP (WCDMA3G), Bluetooth, 802.16e (WiMax), CDMA 2000, XM radio, EDGE, and Digital TV. Table II presents characteristics of the 9 designs, including the number of actors, number of edges (single-rate/multirate), and approximate multirate complexities.

Table III. Total Buffer Requirements (tokens)

Design	CLS	SOS	Ratio (CLS/SOS)
1	50445629	229119	220
2	9073282	43247	210
3	3090	3090	1
4	89428569	669273	134
5	OOM-sc	9957292	N/A
6	48212523	5385031	9
7	1870248382	451862	4139
8	8257858	1976318	4
9	1834606	1832926	1

Table IV. Average Scheduling Time (seconds)

Design	CLS	SOS	Ratio (CLS/SOS)
1	0.08	0.08	1.00
2	279.11	0.16	1744.44
3	0.06	0.06	1.00
4	0.49	0.45	1.09
5	OOM-sc	13.50	N/A
6	10.72	0.67	16.00
7	0.92	0.87	1.06
8	2.73	0.53	5.15
9	3.59	9.98	0.36

These designs contain from several tens to hundreds of actors and edges, and possess very high multirate complexities. In particular, the multirate complexities in designs 1, 2, 4, 5, 6, 7, and 8 are in the range of millions.

We simulate the 9 designs with our simulation-oriented scheduler (SOS), and the present default cluster-loop scheduler (CLS) in ADS. The simulation results of CLS, SOS, and the performance ratio (CLS/SOS) are shown in three tables: Table III presents the total buffer requirements for SDF edges (in number of tokens); Table IV presents the average scheduling time of ten runs (in seconds); and Table V presents the average total simulation time of ten runs (in seconds). As shown in these tables, SOS outperforms CLS in almost all designs in terms of memory requirements, scheduling time, and total simulation time (except in designs 3 and 9, which are comparable due to their relatively small multirate complexities). In particular, SOS is effective in reducing the buffer requirements within short scheduling time. For design 9, CLS requires less scheduling time because of its capabilities as a fast heuristic. However, for design 2, it requires a very long scheduling time due to its heavy dependence on classical SDF scheduling. CLS fails in design 5 due to an out-of-memory problem during scheduling (OOM-sc), and also fails in designs 1, 4, 6, and 7 due to out-of-memory problems in buffer allocation (OOM-ba). With SOS, we are able to simulate these heavily multirate designs.

Table V. Average Total Simulation Time (seconds)

Design	CLS	SOS	Ratio (CLS/SOS)
1	OOM-ba	7.12	N/A
2	349.33	55.31	6.32
3	930.56	876.72	1.06
4	OOM-ba	203.95	N/A
5	OOM-sc	2534.06	N/A
6	OOM-ba	406.86	N/A
7	OOM-ba	28940.77	N/A
8	636.63	415.40	1.53
9	1566.92	1542.39	1.02

6. CONCLUSION

In this article, we have introduced and illustrated the challenges in scheduling large-scale, highly multirate synchronous dataflow (SDF) graphs for simulation tools that incorporate SDF semantics. We have defined critical SDF graphs as an important class of graphs that must be taken carefully into consideration when designing such tools for modeling and simulating modern large-scale and heavily multirate communication and signal processing systems. We have then presented the simulation-oriented scheduler (SOS). SOS integrates several existing and newly developed graph decomposition and scheduling techniques in a strategic way for joint runtime and memory minimization in simulating critical SDF graphs. We have demonstrated the efficiency of our scheduler by simulating practical, large-scale, and highly multirate wireless communication designs.

GLOSSARY

$buf(e)$	The buffer size required for an SDF edge e.
$cns(e)$	The consumption rate of an SDF edge e.
$del(e)$	The number of initial tokens on an SDF edge e.
$in(v)$	The set of input edges of an actor v.
$MC(G)$	The multirate complexity of an SDF graph G.
$out(v)$	The set of output edges of an actor v.
$prd(e)$	The production rate of an SDF edge e.
$snk(e)$	The sink actor of an SDF edge e.
$src(e)$	The source actor of an SDF edge e.
$tok_G(S,e,k)$	The number of tokens on an edge e in the SDF graph G immediately after the actor firing associated with firing index k in the schedule S.
$\sigma(S,k)$	The actor associated with firing index k in the schedule S.
$\tau(S,v,k)$	The firing count of actor v up to firing index k in the schedule S.

REFERENCES

ADE, M., LAUWEREINS, R., AND PEPERSTRAETE, J. A. 1997. Data memory minimization for synchronous data flow graphs emulated on DSP-FPGA targets. In *Proceedings of the Design Automation Conference*.

BHATTACHARYYA, S. S., LEUPERS, R., AND MARWEDEL, P. 2000. Software synthesis and code generation for DSP. *IEEE Trans. Circ. Syst. II: Analog Digital Signal Process. 47,* 9 (Sept.), 849–875.

BHATTACHARYYA, S. S., MURTHY, P. K., AND LEE, E. A. 1996. *Software Synthesis from Dataflow Graphs*. Kluwer Academic, Hingham, MA.

