Report Number: WUCSE-2014-001

2014

Streaming Computations with Precise Control

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

Streaming Computations with Precise Control

Peng Li, Kunal Agrawal, Jeremy Buhler, Roger D. Chamberlain
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, MO 63130
{pengli, kunal, jbuhler, roger}@wustl.edu

Abstract

Streaming computing is a paradigm of distributed computing that features networked nodes connected by first-in-first-out data channels. Communication between nodes may include not only high-volume *data tokens* but also infrequent and unpredictable *control messages* carrying control information, such as data set boundaries, exceptions, or reconfiguration requests. In many applications, it is necessary to order delivery of control messages *precisely* relative to data tokens, which can be especially challenging when nodes can *filter* data tokens. Existing approaches, mainly data serialization protocols, do not exploit the low-volume nature of control messages and may not guarantee that synchronization of these messages with data will be free of deadlock.

In this paper, we propose an efficient messaging system for adding precisely ordered control messages to streaming applications. We use a credit-based protocol to avoid the need to tag data tokens and control messages. For potential deadlocks caused by filtering behavior and global synchronization, we propose deadlock avoidance solutions and prove their correctness. Experimental results show that with our messaging system, we can substantially improve the performance of a sparse-data streaming application by filtering unnecessary data tokens.

1 Introduction

Streaming computing is a paradigm for parallel and distributed computing. A streaming application is a network of computing nodes connected by first-in-first-out (FIFO) data channels. Each node processes incoming data in streaming (equivalently, online or one-pass) fashion. Streaming can exploit common types of parallelism in applications, such as task parallelism, data parallelism, and pipeline parallelism. Example applications include digital signal processing [11], molecular modeling [7], computational biology [9], multimedia [10], and Internet traffic analysis [4]. Frameworks such as StreamIt [17], Auto-Pipe [8], Storm [3], and MillWheel [4] have been implemented to support development of streaming applications.

If a node emits no data on an output channel in response to some input, we say that the node has *filtered* the input on that channel. Filtering is a natural behavior in applications such as machine learning [21] and biological sequence comparison [9]. Other applications do not naturally filter data but can be implemented in a filtering way for higher performance. We consider a classic statistics problem, computing variance of pixel intensities in an image, as a compelling example.

The canonical formula for population variance, denoted by σ^2, is

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(z_i - \overline{z})^2, \tag{1}$$

where $\overline{z}$ is the average of the N values. Equation 1 seems to requires a two-pass calculation process: one pass to compute the mean, and the second to compute the variance using the mean. However, we can convert this computation to a one-pass algorithm [22, 6] that is more streaming-friendly:

$$\sigma^2 = \overline{z^2} - \overline{z}^2 \tag{2}$$

We can implement Equation 2 as a streaming computation as in Figure 1. The source node u duplicates input data to v and w, which compute $\overline{z}^2$ and $\overline{z^2}$ respectively. These quantities are then merged at node x to compute variance values.

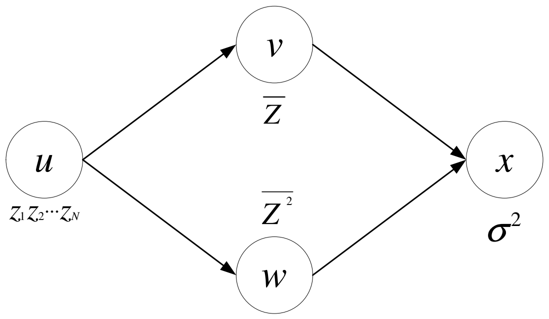

Figure 1: A streaming computation for variance. It occurs as part of large streaming computing systems, including the next generation of VERITAS [20], a ground-based gamma-ray observatory system.

A typical way of computing variances for a stream of images is to process every pixel value until an image boundary is reached, then emit the image's variance. In applications that process sparse images, a lot of pixel values are zero. There is no need for node u to send those zeroes, which consume communication bandwidth and processing time at v and w. Instead, u can filter out all zeroes; however, this means that the number of values received by v and w varies from image to image, and u must promptly notify v and w when an image boundary is reached.

