Polyhedral Constraints for Bounded-memory Execution of Synchronized Filtering Dataflow

Peng Li and Jeremy Buhler
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, MO 63130
{*pengli, jbuhler*}*@wustl.edu*

***Abstract*—Dataflow models are important in parallel computing. A key question for a dataflow model is whether it can safely be executed within bounded memory. For dataflow models with static data producing and consuming rates, such as synchronous dataflow, the question can be answered before computations start. A bounded-memory static schedule can be constructed at the mean time if there exists one. When data rates are determined data-dependently at runtime, however, bounded-memory execution can be quite challenging. In this paper, we study synchronized filtering dataflow (SFDF), a model in which nodes can filter tokens data-dependently. When two data streams join at a node, they must be synchronized according to their global indices. These filtering and synchronization behaviors pose a challenge in scheduling SFDF applications, as they might require unbounded memory and therefore deadlock.**

We propose a general method to guarantee bounded-memory execution in the SFDF model. In our method, when a node filters a data token, it can send a special token called *dummy message* to notify receivers about the filtering. Each node schedules the sending of dummy messages according to specified *dummy intervals*, which must be chosen to ensure safe yet efficient execution.

To characterize the space of safe dummy intervals for SFDF applications, we define a set of polyhedral constraints that bound the space of feasible intervals. We prove that these constraints are sufficient and necessary to guarantee bounded-memory execution of SFDF applications. For dataflow graphs that are series-parallel DAGs, we propose an efficient algorithm to eliminate redundant constraints. We also give experiments to show the impact of dummy interval configuration on scheduling dummy messages.

I. Introduction

In the era of big data, parallel computing is commonly used to process massive data for high performance. Although parallel computing architectures abound, programming such architectures is hard. When multiple programs work together, they may suffer problems such as deadlock and nondeterminism. To solve these problems for a particular application, we can use ad hoc solutions; however, to solve a problem common to many applications, it is helpful to abstract these applications into a model in which the problem can be solved in a unified way.

In this paper, we focus on dataflow models. A dataflow application is a network of computing nodes connected by first-in first-out (FIFO) data channels. We consider mainly *coarse-grained* dataflow applications, where compute nodes may execute complex operations on each input. Applications that fit this paradigm include signal processing [1], computational biology[2], and multimedia [3]. Many parallel computing frameworks have been developed to support coarse-grained dataflow applications, including AutoPipe [4], StreamIt [5], and Ptolemy [6].

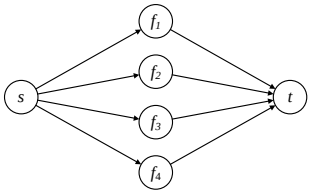

Figure 1. An example of streaming application graph.

This work focuses in particular on dataflow computations with filtering behavior. We say that a node *filters* data when it receives data and computes on them but does not produce any output. Filtering is data-dependent, which means it cannot be predicted statically. For example, a node may examine data items according to some predicate: if the predicate is true, an item is passed on; otherwise, it is discarded. Filtering behavior is a characteristic of many streaming dataflow applications, including e.g. cleaning and integration of sensor data, computer vision pipelines, and biological sequence analysis.

When channels in a dataflow application graph converge at a node, we must consider the question of *synchronization*. A node that has multiple input streams must determine the number of data items to be consumed from each of them to effect one computation. In this paper, we study data-dependent synchronization, which means the consumption rate of each input stream at a node depends on the input data.

Figure 1 shows a simple application with both filtering and synchronization behaviors. The node s generates samples, which will be filtered by f_1, f_2, f_3, and f_4 in a data-parallel fashion. To guarantee determinism, the filtered streams are

synchronized at node t to reassemble the original input order.

Correctly implementing filtering and synchronization behavior in a dataflow application is straightforward if the application has unbounded memory. However, trying to implement these behaviors in a real system with finite memory on each channel can cause incorrect behavior, in particular deadlock. In our previous work [7], [8], we proposed a runtime protocol for deadlock avoidance, the *dummy message protocol*, which required static computation of a number of application-dependent parameters ahead of runtime. We gave detailed algorithms for choosing these parameters so as to provably avoid runtime deadlocks. However, we subsequently realized that those algorithms sampled only one point in a large space of "correct" parameters. We therefore sought to define the boundaries of this space and to explore its implications for dataflow application performance.

In this work, we characterize the space of all correct parameter values for the dummy message protocol for a given application, using a collection of linear constraints to bound the parameter space. We then demonstrate that different parameter choices from this space, even those that represent different extrema of the constraint polyhedron, have differing impacts on the communication cost of deadlock avoidance, and hence potentially on application performance. Our work provides a strong formal basis for investigation into safe optimization of application performance in a filtering, synchronizing data flow model.

