Technical Report #UMIACS-TR-2007-12, Institute for Advanced Computer Studies,
University of Maryland, College Park, MD 20742, February 2007

Cycle-Breaking Techniques for Scheduling Synchronous Dataflow Graphs

CHIA-JUI HSU and SHUVRA S. BHATTACHARYYA
Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742
{jerryhsu, ssb}@umd.edu

System-level modeling, simulation, and synthesis using the synchronous dataflow (SDF) model of computation is widespread in design automation for communication and digital signal processing (DSP) systems. SDF scheduling has a large impact on the performance and memory requirements of implementations. One of the major problems in scheduling SDF graphs is that the existence of cycles in the targeted systems prevents or greatly restricts application of many useful optimization techniques that are available for acyclic SDF graphs. The *loose interdependence algorithms framework* (LIAF) has been developed to decompose cycles in SDF graphs into hierarchies of acyclic subgraphs whenever possible. However, LIAF does not specify any specific algorithm to break cycles, but rather it specifies the constraints that such an algorithm must satisfy. In this report, we present a low-complexity (linear-time) *cycle-breaking* algorithm for decomposing and breaking cycles in SDF graphs so that subsequent acyclic scheduling techniques can be easily applied. We also present a technique for computing buffer bounds on edges that are removed during the cycle-breaking process so that buffer sizes can be computed for such edges. We have implemented our cycle-breaking algorithm and buffer size computation technique in the state-of-art *simulation-oriented scheduler*, and our results demonstrate the effectiveness of these techniques.

1. INTRODUCTION

Modeling systems using synchronous dataflow (SDF) [Lee and Messerschmitt 1987] is widespread in design automation tools for design of communication and signal processing systems. 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 a 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 non-negative 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 sufficient numbers of data values (tokens) on all of its input edges $in(v)$. When firing, v consumes certain numbers of tokens from its input edges, executes its computation, and produces certain numbers 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 is denoted as $prd(e)$ ($cns(e)$).

Before execution, a *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*:

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

When it exists, the minimum positive integer solution for the vector $\boldsymbol{x}$ is called the *repetitions vector* of G, and is 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 report) 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.

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* (SAS), otherwise, S is called a *multiple appearance schedule* (MAS).

SDF clustering is an important scheduling operation. 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, e.g., by using different optimization techniques. SDF clustering 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 [Bhattacharyya et al. 1996].

Once a schedule is determined, 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, $buf(e)$, 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 the buffer sizes of all edges:

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

Generally, the design space for 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, and scheduling in this context has been addressed extensively in the literature (see Section 2). On the other hand, for system simulation, simulation time (including time for scheduling and execution) is the primary objective. In the simulation context, we have developed the *simulation-oriented scheduler* (SOS) [Hsu et al. 2006] that strategically integrates several techniques for joint minimization of time and memory requirements in simulating complex, multirate SDF graphs.

In both contexts (synthesis and simulation), presence of cycles in the targeted systems generally complicates the scheduling problem because cyclic data dependences must be taken care of in order to prevent deadlock. In addition, existence of cycles in SDF graphs prevents or greatly restricts application of many useful optimization techniques that are available for acyclic SDF graphs. Bhattacharyya et al. [1996] have presented the *loose interdependence algorithm framework* (LIAF) to construct SASs whenever they exist. The LIAF framework applies to all consistent SDF graphs, whether or not cycles are present. Even though the original motivation of LIAF is toward code-size minimization in software synthesis [Bhattacharyya et al. 2000], the concept of decomposing cycles in SDF graphs into hierarchies of acyclic subgraphs is applicable to other useful contexts as well. LIAF has been incorporated into the Ptolemy environment for design of heterogeneous embedded systems [Buck et al. 1994], the DIF (dataflow interchange format) package for dataflow-based design and synthesis [Hsu et al. 2004; Hsu et al. 2005], and the SOS scheduler for simulation of complex, multirate signal processing systems [Hsu et al. 2006].

