Efficient Deadlock Avoidance for Streaming Computation with Filtering

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

Jeremy D. Buhler, Kunal Agrawal, Peng Li, and Roger D. Chamberlain, "Efficient Deadlock Avoidance for Streaming Computation with Filtering," in *Proc. of 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP), Feb. 2012, pp. 235-246.

Dept. of Computer Science and Engineering
Washington University in St. Louis

Efficient Deadlock Avoidance for Streaming Computation with Filtering

Jeremy D. Buhler Kunal Agrawal Peng Li Roger D. Chamberlain
Department of Computer Science and Engineering, Washington University in St. Louis
{jbuhler,kunal,pengli,roger}@wustl.edu

Abstract

Parallel streaming computations have been studied extensively, and many languages, libraries, and systems have been designed to support this model of computation. In particular, we consider acyclic streaming computations in which individual nodes can choose to *filter*, or discard, some of their inputs in a data-dependent manner. In these 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 paper, we show that deadlock avoidance for streaming computations with filtering can be performed efficiently for a large class of DAG topologies. We first present a new method where each dummy message is tagged with a destination, so as to reduce the number of dummy messages sent over the network. We then give efficient algorithms for dummy interval computation in series-parallel DAGs. We finally generalize our results to a larger graph family, which we call the *CS4 DAGs*, in which every undirected Cycle is Single-Source and Single-Sink (CS^4). 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 overhead.

Categories and Subject Descriptors C.2.4 [*Computer-Communication Networks*]: Distributed Systems—Distributed applications; D.1.3 [*Programming Techniques*]: Concurrent Programming—Distributed programming; F.1.2 [*Computation by Abstract Devices*]: Modes of Computation—Parallelism and concurrency

General Terms Algorithms, Design, Theory

Keywords Deadlock Avoidance, Graph Theory, Streaming Computation

1. 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 [7], signal processing [15], computational science [12], data mining [6], and others [16]. Languages that explicitly support streaming semantics include Brook [1], Cg [13], StreamIt [17], and X [5]. 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. When a node *fires*, it may consume messages from some subset of its input channels and produce messages on some subset ot its output channels. In this paper, we consider only directed acyclic multigraphs.

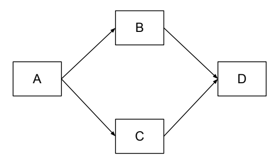

Figure 1: A simple split/join streaming topology.

Many streaming languages and libraries support the synchronous dataflow (SDF) [9] 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.

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 [11].

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 finite channel buffers. With finite buffers, a filtering application can deadlock, even if it has no directed cycles (which is not true for SDF DAGs).

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.

Table 1: Mercury BLAST performance

Dummy Interval	16	128
Dummy Msgs	8.51e+10	5.59e+10
Avg. Time (s)	639.8	421.0

One viable strategy for preemptively avoiding deadlocks in the presence of filtering is to send occasional *dummy messages* in addition to the regular stream of messages generated by the computation. Application nodes send dummy messages at pre-defined intervals, computed at compile time for the whole application, that are chosen to minimize the total number of dummies sent. We previously described this basic strategy and gave algorithms for computing the intervals for dummy transmission in [10].

Keeping the number of dummy messages added to a computation low, as we attempt to do, is beneficial for application performance. For example, we adopted dummy-based deadlock avoidance in Mercury BLAST [3], a streaming FPGA-based application for biological sequence comparison that exhibited deadlocks in practice. Using fewer dummy messages when possible by increasing the interval between them substantially reduced the performance impact of deadlock avoidance on Mercury BLAST, as shown in Table 1. While not every application may see such a dramatic impact, it is important to limit dummy message frequency when, as in Mercury BLAST, doing so frees up bandwidth and on a heavily loaded communication channel.

Unfortunately, the intervals at which nodes in an application must emit dummy messages to avoid deadlock while minimizing dummy message traffic are challenging to compute. In particular, our fastest algorithms for computing a safe set of such intevrals in [10] run in worst-case time exponential in the size of the application's topology, raising the question of whether a deadlock-free filtering can be implemented efficiently as part of compiling a 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. Our contributions are:

1. We present a new deadlock avoidance strategy, the *Destination-Tagged Propagation Algorithm*, in which every dummy message is tagged with a specific destination and does not propagate past this destination. This strategy improves on the Propagation Algorithm of [10] by reducing the total number of dummies sent and the associated computation and communication overhead.
2. We provide small polynomial-time algorithms to compute dummy message schedules that guarantee deadlock freedom when the application topology is a series-parallel DAG, or SP-DAG [18]. Our results cover two specific runtime deadlock avoidance strategies: the Destination-Tagged Propagation Algorithm above, and an alternate, Non-Propagation Algorithm described in [10].
3. We extend our results to a larger family of topologies, which we call 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 [8] and Buck [2]. In a DDF graph, firing of nodes can be determined through the use of of an explicit boolean-valued [8] or integer-valued [2] control input. In our streaming computation model, 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.

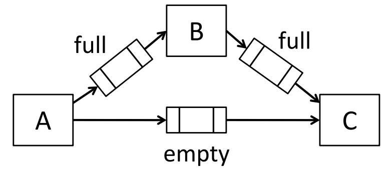

Figure 2: A deadlock condition in a streaming application.

StreamIt [17] is a streaming language and compilation toolkit that supports slightly generalized SDF semantics. Applications in 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 [16]. In this paper, 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.

2. Background

In this section, we review our formal model for streaming applications with filtering, the conditions for deadlock in the model, and the dummy-message approach to deadlock avoidance. We first formulated and investigated these formalisms in [10]. We also briefly review the definition of SP-DAGs, a well-known class of graph that we will use extensively later on.

