Report Number: WUCSE-2011-59

2011

Efficient Deadlock Avoidance for Streaming Computation with Filtering

Authors: Jeremy Buhler, Kunal Agrawal, Peng Li, and Roger D. Chamberlain

In this report, we show that deadlock avoidance for streaming computations with filtering can be performed efficiently for a large class of DAG topologies. We first give efficient algorithms for dummy interval computation in series-parallel DAGs, then generalize our results to a larger graph family, the CS4DAGs, in which every undirected cycle has exactly one source and one sink. Our results show that, for a large set of application topologies that are both intuitively useful and formalizable, the streaming model with filtering can be implemented safely with reasonable compilation overhead.

2011-59

Efficient Deadlock Avoidance for Streaming Computation with Filtering

Authors: Jeremy Buhler, Kunal Agrawal, Peng Li, Roger D. Chamberlain

Corresponding Author: pengli@wustl.edu

Abstract: In this report, we show that deadlock avoidance for streaming computations with filtering can be performed efficiently for a large class of DAG topologies. We first give efficient algorithms for dummy interval computation in series-parallel DAGs, then generalize our results to a larger graph family, the CS4DAGs, in which every undirected cycle has exactly one source and one sink. Our results show that, for a large set of application topologies that are both intuitively useful and formalizable, the streaming model with filtering can be implemented safely with reasonable compilation overhead.

Type of Report: Other

Efficient Deadlock Avoidance for Streaming Computation with Filtering

Jeremy Buhler, Kunal Agrawal, Peng Li, and Roger D. Chamberlain

***Abstract*—Parallel streaming computation has been studied extensively, and many languages, libraries, and systems have been designed to support this model of computation. While some streaming computations send data at *a priori* predictable rates on every channel between compute nodes, many natural applications lack this property. In particular, we consider acyclic streaming computations in which individual nodes can choose to *filter*, or discard, some of their inputs.**

While streaming computation with filtering is an attractive model for many applications, if the channels between nodes have finite buffers, the computation can *deadlock*. One method of deadlock avoidance is to augment the data streams between nodes with occasional *dummy messages*; however, for general DAG topologies, no polynomial time algorithm is known to compute the intervals at which dummy messages must be sent to avoid deadlock.

In this report, we show that deadlock avoidance for streaming computations with filtering can be performed efficiently for a large class of DAG topologies. We first give efficient algorithms for dummy interval computation in series-parallel DAGs, then generalize our results to a larger graph family, the *CS4 DAGs*, in which every undirected cycle has exactly one source and one sink. Our results show that, for a large set of application topologies that are both intuitively useful and formalizable, the streaming model with filtering can be implemented safely with reasonable compilation overhead.

I. Introduction

Streaming is an effective paradigm for parallelizing complex computations on large datasets across multiple computing resources. Examples of application domains that use the streaming paradigm include media [1], signal processing [2], computational science [3], [4], data mining [5], and others [6]. Languages that explicitly support streaming semantics include Brook [7], Cg [8], StreamC/KernelC [9], StreamIt [10], Streams-C [11], and X [12].

A streaming application is typically implemented as a network of *compute nodes* connected by unidirectional communication *channels*. Abstractly, the streaming application is a directed dataflow multigraph, with the node at the tail of each edge (channel) able to transmit data, in the form of one or more discrete *messages*, to the node at its head. In this report, we consider only directed acyclic multigraphs.

Many streaming languages and libraries support the synchronous dataflow (SDF) [13] model, where, for a given input message stream, the number of messages consumed and produced by each node on each channel incident on it is known at compile time. However, the assumptions of SDF are not an intuitively good fit for all streaming applications. In particular, the node's decision on whether to send an output message in response to an input, and which subset of output channels to send messages on, may naturally be data-dependent. We say that nodes that can make such decisions at run-time exhibit *filtering* behavior.

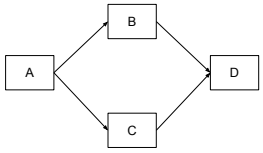

Fig. 1: A simple split/join streaming topology.

Consider, for example, the simple split/join topology shown in Figure 1. In a streaming application, the split node A might analyze an input and decide to send it to some subset of its children for further processing. For example, an object recognition system might receive a video frame and, based on some initial segmentation and analysis in the split node, might forward that frame to one or more dedicated modules that recognize particular types of object. Each recognizer in turn might or might not trigger a "success" message to the join node D. Finally, any information collected at D might be sent downstream to be merged with other analyses that were performed in parallel on the same frame. Two applications of this type are considered in [14].

While filtering behavior can be simulated in an SDF framework by sending enough extra messages, such workarounds may be both unnatural for the application programmer and a waste of channel bandwidth. In particular, if many channels in the streaming application share the same physical resource (e.g. a common bus or network connection, or one CPU handling multiple message queues), the ability to filter might make the difference between an efficient application and one that suffers communication bottlenecks.

This work addresses the challenge of safely realizing streaming applications when nodes are permitted to filter. For most streaming languages, the programmer is allowed to assume infinite buffer capacity on channels that connect compute nodes. In practice, however, the compiler allocates a finite channel buffers. With finite buffers, a filtering application can deadlock, even if it has no directed cycles (this is not true for SDF DAGs). If a language provides infrastructural support for combining computational modules into a streaming topology with filtering, that language's compiler and runtime should ensure that such deadlocks are avoided.

We formally modeled streaming computation DAGs with filtering and derived the precise conditions under which deadlock can occur in such DAGs [15]. We gave algorithms for

deadlock avoidance that work by sending occasional "dummy messages" between nodes. However, the intervals at which each node must emit these messages to avoid deadlock while minimizing dummy message traffic are in general challenging to compute. In particular, Our algorithms for computing dummy-message intervals run in worst-case time exponential in the size of the application's topology, raising the question of whether a safe filtering paradigm can be implemented efficiently as part of compiling a streaming streaming application.

In this work, we show that for a large class of intuitive and useful DAG topologies, deadlock avoidance in the presence of filtering can be guaranteed efficiently. We first show that for series-parallel (SP) DAG topologies [16], which describe applications constructed by pipelining and parallel split-join, our dummy-message interval algorithms can be run in small polynomial time in the size of the DAG. We then extend these results to a larger family of topologies, the CS4 DAGs, that permit limited communication between parallel branches of a computation. We precisely characterize the structure of CS4 DAGs and use this structure to extend our efficient deadlock avoidance algorithms to them. The CS4 DAGs represent an abstraction that balances expressibility with efficiency of deadlock avoidance.

Related Work

SDF was generalized to Dynamic Data Flow (DDF) by Lee [17] and Buck [18]. In a DDF graph, firing of nodes can be determined through the use of of an explicit boolean-valued [17] or integer-valued [18] control input. In our model [15], this control information is encapsulated within the node and is therefore unavailable to the compiler and/or scheduler. Here, synchronization between multiple streams into each node is supported via the use of a non-negative sequence number associated with each data item.