LIAF does not specify any specific algorithm for breaking cycles, but rather specifies what kinds of edges can be removed so that scheduling without considering these edges does not deadlock the graph. In this report, we present a novel cycle-breaking algorithm that can easily be incorporated into LIAF for decomposing and breaking cycles . A key feature of our cycle-breaking algorithm is its low complexity, which is important for use in SOS, as well as in other environments where scheduling runtime is critical. In particular, our cycle breaking technique runs in time that is linear in the number of actors and edges in the input SDF graph.

In both synthesis and simulation contexts, computing buffer sizes of SDF edges is needed to statically allocate memory space to edge buffers. Even though scheduling acyclic graphs that emerge from the LIAF decomposition process without considering the removed edges never violates data precedence constraints, buffer sizes of the removed edges should still be properly computed based on the scheduling results. Otherwise, the graph may deadlock or produce memory corruption during

execution due to buffer overflow. In this report, we analyze the buffer bounds on edges that are removed by cycle breaking so that the buffer sizes of these edges can be set efficiently.

The organization of the report is as follows: In Section 2, we review related work. We then review the LIAF framework in Section 3. In Section 4, we present the cycle-breaking algorithm as well as some associated theory. In Section 5, we analyze the buffer bounds on edges that are removed by the cycle-breaking process.

2. RELATED WORK

Various scheduling algorithms and techniques have been developed for different applications of SDF graphs. In general, the problem of computing a buffer-optimal SDF schedule is NP-complete. Bhattacharyya et al. [1996] has presented a heuristic for minimum buffer scheduling. A simpler variant of this algorithm has been used in both the Gabriel [Lee et al. 1989] and Ptolemy [Buck et al. 1994] environments. We refer to these demand-driven, minimum-buffer scheduling heuristics as *classical SDF scheduling*. This form of scheduling is effective at reducing total buffer requirements, but its time complexity, and the lengths of its resulting schedules (which are usually MASs) generally grow exponentially in the size of multirate SDF graphs.

An SAS is often preferable due to its optimally compact implementation containing only a single copy of code for every actor. A valid SAS exists for any consistent, acyclic SDF graph and can be easily derived from *flat scheduling*, which is a strategy that computes a topological sort of the SDF graph and iterates each actor based on its repetition count. However, flat scheduling may also lead to relatively large buffer requirements and latencies in multirate systems [Bhattacharyya et al. 1996].

For joint code and data minimization, several scheduling algorithms have been developed for acyclic SDF graphs to minimize data memory requirements within SASs. The *dynamic programming post optimization* (DPPO) [Bhattacharyya et al. 1996] performs dynamic programming over a given actor ordering (topological sort) to generate a buffer-efficient looped schedule. It has several forms for different cost functions, e.g., GDPPO [Bhattacharyya et al. 1996], CDPPO [Zitzler et al. 2000], and SDPPO [Murthy and Bhattacharyya 2001]. The *acyclic pairwise grouping of adjacent nodes* (APGAN) [Bhattacharyya et al. 1996] technique is an adaptable (to different cost functions), low-complexity heuristic that generates a looped schedule and an embedded topological sort.

Beyond SASs, the work of [Ko et al. 2004] presents a *recursive procedure call* (RPC) based technique for software synthesis that generates MASs for acyclic SDF graphs through recursive graph decomposition. For delayless SDF graphs, the resulting schedules are proven to be buffer-optimal at each (two-actor) level of the cluster hierarchy, and also polynomially bounded in the graph size. This technique does not optimally handle the scheduling flexibility provided by edge delays, and therefore, it does not always achieve minimum buffer sizes in presence of delays. In [Hsu et al. 2007], we have then developed the *buffer-optimal two-actor scheduling* algorithm that computes a buffer-optimal schedule for a general (with or without delays), acyclic, two-actor SDF graph. This algorithm has been integrated in the simulation-oriented scheduler (SOS) [Hsu et al. 2006] to schedule nested two-actor graphs in the cluster hierarchy.