II. Background

This section describes the dataflow model, *synchronized filtering data flow* (SFDF), used in this paper. We introduced this model in [7], focusing on its filtering behavior. Here, we emphasize both its filtering *and* synchronization behaviors.

A. Application Graph and Data Tokens

An SFDF application is abstracted as a directed acyclic multigraph (DAMG). Vertices represent computational *nodes*, while edges represent FIFO communication *channels*. A node can emit *tokens* on one or more of its outgoing channels. There can be multiple channels between two nodes. SFDF application graphs cannot have directed cycles.

Each token emitted by a node is tagged with non-negative integer index. A node emits a stream of tokens with strictly increasing indices, but these indices need not be consecutive. Gaps in the token sequence correspond to filtered (i.e. discarded, not emitted) tokens.

B. Input Synchronization

An SFDF application is scheduled in a data-driven fashion, which means that a node is able to execute as soon as its inputs are all available. *Source* nodes have no inputs and so can spontaneously create and send output tokens. All other nodes must consume a token from one or more input channels in order to execute.

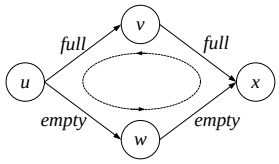

Figure 2. An example of deadlock in the SFDF model.

All tokens consumed in a single execution of a node must have the same index, and every token with this index *must* be consumed by the node at the same time. Because tokens in each FIFO channel have strictly increasing indices, each execution of a node consumes either zero or one tokens from each of its input channels. To ensure that all tokens with a given index are consumed simultaneously, a node cannot execute until it knows that no further tokens with that index will arrive. Hence, to safely consume all tokens with index k, a node must see that all its input channels have an available token with index $\geq k$.

C. Output Filtering

When a node executes, it sends either zero or one tokens on each of its output channels, depending on the result of its computation. Any output token sent has the same index as the inputs for that execution. If a node sends no token on a given output channel after an execution, we say that the output for that channel was *filtered*.

D. SFDF Applications Can Deadlock

An SFDF application has buffers to hold the tokens "in flight" on each communication channel. If these buffers are of finite size, it is possible for some SFDF applications to deadlock, even though the application graph is acyclic.

Figure 2 illustrates the mechanism of deadlock in SFDF. In this dataflow graph, suppose now that no output tokens are filtered by source u along channel uv, or by node v along channel vx. However, *all* tokens are filtered by u along channel uw, so that uw and wx remain empty. Node x is unable to consume data from vx because no token ever arrives on wx to prove that it is safe to do so. Eventually, the finite buffers on channels uv and vx fill, and u is unable to send, while x is unable to receive – hence, deadlock.

Deadlocks like the above cannot happen in the presence of unbounded channel buffers, such such buffers cannot become full. In practice, however, applications have finite, sometimes quite small, buffers (e.g. the queue between two stages on an FPGA). We must therefore consider the problem of scheduling execution of bounded-memory SFDF applications without deadlocks. Because finite buffers are the only source of deadlocks in SFDF, we use the terms *bounded-memory execution* and *deadlock avoidance* interchangeably in this work.

E. When is Deadlock Possible?

During SFDF execution, nodes may sometimes be *blocked* from continuing, either because it is unable to emit its output onto a full channel, or because it is waiting for a token to appear on an empty channel. These situations are typically temporary, but when they are permanent, the application is deadlocked.

To support the novel theoretical results of the next section, we now review some essential definitions and theorems about deadlock from our previous work [7].

Definition (Deadlock). An application is said to *deadlock* if no node is able to execute further in finite time, but some channel still retains unprocessed tokens (so that the computation is incomplete).

Definition (Blocking Relation). If a node v is waiting for input from an upstream neighbor u, or if v is waiting to send output to a downstream neighbor u because the channel buffer between them is full, we say that u *blocks* v.

Definition (Undirected Cycle). Given an application abstracted as a DAMG G, an *undirected cycle* of G is a cycle in the undirected graph G' that is the same as G, except that all edge directions have been removed.

Definition (Blocking Cycle). During execution of an SFDF application represented by graph G, an undirected cycle in G is said to be a *blocking cycle* if, starting from any node on the cycle and proceeding in a given direction, each node on the cycle blocks its neighbor.

In the example of Figure 2, $uvxw$ forms a blocking cycle: u blocks w, w blocks x, x blocks v, and v blocks u.

Theorem II.1 (Deadlock Theorem). *An SFDF application eventually deadlocks if and only if, at some point in the computation, there exists a blocking cycle.*

Theorem II.2 (Filtering Theorem). *If no node in an SFDF application ever filters any output token, then the application cannot deadlock.*

In fact, if no node ever filters its output, SFDF reduces to homogeneous synchronous dataflow (HSDF) [9], which is deadlock-free with a data-driven schedule.