StreamIt [10] is a streaming language and compilation toolkit, that supports slightly generalized SDF semantics. Applications of StreamIt are constructed from three topology primitives: pipeline, split-join, and feedback. While these three primitives generate hierarchical application topologies that facilitate compiler analysis, they limit the kinds of streaming topologies that StreamIt can support well [6]. In this report, we will discuss broader classes of DAG topologies than those that StreamIt supports. Moreover, unlike StreamIt's split/join structures, which have special, language-defined semantics such as round-robin or broadcast, split and join nodes in this work can perform arbitrary computation and filtering just like any other node.

II. Background

A. Model of Streaming Applications with Filtering

A streaming application in our model has a DAG topology of computation nodes connected by reliable, one-way communication channels, each of which has a finite channel buffer. Input messages to the application are assumed to arise at *source* nodes. Inputs are labeled with monotonically increasing sequence numbers, and all channels are assumed to deliver messages in FIFO order. A node accepts an input with sequence number i when, for each of its input channels, the head of the channel buffer contains a message with sequence number $\geq i$. All messages with sequence number $= i$ are consumed together, and they may result in messages with sequence number i being sent on any subset of the node's output channels. If an input to a node does not result in an output on a given channel, we say that the node *filters* the input with respect to that channel.

We observed that in the presence of finite buffers between nodes, filtering behavior can lead to deadlock, as illustrated in Figure 2. If the buffer from A to C is empty because A filters its output to C and the buffers from A to B and B to C are full, the application is deadlocked. A must wait for B to consume an input before it can proceed; B must wait for C to consume an input; and C must wait until it sees an input from A.

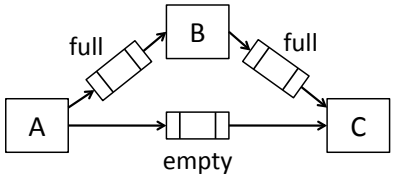

Fig. 2: A deadlock condition in a streaming application.

More generally, it can be proven that every potential deadlock in a DAG G corresponds to some undirected cycle C of G, which can be decomposed as a sequence of nodes with either two incoming or two outgoing directed paths on C. Roughly, a deadlock can occur whenever each of these nodes has a directed path with completely full buffers on one side, and an oppositely directed path with completely empty buffers (due to filtering) on the other side.

B. Deadlock Avoidance Through Dummy Messages

To avoid deadlock, We proposed two algorithms in which nodes periodically send *dummy messages* – content-free messages whose sequence number is that of some input that was filtered by the node. The idea of dummy messages originates in the parallel discrete-event simulation (PDES) literature [19], which used null messages for deadlock avoidance in conservative PDES algorithms.

In the first algorithm, the "Propagation Algorithm", only nodes with two outgoing edges on some undirected cycle send dummy messages. A dummy is sent on a channel whenever its source has gone too long without sending a data message on the channel. Dummy messages may not themselves be filtered but must be propagated on all output channels of any node they reach. In the second algorithm, the "Non-Propagation Algorithm", *every* node sends dummy messages, but the dummies need not be propagated past the channel on which they are emitted. Either algorithm can be implemented as a "wrapper" around each computational node in an application

by the language compiler and runtime, with no participation by the application programmer.

To implement dummy message-based deadlock avoidance, the language compiler must use the finite lengths of each channel buffer to calculate the intervals at which every node must send dummy messages to ensure safety. The basic idea behind the calculating dummy intervals for the Propagation Algorithm is the following. Consider an edge e from node u to node v. Let F be the set of edges starting at u, and let $\mathcal{C}$ be the set of undirected simple cycles that contain both e and another edge from $F - \{e\}$. For a cycle $C \in \mathcal{C}$, let $L(C, e)$ be the length of a shortest directed path on C (edge lengths are the buffer sizes on the edges) starting at u and not containing e. The dummy interval $[e]$ for e is then given by

$$[e] = \min_{C \in \mathcal{C}} L(C, e).$$

The corresponding idea for the Non-Propagation Algorithm is the following. Consider an edge e from node u to node v. Let $\mathcal{C}'$ be the set of undirected simple cycles containing e, and for each cycle $C \in \mathcal{C}'$, let $L(C, e)$ be as above. Let $h(C, e)$ be the number of edges in a longest directed path (in terms of edge count) on C that contains e. The dummy interval for e is then given by

$$[e] = \min_{C \in \mathcal{C}'} L(C, e)/h(C, e).$$

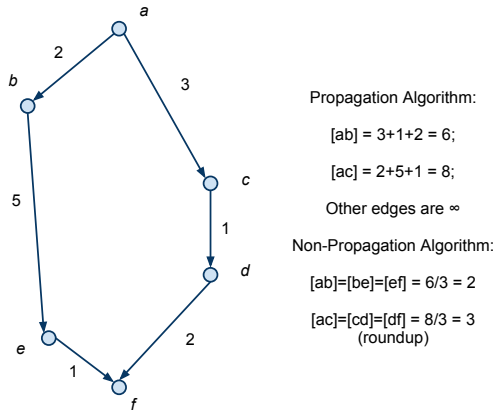

Fig. 3: Calculating dummy intervals on an undirected cycle using our algorithms

The above methods apply to general DAGs, but a direct implementation of them to compute dummy intervals requires worst-case time exponential in the size of the DAG (since a DAG may have exponentially many undirected simple cycles). It is currently unknown whether polynomial-time algorithms exist for dummy interval computation on general DAGs. We conjecture that there is no such algorithm for general DAGs that runs in small polynomial (linear or quadratic) time; hence, we seek efficiently solvable restrictions of these problems to simpler graph classes of practical interest.

III. SP-DAGs and their properties

In this section, we define the class of *series-parallel* (SP) DAGs. These graphs, which were defined by Valdes et al. [16], intuitively describe a large class of natural streaming topologies that can be built up recursively via pipelining and parallel splits and joins.

Definition (**Series-parallel DAG**)**.** A series-parallel DAG (SP-DAG) is a connected, directed acyclic multigraph with two distinguished terminals, a source and a sink. The set of all SP-DAGs is defined recursively as follows:

Base: a source and sink connected by any non-zero multiplicity of edges is an SP-DAG.

Ind. 1 (Serial composition, *Sc*): if H_1 and H_2 are SP-DAGs, connecting them by merging the sink of H_1 and the source of H_2 yields an SP-DAG $Sc(H_1, H_2)$.

Ind. 2 (Parallel composition, *Pc*): if H_1 and H_2 are SP-DAGs, connecting them by merging the sources of H_1 and H_2, and the sinks of H_1 and H_2, yields an SP-DAG $Pc(H_1, H_2)$.

Serial and parallel composition intuitively correspond to connecting computational nodes in a pipeline and with a parallel split-join, respectively. However, a composition confers no special semantics on the nodes used for the join. We sometimes refer to subgraphs H_1 and H_2 in the composition operations as *components* of the composed graph.

The next few lemmas elucidate the undirected cycle structure of SP-DAGs, which we will exploit in the next section to define efficient deadlock avoidance algorithms. Briefly, the only undirected simple cycles in an SP-DAG connect the sources and sinks of its parallel compositions, and each such cycle consists of a pair of directed paths from the source to the sink of some component. In contrast, general DAGs can have undirected cycles that contain any number of sources and sinks.