The aforementioned acyclic SDF scheduling algorithms can be integrated with LIAF to schedule acyclic graphs that emerge from the LIAF decomposition process. In fact, the SOS approach [Hsu et al. 2006] adapts and integrates LIAF and our cycle-breaking algorithm with a variety of acyclic scheduling techniques in new ways that efficiently address the novel constraint of highly multirate simulation.

3. LOOSE INTERDEPENDENCE ALGORITHMS FRAMEWORK

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 operate on acyclic graphs.

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 *inter-iteration edge* in our context.

Now suppose that ${G_i}^*$ denotes the graph that results from removing all inter-iteration edges from the strongly connected subgraph G_i. 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.

In general application of LIAF, tightly interdependent subgraphs are scheduled by *classical SDF scheduling*. As discussed in Section 2, classical SDF scheduling is a demand-driven, minimum-buffer scheduling heuristic, but its complexity is not polynomially-bounded in the size of the input graph. Fortunately, this does not cause any major practical limitation because tightly interdependent subgraphs rarely arise in practice [Bhattacharyya et al. 1996].

The acyclic graphs that emerge from LIAF decomposition can be further processed by acyclic scheduling techniques such as *single-rate clustering* [Hsu et al. 2006], flat scheduling, APGAN, DPPO, and buffer-optimal two-actor scheduling, as we discussed in Section 2. For more details, we refer the reader to [Hsu et al. 2007].

4. CYCLE-BREAKING

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 that is used to break cycles, but rather specifies the constraints that such an algorithm must satisfy so that schedulers derived from the framework

[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.

```
CYCLE-BREAKING(G ≡ (V, E)) /*The input G is a strongly connected SDF graph*/
 1  E' ← ∅
 2  for e ∈ E
 3     if del(e) ≥ cns(e) × q_G[snk(e)]     E ← E − e,  E' ← E' + e     end
 4  end
 5  if IS-CONNECTED(G)
 6     {SCC_1, SCC_2, ..., SCC_N} ← TOPOLOGICALLY-SORTED-SCC(G)
 7     if N = 1     G is tightly interdependent, E ← E + E', ...
 8     else
 9        for e ∈ E'
10           if !(src(e) ∉ SCC_1 and snk(e) ∈ SCC_1)     E ← E + e,  E' ← E' − e     end
11        end
12        G is no longer strongly connected ...
13     end
14  else
15     {CC_1, CC_2, ..., CC_M} ← CONNECTED-COMPONENTS(G)
16     {SCC_1, SCC_2, ..., SCC_P} ← TOPOLOGICALLY-SORTED-SCC(G_CC1 ≡ (CC_1, E_CC1))
17     for e ∈ E'
18        if !(src(e) ∉ SCC_1 and snk(e) ∈ SCC_1)     E ← E + e,  E' ← E' − e     end
19     end
20     G is no longer strongly connected ...
21  end
```

Fig. 1: Cycle-breaking algorithm.

can construct single appearance schedules whenever they exist and satisfy other useful properties [Bhattacharyya et al. 1996; Bhattacharyya et al. 2000].

Figure 1 presents our cycle-breaking algorithm, which is designed for low complexty and to be better suited for the acyclic scheduling techniques that we discussed in Section 2.

In Figure 1, given a strongly connected SDF graph $G = (V, E)$ to CYCLE-BREAKING, we first remove all inter-iteration edges from G (lines 2-4). If G is connected (line 5), we compute the strongly connected components SCC_1, SCC_2, ..., SCC_N of G in topologically sorted order (line 6). By a topologically sorted order of SCCs, we mean a topological sort [2] of the acyclic graph that results from clustering the SCCs in G. In addition, for a vertex that does not belong to any SCC that contains at least two vertices, we say that this vertex is an SCC by itself.

If G is still strongly connected ($N = 1$ in line 7), we conclude that G is tightly interdependent; restore G to its original state (before any edge removals); and mark it for processing by the tightly interdependent scheduling techniques, e.g., classical scheduling discussed in Section 2. On the other hand, if G is connected, but not strongly connected ($N > 1$ in line 7), then we put all previously removed edges (which are stored in E') back in G, except edges from $\{V - SCC_1\}$ to SCC_1 (lines 9-11).