III. Bounded-memory Execution of SFDF

We now present a provably correct approach to deadlock avoidance in SFDF applications. We use the idea of carefully scheduled emission of *dummy messages* introduced in [7]; however, whereas previous work gave only one possible way to schedule the emission of these messages, we precisely characterize the space of possible schedules that avoid deadlock. The characterization takes the form of a set of linear inequalities.

A. Dummy Messages

Theorem II.2 states that if no node filters data, an application cannot deadlock. We can effectively turn a filtering application into a non-filtering one by replacing filtered output tokens with *dummy messages* – special tokens with an index but no data content. The sole purpose of dummy messages is to communicate the index of the tokens in a node's most recent computation to its descendants in the system, whether or not that computation produced output.

If node u has output channel q, and it performs a computation on inputs with index i that does not result in an output on q, then it instead emits a dummy message on q with index i. Receiving dummy messages with index i on all inputs causes a node to perform a "null computation," emitting dummies with index i on all outputs.

The above use of dummy messages effectively transforms SFDF into HSDF and so prevents deadlock. However, this approach can send many more dummy messages than are strictly needed to prevent deadlock. Real distributed systems have limited channel bandwidth, so that communication costs can become a bottleneck. Moreover, in some applications, such as biological sequence comparison [10], efficient execution is predicated on having nodes filter the vast majority of their outputs. Replacing these outputs with dummies negates much of the application's intended efficiency.

To avoid the high computational and bandwidth costs of naively using dummy messages, we consider how to *schedule* their emission, at much-reduced frequencies, so as to guarantee that the application is still deadlock-free.

B. Scheduling Dummy Messages

Theorem II.1 states that deadlock happens only in the presence of a blocking cycle. If we can emit dummy messages just often enough to prevent blocking cycles, then we can eliminate deadlock. We use dummy messages to break blocking cycles by emitting tokens on what would otherwise be an empty channel, thereby preventing the sender of the channel from blocking its receiver.

Algorithm 1 describes the general mechanism for scheduling dummy messages. Each node maintains a *compute index*, which is the index of the last set of input tokens it consumed, as well as a *last output index* for each output channel, indicating when it last sent a token on that channel. Each channel q has a (finite) buffer length $|q|$ and a *dummy interval* $[q]$, which is statically determined given only the graph topology and buffer lengths. A node emits a dummy message on channel q only if it has no output data to send *and* the node's compute index has increased by at least $[q]+1$ since it last sent a token on q. If $[q] = 0$ for every channel, the application behavior reverts to the HSDF case described above; if $[q] = \infty$, then a node need never send any dummy messages on channel q.

We note that Algorithm 1 is essentially the "non-propagation algorithm" of [7].

Algorithm 1: Behavior of a single SFDF node.

```
ComputeIndex ← 0 ;
foreach output port q do
    LastOutputIndex_q ← 0 ;
while computing do
    if not source node then
        wait until every input channel has a pending
            token ;
        let T be minimum index of any pending token ;
        consume pending tokens with index T ;
    else
        T ← ComputeIndex + 1 ;
    ComputeIndex ← T ;
    perform computation on data tokens with index T ;
    foreach output channel q do
        if a data token with index T will be emitted on
        q then
            schedule a token with index T for output q ;
            LastOutputIndex_q ← T ;
        else if T − LastOutputIndex_q > [q] then
            schedule a dummy msg w/ index T for q ;
            LastOutputIndex_q ← T ;
    if not sink node then
        emit output tokens with index T;
```

C. Polyhedral Characterization of Safe Dummy Intervals

We now address the key question of how to choose the dummy interval for each channel in an application so that deadlock cannot occur. Setting all such intervals to 0 is, as we have seen, safe, but there may be less bandwidth-intensive alternatives.

We will prove that the space of deadlock-free dummy intervals for a given application graph G is precisely defined by a set of linear constraints on these intervals, determined by G's buffer lengths. We introduce two constraints for each undirected cycle in G, which together ensure that this cycle cannot become a blocking cycle for more than finite time. Because space of deadlock-free dummy intervals is defined by linear constraints, we can speak of the *safe dummy interval polyhedron* for a given application graph.

To describe the necessary constraints, consider Figure 3, which illustrates a blocking cycle C in an application. Channels on this cycle are directed either clockwise or counterclockwise. If C is blocking, then either all its clockwise channels are full and all its counterclockwise channels are empty, or vice versa.