2.1 Model of Streaming Applications with Filtering

A streaming application is a DAG of computation nodes connected by reliable, one-way communication channels, each of which has a finite channel buffer. We assume that channel buffer sizes are fixed *a priori*. Input messages arrive at a unique first node of the application and are labeled with monotonically increasing sequence numbers. All channels are assumed to deliver messages in FIFO order. A node always consumes all messages with sequence number $= i$ together and may then produce messages with sequence number i on any subset of its input channels.

A node does not necessarily need messages with sequence number i on all its input channels, but it must be sure that no message with sequence number i will arrive on a channels after it has already consumed other messages with sequence number i. Therefore, a node can only accept input with sequence number i when, for each of its input channels, the head of the channel buffer contains a message with sequence number $\geq i$. If an input of sequence number i to a node does not result in an output on a given channel, we say that the node *filters* the input i on that channel.

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.

Any cycle of G can be decomposed into a sequence of nodes, where alternating nodes have two incoming and two outgoing directed paths on C. As Figure 2 illustrates, a deadlock arises in a DAG G through the creation of a *blocking cycle*. 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. Therefore, any undirected cycle has a potential to become a blocking cycle and cause a deadlock.

We verified the precise conditions under which deadlock can occur in this model in [10].

2.2 Deadlock Avoidance Through Dummy Messages

Our strategy to avoid deadlock is to have 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 [14], which used null messages for deadlock avoidance in conservative PDES algorithms. A dummy message prevents a blocking cycle from forming by ensuring that the "empty" side of a potentially blocking cycle cannot remain empty (i.e., free of message traffic) while the "full" side fills completely. To meet this goal, dummy messages are sent at fixed intervals calculated from the buffer sizes of channels in the application's network.

The choice of dummy message intervals can be made in multiple ways, each corresponding to a different runtime strategy for sending dummies. Consider a DAG with node u and a potentially blocking cycle C, such that u has two outgoing edges. In order to prevent C from becoming a blocking cycle, we want to ensure that at runtime, one of the two directed paths out of u cannot become full of blocked messages while the other path remains empty.

We described two runtime strategies for sending dummy messages in [10]. In one strategy, which we call the "Propagation Algorithm", only nodes with two outgoing edges on some undirected cycle (like node u) send dummy messages, which may not be filtered but must be propagated on *all* output channels of any node they reach. In a second strategy, the "Non-Propagation Algorithm," *every* node may send dummy messages, but the dummies are never propagated beyond the channel on which they are emitted. In this case, not only u but all nodes on cycle C work together to prevent C from becoming a blocking cycle.

In the Propagation Algorithm, we compute dummy intervals at compile time as follows. Consider an edge e leaving a node u with at least two outgoing edges. Let F be the set of edges leaving 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 e' be other edegs out of u on cycle C, and let $L(C, e)$ be the total buffer size of the maximal directed path on C starting from u via e'. The dummy interval $[e]$ for e is then given by

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

Intuitively, $[e]$ is chosen short enough to ensure that for every potentially blocking cycle C involving e and another edge e' out of u, u sends a dummy along e often enough that the path along e' cannot become full while the path along e remains empty.

We use Figure 3 as a dataflow graph to illustrate the dummy interval calculation for the Propagation Algorithm. Let $[ab]_p$ be edge ab's dummy interval, and let $L(abef)$ be the path $abef$'s total buffer size. We show the calculation process for some edges rather than all of them for illustrative purpose. Let cycles $C_1 = abdc$, $C_2 = abefdc$, and $C_3 = bdfe$, respectively. Then the algorithm dictates

$[ac]_p = \min(L(C_1, ac), L(C_2, ac)) = 7;$
$[bd]_p = L(C_3, bd) = 6.$

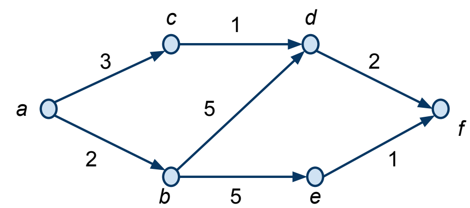

Figure 3: A DAG with three undirected cycles

The intervals $[cd]_p$, $[df]_p$, and $[ef]_p$ are ∞, since no dummy message need be generated on these channels.

Given a calculated interval I_p, during runtime, a node sends a dummy message to the channel if the current sequence number $CSN \geq I_p + LDS$, where LDS is the sequence number of last dummy message sent to this channel. For example, on channel bd, if a dummy message is sent with $LDS = 10$, after the node b computes on sequence number 16, it sends a dummy message to bd regardless of filtering. If b never receives a token with sequence number 16, which is filtered by a, b then sends a dummy message after it computes on a next higher sequence number.

In the Propagation Algorithm, every node on a directed path out of a dummy-emitting source receives a dummy message as often as the source sends one on its first edge. In contrast, the Non-Propagation Algorithm does not forward messages beyond a single edge, so the edges in a path must collaborate to ensure that the end of the path receives a dummy often enough to prevent deadlock. Consider an edge e leaving node u. 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 the maximal directed path out of u on C that begins with e. Then the dummy interval for e is then given by

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

In effect, we divide the dummy interval needed to prevent deadlock on each cycle C evenly among the edges on the path starting with e, so that the end of this path receives a dummy at least as often as it did in the Propagation Algorithm.