If G becomes disconnected after removing all inter-iteration edges (that is, if control passes to the *else* branch rooted at line 14), then we compute the connected components (CCs) $CC_1, CC_2, \ldots, CC_M$ (line 15). Here, $M > 1$, and the

[2] A topological sort of a directed acyclic graph $G = (V, E)$ is a linear ordering of V such that for every edge (u, v) in G, u appears before v in the ordering.

CCs can be ordered arbitrarily. Next, we compute the strongly connected components $SCC_1, SCC_2, \ldots, SCC_P$ ($P \geq 1$) in some topologically sorted order for one of the connected subgraphs $G_{CC_1} = \{CC_1, E_{CC_1}\}$ (line 16). Lastly, we return all previously-removed edges back to G, except edges from $\{V - SCC_1\}$ to SCC_1 (lines 17-19), and complete the process.

The following theorem proves the correctness of the cycle-breaking algorithm.

THEOREM 4.1. *Suppose a strongly connected SDF graph $G = (V, E)$ is applied as input to the CYCLE-BREAKING algorithm, then G is determined to be tightly interdependent in line 7, is determined (after modification) to not be strongly connected in line 12, or is determined (again, after modification) to not be strongly connected in line 20.*

PROOF. CASE I: In line 7, G is tightly interdependent because after removing all inter-iteration edges (lines 2-4), it is still strongly connected (line 6-7).

CASE II: Just after line 8, the modified version of G (after removing all inter-iteration edges) is connected and has $N > 1$ SCCs, $SCC_1, SCC_2, \ldots, SCC_N$, ordered in a topologically sorted fashion. We can then determine that G is not strongly connected (since there are $N > 1$ SCCs), and there is no edge from $\{V - SCC_1\}$ to SCC_1 (since SCC_1 is the first SCC in topologically sorted order). By putting back all previously removed edges, except edges from $\{V - SCC_1\}$ to SCC_1 (lines 9-11), the resulting graph G (line 12) is not strongly connected because for any $u \in SCC_1$ and $v \in \{V - SCC_1\}$, there is a path from u to v (since the original input G is strongly connected), but no path from v to u (since there is no edge from $\{V - SCC_1\}$ to SCC_1).

CASE III: Just after line 16, the modified version of G (after removing all inter-iteration edges) is disconnected, and SCC_1 here is the first SCC in topologically sorted order in the connected subgraph $G_{CC_1} \equiv \{CC_1, E_{CC_1}\}$. We can then derive that there is no edge from $\{V - SCC_1\}$ to SCC_1 (since there is no edge between CCs, and SCC_1 is the first SCC in topologically sorted order in G_{CC_1}). By putting back all previously removed edges, except edges from $\{V - SCC_1\}$ to SCC_1 (lines 17-19), the resulting graph G (line 20) is connected but not strongly connected because for any $u \in SCC_1$ and $v \in \{V - SCC_1\}$, there is a path from u to v (since the original input G is strongly connected), but no path from v to u (since there is no edge from $\{V - SCC_1\}$ to SCC_1). □

The following theorem establishes key properties provided by our CYCLE-BREAKING algorithm.

THEOREM 4.2. *If a loosely interdependent, strongly connected SDF graph $G = (V, E)$ is applied as input to the CYCLE-BREAKING algorithm, then the resulting graph G is connected. Also, suppose that $SCC'_1, SCC'_2, \ldots, SCC'_L$ are the $L > 1$ SCCs in **any** topologically sorted order of the resulting graph G (line 12 or line 20). Then the edges removed by the CYCLE-BREAKING algorithm are edges from $\{V - SCC'_1\}$ to SCC'_1. Furthermore, SCC'_1 is equal to SCC_1 in line 6 or line 16.*

PROOF. Continuing from the proof of Theorem 4.1 for both CASE II and CASE III, we can derive that 1) SCC_1 is a strongly connected component in the resulting

graph G; and for any $u \in SCC_1$ and $v \in \{V - SCC_1\}$, 2) there is a path from u to v, but 3) there is no path from v to u. As a result, the resulting graph G is connected. In addition, SCC_1 must be the first SCC in *any* topologically sorted order of the resulting graph G, i.e., $SCC_1 = SCC'_1$; and the removed edges, i.e., inter-iteration edges from $\{V - SCC_1\}$ to SCC_1, must be edges from succeeding SCCs, $SCC'_2, SCC'_3, \ldots, SCC'_L$, to the first SCC'_1 in the resulting graph G. □

The following theorem pertains to the complexity of our cycle-breaking algorithm.