Observation. *In an SP-DAG, every node has an immediate postdominator (follows trivially from single-sink property).*

Lemma III.1. *In an SP-DAG G, let Z be a node with at least two outgoing edges. Let W be the immediate postdominator of Z. Then for any directed path P from Z to W, Z dominates all nodes of P other than W.*

Proof: By induction on the structure of G.

Base: in an SP-DAG with a single multi-edge, P is a single edge from Z to W. Z trivially dominates itself.

Ind.: Otherwise, G is either $Sc(H_1, H_2)$ or $Pc(H_1, H_2)$ for SP-DAGs H_1, H_2. If Z is the source of G, then Z trivially dominates all of G, since SP-DAGs have a single source. Z can not be the sink of G since the sink has no outgoing edges.

Now Z lies either in $H_1 - H_2$ or in $H_2 - H_1$, or $G = Sc(H_1, H_2)$ and Z is the sink of H_1 and the source of H_2. If Z is in $H_1 - H_2$, then H_1's sink always postdominates Z, so W, the immediate postdominator of Z, is a node in H_1. Applying the IH to subgraph H_1, the Lemma holds for Z and W. Analogous reasoning holds if Z is in $H_2 - H_1$. Finally, if

Z is the source of H_2 and the sink of H_1, then W is in H_2 and Z dominates all of H_2. ∎

Proof for Lemma III.4.

Proof: By induction on the structure of G.

Base: Trivially true for a single multi-edge.

Ind.: If $G = Sc(H_1, H_2)$, then the property holds for H_1 and H_2, and their serial composition creates no new cycles. Hence the property holds for every cycle of G.

If $G = Pc(H_1, H_2)$, then every new cycle created by their parallel composition connects the common source X of G to its common sink Y by directed paths passing through H_1 and H_2, respectively. All such cycles therefore have one source X and one sink Y. ∎

Lemma III.2. *Let $G = Pc(H_1, H_2)$ be an SP-DAG, where X is its source and Y is its sink. Let Z be a node of $H_1 - \{X, Y\}$ that has at least two outgoing edges e and e' in G. Let C be an undirected simple cycle that contains both e and e'. Then C contains no edge edge $e'' \in H_2$.*

Proof: Suppose not. WLOG, let the counterexample simple cycle C leave Z via edge $e = Z \to U$ and return via edge $e' = Z \to V$. Since C passes through an edge in H_2, it must also pass through both X and Y, since those are the only two nodes that connect H_1 and H_2. So there must be two vertex-disjoint undirected paths in H_1: P_1 goes from Z to U to Y, and P_2 (entirely in H_1) goes from Z to V to X.

Let W be the immediate postdominator of Z, which lies in H_1. We claim that both paths P_1 and P_2 must pass through W.

Suppose path P_1 does not pass through W. Now U is a predecessor of W, while Y is not, so there is some first edge in P_1 that connects a predecessor A of W to a non-predecessor B. We have two cases.

1) First, say this edge is oriented $A \to B$, then there is a directed path from Z to A to B to Y that bypasses W, which contradicts W's postdomination of Z.
2) If the edge is oriented $B \to A$, then B is not a successor of W, since G is acyclic. There is then a directed path from X to B to A that bypasses Z, which contracts Z's domination of A by Lemma III.1.

Conclude that P_1 must indeed pass through W.

Suppose P_2 doesn't pass through W. Now V is a successor of Z, while X is not; hence, there is some first edge on path P_2 that connects a successor A of Z to a non-successor B. This edge must be oriented $B \to A$, else B would be a successor of Z.

Now A cannot be a predecessor of W; otherwise, there would be a directed path from X to B to A that bypasses Z, contradicting Z's dominance of A by Lemma III.1. Hence, A is a successor of W. The subpath of P_2 from V to A therefore contains some first edge connecting a predecessor C of W to a successor D of W. This edge must be oriented $C \to D$, since G is acyclic. But then there is a directed path from Z to C to D to Y that bypasses W, which contradicts W's postdomination of Z. Conclude that P_2 must indeed pass through W.

Since P_1 and P_2 both contain W, they are not vertex disjoint, leading us to a contradiction. ∎

Corollary III.3. *For an SP-DAG $G = Pc(H_1, H_2)$, any undirected simple cycle C in G that has edges in both H_1 and H_2 consists of a pair of directed paths P_1 through H_1 and P_2 through H_2 that connect the source X of G to its sink Y.*

Proof: We know from Lemma III.2 that undirected simple cycles in G that traverse edges of both H_1 and H_2 do not pass through two outgoing edges of any node other than X. Moreover, each such cycle passes through two incoming edges of node Y, since Y does not have any outgoing edges.

Let P_1 be the directed path on C that exits X in (WLOG) H_1. If this path were to terminate at some node Z prior to Y, then the portion of cycle following P_1 would traverse two adjacent incoming edges of Z. But if the cycle leaves Z via an edge that points into Z and eventually reaches Y via an edge that points into Y, it must at some point "change direction" by passing through two outgoing edges of a node Q other than X, which is impossible by Lemma III.2.

Conclude that C must be fully directed from X to Y in both components. ∎

Lemma III.4. *Each undirected simple cycle in an SP-DAG G has a single source and a single sink.*

IV. Efficient Deadlock avoidance for SP-DAGs

The structured nature of series-parallel graphs is known to permit efficient algorithms for a variety of combinatorial problems, including some that are NP-hard on general graphs [20]. Similarly, we show in this section that the nice cycle structure of SP-DAGs allows us to efficiently prevent deadlock. In particular, we prove that we can compute the dummy intervals for all edges for the Propagation Algorithm in $O(|G|)$ time and for the Non-Propagation algorithm in $O(|G|^2)$ time.

A. Computing dummy intervals for the Propagation Algorithm

Dummy interval computation consists of the following steps:

1) We first recursively decompose G according to the construction rules for SP-DAGs, using e.g. the linear-time recognition algorithm of Valdes, Tarjan, and Lawler [16]. The decomposition results in a tree T whose leaves are single (multi-)edge graphs and whose internal nodes are labeled with the composition operators *Sc* or *Pc*, such that applying the composition operations in post-order results in graph G. The size of this tree is $O(|G|)$.
2) For every component H of G, we compute $L(H)$, which is the length of a shortest directed path (with buffer lengths as edge weights) from the source of H to its sink. This calculation can be done bottom-up on the tree T in $O(|G|)$ time.
3) We then compute the dummy intervals for each edge. Naively, we can update intervals once for each edge for each component of T, for a total time $O(|G|^2)$ (since SP-DAGs are planar and so have $O(|G|)$ edges). However,

we also give a more sophisticated algorithm that can compute all dummy intervals in $O(|G|)$ time with a top-down algorithm.