BILSEN, G., ENGELS, M., LAUWEREINS, R., AND PEPERSTRAETE, J. A. 1996. Cyclo-Static dataflow. *IEEE Trans. Signal Process. 44*, 2 (Feb.), 397–408.

BUCK, J. T. 1993. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. thesis UCB/ERL 93/69, Department of EECS, U. C. Berkeley.

BUCK, J. T., HA, S., LEE, E. A., AND MESSERSCHMITT, D. G. 1994. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. J. Comput. Simul. 4*, 155–182.

BUCK, J. T. AND VAIDYANATHAN, R. 2000. Heterogeneous modeling and simulation of embedded systems in El Greco. In *Proceedings of the International Workshop on Hardware/Software Codesign* (San Diego, CA).

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms*, 2nd ed. The MIT Press, Cambridge, MA.

CUBRIC, M. AND PANANGADEN, P. 1993. Minimal memory schedules for dataflow networks. In *Proceedings of the International Conference on Concurrency Theory (CONCUR)*, 368–383.

EKER, J., JANNECK, J. W., LEE, E. A., LIU, J., LIU, X., LUDVIG, J., NEUENDORFFER, S., SACHS, S., AND XIONG, Y. 2003. Taming heterogeneity—The Ptolemy approach. *Proc. IEEE 91,* 1 (Jan.), 127–144.

GEILEN, M., BASTEN, T., AND STUIJK, S. 2005. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Proceedings of the Design Automation Conference* (Anaheim, CA), 819–824.

HSU, C. AND BHATTACHARYYA, S. S. 2007. Cycle-Breaking techniques for scheduling synchronous dataflow graphs. Tech. Rep. UMIACS-TR-2007-12, Institute for Advanced Computer Studies, University of Maryland at College Park.

HSU, C., KO, M., AND BHATTACHARYYA, S. S. 2005. Software synthesis from the dataflow interchange format. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems* (Dallas, TX), 37–49.

HSU, C., RAMASUBBU, S., KO, M., PINO, J. L., AND BHATTACHARYYA, S. S. 2006. Efficient simulation of critical synchronous dataflow graphs. In *Proceedings of the Design Automation Conference* (San Francisco, CA), 893–898.

KO, M., MURTHY, P. K., AND BHATTACHARYYA, S. S. 2004. Compact procedural implementation in DSP software synthesis through recursive graph decomposition. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems* (Amsterdam, The Netherlands), 47–61.

LAUWEREINS, R., ENGELS, M., ADE, M., AND PEPERSTRAETE, J. A. 1995. Grape-II: A system-level prototyping environment for DSP applications. *IEEE Comput. Mag. 28*, 2 (Feb.), 35–43.

LEE, E. A., HO, W. H., GOEI, E., BIER, J., AND BHATTACHARYYA, S. S. 1989. Gabriel: A design environment for DSP. *IEEE Trans. Acoustics, Speech, Signal Process. 37*, 11 (Nov.), 1751–1762.

LEE, E. A. AND MESSERSCHMITT, D. G. 1987. Synchronous dataflow. *Proc. IEEE 75*, 9 (Sept.), 1235–1245.

MARWEDEL, P. AND GOOSSENS, G., EDS. 1995. *Code Generation for Embedded Processors*. Kluwer Academic, Hinghan, MA.

MURTHY, P. K. AND BHATTACHARYYA, S. S. 2006. *Memory Management for Synthesis of DSP Software*. CRC Press, Baca Raton, FL.

MURTHY, P. K., BHATTACHARYYA, S. S., AND LEE, E. A. 1997. Joint minimization of code and data for synchronous dataflow programs. *J. Formal Methods Syst. Des. 11*, 1 (Jul.), 41–70.

OH, H., DUTT, N., AND HA, S. 2006. Memory optimal single appearance schedule with dynamic loop count for synchronous dataflow graphs. In *Proceedings of the Asia and South Pacific Design Automation Conference* (Yokohama, Japan), 497–502.

PATEL, H. D. AND SHUKLA, S. K. 2004. *SystemC Kernel Extensions for Heterogeneous System Modeling*. Kluwer Academic, Hingham, MA.

PINO, J. L. AND KALBASI, K. 1998. Cosimulating synchronous DSP applications with analog RF circuits. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers* (Pacific Grove, CA).

ROBBINS, C. B. 2002. Autocoding toolset software tools for automatic generation of parallel application software. Tech. Rep., Management Communications and Control, Inc.

STEFANOV, T., ZISSULESCU, C., TURJAN, A., KIENHUIS, B., AND DEPRETTERE, E. 2004. System design using Kahn process networks: The Compaan/Laura approach. In *Proceedings of the Design, Automation and Test in Europe Conference*.

SUNG, W., OH, M., IM, C., AND HA, S. 1997. Demonstration of hardware software codesign workflow in PeaCE. In *Proceedings of the International Conference on VLSI and CAD*.

Received September 2006; revised March 2007; accepted March 2007