THEOREM 4.3. *Given a strongly connected SDF graph $G = (V, E)$, the complexity of the CYCLE-BREAKING algorithm is $\Theta(|V| + |E|)$.*

PROOF. Determining whether a graph is connected (IS-CONNECTED) as well as computing connected components of a disconnected graph (CONNECTED-COMPONENTS) can be implemented in linear time (i.e., in time that is linear in the number of actors and edges in G). This can be done, for example, by using depth-first search. A linear time algorithm to compute SCCs of a directed graph in topologically sorted order (TOPOLOGICALLY-SORTED-SCC) can be found in [Cormen et al. 2001]. Computing the repetitions vector of an SDF graph can also be implemented in linear time [Bhattacharyya et al. 1996]. Furthermore, with efficient data structures, operations in lines 2-4, lines 9-11, and lines 17-19, can be implemented in linear time. As a result, the complexity of CYCLE-BREAKING is $\Theta(|V| + |E|)$. □

With the CYCLE-BREAKING algorithm, operations for 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.

5. BUFFERING FOR REMOVED INTER-ITERATION EDGES

In both hardware/software synthesis and system simulation, computing buffer sizes of dataflow edges is important for statically allocating memory space to edge buffers. Even though scheduling acyclic graphs that emerge from the LIAF decomposition process without considering the removed inter-iteration edges never violates data precedence constraints, buffer sizes of the removed edges should still be properly computed based on the scheduling results. Otherwise, during execution, the graph may deadlock or produce memory corruption due to buffer overflow. In this section, we analyze buffer bounds for inter-iteration edges that are removed by cycle-breaking. Our analysis here assumes that the acyclic graphs that emerge from LIAF are scheduled based on *R-schedules*.

A valid single appearance schedule S is an *R-schedule* if S and each of the nested schedule loops in S has *either* 1) a single iterand, and this single iterand is an actor, *or* 2) exactly two iterands, and these two iterands are schedule loops having coprime iteration counts [Bhattacharyya et al. 1996] . In general, an R-schedule can be viewed as providing a single appearance, minimal periodic schedule for each two-actor graph in the *R-hierarchy*. Here by R-hierarchy, we mean the *nested two-actor cluster hierarchy* that is obtained from the looped binary structure in the R-schedule.

A variety of single appearance scheduling techniques fall into the domain of R-schedules — for example, APGAN [Bhattacharyya et al. 1996], DPPO [Bhat-

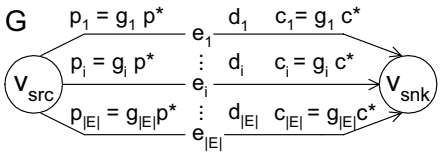

Fig. 2: Consistent, acyclic, two-actor SDF graph.

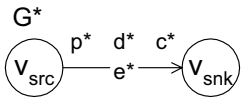

Fig. 3: Primitive two-actor SDF graph.

tacharyya et al. 1996], and RPMC [Murthy 1996]. Furthermore, Ko et al. [2004] has developed a technique that works on recursive, multiple appearance schedules of each two-actor graph in the R-hierarchy; and Hsu et al. [2006] has adapted and integrated this technique in the SOS approach.

Analysis of buffer bounds on the removed inter-iteration edges can be performed by studying the configuration of the removed edges in the R-hierarchy from the given R-schedule. Before such analysis, we first discuss properties of consistent, acyclic, two-actor SDF graphs that are useful for simplifying our analysis. These properties have also been presented in [Hsu et al. 2007].