We still use Figure 3 to explain the calculation. Let and $[ab]_n$ be edge ab's dummy intervals for the Non-Propagation Algorithm, and let $h(abef)$ be the number of hops on path $abef$. For the Non-Propagation Algorithm, we have

$[ac]_n = \min(\lceil L(C_1, ac)/h(acd) \rceil, \lceil L(C_2, ac)/h(acdf) \rceil) = 3;$
$[bd]_n = \min(\lceil L(C_1, bd)/h(abd) \rceil, \lceil L(C_3, bd)/h(bdf) \rceil) = 2;$
$[cd]_n = \min(\lceil L(C_1, cd)/h(acd) \rceil, \lceil L(C_2, cd)/h(acdf) \rceil) = 3.$

The runtime node behavior in the Non-Propagation Algorithm is also different from that in the Propagation Algorithm. Given a calculated interval I_n, a node sends a dummy message to the channel if the current output is filtered and the current sequence number $CSN = I_n + LTN$, where LTN is the sequence number of last token, whichever a data token or a dummy message, sent to this channel. For example, if a token with sequence number 10 is sent to bd, and b filters sequence numbers 11 and 12. After filtering 12, b should send a dummy message to bd with sequence number 12. However, if b does not filter 12, no dummy message is needed and b simply updates bd's LTN to 12.

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.

2.3 SP-DAGs

Series-parallel (SP) DAGs, which were defined by Valdes et al. [18], intuitively describe a large class of natural streaming topologies that can be built up recursively via pipelining and parallel splits and joins.

DEFINITION 1 (**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 $Pc(H_1, H_2)$.*

For example, in Figure 1, each of the four edges AB, BD, AC, and CD is a base-case SP-DAG; we have $ABD = Sc(AB, BD)$, $ACD = Sc(AC, CD)$, and $ABCD = Pc(ABD, ACD)$. We sometimes refer to subgraphs H_1 and H_2 in the composition operations as *components* of the composed graph.

3. Destination-Tagged Dummy Messages

In the Propagation Algorithm, whenever any node receives a dummy message, it propagates it along all its outgoing edges. Therefore, if a node u generates a dummy message on edge (u, v), it is received by all the successors of v in the DAG, even if it is no longer useful. These extra propagation steps incur needless communication overhead in the DAG.

To avoid unnecessary overhead, we devise a new method, the *Destination-Tagged Propagation Algorithm*. As before, only source nodes can generate dummy messages, but these messages are now tagged with a destination node d. When a node receives a dummy message with destination d, it does not necessarily forward it along all its outbound edges; rather, it forwards the message only along edges that can reach d. (Node d itself need not propagate the message at all.) Under this scheme,unlike the previous algorithm, a message need never propagate to successors of its destination node.

Because each source can generate dummy messages for multiple destinations, each edge can have more than one dummy interval associated with it. Formally, we represent the *dummy message schedule* of an edge e as a set $[e] = \{p_1, p_2, ..., p_k\}$, where each $p_i = (\tau_i, d_i)$ is a *dummy interval-destination pair*. τ_i represents an interval at which a dummy message must be sent, while d_i represents its destination node. In addition, each dummy message pair p_i has a counter c_i associated with it, and the maximum value of the counter is c_i. A source node uses the dummy message schedule and the counters to decide when to send dummy messages along e. In Sections 4 and 6, we show how to efficiently compute the dummy message schedules for SP-DAGs and CS4 DAGs respectively, and also how nodes must behave at run-time in order to correctly propagate tagged dummy messages.

4. Efficient Deadlock Avoidance for SP-DAGs

We now show that restricting filtering application topologies to SP-DAGs permits efficient implementations of both the Destination-Tagged Propagation Algorithm and the Non-Propagation Algorithm for deadlock avoidance. We first briefly cover the properties of SP-DAGs that allow us to efficiently calculate dummy schedules for these topologies. We then describe how to compute dummy schedules for both avoidance algorithms in small polynomial time. We also specify the runtime behavior required of nodes in each algorithm. Finally, we argue that this behavior, in conjunction with its companion method for determining dummy intervals, guarantees deadlock freedom for SP-DAGs.

4.1 SP-DAG preliminaries

The next few lemmas elucidate the undirected cycle structure of SP-DAGs, which we will exploit later to build efficient deadlock avoidance algorithms. In particular, we use the property, verified in Lemma 4.4, that every undirected cycle on an SP-DAG has a single source and a single sink. We also use the hierarchical decomposition structure of SP-DAGs to efficiently compute dummy message schedules.

OBSERVATION 1. *In an SP-DAG, every node has an immediate postdominator (follows trivially from single-sink property).*

LEMMA 4.1. *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. □

LEMMA 4.2. *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. If the 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 4.1.

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 4.1. 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. □

LEMMA 4.3. *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 4.2 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 4.2.

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

LEMMA 4.4. *Each undirected simple cycle in an SP-DAG G has a single source and a single sink.*

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 4.5. *If X is the source for two components with sinks Y and Z, and these components share a common edge, then either Y is a successor of Z in G or vice versa.*

(Proof omitted due to space limitations.)

4.2 Destination-Tagged Propagation Algorithm

We now present the Destination-Tagged Propagation Algorithm as applied to SP-DAGs. We will describe both the compile-time algorithm used to compute dummy schedules for each edge, and the runtime behavior of nodes. The calculation of dummy schedules at compile time requires $O(|G|^2)$ time.

In our approach, the source node of each component H of an SP-DAG is responsible for preventing deadlock on undirected cycles of H that cross more than one of its sub-components. Since a node can be a source for multiple distinct components, it may need to send dummy messages that target multiple sinks. Therefore, an edge e from source u has a dummy message schedule $[e] = \{p_1, p_2, ..., p_k\}$, where in each pair $p_i = (\tau_i, d_i)$, d_i is a sink of some component for which u is the source. τ_i is the interval at which a dummy message must be sent to sink d_i. We keep this list of pairs sorted by τ_i. In addition, for each edge, we have at most one pair for a particular destination.