Given an undirected cycle C, suppose the set of clockwise channels is H_1 and the set of counterclockwise channels is H_2. $[q]$ is the dummy interval for channel q and $|q|$ is the buffer length of q. We establish the following inequality

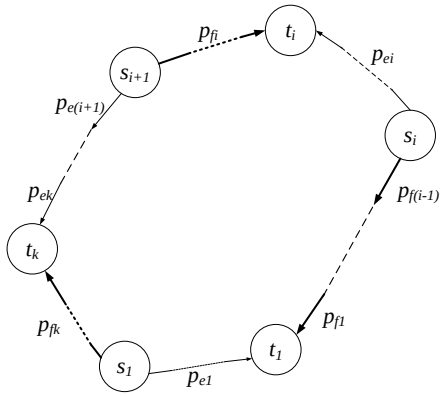

Figure 3. The division of a blocking cycle. Node and channel labels are used in the proofs of Theorem III.1 and Theorem III.2.

constraints for cycle C:

$$\begin{aligned} \Sigma_{q\in H_1}[q] &< \Sigma_{q\in H_2}|q| \\ \Sigma_{q\in H_2}[q] &< \Sigma_{q\in H_1}|q|. \end{aligned}$$

An application graph may have more than one undirected cycle, each of which generates a pair of constraints as described. The union of all these constraints defines a feasible polyhedron of dummy intervals for the application. In general, the number of undirected cycles in a graph, and hence the number of constraints, may be exponential in the number of nodes.

1) Proof for Sufficiency:

Theorem III.1. *If all nodes behave according to Algorithm 1, using dummy intervals that satisfy constraints as defined above, then the application cannot deadlock.*

Proof: We will assume that a blocking cycle C occurs in an application using deadlock avoidance with a set of dummy intervals that satisfy all constraints, then derive a contradiction. WLOG, we assume that the empty channels of C are oriented counterclockwise.

Divide cycle C into paths p of consecutive, similarly directed channels connecting *sources* s and *sinks* t. In particular, label the nodes on path p_{ei} $v_0, \ldots v_n$, with $v_0 = s_i$ and $v_n = t_i$, as in Figure 3.

We next define the concepts of *minval* and *maxval*.

Definition. (**minval** and **maxval**) For a non-empty channel or path q, minval(q) is defined to be *lowest* index of any token queued on q, while maxval(q) is defined to be the *highest* such index. For an empty channel or path q', minval(q') is defined to be the index of the token that has most recently traversed q'.

We first prove that if t_i has received no token with index greater than some T, then the last token received by node v_j of p_{ei} must have index at most $T + \Sigma_{k=j}^{n-1}[v_k v_{k+1}]$. The proof is by induction on j in decreasing order. In the base case, when $j = n$, the claim is trivially true, since $v_n = t_i$.

For the general case, by the inductive hypothesis, the last token received by v_{j+1} had index at most $M_{j+1} = T + \Sigma_{k=j+1}^{n-1}[v_k v_{k+1}]$, and so v_j's last token sent to v_{j+1} had index at most M_{j+1}. Now suppose that v_j has received a token with an index M' greater than

$$M_j = T + \Sigma_{k=j}^{n-1}[v_k v_{k+1}].$$

We have that $M_j - M_{j+1} = [v_j v_{j+1}]$, and so $M' - M_{j+1} > [v_j v_{j+1}]$, which means the interval between v_j's last received and last sent tokens is at least $[v_j v_{j+1}]$, the dummy interval of $v_j v_{j+1}$. Hence, Algorithm 1 implies that v_j must have sent a token, either real or dummy, to v_{j+1} with index $> M_{j+1}$. But this contradicts our IH. Conclude that the last token received by v_j has index at most M_j, as desired.

Next, we observe a special case of the above claim: if t_i's most recently received token from s_i (v_0) has index minval(p_{ei}), then s_i's most recently received token has some index τ (or s_i's most recent computing index is τ if s_i is a global source), where

$$\tau \leq \text{minval}(p_{ei}) + \Sigma_{q \in p_{ei}}[q].$$

We know $\tau \geq \text{maxval}(p_{f(i-1)}) \geq \text{minval}(p_{f(i-1)}) + |p_{f(i-1)}| - 1$, as $p_{f(i-1)}$ buffers at most $|p_{f(i-1)}|$ tokens, so we have

$$\text{minval}(p_{ei}) \geq \text{minval}(p_{f(i-1)}) + |p_{f(i-1)}| - [p_{ei}] - 1, \quad (1)$$

where $[p_{ei}] = \Sigma_{q \in p_{ei}}[q]$. Moreover, since C is a blocking cycle, at every sink t_i, we have