For step 2 of the above procedure, the following algorithm allows us to use the component tree T to compute shortest paths from source to sink for each component H of G in $O(|G|)$ time. For a single multi-edge $X \to Y$, we can compute a shortest path from X to Y in time proportional to the number of edges. Given shortest path lengths $L(H_1)$ and $L(H_2)$ from source to sink in SP-DAGs H_1, H_2, we can compute this length for their composition H in constant time as follows:

- If $H = Sc(H_1, H_2)$, $L(H) = L(H_1) + L(H_2)$.
- If $H = Pc(H_1, H_2)$, $L(H) = \min(L(H_1), L(H_2))$.

For step 3 above, we can naively compute all dummy intervals in $O(|G|^2)$ time during a post-order traversal of T as follows. Let H be a component of G.

Case 1: Say H is a leaf of T corresponding to a multi-edge $X \to Y$. The dummy interval $[e]$ for edge e is the minimum buffer size over all edges other than e between X and Y. If $X \to Y$ is only a single edge, then $[e] = \infty$.

Case 2: Say $H = Sc(H_1, H_2)$. Since H_1 and H_2 are joined by a single articulation point, their composition creates no new simple cycles. The dummy intervals for edges in H_1 and H_2 do not change.

Case 3: Say $H = Pc(H_1, H_2)$, and X is H's source. For each node $Z \in H - \{X\}$ with two outgoing edges e and e', Lemma III.2 proves that the parallel composition introduces no new simple cycle that leaves Z via e and returns via e'. Hence, the dummy intervals for these nodes' outgoing edges do not change. For the source X itself, the composition creates undirected cycles consisting of two directed paths from X to Y, as shown in Corollary III.3. Each such cycle leaves X via some e in H_1 and returns via some e' in H_2. Recall that we precomputed $L(H_1)$ and $L(H_2)$ as the total buffer lengths on some shortest (by total buffer length) directed path from X to Y in H_1 and H_2, respectively. Now each e out of X in H_1 already has some dummy interval $[e]$. Then we update $[e]$ as follows. For each edge e out of X in H_1,

$$[e] \leftarrow \min([e], L(H_2)).$$

Similarly, for each edge e' out of X in H_2, we set

$$[e'] \leftarrow \min([e'], L(H_1)).$$

By definition of L, we can always find a new cycle through e/e' that realizes the claimed shortest dummy interval.

Naively, we must update dummy intervals for every edge in a component when that component is processed, which could entail $O(|G|^2)$ edge updates over the entire tree. However, Algorithm 1 computes dummy intervals for all edges of G in top-down manner in time $O(|G|)$. We initially call SETIVALS(G, ∞) on the full graph G.

Definition. A cycle in graph G is external to a subgraph H if it passes through some edge of $G - H$. If the cycle does not pass through any edge in $G - H$, it is internal to H.

Algorithm 1: SETIVALS(G,V)

```
/* Set dummy interval on SP-DAGs for Prop. Algo. */
/* [e]: dummy interval for e; |e|: length of e */
if G is a single multi-edge X → Y then
    foreach edge e in the multi-edge do
        [e] ← min(V, min_{e'∈{X→Y−e}} |e'|)
else if G = Pc(H1, H2) then
    /* parallel composition */
    SETIVALS(H1,min(V, L(H2)))
    SETIVALS(H2,min(V, L(H1)))
else
    /* series composition */
    SETIVALS(H1,V)
    SETIVALS(H2,∞)
```

Claim IV.1. *Let H be a component of G, and let X be its source. When we call* SETIVALS(H, V)*, V is the smallest dummy interval for edges out of X that is required by any cycle that passes through X but is external to H.*

Proof: By induction from larger to smaller components.

Base: In the initial call to SETIVALS, $V = \infty$, which is appropriate because there is no cycle passing through the source of G that is not confined to G.

Ind.: Suppose V is set as claimed on entry to SETIVALS for component H. We will show that the claim holds for the recursive calls that result if that $H = Pc(H_1, H_2)$ or $H = Sc(H_1, H_2)$.

Say $H = Pc(H_1, H_2)$. Let e be an edge out of X in H_1. $L(H_2)$ represents the minimum dummy interval required for edge e due to any cycle that passes through X, is external to H_1, but is internal to H. By our IH, V is the minimum interval required for e due to any cycle that passes through X and is external to H. Hence, the code correctly updates V for H_1. Analogous reasoning holds for H_2.

If $H = Sc(H_1, H_2)$, then H_1 and H_2 are connected by a single vertex which is the sink of H_1 and the source of H_2. Hence, there is no simple cycle that traverses both H_1 and H_2 and is internal to H. V therefore remains the correct value to pass for H_1. For H_2, we know by Lemma III.2 that no cycle considered thus far enters and leaves H_2's source through outgoing edges, so there is not yet any constraint on the smallest dummy interval for these edges. Hence, we set correctly set $V = \infty$.

Conclude in all cases that V is correctly updated for the recursive calls. ■

Corollary IV.2. SETIVALS *correctly sets $[e]$ for all edges of G in time $O(|G|)$.*

Proof: The dummy intervals for edges are set only in the base case of SETIVALS. By our Claim above, when we call SETIVALS in this base case, V is the smallest dummy interval required for any edge in the given multi-edge, other than for trivial cycles involving two edges of the same multi-edge. Hence, the base case correctly sets $[e]$ for all edges in

the multi-edge.

The base case can be implemented in time proportional to the number of edges in the multi-edge, for a total cost of $O(|G|)$ over all edges in G, while the other cases require constant time per component of G, of which there are $O(|G|)$. Conclude that the entire algorithm is $O(|G|)$, as desired. ■

B. Computing dummy intervals for the Non-Propagation Algorithm

We now show how to efficiently calculate dummy intervals for the Non-Propagation Algorithm. The approach is broadly similar to that for the Propagation Algorithm, except that the interval $[e]$ now minimizes a ratio between the length of a component-dependent shortest path and the number of hops in an edge-dependent longest path.

1) Decompose the graph into a tree of components.
2) Compute $L(H)$ for each component H.
3) Compute $h(H)$ for each component H, where $h(H)$ is the longest path (in terms of the number of hops) from the source of H to its sink.
   - For a single multi-edge, $h(H) = 1$.
   - If $H = Sc(H_1, H_2)$, $h(H) = h(H_1) + h(H_2)$.
   - If $H = Pc(H_1, H_2)$, $h(H) = \max(h(H_1), h(H_2))$.
4) Compute $h(H, e)$ for each edge $e \in H$, where $h(H)$ is the longest path (in terms of the number of hops) from the source of H to its sink that passes through e. For a single multi-edge, $h(H, e) = 1$. For a series composition, for all $e \in H_1$, $h(H, e) = h(H_1, e) + h(H_2)$. Similarly for $e \in H_2$, $h(H, e) = h(H_2, e) + h(H_1)$. For parallel composition, if $e \in H_2$, $h(H, e) = h(H_1, e)$. Similarly for $e \in H_2$. All these computations can be done in $O(|G|^2)$ time.
5) Compute the dummy interval $[e]$ for each edge e in a bottom-up fashion.

The first four steps in the above procedure are straightforward. For the fifth step, we visit the components of T in post-order. When considering component H, we update $[e]$ for all the edges in H considering only cycles internal to H.