Notifications of image boundaries are a type of *control messages* that are distinct from the stream of pixel values. They are inserted by a node into the filtered stream unpredictably and infrequently, and they impact the behavior of downstream nodes when they arrive. Importantly, control messages must be *precisely ordered* relative to the data stream – it is incorrect for a node to group a pixel from before a boundary with the image after the boundary, or vice versa.

Precisely ordered control messages arise in many streaming computations to ensure correctness and/or to boost performance. The variance example is one case of communicating boundaries between finite-length streams. Exceptions in out-of-order CPUs are another common case where precise ordering is needed relative to an instruction stream, only parts of which are sent to each functional unit. In streaming applications with filtering and synchronization, control messages can also be used to avoid deadlocks [13]. These examples involve nodes that can filter their inputs, though precise ordering is also useful for non-filtering paradigms like SDF.

In this work, we describe an efficient strategy for adding precisely ordered control messages to streaming applications with filtering behavior. We pay particular attention to applications in which the communication channels connecting compute nodes have small, statically determined buffer sizes, and in which control messages are kept separate from the data stream for reasons of performance or ease of implementation. These properties are expected for channels implemented in hardware (e.g. connections among CPU cores or FPGAs) and in software systems where data is strongly typed. Under these circumstances, careful attention must be paid to preserve the desired semantics and to avoid the possibility of deadlocks if buffers become full. We give protocols to ensure precise ordering and deadlock freedom, prove their correctness, and demonstrate experimentally that an optimization enabled by control messages can benefit application throughput.

The rest of the report is organized as follows. Section 2 provides details of our dataflow model and control message delivery requirements. In Section 3, we discuss a protocol to ensure precise ordering semantics between data and control for a pair of nodes. Section 4 extends this protocol to work on arbitrary directed acyclic graphs of nodes with filtering behavior. Section 5 gives experimental results. Related work is discussed in Section 6, and Section 7 concludes.

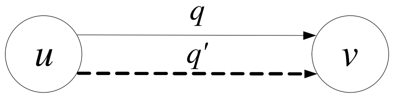

Figure 2: An edge with paired data and control channels q and q'.

2 Background

This section describes the *synchronized filtering dataflow* (SFDF) computing model that forms the basis of our work. We introduced SFDF in [13]; this work extends that model to accommodate separate control and data paths between nodes.

Our model has two tiers: for the first tier, we focus on communication between a pair of sender and receiver, ignoring global synchronization; for the second tier, we bring global synchronization into consideration. If the interested user wants to apply our result, he/she can decide which model to use according the characteristics of the applications. In both tiers, we try to make as few assumptions as possible so that our model can be applied to a broad range of applications.

2.1 Application Topology

An application consists of compute nodes organized in a directed acyclic multigraph. Nodes are connected by one-way *channels*, each of which reliably delivers data from a sender to a receiver in FIFO order. However, channels have no timing guarantee. Each channel has a known, finite buffer capacity that does not change at runtime. We denote by $|q|$ the buffer size of channel q. In practice, it might be possible to shrink or expand the channel buffers of software applications, but in some circumstances (e.g. FPGA applications), it is difficult to change buffer size at runtime, so as a general rule, we assume channel buffers cannot be resized dynamically.

There are two types of channels: *data* and *control*, which carry *data tokens* and *control messages* respectively. As shown in Figure 2, for each data channel q connecting two nodes, there is a parallel control channel q'. (We refer to this pair of channels as the *edge* between the nodes.) A node can choose to listen for input on at most one channel at a time; once a channel is chosen for listening, the node can take no further action until input appears on that channel.

2.2 Filtering and Data Channel Synchronization

When a node of an SFDF application receives and processes input data, it may produce zero or one data token on each of its output data channels. If no token is produced for some input on a given output channel, we say that the node has *filtered* its input on that channel.

When a node takes as input two or more data streams, each of which may be subject to filtering by upstream nodes, the semantics of joining these multiple streams must be clearly defined.