$$\text{minval}(p_{fi}) \geq \text{minval}(p_{ei}) + 1. \quad (2)$$

Starting from a sink t_i, traversing the cycle clockwise, and applying inequalities 1 and 2 alternately, we have

$$\begin{aligned} \text{minval}(p_{ei}) &\geq \text{minval}(p_{f(i-1)}) + |p_{f(i-1)}| - 1 - [p_{ei}] \\ &\geq \text{minval}(p_{e(i-1)}) + |p_{f(i-1)}| - [p_{ei}] \\ &\geq \cdots\cdots \\ &\geq \text{minval}(p_{ei}) + \Sigma_{q \in F}|q| - \Sigma_{q \in C \setminus F}[q] \\ &\geq \text{minval}(p_{ei}) + 1, \end{aligned}$$

where F is the set of full channels on C, and $C \setminus F$ is the set of empty channels. We observe contradiction and so conclude that blocking cycle C cannot exist, and hence no deadlock occurs. ■

2) Proof for Necessity:

Theorem III.2. *If all nodes behave according to Algorithm 1, but dummy intervals do* not *satisfy the constraints defined above, then the application can potentially deadlock.*

Proof: We assume that some constraint in the set is violated and construct a *filtering history* — that is, a record of execution describing which outputs are filtered by each node — that makes the application deadlock.

Let C be an undirected cycle in the application, again as shown in Figure 3, for which the chosen dummy intervals violate a constraint. WLOG, suppose in particular that $\Sigma_{q \in H}[q] \geq \Sigma_{q \in C \setminus H}|q|$, where H is the set of counterclockwise channels on C.

We now construct the application's filtering history as follows. For each counterclockwise channel uv on C, node u emits data on uv *only* for inputs with index $\leq N(v)$, for a value $N(v)$ to be specified below. For each clockwise channel wx on C, node w emits data on wx for every input received. Finally, for all channels yz that are not part of C, node y also emits output on yz for every input received. Hence, the only channels on which outputs may be filtered are the counterclockwise channels of C.

We set the node-specific values $N(u)$ for all nodes of C as follows. Suppose C has k sinks. Starting with some arbitrary sink labeled t_k, we set $N(t_k) = T = \Sigma_{q \in H}[q]$. (We use this value to avoid negative indices; any other sufficiently large integer also works.) We then traverse the cycle *clockwise* starting from t_k. We update $N(v)$ for v according to the equation

$$N(v) = N(u) + [uv] \quad (3)$$

if vu (not uv) is a counterclockwise channel, or

$$N(v) = N(u) - |uv| \quad (4)$$

if uv is a clockwise channel.

Now we prove that the compute index of u, denoted as $I(u)$, cannot advance to $N(u) + 1$.

Lemma III.3. *For any node u, suppose its clockwise neighbor is v, the event $I(u) = N(u) + 1$ happens after the event $I(v) = N(v) + 1$.*

Proof: According to the direction of channels, there are two cases.

Case 1: uv is a clockwise channel. If $v \neq t_k$, $N(u) - N(v) = |uv|$; if $v = t_k$, $N(u) - N(v) = \Sigma_{q \in H}[q] - \Sigma_{q \in C \setminus H}|q| + |uv| \geq |uv|$. In both cases, $N(u) - N(v) \geq |uv|$. Since uv does not filter any tokens, if v does not advance $I(v)$ to $N(v) + 1$ first, u cannot advance $I(u)$ to $N(u) + 1$, otherwise there would be at least $|uv| + 1$ tokens buffered in channel uv.

Case 2: vu (not uv) is a counterclockwise channel. v filters all tokens with indices greater than $N(u)$. If $I(v) < N(v) + 1$, v does not send any dummy message, then u does not receive any token with an index greater than $N(u)$. So $I(v)$ has to be $N(v) + 1$ before u advances its index to $N(u) + 1$. ■

According to Lemma III.3, we have a temporal contradiction if any $I(u)$ advances to $N(u) + 1$. Hence for any node u on C, it can only advance its compute index to $N(u)$. But there are still unprocessed tokens. According to the definition of deadlock, the system is deadlocked. ■

D. Deadlock Avoidance for Series-parallel DAGs

The set of linear constraints defining the deadlock-free dummy intervals for a graph G includes constraints for

all undirected cycles in G and can therefore grow exponentially with the size of G. However, for some graphs with particular structures, it is possible to enumerate a polynomial-sized set of constraints that is equivalent to the feasible polyhedron. Consider, for example, series-parallel DAGs (SP-DAGs), which can be constructed via a sequence of series compositions and/or parallel compositions starting from single edges. The interested reader can refer to [11], [8] for details. We now sketch an algorithm for defining the polyhedron of deadlock-free dummy intervals for an SP-DAG G. $H = Sc(H_1, H_2)$ means H is a series composition of H_1 and H_2. $H = Pc(H_1, H_2)$ means H is a parallel composition of H_1 and H_2.