Case 1: If H is a multi-edge from $X \to Y$, let e be an edge from X to Y. If we consider only cycles internal to H, $L(H, e)$ is the minimum buffer size over all edges other than e between X and Y, and $h(H, e) = 1$. Therefore, the calculation in this case is identical to the calculation for the Propagation Algorithm.

Case 2: If $H = Sc(H_1, H_2)$, serial composition introduces no new simple cycles through e, so $[e]$ is unchanged.

Case 3: If $H = Pc(H_1, H_2)$, suppose WLOG that e is in H_1. Let X be the source of H, and let Y be its sink. Every new cycle created by the parallel composition consists of two confluent paths from X to Y, one in each of H_1 and H_2. Let C be the newly created cycle that traverses a longest (in hop count) directed path in H_1 that includes e and returns via a shortest (in buffer length) path in H_2. Then the ratio $L(C, e)/h(C, e)$ for C is minimum among all new cycles created by the composition. Since, $L(C, e) = L(H_2)$ and $h(C, e) = h(H_1, e)$, we have $[e] = \min([e], L(H_2)/h(H_1, e))$. The symmetric computation applies if e is in H_2.

Each case above takes constant time per edge in the component H, or $O(|G|)$ time per component. Conclude that the entire tree traversal is $O(|G|^2)$.

V. CS4 DAGs: A Larger Set of Simple Streaming Topologies

We have shown how to efficiently prevent deadlock in SP-DAGs – a large, practically useful class of DAG topologies that can be constructed with simple composition operations. A natural question at this point is, do there exist "natural" topologies that are not SP-DAGs? Might these topologies also have efficient algorithms for deadlock avoidance?

Figure 4 shows two simple two-terminal DAGs that are not SP-DAGs. The topology on the left augments a trivial split/join with a one-way communication channel linking its two sides; it is perhaps the simplest DAG that is not series-parallel. The topology on the right adds slightly more complexity, creating a "butterfly" structure like that commonly used to decompose large FFT computations. A key feature distinguishing the two graphs is that, in the left-hand example, every undirected simple cycle has only one source and one sink. This property is true for SP-DAGs, and we exploited it implicitly in the algorithms of the previous section. On the other hand, the butterfly graph contains a cycle *a-c-b-d* with two sources and two sinks.

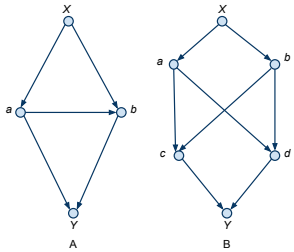

Fig. 4: two simple non-SP DAGs.

In this section, we characterize the set of all DAGs whose undirected cycles each contain one source and one sink. The next section shows that all such DAGs are amenable to efficient deadlock avoidance using generalizations of our algorithms from Section IV. In contrast, we can provide no such guarantee for graphs like the butterfly that have cycles with multiple sources and sinks. For such graphs, the programmer (or possibly the compiler) would need to construct an alternative topology to obtain efficient deadlock avoidance using our methods.

Definition. Let G be a DAG with a single source and sink. We say that G is "CS4" if every undirected simple **c**ycle in G has a **s**ingle **s**ource and a **s**ingle **s**ink.

Lemma V.1. *G is CS4 only if no subgraph of G is homeomorphic to K_4, the complete graph on 4 vertices.*

Proof: Suppose G has a subgraph H homeomorphic to K_4. H has 4 "corner" vertices and 6 connections (which may in general be paths rather than single edges) connecting them in the pattern of K_4. There are therefore 12 incidences of connections on corner vertices in H. WLOG, suppose that at least 6 of these are incoming. Now we have two cases.

1) Two vertices X and Y of H have exactly two incoming edges apiece.
2) One vertex has 3 incoming edges.

Consider case 1. If the (unique) shared connection between X and Y is oriented identically w/r to X and Y (either into both or out of both), then it is possible to find a cycle through X and Y with two sinks. Now consider the case when the connection $X - Y$ is directed out of one vertex and into the other. Suppose WLOG that connection $x - y$ is directed out of X and into Y. Let W and Z be the other two corner vertices of H.

Exactly one of the connections $Y - W$ and $Y - Z$ must be directed out of Y. Suppose WLOG that $Y - Z$ is directed out of Y. Because each of X and Y have exactly two incoming edges, we know the following: (1) $X - Z$ must be directed into X; (2) $W - X$ must be directed into X; (3) $W - Y$ must be directed into Y. Now $Y - Z$ must be directed into Z; otherwise, there must be a sink on this connection, and the cycle $XWYZ$ would contain two sinks. It follows that $X - Z$ is directed out of Z; otherwise, X and Z would constitute the forbidden case (1).

Now we established above that $Y - Z$ may not contain a sink. Similarly, $X - Z$ may not contain a sink because of cycle XWZ, and $X - Y$ may not contain a sink because of cycle XWY. Hence, cycle XYZ must be a directed cycle, which is forbidden because G is a DAG.

Consider case 2 above, where one corner vertex v of H has three incoming edges. Then no other corner vertex of H can have two incoming edges without creating a cycle with two sinks. Since H has at least six incoming edges on its corner vertices, it follows that the other three corner vertices of H each have exactly one incoming, and hence two outgoing, edges. Repeat the argument of Case (1) for any two of these vertices, swapping "in" and "out".

Conclude that there is no way to direct the edges of H so as to ensure that all its cycles have one source and one sink. ∎

Corollary V.2. *Every CS4 graph is planar.*

Proof: If G contains no subgraph homeomorphic to K_4, then it contains no subgraph homeomorphic to either K_5 or $K_{3,3}$, both of which contain subgraphs homeomorphic to K_4. By Kuratowski's Theorem, G must be planar. ∎

Now absence of K_4 is a characteristic property of *undirected* series-parallel graphs [21]. Hence, we may expect that CS4 DAGs have an undirected series-parallel structure. However, this does not imply that a CS4 DAG is an SP-DAG; our simple four-node graph above provides a counterexample. Fortunately, as we now show, it turns out that just a small amount of extra complexity is needed to capture all CS4 DAGs.

Definition. A 2-path cycle is a DAG consisting of a single source X, a single sink Y, and two directed paths connecting X to Y that are disjoint except at their endpoints.

Definition. Let C be a cycle. A chord graph H is a DAG with a single source and sink that connects two vertices of C, and H's source and sink lie on C.

Definition. Let C be a 2-path cycle with paths P_1 and P_2. A cross-link is a chord graph that connects a vertex of P_1 to a vertex of P_2, where neither endpoint of the connection is C's source or sink.

Definition. An *SP-ladder* G is a DAG consisting of a 2-path cycle with paths P_1 and P_2, called the outer cycle of G, and one or more chord graphs $H_1 \dots H_k$, such that:

- Each H_i is an SP-DAG;
- At least one H_i is a cross-link;
- If G contains two chord graphs with endpoints (u_1, v_1) and (u_2, v_2), then these chord graphs do not cross; that is, in tracing the outer cycle around G, we never encounter both u_2 and v_2 between u_1 and v_1.