Computing Dummy Message Schedules

At compile time, we compute the dummy message schedule for each edge using a recursive decomposition of the SP-DAG as follows:

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 [18]. 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 schedules for all edges in total time $O(|G|^2)$ as follows.

The schedule computation algorithm performs a post-order traversal of G's component decomposition tree T. For each component H of G, we have three possibilities.

Case 1: Say H is a leaf of T corresponding to a multi-edge $X \to Y$. Let e be one edge of this multi-edge, and let τ be the minimum buffer size over all edges other than e between X and Y. Set $[e] = \{(\tau, Y)\}$. If $X \to Y$ is only a single edge, then $[e] = \emptyset$.

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 schedules for edges in H_1 and H_2 do not change.

Case 3: Say $H = Pc(H_1, H_2)$, where X is H's source and Y is H's sink. Now we add new pairs for each edge e out of X in H_1 as follows:

$$[e] \leftarrow [e] \cup \{(L(H_2), Y)\}\,.$$

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

$$[e'] \leftarrow [e'] \cup \{(L(H_1), Y)\}\,.$$

Finally, to eliminate unneeded dummy messages, we postprocess the schedule of each edge e as follows.

- If $[e]$ has more than one pair with the same destination, we retain only the pair with the smallest interval τ_i.
- If $[e]$ contains two pairs $p_a = (\tau_a, d_a)$ and $p_b = (\tau_b, d_b)$, such that d_b succeeds d_a and $\tau_b \leq \tau_a$, then we remove p_a.

This postprocessing requires only $O(|G|)$ time per edge. We now prove that this calculation preserves the invariants we require.

LEMMA 4.6. *In any edge's dummy schedule $[e]$, there is at most one dummy interval per destination, and the dummy messages are sorted by increasing τ.*

Proof. The first step of postprocessing ensures that there is at most one dummy message per destination on an edge. In addition, since the dummy intervals are calculated in post-order, if pair $p_i = (\tau_i, d_i)$ comes before pair $p_j = (\tau_j, d_j)$ in the original calculation, then d_j is a successor of d_i. Therefore, after step 2 of postprocessing, the schedule is sorted by increasing τ_i. □

Runtime Node Behavior

We now describe how the schedules of each edge are used at runtime to decide when to send dummy messages. We assume that the pairs of each edge's schedule $[e]$ are ordered by increasing τ. To track the time between successive dummy messages to each destination, edge e maintains a counter c_i for each pair p_i. The value of counter c_i ranges from 0 to τ_i.

Each time node X processes an incoming message, it acts as follows:

- If the message is a dummy (or a real message that is also marked as dummy), and X is not its destination, then X schedules a dummy message on all its outgoing edges and zeros out all counters on these edges.
- If the message is not a dummy, or is a dummy message with destination X, then X increments all counters on all outgoing edges, starting with the largest τ_i (end of the list). If a counter c_i on edge e reaches its maximum value, then X schedules a dummy message with destination d_i along e and zeroes out all counters c_j on e with $j \leq i$.

In all cases, if X has scheduled a dummy message on an edge e, and is also sending a real message on edge e, then it merges the dummy message with the real message and sends them as a single message.

Proof of Freedom from Deadlock

We now argue that the Destination-Tagged Propagation Algorithm ensures freedom from deadlock for SP-DAGs. As noted in Section 2.1, deadlock can arise in a DAG G only through the creation of a blocking cycle. Since SP-DAGs have exactly one source and one sink on each cycle, a blocking cycle consists of one path from the source to the sink with full buffers and another path from the source to the sink with empty buffers.

We claim that, because of the design of our dummy message scheme above, no sequence of messages sent on G can ever give rise to a blocking cycle, no matter how nodes choose to filter the non-dummy messages. The following sequence of results proves this claim.

LEMMA 4.7. *Let H be a component of G with source X and sink Y. If X propagates an incoming dummy message, then that message will reach Y.*

Proof. A dummy message arriving at X was generated by the source of some super-component H' of H with sink Z. By the properties of SP-DAGs, Z must be either Y or a successor of Y. In either case, all paths from X to Z lead through Y, so Y will eventually receive the dummy message. □

LEMMA 4.8. *If an edge's schedule includes pairs $p_i = (\tau_i, d_i)$ and $p_j = (\tau_j, d_j)$, and $\tau_i < \tau_j$, then d_j is a successor of d_i.*

Proof. Step 1 of postprocessing ensures that $d_i \neq d_j$. By Lemma 4.5, one of these nodes is a successor of the other. If d_i were a successor of d_j, then step 2 of postprocessing would have removed p_j. □

LEMMA 4.9. *Suppose that, for edge e out of node X, pair $(\tau_i, d_i) \in [e]$. For each τ_i messages that X receives, it sends at least one dummy message along e that will reach d_i.*

Proof. Consider a span of τ_i consecutive messages received by X. Before these messages arrive, counter c_i on e has some value $< \tau_i$. One of two cases will occur:

1. If one of the messages is a dummy that does not target X, then by Lemma 4.7, the dummy will reach d_i.
2. If all the messages either are non-dummies or target X, then either counter c_i will increase until it reaches τ_i, triggering a dummy message to d_i, or some other counter c_j, $j > i$, will reach τ_j, triggering a dummy message to d_j. By Lemma 4.8, we know that d_j is a successor of d_i, and so this message will pass through d_i.