PROPERTY 5.1. *A consistent, acyclic, two-actor SDF graph* $G = (\{v_{src}, v_{snk}\}, E)$ *has a general form as shown in Figure 2, 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 5.2 *Primitive Two-Actor SDF Graph.* Given a consistent, acyclic, two-actor SDF graph $G = (\{v_{src}, v_{snk}\}, E)$ as described in Property 5.1, its *primitive form* is defined as a two-actor, single-edge SDF graph $G^* = (\{v_{src}, v_{snk}\}, \{e^*\})$ as shown in Figure 3, 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.

The following lemma is useful in simplifying analysis for acyclic, two-actor SDF graphs. The proof can be found in [Hsu et al. 2007].

LEMMA 5.3. *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.

Next, we define some notation that is important to our analysis. Suppose that we are given a consistent SDF graph $G = (V, E)$ and a valid minimal periodic schedule S for G. By a *firing index* for S, we mean a non-negative 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. 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

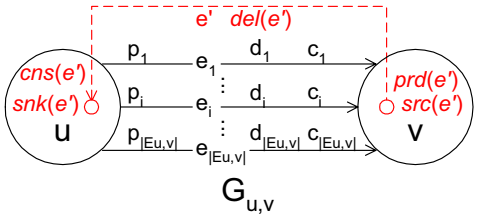

Fig. 4: Presence of cycle-broken edge in the two-actor graph.

let

$$tok_G(S, e, k) = \tau(S, src(e), k) \times prd(e) - \tau(S, snk(e), k) \times cns(e) + del(e) \quad (3)$$

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 $k = 0$, $\tau(S, v, 0)$ 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 τ and tok_G are uniquely determined by k, and are not dependent on the schedule iteration [Bhattacharyya et al. 1996].

Now, we develop our analysis of buffer bounds on the removed inter-iteration edges. Suppose that we are given a consistent, loosely interdependent, strongly connected SDF graph G. Suppose also that the CYCLE-BREAKING algorithm (see Section 4) removes a subset of inter-iteration edges E' from G, and suppose G' is the acyclic SDF graph that is constructed by clustering the SCCs of the resulting graph G. As described earlier, we assume that R-schedule techniques are applied to scheduling G'. Then we have the following observations: 1) By joint analysis of G' and the given R-schedule, a R-hierarchy H can always be constructed such that each two-actor graph in H is consistent and acyclic, and the order of the leaf actors encountered in depth-first, source-to-sink traversal of H gives a topological sort of G'. 2) According to Theorem 4.2, a removed inter-iteration edge $e' \in E'$ must connect a succeeding actor (or SCC supernode) to the first actor (or SCC supernode) in the topological sort. 3) The final schedule S' of G' can be decomposed such that for each two-actor SDF subgraph in H, there exists a corresponding minimal periodic sub-schedule (single appearance or multiple appearance) in S'.

Based on 1), 2), and 3), analysis of buffer bounds on a removed inter-iteration edge e' can be performed in the *unique* two-actor graph in H, $G_{u,v} = (\{u, v\}, E_{u,v} = \{e \mid src(e) = u \text{ and } snk(e) = v\})$, such that $src(e')$ is in the v-cluster and $snk(e')$ is in the u-cluster. In other words, there exists a unique, consistent, acyclic, two-actor SDF graph $G_{u,v}$ in H such that the presence of e' is in the reverse direction across the two actors in $G_{u,v}$. Figure 4 shows a general form of such configuration, where for each $e_i \in E_{u,v}$, $p_i = prd(e_i)$, $c_i = cns(e_i)$, $d_i = del(e_i)$.

The following theorem pertains to the buffer bounds on the removed inter-iteration edges.