Intuitively, we call G an SP-ladder because it can be viewed as a 2-path cycle "decorated" with non-cross-link chord graphs, plus one or more cross-links connecting the paths, none of which cross each other. The cross-links are similar to the rungs of a ladder. Examples of simple and complex SP-ladders are given in Figure 5.

Definition. Say that a cycle C of SP-ladder G traverses a chord graph H if C passes through a node of H other than its source or sink but is not confined to H.

Fact V.3. *If an undirected simple cycle C in G traverses a chord graph H, then C contains a directed path in H from its source u to its sink v.*

Proof: C reaches an internal vertex of H from outside, so it must consist of a simple path P in H that connects u to v, plus a path to return from v to u outside H. We claim that path P is directed. Suppose not; P enters and leaves H through edges directed out of its source and into its sink, so P must contain an internal source at some node Z. But Lemma III.2 showed that there is no simple path connecting the source and sink of H that contains an internal source. ∎

Lemma V.4. *Suppose that C traverses $k \geq 0$ cross-links of G. Then there is a cycle C' in G with at least as many sources/sinks as C that does not traverse any cross-link of G.*

Proof: By induction on k.

Base: Trivially true if $k = 0$; set $C' = C$.

Ind.: Suppose that C traverses k cross-links of G. Order these links as $H_1 \dots H_k$ in topologically increasing order of their endpoints (which is possible, because they cannot cross). Let $u_i < v_i$ be the endpoints of H_i in G.

We claim that either C does not pass through any strict predecessor of u_1 or v_1, or that it does not pass through any strict successor of u_k or v_k. Since C traverses H_1, it contains a directed path from u_1 to v_1. Starting from v_1, C must return by some undirected path P to u_1. Now if the first edge on this path touches a predecessor of v_1, then C must return to u_1 without touching any successor w of u_1 or v_1; indeed, to reach w without passing through u_1 or v_1 itself, the path would have to traverse a chord graph that crosses H_1, which cannot exist. If, on the other hand, P's first edge touches a successor of v_1, then C must return to u_1 without touching any predecessor w of u_1 or v_1, for the same reason.

Suppose that C does not touch a predecessor of u_1 or v_1. Construct C' from C by removing the path through H_1 and replacing it with the path on G's outer cycle that connects u_1 and v_1, passing through G's source X. C' does not contain the source that lies at endpoint u_1 of H_1 in C, but it does contain a new source at X. Removing H_1 cannot eliminate any other source or sink of C, so C' has as many sources/sinks as C.

If instead C does not touch a successor of u_k or v_k, construct C' from C by removing the path through H_k and replacing it with the path on G's outer cycle that directly connects u_k and v_k, passing through G's sink Y. C' does not contain the sink that lies at endpoint v_k of H_k in C, but it does contain a new sink at Y. Removing H_k cannot eliminate any other source or sink of C, so C' has as many sources/sinks as C.

By the IH, there is a cycle C'' in G with at least as many sources/sinks as C' that does not pass through any cross-link of G. ∎

Corollary V.5. *Every SP-ladder is CS4.*

Proof: Let C be any cycle in an SP-ladder G. If C traverses $k > 0$ cross-links of G, Lemma V.4 guarantees that there is a cycle C' that does not traverse any cross-links of G with at least as many sources/sinks as C.

Now either C' is confined to some chord graph H of G, or C' lies in the graph G' obtained by removing all cross-links from G. H and G' are both SP-DAGs, which are CS4 by Lemma III.4. Hence, C' has only one source and one sink.

Conclude that C has only one source and one sink, and so G is CS4. ∎

Lemma V.6. *Let G be a DAG with a single source and sink that is CS4. Then G is a serial composition of one or more graphs $G_1 \dots G_k$, s.t. each G_i is either an SP-DAG or an SP-ladder.*

Proof: Divide G into subgraphs $G_1 \dots G_k$ at its articulation points, so that G is the serial composition of $G_1 \dots G_k$. If every G_i is an SP-DAG, we are done. Otherwise, let G^* be a component of G that is not an SP-DAG. Now G^* has no internal articulation points, so it is composed of a 2-path outer cycle cut by one or more chord graphs.

Let H_1, H_2 be two chord graphs in G^*, with endpoints u_1/v_1 and u_2/v_2. If these subgraphs cross, then there exist paths P_1 connecting u_1 and v_1 in H_1 and P_2 connecting u_2 and v_2 in H_2. Moreover, G^*'s outer cycle contains u_1, v_1, u_2, and v_2 in some alternating order. Hence, the union of P_1, P_2, and this cycle is homeomorphic to K_4, and so G^* (and hence G) cannot be CS4. Conclude that no two chord graphs of G^* cross.

Now suppose that some chord graph H is not an SP-DAG. Let H^* be a smallest subgraph of H that is not an SP-DAG. H^* cannot be a serial composition of multiple subgraphs, so it is a 2-path outer cycle with one or more chord graphs, all of which are SP-DAGs. If H^* had no cross-link, we could decompose it as an SP-DAG via repeated parallel compositions to extract all of its chord graphs. Hence, some chord graph J of H^* is a cross-link.

Let u, v be the endpoints of J, and let x, Y be its source and sink. The outer cycle of H^* connects these vertices in the order $x - u - y - v$. Moreover, there is a path from u to v bypassing X and Y (through the cross-link) and a path from X to Y bypassing u and v (from X outwards to the source of H, then via the outer cycle of G^* to the sink of H, and finally inwards to y). The union of these two paths and the outer cycle of H^* is therefore homeomorphic to K_4 , and so H^* (and hence G) cannot be CS4. Conclude that H^*, and therefore H, cannot exist, and so every chord graph of G^* is indeed an SP-DAG.

Finally, if no chord graph of G^* is a cross-link, G^* can be decomposed via repeated parallel compositions to expose all its chord graphs and so is an SP-DAG. Otherwise, it is an SP-ladder. Conclude that every component of G is either an SP-DAG or an SP-ladder. ∎

Theorem V.7. *The set of single-source, single-sink CS4 DAGs is exactly the family of graphs of which each one is a serial composition of one or more graphs $G_1 \dots G_k$, s.t. each G_i is either an SP-DAG or an SP-ladder.*

Proof: Lemma V.6 shows that every single-source, single-sink CS4 DAG is in the claimed family. Conversely, Lemma V.1 and Corollary V.5 show that SP-DAGs and SP-ladders respectively are CS4. Serial composition of such graphs cannot introduce new cycles, so all such compositions remain CS4. ∎

VI. Efficient Deadlock Avoidance for CS4 DAGs

We now present algorithms to compute optimal dummy intervals for deadlock avoidance on CS4 graphs. Since a CS4 graph is serial composition of SP-DAGs and SP-ladders, edges on different SP-DAGs and SP-ladders cannot be on the same simple cycle. Hence, we can first decompose a CS4 graph into SP-DAGs and SP-ladders, then compute dummy intervals on these subgraphs separately. We have already described algorithms for SP-DAGs, so here we focus on SP-ladders.