□

LEMMA 4.10. *Consider a parallel component $H = Pc(H_1, H_2)$ with source X and sink Y. Let $L(H_1)$ be the length of a shortest path from X to Y through H_1. Consider any edge $e \in H_2$ that starts at X. In any time period during which X receives $L(H_1)$ messages, it sends (or forwards) at least one dummy message on e with destination either Y or a successor of Y.*

Proof. When the schedule-setting algorithm first processes H, it adds the pair $(L(H_1), Y)$ to $[e]$. Postprocessing will remove this pair only if X is also scheduled to send a more frequent dummy message to Y or to one of its successors. Hence, Lemma 4.9 guarantees that X will send at least one dummy message along e that reaches Y for each $L(H_1)$ messages it receives. □

THEOREM 4.11. *If dummy messages are sent as described in Section 4.2, using the interval-destination pairs computed as described in Section 4.2, then deadlock cannot occur in G.*

Proof. Suppose a deadlock does occur in G. Then there must be a blocking cycle C in G. Since G is an SP-DAG, C lies in some smallest parallel component H and consists of two directed paths s_1 and s_2 joining H's source X to its sink Y.

Suppose WLOG that s_1 is full and s_2 is empty. We can decompose H into parallel sub-components H_1 and H_2 such that $s_1 \subseteq H_1$ and $s_2 \subseteq H_2$. By construction, the total length of all edges' buffers along path s_1 is $\geq L(H_1)$, while that along s_2 is $\geq L(H_2)$.

Now consider the first edge e on path s_2, which leaves source X. This edge lies in component H_2. For s_1 to fill, X must have received and passed on at least $L(H_1)$ messages. But then Lemma 4.10 guarantees that X has sent a dummy message along e within its last $L(H_1)$ received messages. This dummy will eventually propagate to Y, where it will allow Y to consume at least one of the buffered messages from s_1. Since s_1 remains full, we conclude that the dummy must still be somewhere on path s_2, and so s_2 cannot be empty. This contradicts our assumption that cycle C is blocking. □

4.3 The Non-Propagation Algorithm on SP-DAGs

We now show how to efficiently calculate dummy intervals for the Non-Propagation Algorithm when the graph topology is restricted to be an SP-DAG. The approach is broadly similar to that for the Destination-Tagged Propagation Algorithm, except that the schedule $[e]$ for an edge e now consists of only a single pair whose destination is the node at the end of the edge. For this section, we therefore adopt the convention that $[e]$ is a single number, the dummy interval for e. In addition, all nodes, not just sources, may generate dummy messages on their outgoing edges.

Dummy interval calculation

Our algorithm for dummy interval computation is as follows.

1. Decompose the graph into a tree of components.
2. Compute $L(H)$ for each component H, where $L(H)$ is the shortest path from H's source to H's sink, with buffer lengths as edge weights.

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 that for the Dummy-Tagged 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 $O(|G|^2)$.

Runtime node behavior and correctness

We previously described the runtime behavior of nodes for the Non-Propagation Algorithm in a general graph in [10]. Briefly, a node sends a dummy message along an edge e if it filters $[e]$ continuous messages on edge e. This behavior applies unchanged to SP-DAGs. The dummy intervals $[e]$ of the previous section minimize a ratio between the length of a component-dependent shortest path and the number of hops in an edge-dependent longest path, as for the computation we previously gave for general graphs. Correctness for SP-DAGs therefore follows by the proof given for the algorithm on general graphs [10].

5. 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?

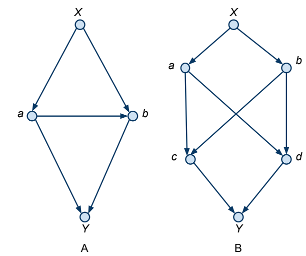

Figure 4: two simple non-SP-DAGs.

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.

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 Sections 4.2 and 4.3.

DEFINITION 2. *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 (for short, CS^4).*

A streaming application with the butterfly topology of Figure 4B is neither an SP-DAG nor even a CS4 DAG. However, it can be transformed to topologies with these properties by removing and redirecting certain graph edges. To transform this topology to a CS4 DAG without adding or removing nodes, we remove edge ad and add a directed edge from c to d. All messages passed from a to d directly in the original topology would then be routed via node c. However, if we are limited to using only SP-DAGs, besides removing ad and adding cd, we would also need to remove edge bd and route messages from b to d via node c, as Figure 5 shows. Hence, we can realize the original topology as a CS4 DAG with fewer changes than are needed to realize it as an SP-DAG.

A practical consequence of the difference between the CS4 and SP-DAG realizations of Figure 4B is that the CS4 DAG requires removing fewer edges, and hence less forwarding of messages that were delivered directly in the original topology. Moreover, the total number of messages sent is greater for the SP-DAG than for the CS4 DAG. As our experiments illustrate, reducing the total number of messages sent by a given node can significantly improve its real-world performance.

We can formally characterize CS4 graphs by the absence of a forbidden graph minor as follows.

LEMMA 5.1. *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.

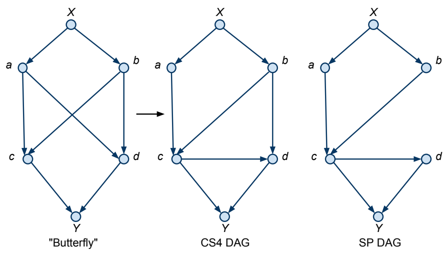

Figure 5: transforming butterfly to CS4 DAG and SP-DAG.

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. □

Now absence of K_4 is a characteristic property of *undirected* series-parallel graphs [4]. 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 3. *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 4. *Let C be a cycle. A chord graph H is a DAG with a single source and sink that connects two vertices of C, such that H's source and sink lie on C.*

DEFINITION 5. *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. A down-link is a chord graph that is not a cross-link.*

DEFINITION 6. *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 $H_1 \ldots H_k$, 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 6.