In SFDF, data tokens emitted into a channel bear *strictly increasing* integer indices. In a single computation, a node may consume only data tokens with a common index i, and any output tokens produced by this computation will also have index i. Moreover, the node may not begin computing on data tokens with index i until, for each of its input channels, either the next token on that input has index i, or it is known that *no* token with index i will ever appear. These semantics ensure that, even though different channels are not synchronized, all tokens with a common index, and *only* such tokens, are processed together by a single node. In other words, in a single computation, a node should only consume data tokens with the same index, and all data tokens with the same index emitted by senders must be consumed by the node in only one computation.

Note that, if it is possible for a node to receive inputs on only a subset of its input channels due to filtering, then the application designer must specify the meaning of the node's computation for all such possible subsets.

2.3 Control Channel Behavior

Control channels carry *control messages*, which have one of a finite set of types and can contain arbitrary content. The order in which control and data are processed is *precise*: if a node sends a data token with index i on data channel q of an edge, followed immediately by a control message on the associated control channel q', then this message should be processed by the receiving node after computing on all input data with index i but before consuming data with any index $> i$. A node may send multiple control messages on an edge between two consecutive data indices.

Intuitively, control messages are sent only rarely compared to data tokens. By splitting these messages out into their own channels, we avoid multiplexing them with the data tokens in the higher-volume data channels. This separation permits strong typing assumptions about data channels, which may lead to more efficient implementation; moreover, it simplifies the common case of sending and receiving data between nodes, which may benefit the application's latency and throughput. Unfortunately, while multiplexing data and control in one channel trivially guarantees precise ordering, the same is not true for separate, unsynchronized control and data channels.

In what follows, we first give a protocol to ensure precise ordering of control messages and data tokens on a single edge. This protocol is *independent of SFDF's global synchronization requirements* between data channels. We then show how to extend our protocol to ensure that SFDF applications with control channels execute safely without global deadlock.

3 Ensuring Precise Control-Data Ordering

To guarantee precise ordering of control messages and data tokens between a sender and receiver, we design a protocol in which the sender uses the control channel to enforce the ordering at the receiver. Enforcement is mediated through the use of *credits*.

Consider an edge e consisting of two nodes connected by data and control channels q and q'. We will enforce precise ordering of control messages and data tokens on this edge through the use of *credits*. The sender and receiver each maintain internal credit balances, which are integer values that are initially zero. When a receiver receives some number c of credits on e, its credit balance RCB_e is incremented by c; when it *consumes* a data token on e, RCB_e is decremented by one. The sender's credit balance SCB_e is incremented by one whenever it *sends* a data token; when it sends c credits to the receiver on e, SCB_e is decremented by c.

Credits can be attached to any control message. If credits must be sent but no other control message is pending, the sender may send a *credit message* with no intrinsic content but its attached credit. When the receiver sees a control message with attached credit, it immediately increments its credit balance and may then switch to the data channel and attempt to read data tokens without first processing the control message itself.

3.1 Credit Balance Protocols

Intuitively, a credit represents permission from the sender for the receiver to consume a data token. It implies that *there are no pending control messages that must be processed before consuming the next data token.* The receiver may consume data tokens as long as its credit balance is positive, but when the balance goes to zero, it must wait for the sender either to supply more credits or to send control messages that should be processed before the next data token. The formal protocol followed by the receiver is given in Algorithm 1.

The sender, for its part, must issue credit to consume a pending data token only after it knows that no control message should precede that token. It is not necessary for the sender to issue the credit immediately after the token; indeed, the volume of credit messages can be reduced by letting the data channel buffer tokens until either a control message must be issued or the buffer is full. To this end, Algorithm 2 gives a sender's protocol parametrized by a threshold T, which should be set less than the buffer size of the outgoing data channel. When the threshold is exceeded with no intervening control messages, the sender issues credit to drain the data channel's buffer. Note that, in this and all following protocols, all **emit** operations block until the output channel is not full.