An SP-ladder can be decomposed into its constituent SP-DAGs as shown in Figure 5, where each edge represents an SP-DAG directed the same way as the edge. This simplified representation of an SP-ladder has two paths from the source X to the sink Y. For convenience, we assume the two paths go from top to the bottom and distinguish them as the "left

path" and the "right path". We mark the vertices that connect these paths to cross-links from top to bottom, with the vertices on the left labeled $u_0, u_1, u_2, \ldots, u_{k+1}$ and the vertices on the right path from top to bottom labeled $v_0, v_1, v_2, \ldots, v_{k+1}$. The source $X = u_0 = v_0$ and the sink $Y = u_{k+1} = v_{k+1}$. This graph has k cross-links, which are numbered from top to bottom as K_1 through K_k, and the SP-DAGs on the outer cycle are numbered S_0 through S_k on the left and D_0 through D_k on the right. Note that in some cases, $u_i = u_{i+1}$ and then S_k is a graph with a single node. Figure 6 illustrates the general decomposition and this special case.

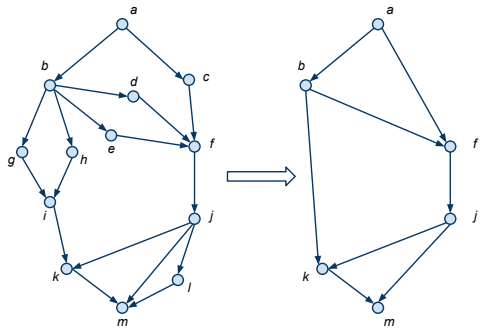

Fig. 5: decomposition of an SP-ladder graph

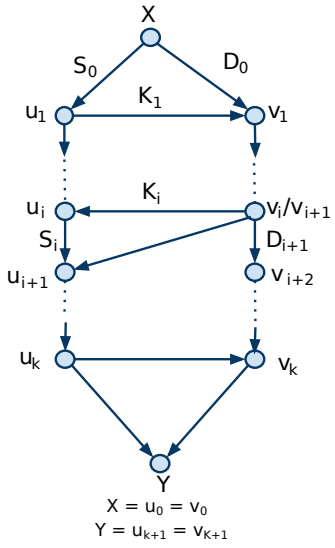

Fig. 6: general structure of a decomposed SP-ladder graph, including an example of cross-links sharing an endpoint.

Definition. We say that an undirected simple cycle is *external* if it traverses at least two of the constituent SP-DAGs.

Definition. A node s is an *internal source* if there exists an external cycle C that has s as its source, and $s \neq X$.

Fact VI.1. *An internal source is either (1) u_i if K_i goes from left to right, or (2) v_i if K_i goes from right to left.*

Fact VI.2. *Any external cycle with source $X = u_0 = v_0$ has a path through S_0 and another path through D_0. In addition, any external cycle with internal source u_i ($i \neq 0$) has one path going through S_i and another path going through K_i. Similarly for internal source v_i.*

Lemma VI.3. *Consider any external cycle C with source u_i. The sink of this cycle has 3 possibilities: (1) u_j, where $j > i$ and K_j goes from right to left, (2) v_j, where $j > i$ and K_j goes from left to right, or (3) Y, the sink of the ladder.*

We call the sinks defined in Lemma VI.3 the *potential sinks* of u_i. We can similarly define potential sinks for an internal source v_i.

A. Computing dummy intervals for Propagation Algorithm

The overall algorithm for finding dummy intervals for the propagating algorithm for an SP-ladder is given below; it runs in time $O(|G|)$, just as for SP-DAGs.

1) Decompose the SP-ladder into the component SP-DAGs, identifying the u_i's, v_i's, S_i's, D_i's and K_i's. In addition, we mark each edge as either belonging to a cross-link or not. This can be done in $O(|G|)$ time.
2) Compute dummy intervals for all edges due to cycles internal to each component SP-DAG, using the algorithm of Section IV.
3) For all $H \in \bigcup_{0 \leq i \leq k} S_i \cup D_i \cup K_i$, compute $L(H)$, which is the length of the shortest path from H's source to its sink (in terms of buffer sizes). Again, this is done as shown in Section IV.
4) Starting at the bottom of the SP-ladder, for each u_i, we compute $L_s(u_i)$, which is defined as the shortest directed path starting at u_i, going through S_i and ending at a potential sink of u_i. Similarly define $L_k(u_i)$ as the shortest directed path starting at u_i, going through K_i and ending at one of the potential sinks of u_i. If u_i is not the source of K_i, then we just set $L_k(u_i) = 0$. We define $L_d(v_i)$ and $L_k(v_i)$ in a similar manner, and compute those too.
5) Using these L values, we can update the dummy intervals for all edges that start at internal sources and at source X. No other dummy intervals change.

For step 1, we can decompose an SP-ladder into its constituent SP-DAGs in $O(|G|)$ time as follows: Identify an outer cycle C for G with left and right sides, using DFS in linear time. For each vertex u on the left side of C, determine (via DFS) whether any directed path leaving u encounters the right side of C at some vertex v before it encounters the left side again. If so, the nodes and edges on all such paths from u to v form a cross link. Repeat for the right side of C to identify cross-links directed from right to left. Now that we have identified all u_i's and v_i's, we can easily compute S_i's, D_i's and K_i's.

For step 4 above, we can compute $L_s(u_i)$ and $L_k(u_i)$ starting at the bottom of the SP-ladder as follows:

$$
\begin{aligned}
L_k(u_{k+1}) &= 0 \\
L_s(u_{k+1}) &= 0 \\
L_s(u_i) &= L(S_i) + \\
&\quad \min \left\{ \begin{array}{cc} & L_s(u_{i+1}), \\ \left[\begin{array}{cc} L(K_{i+1}) & \text{if } u_{i+1} \text{ is } K_{i+1}\text{'s source} \\ 0 & \text{otherwise} \end{array} \right. & \end{array} \right. \\
L_k(u_i) &= \left\{ \begin{array}{cc} L(K_i) + L_d(V_i) & \text{if } u_i \text{ is } K_i\text{'s source} \\ 0 & \text{otherwise} \end{array} \right.
\end{aligned}
$$

The calculations for v_i are analogous. Now for the propagation algorithm, suppose u_i is the source of a cross-link K_i directed left-to-right. Then for each edge e out of u_i, if e lies in K_i, then $[e] = \min([e], L_s(u_i))$. Recall that $L_s(u_i)$ is length of a shortest path, say p, that starts at u_i, goes through S_i, and ends at some potential source, say t of u_i. This dummy interval update covers all cycles that go to t via p and then returns to u_i via K_i. Similarly $[e] = \min([e], L_k(u_i))$ if e lies in S_i. We can similarly update all edges out of v_i. To update each edge e out of X, $[e] = \min([e], L(v_0))$ if e lies in S_0, and $[e] = \min([e], L(u_0))$, if e lies in D_0.

B. Computing dummy intervals for Non-Propagation Algorithm

Computing the dummy intervals for the Non-Propagation Algorithm takes longer. Here we give an $O(|G|^3)$ algorithm.