DEFINITION 7. *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.*

LEMMA 5.2. *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 4.2 showed that there is no simple path connecting the source and sink of H that contains an internal source. □

LEMMA 5.3. *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 $H_1 \ldots H_k$ 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 5.4. *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 5.3 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 4.4. 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 5.5. *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 5.6. *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 5.5 shows that every single-source, single-sink CS4 DAG is in the claimed family. Conversely, Lemma 5.1 and Corollary 5.4 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. □

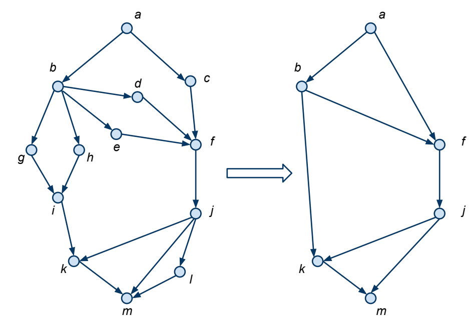

Figure 6: decomposition of an SP-ladder graph.

6. Efficient Deadlock Avoidance for CS4 DAGs

We now present algorithms to compute optimal dummy message schedules 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 schedules for edges in each of 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 6, 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 call the vertices that connect these paths to cross-links *corner vertices* and mark them from top to bottom, with the vertices on the left labeled $u_0, u_1, u_2, \dots, u_{k+1}$ and the vertices on the right path from top to bottom labeled $v_0, v_1, v_2, \dots, v_{k+1}$. The source $X = u_0 = v_0$ and the sink $Y = u_{k+1} = v_{k+1}$. All other nodes are called *internal nodes*. 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, $u_i = u_{i+1}$, in which case S_k is a graph with a single node. Figure 7 illustrates the general decomposition and this special case.

DEFINITION 8. *We say that an undirected simple cycle is* external *if it traverses at least two of the constituent SP-DAGs.*

The following facts about external cycles can be derived using structural properties of SP-ladders.

FACT 6.1. *Any external cycle with source $X = u_0 = v_0$ has a path through S_0 and another path through D_0. Any external cycle with source u_i ($i \neq 0$) has one path going through S_i and another path going through K_i. Similarly for source v_i ($i \neq 0$). All external cycles have corner nodes as sources and sinks.*

FACT 6.2. *Consider any external cycle C with source u_i. There are three possibilities:*

- *The sink of this cycle is u_k, where $i < k < m$ and K_k goes from right to left. In this case, one path on the cycle crosses K_j, goes through all v_j where $i \leq j \leq k$, and then traverses K_j. The other path traverses S_i, goes through all u_j where $i < j < k$.*

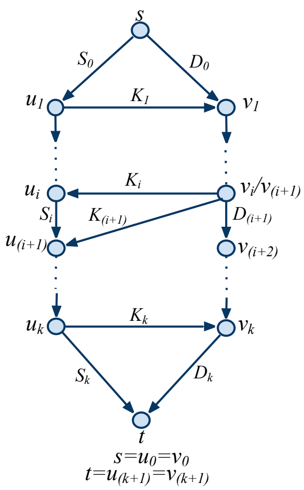

Figure 7: general structure of a decomposed SP-ladder graph, including an example of cross-links sharing an endpoint.

- *The sink of the cycle is v_k, where $i < k < m$ and K_k goes from left to right. In this case, one path on the cycle crosses K_i and passes through all v_j where $i \leq j < k$. The other path traverses S_i, goes through all u_j where $i \leq j \leq k$ and then crosses K_k.*
- *The sink of the cycle is $Y = u_m = v_m$, the sink of the ladder. One path on the cycle crosses K_i and passes through all v_j where $i \leq j$. The other path traverses S_i, goes through all u_j where $i \leq j$.*

We call the sinks defined in Fact 6.2 the *potential sinks* of u_i. We can similarly define potential sinks for an internal source v_i.

6.1 Destination-Tagged Propagation Algorithm

We now give an efficient version of the Destination-Tagged Propagation Algorithm specialized for SP-ladders. Again, only sources send dummy messages. An SP-ladder has two types of cycle sources: *internal sources* and *corner sources*. The algorithms for internal nodes are similar to those described in Section 4. We will concentrate on describing the algorithms for the corner sources. We will describe all the algorithms for some u_i, where u_i is a corner node on the left path of the ladder. Analogous algorithms can be derived for nodes on the right path.

The corner sources have two kinds of edges: edges on cross links K_i, and edges on down-links (S_i or D_i). An edge going out of a corner source u_i has three types of dummy interval-destination pairs:

1. $[e]_i$ consists of pairs for messages that stay within the chord for which u_i is a source (S_i for down-link, and K_i for cross-link). These are kept sorted by increasing τ as in the case of SP-DAGs.
2. $[e]_X$ consists of pairs for nodes v_k where $k > i$, i.e. corner nodes on the opposite side of the ladder from u_i
3. $[e]_W$ consists of pairs for nodes u_i where $k > i$, i.e. corner nodes on the same side of the ladder as u_i

The second and third lists are stored separately by increasing k. The schedule $[e] = [e]_i \cup [e]_X \cup [e]_W$.

Computing Dummy Message Schedules

We calculate the dummy message schedules for edges as follows:

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, mark each edge as either belonging to a cross-link or a down-link. This can be done in $O(|G|)$ time.
2. Compute $[e]_i$, schedules for all edges due to cycles internal to each chord graph , using the algorithm of Section 4.2.
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 a shortest path from H's source to its sink (in terms of buffer sizes). Again, this is done as shown in Section 4.2.
4. Starting at the bottom of the SP-ladder, for each u_i, and for each potential sink t of u_i, compute $L_s(u_i, t)$, which is defined as the shortest directed path starting at u_i, going through S_i and ending at t. Similarly, define $L_k(u_i, t)$ as the shortest directed path starting at u_i, going through K_i and ending at t. If u_i is not the source of K_i, then just set $L_k(u_i, t) = 0$. Define and compute $L_d(v_i, t)$ and $L_k(v_i, t)$ in a similar manner.
5. Using these L values, update the set of dummy intervals pairs for all edges that start at internal sources and at source X. No other sets change.