THEOREM 5.4. *Suppose that we are given a consistent, loosely interdependent, strongly connected SDF graph G. Suppose G' is the acyclic SDF graph that is constructed by applying the CYCLE-BREAKING algorithm on G and clustering the*

SCCs of the resulting graph G. Suppose H is the R-hierarchy in scheduling G'. Suppose that e' is an inter-iteration edge that is removed by the CYCLE-BREAKING algorithm. Suppose $G_{u,v} = (\{u, v\}, E_{u,v} = \{e \mid src(e) = u \text{ and } snk(e) = v\})$ is the consistent, acyclic, two-actor SDF graph in H such that $src(e')$ is in the v-cluster and $snk(e')$ is in the u-cluster. Then the buffer size required for e' is bounded by

$$\begin{array}{ll} del(e') + g \times d^* & \text{if } d^* \leq p^* \times c^* \\ del(e') + g \times p^* \times c^* & \text{if } d^* > p^* \times c^*. \end{array} \tag{4}$$

Here, p^*, c^*, and d^* are the primitive production rate, primitive consumption rate, and primitive delay of $G_{u,v}$, respectively; and in addition, $g = \gcd(p, c)$, $p = prd(e') \times \boldsymbol{q}_G[src(e')]/g_v$, $c = cns(e') \times \boldsymbol{q}_G[snk(e')]/g_u$, $g_u = \gcd_{\alpha \in u-\text{cluster}}(\boldsymbol{q}_G[\alpha])$, and $g_v = \gcd_{\alpha \in v-\text{cluster}}(\boldsymbol{q}_G[\alpha])$.

Proof. Based on Definition 5.2 and Lemma 5.3, analysis of $G_{u,v}$ in Figure 4 is equivalent to analysis of its primitive form $G^*_{u,v} = (\{u, v\}, \{e^* = (u, v)\})$ in Figure 5, where for each $e_i \in E_{u,v}$, $p_i = prd(e_i)$, $c_i = cns(e_i)$, $d_i = del(e_i)$, $g_i = \gcd(p_i, c_i)$, $p^* = p_i/g_i$, and $c^* = c_i/g_i$; for e^*, $prd(e^*) = p^*$, $cns(e^*) = c^*$, $\gcd(p^*, c^*) = 1$, and $del(e^*) = d^* = \min_{e_i \in E_{u,v}} (\lfloor d_i/g_i \rfloor)$.

Furthermore, because of the properties of SDF clustering, we can derive that 1) $p^* \times g_u = c^* \times g_v$, 2) execution of u consists of executing $snk(e')$ for $\boldsymbol{q}_G[snk(e')]/g_u$ times, and 3) execution of v consists of executing $src(e')$ for $\boldsymbol{q}_G[src(e')]/g_v$ times. As a result, we can transform e' in Figure 4 to an equivalent edge e in Figure 5 such that $src(e) = v$, $snk(e) = u$, $prd(e) = p$, $cns(e) = c$, and $del(e) = d = del(e')$. Note that adding e to $G^*_{u,v}$ preserves consistency because 1) $p^*/c^* = c/p$ — this is because of the balance equation on e':

$$prd(e') \times \boldsymbol{q}_G[src(e')] = cns(e') \times \boldsymbol{q}_G[snk(e')], \tag{5}$$

and 2) d is large enough for the consumption requirements of u for a complete iteration of $G^*_{u,v}$ — this is because e' is an inter-iteration edge for G so that

$$d = del(e') \geq cns(e') \times \boldsymbol{q}_G[snk(e')] = c \times g_u \geq c \times c^* \tag{6}$$

Based on Lemma 5.3, suppose S is any valid minimal periodic schedule for $G_{u,v}$ as well as $G^*_{u,v}$. According to Equation (3), we can derive that

$$tok_{G^*_{u,v}}(S, e, k) = \tau(S, v, k) \times p - \tau(S, u, k) \times c + d \tag{7}$$

and

$$tok_{G^*_{u,v}}(S, e^*, k) = \tau(S, u, k) \times p^* - \tau(S, v, k) \times c^* + d^* \tag{8}$$

Then, we can derive the following equation based on Equation (7) and Equation (8).

$$tok_{G^*_{u,v}}(S, e, k) = d + g \times (d^* - tok_{G^*_{u,v}}(S, e^*, k)) \tag{9}$$

Because S is a valid minimal periodic schedule, for any firing index k, we can derive that

$$\begin{array}{ll} tok_{G^*_{u,v}}(S, e^*, k) \geq 0 & \text{if } d^* \leq p^* \times c^*, \\ tok_{G^*_{u,v}}(S, e^*, k) \geq d^* - p^* \times c^* & \text{if } d^* > p^* \times c^*. \end{array} \tag{10}$$

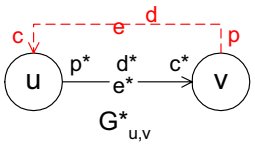

Fig. 5: Presence of cycle-broken edge in the primitive two-actor graph.