1) Decompose the graph G 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 sizes as edge weights.
3) Introduce a variable $d(H)$ for each component H.
   - For a single edge H, add constraint $d(H) = [H]$.
   - If $H = Sc(H_1, H_2)$, add constraint $d(H) = d(H_1) + d(H_2)$.
   - If $H = Pc(H_1, H_2)$, add constraints

$$\begin{aligned} d(H) &\geq d(H_1) \\ d(H) &\geq d(H_2) \\ d(H_1) &< L(H_2) \\ d(H_2) &< L(H_1). \end{aligned}$$

Claim III.4. *The set of constraints defined by the above algorithm defines the polyhedron of deadlock-free dummy intervals for G. (Proof omitted due to space constraints.)*

E. Dummy Interval Selection

We have now defined a polyhedral space of feasible dummy intervals for preventing deadlock. For example, in Figure 2, if uv, vx, uw, and wx all have a buffer size of 8, then any (integer-valued) dummy interval assignment that satisfies $[uv] + [vx] < 16$ and $[uw] + [wx] < 16$ is valid. Given a plethora of possible feasible solutions, which one should we choose? The answer depends on the application designer's performance goals.

Latency and throughput are two common performance measures for parallel applications. In streaming computing, long-running analytical applications may have throughput but no latency requirements (e.g. [2]). Others, such as computational finance [12] and face recognition [13], have a real-time component and so may have both latency and throughput requirements. Balancing latency and throughput requires challenging optimization [14], which is beyond the scope of this paper; here, we focus on selecting dummy intervals purely to optimize throughput.

Generally speaking, larger dummy intervals introduce less communication overhead and so favor throughput; this qualitative statement has been verified by our previous experiments [15]. As a result, we are interested in sets of *maximal* dummy intervals, i.e. those at the boundaries of the feasible polyhedron. In the previous example, we might investigate sets of dummy intervals that satisfy $[uv] + [vx] = 15$ and $[uw] + [wx] = 15$.

Given a non-maximal dummy interval assignment, we can increase some channels' dummy intervals to get a maximal assignment, which will schedule fewer dummy messages than does the non-maximal one does and so may lead to better throughput. But other than fewer dummy messages leading to better throughput in general, the relation between throughput gain and the assignment of dummy intervals is difficult to model. First, data-dependent filtering makes it impossible to predict the number of dummy messages generated during runtime. Second, filtering makes the workloads of different nodes hard to predict; hence, it is hard – if not impossible – to know which set of dummy intervals will schedule the fewest dummy messages and consequently will yield the best throughput. However, if we know *a priori* the filtering behavior of some nodes, we can narrow down candidate solutions. For example, if we know that some nodes do not filter outputs at all, we can set their outputs' dummy intervals to 0, and hence raise some other interval(s) on the same cycle, without introducing extra communication overhead to the application.

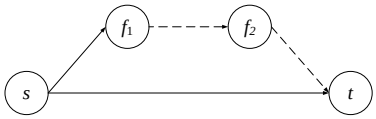

Figure 4. Dataflow graph of a synthetic application.

IV. Experimental Results

In this section, we conduct some preliminary experiments to determine whether the choice of feasible dummy intervals within the feasible polyhedron is likely to have an impact on application performance.

Because the topology of dataflow graphs and filtering patterns vary from application to application, it is impossible to cover all application scenarios with experiments. Moreover, to the best of our knowledge, there is no benchmark suite for SFDF applications. In this work, we study a small, synthetic application topology and simply count the number of dummy messages scheduled for different sets of feasible dummy intervals. We do not compute actual application throughput, as it is influenced by many factors, such as mapping and communication bandwidth, that are not covered by the SFDF model.

Figure 4 shows the dataflow graph of our synthetic application. The two dashed channels filter data. We assign a buffer size of 32 for channel st, so $[f_1 f_2] + [f_2 t] < 32$ is the constraint for avoiding deadlocks. Nodes f_1 and f_2

each filter 45% of the data arriving from the start node s. Applications with such high filtering rates are typically among the most vulnerable to deadlock, since some channels may go a long time without seeing any real data.

We investigate two different temporal patterns of filtering: correlated and uncorrelated. Correlated filtering means that if a data token is filtered out, the next is likely to be filtered as well. In uncorrelated filtering, filtering one token does not increase the filtering probability for the next. Correlated filtering is more likely to stress the dummy message mechanism because it entails longer runs of filtered tokens and hence increases the likelihood of persistently empty output channels. To simulate correlated filtering in this experiment, we simply repeat each input token 1000 times, so that each block of 1000 input tokens is either completely filtered or completely passed through. The number of tokens received by f_1 is 10^7 in both cases.