Algorithm 1: Receiver Credit Balance Protocol

```
while $RCB = 0$ do
    wait for a control message on $q'$
    let $c$ be credit value carried by message
    if $c = 0$ then
        consume message
    else
        Detach $c$ credits from message
        $RCB \leftarrow RCB + c$
wait for a data token on $q$
consume token
$RCB \leftarrow RCB - 1$
```

Algorithm 2: Sender Credit Balance Protocol

```
if token is ready then
    emit token on $q$
    $SCB \leftarrow SCB + 1$
while control message is ready OR $SCB > T$ do
    emit message on $q'$ with $SCB$ credits
    $SCB \leftarrow 0$
```

3.2 Correctness and Safety

We argue that the sender and receiver protocols ensure precise ordering of control messages vs. data tokens.

Theorem 3.1. *If a receiver and sender are connected by an edge and behave as in Algorithms 1 and 2, and the sender issues a data token d followed by a control message m, then the receiver will process m after d but before the next token following d.*

Proof. The sender's protocol never sends the credit necessary to consume a data token before sending the token itself. Hence, when d is sent, the receiver does not have the credit needed to accept it. This credit is sent only with control message m and is sufficient only to process d and any unreceived data tokens sent prior to d. Hence, the receiver sees m, uses its credits to accept d and any prior tokens, and then processes m.

For any data token d' sent after d, the receiver will not receive the credit needed to accept it until after processing m. □

The above argument assumes that the sender and receiver are always able to make progress. Because the data and control channels have finite buffers, the sender could at some point be blocked trying to send a data token or control message into a channel with a full buffer, or the receiver could be blocked waiting for tokens or messages when none are yet visible to it. If both the sender and the receiver are blocked indefinitely, the system is deadlocked. We now verify that our protocol makes such a deadlock impossible.

Theorem 3.2. *If a receiver and sender are connected by an edge and behave as in Algorithms 1 and 2, this pair of nodes will never deadlock.*

Proof. To verify freedom from deadlock, we must check that two bad cases never occur. (These are special cases of the general *blocking cycle* described in [14, 15], which is proved there to be the only way an SFDF application can deadlock.)

- *Case 1.* The sender is blocked writing a full data channel q while the receiver is blocked reading an empty control channel q'.
- *Case 2.* The sender is blocked writing a full control channel q' while the receiver is blocked reading an empty data channel q.

We first address Case 1. If the data channel is full, but the receiver is reading the control channel, then the receiver has no credits to consume data tokens. If no control message with credits is in flight, then the sender has sent $|q|$ data tokens without sending any credits. Since the sender's threshold $T = |q|$, it would have sent credits before trying to send token $|q| + 1$, which contradicts the assumption that no credits are in flight. Hence, the receiver will be able to drain the data channel after finite time, and the nodes are not deadlocked.

We now consider Case 2. If the control channel is full, but the receiver is blocked reading the data channel, then the receiver has at least one unexpended credit. But the sender never issues such credits before issuing the corresponding data tokens. Hence, there must be enough data tokens in flight to expend the receiver's credits, and it will consume them and switch to reading the control channel after finite time. □

4 Combining SFDF with Precise Control

We now explore how to combine SFDF's synchronization of multiple, possibly filtered input streams with the use of separate data and control channels. Recall that an SFDF application is a directed acyclic multigraph. Each edge e of this multigraph now consists of two channels: a data channel q_e, and a control channel q'_e. Each edge also holds variables sufficient to implement the credit protocols of the previous section, including sending and receiving credit balances SCB_e and RCB_e and a threshold T_e that is smaller than data channel buffer size $|q_e|$.

Algorithm 3 describes how to combine SFDF with control channels. To ensure precise data and control ordering, each node implements Algorithm 1 on each of its input edges and Algorithm 2 on each of its output edges. Edges are processed sequentially in an arbitrary order. To synchronize across data channels, the receiving protocol is split into two parts: part one ensures that data tokens are available on all input edges' data channels, while part two decides which tokens to read (based on their indices) in order to start the next computation. The common index i of tokens consumed by a computation at a node is called the *computation index.* Note that no attempt is made to synchronize control messages across edges.