Finally, substituting Equation (10) into Equation (9) gives us

$$\begin{aligned} tok_{G^*_{u,v}}(S,e,k) &\leq d + g \times d^* && \text{if } d^* \leq p^* \times c^*, \\ tok_{G^*_{u,v}}(S,e,k) &\leq d + g \times p^* \times c^* && \text{if } d^* > p^* \times c^*. \end{aligned} \tag{11}$$

The proof is complete. □

6. CONCLUSION

In this report, we have reviewed the loose interdependence algorithms framework for decomposing cycles in SDF graphs into hierarchies of acyclic subgraphs whenever possible. Next, we have presented a low-complexity (linear-time) cycle-breaking algorithm to break cycles and decompose strongly connected SDF graphs. Our algorithm has been carefully designed such that the resulting graphs can be scheduled by the subsequent acyclic scheduling processes in the LIAF framework. With our new cycle-breaking algorithm, LIAF can be efficiently integrated with many useful scheduling techniques for acyclic SDF graphs. We have also presented an analysis of buffer bounds on the removed inter-iteration edges. We have implemented our cycle-breaking algorithm and buffer size computation technique in the *simulation-oriented scheduler*, which is geared toward efficient simulation of complex, multirate SDF graphs, such as those arising from modern wireless communications applications. Our results in [Hsu et al. 2006; 2007] demonstrate the effectiveness of the techniques developed in this report.

REFERENCES

Bhattacharyya, S. S., Leupers, R., and Marwedel, P. 2000. Software synthesis and code generation for DSP. *IEEE Transactions on Circuits and Systems — II: Analog and Digital Signal Processing 47,* 9 (Sept.), 849–875.

Bhattacharyya, S. S., Murthy, P. K., and Lee, E. A. 1996. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers.

Buck, J. T., Ha, S., Lee, E. A., and Messerschmitt, D. G. 1994. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation 4*, 155–182.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. 2001. *Introduction to Algorithms*, Second ed. The MIT Press.

Hsu, C., Keceli, F., Ko, M., Shahparnia, S., and Bhattacharyya, S. S. 2004. DIF: An interchange format for dataflow-based design tools. In *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*. Samos, Greece, 423–432.

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.

Hsu, C., Ramasubbu, S., Ko, M., Pino, J. L., and Bhattacharyya, S. S. 2007. Efficient simulation of critical synchronous dataflow graphs. *ACM Transactions on Design Automation of Electronic Systems - Special Issue on Demonstrable Software Systems and Hardware Platforms*. to be published.

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.

Lee, E. A., Ho, W. H., Goei, E., Bier, J., and Bhattacharyya, S. S. 1989. Gabriel: A design environment for DSP. *IEEE Transactions on Acoustics, Speech, and Signal Processing 37,* 11 (Nov.), 1751–1762.

Lee, E. A. and Messerschmitt, D. G. 1987. Synchronous dataflow. *Proceedings of the IEEE 75,* 9 (Sept.), 1235–1245.

Murthy, P. K. 1996. Scheduling techniques for synchronous and multidimensional synchronous dataflow. Ph.D. thesis, EECS Department, University of California, Berkeley.

Murthy, P. K. and Bhattacharyya, S. S. 2001. Shared buffer implementations of signal processing systems using lifetime analysis techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 20,* 2 (Feb.), 177–198.

Zitzler, E., Teich, J., and Bhattacharyya, S. S. 2000. Multidimensional exploration of software implementations for DSP algorithms. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology 24,* 1 (Feb.), 83–98.