Again, we decompose into constituent SP-DAGs. As in the Non-Propagation Algorithm for SP-DAGs, for each constituent SP-DAG H, we precompute $h(H)$ as the length of the longest path (in terms of the number of hops) from H's source to its sink. In addition, for each edge e in H, compute $h(H, e)$ as the longest path from H's source to its sink that passes through e. In addition, we compute the initial estimate of the dummy intervals considering only the cycles internal to the constituent SP-DAGs.

Now consider every source (internal or not) u_i in the SP-ladder. We can enumerate all the potential sinks t for that source using Lemma VI.3. We know that any cycle from u_i to t must have one path that goes across a cross-link K_i and another path that goes across S_i. We define $L_s(u_i, t)$ as the length of the shortest directed path from u_i to t that goes along S_i and $L_k(u_i, t)$ as the length of the shortest directed path from u_i to t that goes along K_i. Similarly, $h_s(u_i, t)$ is the length of the longest directed path (in terms of hop count) from u_i to t that goes along S_i and $h_k(u_i, t)$ as the length of the longest directed path from u_i to t that goes along K_i.

Now consider an edge e in some constituent SP-DAG H along the path from u_i to t. We can update the dummy interval for e as follows: If e lies along some path from u_i to t that goes across K_i, then $[e] = L_s(u_i, t)/(h_k(u_i, t) - h(H) + h(H, e))$. If on the other hand, e lies along some path from u_i to t that goes across S_i, then $[e] = L_k(u_i, t)/(h_s(u_i, t) - h(H) + h(H, e))$. We can do the analogous procedure for each potential source v_i.

Running time: There are $O(|G|^2)$ source-sink pairs. For a given pair u_i and t, we can calculate $L_s(u_i, t)$, $L_k(u_i, t)$, $h_s(u_i, t)$ and $h_k(u_i, t)$ using L and h values of the constituent SP-DAGs in $O(|G|)$ time. We can also update all dummy intervals for edges on some path from u_i to t in $O(|G|)$ time. Therefore, the overall algorithm takes $O(|G|^3)$ time.

VII. Conclusion

In this work, we have explored the practicality of a flexible, general model of streaming computation. This model, formalized by us, permits computation nodes to arbitrarily filter their inputs. We have shown that, if the allowed streaming topologies are restricted to the CS4 DAGs (or, more stringently, to the SP-DAGs), then we can efficiently parametrize our dummy message-based deadlock avoidance algorithms. Hence, if the streaming application programmer agrees to use such topologies, the compiler and runtime system can guarantee safe execution of the resulting applications, in a way that is non-intrusive to application code and that scales even to large and complex applications.

Our work raises several directions for future research. One open question is whether it is possible to reduce the frequency of dummy messages sent in CS4 graphs below that mandated by deadlock avoidance for general DAGs. It may be that the simpler structure of these DAGs allows not just more efficient implementation of our avoidance scheme but a newer, lower-bandwidth scheme altogether. A second question is whether one can efficiently translate arbitrary DAGs to equivalent CS4 topologies by adding a small number of nodes and edges. For example, the butterfly of Figure 4 can be replaced by an SP-ladder with cross-links a-d and d-c, provided that data to be sent from b to c is routed via an extra hop through d. Finally, we plan to augment an existing language for streaming computation, such as the X language [12], to support the filtering model.

References

[1] B. Khailany, W. Dally, S. Rixner, U. Kapasi, P. Mattson, J. Namkoong, J. Owens, B. Towles, and A. Chang, "Imagine: Media processing with streams," *IEEE Micro*, pp. 35–46, March/April 2001.

[2] J. W. Romein, P. C. Broekema, E. van Meijeren, K. van der Schaaf, and W. H. Zwart, "Astronomical real-time streaming signal processing on a Blue Gene/L supercomputer," in *ACM Symp. on Parallelism in Algorithms and Architectures*, 2006, pp. 59–66.

[3] Y. Liu, N. Vijayakumar, and B. Plale, "Stream processing in data-driven computational science," in *IEEE/ACM Int'l Conf. on Grid Computing*, 2006, pp. 160–167.

[4] M. Erez, J. Ahn, A. Garg, W. Dally, and E. Darve, "Analysis and performance results of a molecular modeling application on Merrimac," in *ACM/IEEE Supercomputing Conf.*, Nov. 2004.

[5] M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy, "Mining data streams: a review," *SIGMOD Rec.*, vol. 34, no. 2, pp. 18–26, June 2005.

[6] W. Thies and S. Amarasinghe, "An empirical characterization of stream programs and its implications for language and compiler design," in *Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2010, pp. 365–376.

[7] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," *ACM Trans. on Graphics*, vol. 23, no. 3, pp. 777–786, Aug. 2004.

[8] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: a system for programming graphics hardware in a C-like language," *ACM Trans. on Graphics*, vol. 22, no. 3, pp. 896–907, July 2003.

[9] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens, "Programmable stream processors," *Computer*, vol. 36, no. 8, pp. 54–62, Aug. 2003.

[10] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Int'l Conf. on Compiler Construction*, 2002, pp. 179–196.

[11] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented FPGA computing in the Streams-C high level language," in *IEEE Int'l Symp. on Field-Programmable Custom Computing Machines*, 2000, pp. 49–58.

[12] M. A. Franklin, E. J. Tyson, J. H. Buckley, P. Crowley, and J. Maschmeyer, "Auto-pipe and the X language: A pipeline design tool and description language," in *IEEE Int'l Parallel and Distributed Processing Symp.*, Apr. 2006.

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

[14] P. Li, K. Agrawal, J. Buhler, R. D. Chamberlain, and J. M. Lancaster, "Deadlock-avoidance for streaming applications with split-join structure: Two case studies," in *IEEE Int'l Conf. on Application-specific Systems, Architectures and Processors*, July 2010, pp. 333–336.

[15] P. Li, K. Agrawal, J. Buhler, and R. D. Chamberlain, "Deadlock avoidance for streaming computations with filtering," in *ACM Symp. on Parallelism in Algorithms and Architectures*, 2010.

[16] J. Valdes, R. E. Tarjan, and E. L. Lawler, "The recognition of series parallel digraphs," in *Proceedings of the eleventh annual ACM symposium on Theory of Computing*, ser. STOC '79. New York, NY, USA: ACM, 1979, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/800135.804393

[17] E. A. Lee, "Consistency in dataflow graphs," *IEEE Trans. on Parallel and Distributed Systems*, vol. 2, no. 2, pp. 223–235, Apr. 1991.

[18] J. T. Buck, "Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams," in *Asilomar Conf. on Signals, Systems, and Computers*, vol. 1, Nov. 1994, pp. 508–513.

[19] J. Misra, "Distributed discrete-event simulation," *ACM Comput. Surv.*, vol. 18, no. 1, pp. 39–65, 1986.

[20] K. Takamizawa, T. Nishizeki, and N. Saito, "Linear-time computability of combinatorial problems on series-parallel graphs," *Journal of the ACM*, vol. 29, pp. 623–641, 1982.

[21] R. J. Duffin, "Topology of series-parallel networks," *Journal of Mathematical Analysis and Applications*, vol. 10, pp. 303–318, 1965.