For step 1 above, we 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 compute $L_s(u_i, t)$ and $L_k(u_i, t)$, where t is a potential sink u_k or v_k of u_i. We consider u_i's in decreasing order of i. In order to compute $[e]_X$ and $[e]_W$ in sorted order, for a particular u_i, we consider t in increasing order of k.

$$
\begin{aligned}
L_s(u_i, u_i) &= 0 \\
L_s(u_i, t) &= L(S_i) + \\
&\quad \begin{cases} L(K_{i+1}) & \text{if } v_{i+1} = t, \\ L_s(u_{i+1}, t) & \text{otherwise} \end{cases} \\
L_k(u_i, t) &= \begin{cases} L(K_i) + L_d(v_i, t) & \text{if } u_i \text{ is } K_i\text{'s source} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

Say $t = v_k$, that is, t is on the opposite side of the ladder as u_i. For each edge e that starts at u_i, if e is a cross-link edge, then set $[e]_X \leftarrow [e] \cup (L_s(u_i, t), t)$, and if e is a down-link edge, set $[e]_X \leftarrow [e] \cup (L_k(u_i, t), t)$. On the other hand, if $t = u_k$, that is, on the same side of the ladder as u_i, then the same updates happen to $[e]_W$. Since we compute t in increasing order of k, these lists are sorted by increasing k The calculations for v_i are analogous.

Now we do some postprocessing to remove some superfluous pairs of dummy messages. For the internal dummy pairs, we do the same processing as SP-DAGs. For the external dummy messages, we do the following for the node u_i.

- If any edge e has an internal pair $p_a = (\tau_a, d_a)$ and an external pair $p_b = (\tau_b, d_b)$, where $\tau_a \geq \tau_b$, then p_a is removed.
- If a particular edge e has more than one interval with the same destination, we keep only the one with the smallest τ.

6.2 Runtime Node Behavior

The behavior of all nodes except the corner source remains the same as in the corresponding algorithm for SP-DAGs. As mentioned above, a corner source u_i has 3 lists of dummy message pairs, $[e]_i$, $[e]_X$ and $[e]_W$, where $[e]_i$ is sorted by increasing τ and

$[e]_X$ and $[e]_W$ are sorted by increasing k, where destination is a corner sink v_k or u_k respectively. Each dummy pair $p_a = (\tau_a, d_a)$ has counter c_a associated with it, and the maximum value of the counter is τ_a. One other difference from SP-dags is that in some cases, a dummy message can have more than one destination. If that is the case, the dummy message carries the list of destinations with it. There are two cases in the runtime behavior of a corner source u_i

Case 1: u_i receives a non-dummy message. For each outgoing edge e, increment the counters for in $[e]_i$, $[e]_X$ and $[e]_W$ starting from the end (decreasing τ for $[e]_i$ and decreasing k for $[e]_X$ and $[e]_W$). If a pair $p_a = (\tau_a, d_a)$ reaches its maximum value, then a dummy message with destination d_a is scheduled along that edge, and the counter for p_a is zeroed out. If d_a is an internal destination, then it behaves in the same way as the SP-dag algorithm. If $d_a = u_k$ $(k > i)$ or $d_a = v_k$ $(k \geq i)$, a corner node, all the counters in $[e]_W$ are zeroed out. In addition, the following occurs.

- If e is in a cross-link, then counters for pairs in $[e]_X$,to all v_j, $j \leq k$, are zeroed out.
- If the e is in a down-link, then counters for pairs in $[e]_W$, to all u_j, $j \leq k$, are zeroed out.

Case 2: u_i receives a dummy message, or a real message also marked as a dummy. If u_i is the only destination, then no action need be taken. Otherwise, destination(s) are always another corner node. Consider a destination $d_a = u_k$ $(k > i)$ or v_k $(k \geq i)$.

- Say d_a is some u_k, or v_k, $k > i$, [1] then the message is scheduled on all the down-link edges, and the counter for the pairs going to this destination are zeroed out. For a down-link edge e, all the counters in $[e]_i$ (for all the internal dummy messages) on these down-links are zeroed out. All the counters on $[e]_W$ with destination u_j, $j \leq k$ are zeroed out. All the counters (on down-links and cross-links) that are not zeroed out are incremented.
- If d_a is some v_k, $k = i$, [2] then the message is scheduled on along all the cross-link edges and all the counters in $[e]_i$ are zeroed out. All the other counters are incremented.

If u_i wants to send multiple dummy messages on the same edge, then they are merged and a list of destinations is created. In this formulation, assuming all buffer sizes are non-zero, there are at most 2 destinations for each dummy message. In both cases, if the node wants to send both a real message and a dummy message along the same edge, then the real message is also marked as dummy, and a total of one message is sent.

6.3 Proof of Correctness

SP-ladders have the CS4 property that each undirected cycle has at most one source and one sink. Therefore, in order for a deadlock to occur one path from the source to the sink must be full and another path must be empty. Here, we show that this can not occur when using the above algorithm for dummy schedules and node behavior.

The following lemma shows why the node can safely zero out the counters as described in the previous subsection.