Table I
MEASURED DUMMY MESSAGE COUNTS FOR CORRELATED FILTERING

	Dummy message count (Thousands)		
Dummy intervals $[f_1 f_2], [f_2 t]$	$f_1 f_2$	$f_2 t$	Total
1, 30	2206	286	2492
5, 26	734	317	1051
9, 22	441	346	787
13, 18	314	399	713
17, 14	244	551	795
21, 10	199	617	816
25, 6	168	825	993
29, 2	146	1679	1825

Table II
MEASURED DUMMY MESSAGE COUNTS FOR UNCORRELATED FILTERING

	Dummy message count (Thousands)		
Dummy intervals $[f_1 f_2], [f_2 t]$	$f_1 f_2$	$f_2 t$	Total
1, 30	1397	39	1436
5, 26	46	56	102
9, 22	2	88	90
13, 18	0.092	140	140
17, 14	0.005	230	230
21, 10	0	399	399
25, 6	0	769	769
29, 2	0	1965	1965

Table I and Table II show our experimental results. All sets of dummy intervals studied are maximal with respect to the polyhedron of feasible solutions, so no one set dominates another *a priori*. As expected, correlated filtering, which tends to produce more persistently empty channels, results in a larger number of dummy messages overall.

The most important observation arising from this experiment is that the choice among feasible sets of intervals has a substantial impact on dummy message traffic. If we take as our optimization criterion minimizing the extra message traffic incurred by the mechanism, certain feasible choices are much more efficient (by up to an order of magnitude) than others. We hypothesize that these differences in raw message count will likely propagate to differences in "real-world" objective functions such as application throughput.

We conclude that there are interesting performance optimization questions to be explored in choosing among the many deadlock-free solutions permitted by our approach. Even this limited experiment suggests directions for optimization; for example, more balanced allocation of dummy intervals on a path empirically seems to incur fewer dummy messages.

V. RELATED WORK

Many dataflow models have been proposed previously. G. Kahn introduced process network (KPN) where deterministic processes communicate through unbounded FIFO channels [16]. The data consuming and producing rates are not defined in KPNs. Scheduling KPN applications is therefore challenging, as they do not deadlock in theory but do deadlock in practice due to bounded channels. Lee et. al. proposed synchronous dataflow (SDF) [9], which is a restricted KPN. In SDF, data consuming and producing rates of each channel are known statically. The static data rates allow the construction of a deadlock-free static schedule during compile time. Extensions of the SDF model include cyclo-static dataflow [17], multidimensional SDF [18], and the synchronous piggybacked dataflow (SPDF) [19]. The SDF model and its extensions are unable to characterize applications with data-dependent data rates, which (like SFDF) fall within the scope of dynamic dataflow. Buck et. al. proposed boolean dataflow (BDF) [20], which is a special dynamic dataflow. In BDF, for each channel with dynamic rates, a boolean switch is added to select an input or an output port. Buck proved [21] that the problem of deciding whether a BDF graph can be scheduled with bounded memory is undecidable. Like BDF, SFDF is also a special dynamic dataflow, but we can guarantee bounded-memory execution. Some recent work on streaming systems also considered dynamic data rates. Sbirlea et al. proposed Streaming Concurrent Collections (SCnC), a streaming model that supports Concurrent Collection programs in streaming domain [22]. SCnC allows variable data rates and thus has deadlock issues, which is resolved by restricting filtering behaviors so that sufficiently large buffers can be used to avoid deadlocks. Soule et al. designed a scheduler for streaming applications with hybrid static/dynamic data rates [23]. The scheduler partitions dataflow graph into static subgraphs connected by dynamic rate channels so that both static and dynamic scheduling techniques can be applied to achieve good performance.

Our work is an extension of previous work on using dummy messages to avoid deadlocks in streaming comput-

ing [7], [8]. In the previous work, for each undirected cycle, we assigned the same dummy interval for all channels on the same directed path. In this work, we extend the solution space of dummy interval assignment to a polyhedron. The benefit of this new contribution is that we can safely vary a channel's dummy interval according to its filtering characteristics.

VI. Conclusion

Synchronized filtering dataflow (SFDF) is a dataflow model that allows data-dependent filtering. This filtering, together with the model's synchronization behavior, can cause SFDF applications to require unbounded memory and therefore to deadlock given bounded memory. The dummy message technique can provide guaranteed safe execution in bounded memory. This work precisely characterizes the set of deadlock-free schedules for sending dummy messages and demonstrates that, even among schedules that are both deadlock-free and "maximal" (i.e. no one dummy interval can safely be increased), different choices lead to substantially more or less communication in practice.