LEMMA 6.3. *The following claims are true.*

1. *If a corner source u_i forwards a dummy message along an edge of a chord graph, it will go through all the nodes within that chord.*
2. *If a corner source u_i sends or forwards a dummy message along a down-link to some sink u_k or v_k, where $k \geq i$, this message will go through all the sinks u_j, $i \leq j \leq k$.*
3. *If a corner source u_i sends or forwards a dummy message along a cross link K_i intended for v_k or u_k, where $k \geq i$, it reaches all the nodes v_j, $i \leq j \leq k$.*

Proof. From the For claim 1, if a source forwards a dummy message, it is an external dummy message, and therefore its sink must be a corner node, and it must traverse the entire chord graph on which it is forwarded. In addition, when a corner source gets a dummy message not intended for itself, it forwards it along all its edges. Therefore, it must go through all the nodes of the chord graph before it reaches the sink.

Claims 2 and 3 are true due to Lemma 6.2. □

The following lemmas are analogous to Lemmas 4.9 and 4.10 for SP-dags.

LEMMA 6.4. *Suppose that, for edge e out of node X, pair $(\tau_i, d_i) \in [e]$. For each τ_i messages that X receives, it sends at least one dummy message along e that will reach d_i.*

Proof. Consider a span of τ_i consecutive messges received by X. Before these messages arrive, c_i has some value $< \tau_i$. For each incoming message, one of the following will occur.

1. The counter will be incremented until it reaches τ_i, triggering a dummy message to d_i.
2. The counter will be zeroed out because some other dummy message is sent or forwarded. From node behavior and Lemma 6.3, the counter is zeroed out only if the dummy message sent or forwarded will pass through d_i.

□

LEMMA 6.5. *Suppose that an external cycle in G starts at u_i and ends at t. Every time u_i receives $L_S(u_i, t)$ messages, it sends at least one dummy message with destination t along all its cross-link edges. Every time u receives $L_K(u_i, t)$ messages, it sends at least one dummy message along all its down-link edges.*

Proof. Using the above procedure for setting intervals, to start with, every cross-link edge will have a dummy interval with $p_a = (L_K(u_i, t), t)$ set. If the dummy interval was later removed, it is because another dummy pair p_b causes a dummy message with the same or higher frequency to be sent, and this dummy message will traverse all the paths that a dummy message due to p_b would take.

Therefore, by Lemma 6.4 implies the proof. □

Using the above lemmas, we can prove the correctness theorem.

THEOREM 6.6. *If dummy messages are sent as described in Section 4.2, using the interval-destination pairs computed by the above procedure, then deadlock cannot occur in G.*

Proof. Suppose a deadlock does occur in G. Then there must be a blocking cycle C in G. WLOG, say that the blocking cycle starts at u_i and ends at some sink t, and one path from u_i to t goes through K_i and another one goes through S_i. Say that the path s_1 through K_i is full and the path s_2 through S_i is empty.

We know that *length*$(s_1) \geq L_K(u_i, t)$. If we consider the first edge of path s_2, it leaves u_i through its cross-link. From Lemma 6.5, u_i sends a dummy message along this edge every time it gets $L_K(u_i, t)$ messages. Since this message is propagated all the way to t, s_2 cannot be completely empty, which contradicts our assumption that cycle C is blocking. □

[1] If there are two cross-links out of u_i, then we use the larger index i to make this decision.

[2] If there are two cross-links from i, we forward along the one that is equal.

6.4 Non-Propagation Algorithm

Computing the dummy intervals for the Non-Propagation Algorithm takes longer than for the Destination-Tagged Propagation Algorithm on SP-ladders. 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 u_i in the SP-ladder. We can enumerate all the potential sinks t for that source using Lemma 6.2. As we defined $L_s(u_i, t)$ and $L_K(u_i, t)$ we define $h_s(u_i, t)$ 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.

7. Conclusions

In this work, we have explored the practicality of a flexible, general model of streaming computation that 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 compute dummy message intervals for all edges. In addition, we have extended one of their dummy message-based algorithms to reduce the amount of propagation, thereby potentially reducing overheads. 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 one devise alternate dummy-based deadlock avoidance algorithms can further reduce the number of dummy sent; alternatively, can one derive lower bounds for the number of messages that *must* be sent by any algorithm to avoid deadlock? A second question is whether one can efficiently and systematically translate arbitrary DAGs to equivalent CS4 topologies by adding a small number of nodes and edges. Finally, we plan to augment an existing language for streaming computation, such as the X language [5], to support the filtering model.

Acknowledgments

We sincerely thank the anonymous reviewers for their devoted time and insightful comments, without which we could not have brought the paper to the final shape.

References

[1] 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. Graphics*, 23(3):777–786, 2004.

[2] 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*, pages 508–513, Nov. 1994.

[3] J. Buhler, J. M. Lancaster, A. C. Jacob, and R. D. Chamberlain. Mercury BLASTN: Faster DNA sequence comparison using a streaming hardware architecture. In *Proc. Reconfigurable Systems Summer Institute*, Urbana, IL, July 2007.

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

[5] 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.

[6] M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Mining data streams: a review. *SIGMOD Rec.*, 34(2):18–26, 2005.

[7] 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*, pages 35–46, March/April 2001.

[8] E. A. Lee. Consistency in dataflow graphs. *IEEE Trans. on Parallel and Distributed Systems*, 2(2):223–235, Apr. 1991.

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

[10] 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.

[11] 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*, pages 333–336, July 2010.

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

[13] 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*, 22(3):896–907, July 2003.

[14] J. Misra. Distributed discrete-event simulation. *ACM Comput. Surv.*, 18(1):39–65, 1986.

[15] 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*, pages 59–66, 2006.

[16] 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*, pages 365–376, 2010.

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

[18] J. Valdes, R. E. Tarjan, and E. L. Lawler. The recognition of series parallel digraphs. In *ACM Symposium on Theory of Computing*, 1979.