Our primary future work will investigate how to exploit the differences among deadlock-free schedules to maximize real application performance while maintaining safety. The relationship of dummy intervals to application throughput and/or latency is a complex, nonlinear function of these intervals. Moreover, the importance of minimizing additional traffic on a given channel depends on how the application is mapped to resources; for example, a high-bandwidth and low-utilization link might be configured to support a high rate of dummy messages in order to minimize additional traffic on some other, more heavily used link. We will investigate algorithmic approaches to optimize performance within the safety constraints defined by the feasible interval polyhedron. These ideas will be implemented and tested empirically in the Auto-Pipe system [4].

Other important theoretical questions involve the SFDF model and the computational cost of working with it. Finding a feasible point in the interval polyhedron naively entails solving an integer linear program; moreover, the number of linear constraints induced for a general DAG may be exponential in its size (though special cases, such as SP-DAGs, admit more compact constraint sets). We will investigate the worst-case complexity of performance optimization within the safe polyhedron, as well as its empirical complexity for real-world application graphs. Finally, we will consider extensions of the SFDF model itself to the case where a single input datum to a node may produce more than one output datum; this extension may require modification of the dummy message protocol, and hence of our other results herein, to maintain safety.

References

[1] E. J. Tyson, J. Buckley, M. A. Franklin, and R. D. Chamberlain, "Acceleration of atmospheric Cherenkov telescope signal processing to real-time speed with the Auto-Pipe design system," *Nuclear Instruments and Methods in Physics Research A*, vol. 585, no. 2, pp. 474–479, Oct. 2008.

[2] A. C. Jacob, J. M. Lancaster, J. Buhler, B. Harris, and R. D. Chamberlain, "Mercury BLASTP: Accelerating protein sequence alignment," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 1, no. 2, 2008.

[3] 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.

[4] R. D. Chamberlain, M. A. Franklin, E. J. Tyson, J. H. Buckley, J. Buhler, G. Galloway, S. Gayen, M. Hall, E. B. Shands, and N. Singla, "Auto-Pipe: Streaming applications on architecturally diverse systems," *IEEE Computer*, vol. 43, no. 3, pp. 42–49, Mar. 2010.

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

[6] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity-the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[7] 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.

[8] J. D. Buhler, K. Agrawal, P. Li, and R. D. Chamberlain, "Efficient deadlock avoidance for streaming computation with filtering," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. ACM, 2012, pp. 235–246.

[9] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, Sep. 1987.

[10] 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, Jul. 2007.

[11] 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.

[12] N. Singla, M. Hall, B. Shands, and R. Chamberlain, "Financial monte carlo simulation on architecturally diverse systems," in *Workshop on High Performance Computational Finance*, nov. 2008.

[13] P. Viola and M. Jones, "Robust real-time object detection," *International Journal of Computer Vision*, vol. 4, 2001.

[14] S. Padmanabhan, Y. Chen, and R. D. Chamberlain, "Convexity in non-convex optimizations of streaming applications," in *ICPADS*. IEEE, 2012.

[15] 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*, Jul. 2010, pp. 333–336.

[16] G. Kahn, "The semantics of simple language for parallel programming," in *IFIP Congress*, 1974, pp. 471–475.

[17] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cycle-static dataflow," *Signal Processing, IEEE Transactions on*, vol. 44, no. 2, pp. 397–408, 1996.

[18] P. K. Murthy and E. A. Lee, "Multidimensional synchronous dataflow," *Signal Processing, IEEE Transactions on*, vol. 50, no. 8, pp. 2064–2079, 2002.

[19] C. Park, J. Jung, and S. Ha, "Extended synchronous dataflow for efficient dsp system prototyping," *Design Automation for Embedded Systems*, vol. 6, no. 3, pp. 295–322, 2002.

[20] J. T. Buck and E. A. Lee, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," in *ICASSP-93*, vol. 1. IEEE, 1993, pp. 429–432.

[21] J. T. Buck, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," Ph.D. dissertation, University of California, Berkeley, 1993.

[22] D. Sbirlea, J. Shirako, R. Newton, and V. Sarkar, "Scnc: Efficient unification of streaming with dynamic task parallelism," in *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2011 First Workshop on*. IEEE, 2011, pp. 58–65.

[23] R. Soulé, M. I. Gordon, S. P. Amarasinghe, R. Grimm, and M. Hirzel, "Dynamic expressivity with static optimization for streaming languages," in *DEBS*, 2013, pp. 159–170.