Not every node in an application may have inputs or outputs. In particular, *source nodes* have no inputs but rather generate tokens and messages spontaneously, following only the output part of the protocol.

Unfortunately, this straightforward combination of SFDF and the credit protocols is prone to deadlock. We explore this issue and its remediation next.

4.1 Deadlocks Due to Full Data Channels

To classify possible deadlocks in SFDF networks with control, we first review the notion of *blocking.*

Definition 1 (**Blocking Relation**)**.** If a node v is waiting for input (either control or data) 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, denoted $u \dashv v$. If there exists a sequence of nodes $v_1 \ldots v_n$ such that $v_i \dashv v_{i+1}$ for $1 \leq i < n$, we write $v_1 \dashv^+ v_n$.

The following result of [13] also applies to SFDF with control channels, using essentially the same proof:

Theorem 4.1 (**Deadlock Theorem**)**.** *An SFDF application deadlocks iff during the computation, there exists a node u s.t. $u \dashv^+ u$ and there are unprocessed data tokens or control messages in some channel.*

In other words, deadlock is equivalent the presence of a *blocking cycle* in the application graph.

To further focus the discussion, we make two simplifications. First, we will assume until otherwise stated that *no control channel ever becomes full* during a computation. This is intuitively reasonable if control messages are sent much less frequently than data tokens. Second, we observe that, because a node always sends the credit to receive a data token after the token itself, *a node cannot block indefinitely on an empty data channel.* Indeed, if a node is waiting on a data channel, then it has unexpended credit, which means the corresponding data token is already in flight.

With the above simplifications, a blocking cycle must contain only two types of edges: *full* data channels and *empty* control channels. The following example shows that such a deadlock is possible. Consider four nodes connected as in Figure 1 above, with edges uv, vx, uw, and wx. Every computation of u produces

Algorithm 3: Single-node behavior in SFDF with control messages.

```
foreach input edge $e$ do
    while $RCB_e = 0$ do
        wait for a control message on $q'_e$
        let $c$ be credit value carried by message
        if $c = 0$ then
            consume message
        else
            Detach $c$ credits from message
            $RCB_e \leftarrow RCB_e + c$
    wait for a data token on $q_e$
let $i$ be least index among data tokens on all edges
foreach input edge $e$ do
    if token on $q_e$ has index $i$ then
        consume token
        $RCB_e \leftarrow RCB_e - 1$
perform computation for index $i$
foreach output edge $e$ do
    if token is ready on $e$ then
        emit token on $q_e$ with index $i$
        $SCB_e \leftarrow SCB_e + 1$
    while control message is ready on $e$ OR $SCB_e > T_e$ do
        emit message with $SCB_e$ credits
        $SCB_e \leftarrow 0$
```

data tokens on q_{uv} and q_{uw}, and every computation of v produces a data token on q_{vx}; however, w filters more than half of its inputs on q_{wx}. Assume the data channels on all four edges have the same buffer size 32, the threshold for scheduling credit messages is $T = 31$ (recall that a credit message is prompted if buffered tokens are more than T), and that no control messages are sent other than credit messages. After some computations, the system reaches the state shown in Figure 3.

At this point, if u does one more computation (and w filters the resulting data token), then we have that (1) u is blocked by v on a full q_{uv}; (2) v is blocked by x on a full q_{vx}; (3) x is blocked by w waiting for credit on an *empty* control channel q'_{wx}; and (4) w, which has no pending tokens and hence no credit, is blocked by u waiting for credit on the empty control channel q'_{uw}. Hence, the system is deadlocked with a blocking cycle.

This example actually illustrates two related but distinct causes of deadlock. If w sends *no* data on q_{wx}, then deadlock occurs because x has no input on this channel but does not know that none will arrive. This kind of deadlock also occurs in SFDF networks without control channels [13]. If, however, w sends *some* data on q_{wx}, x has enough data to make progress, but in the absence of control messages, nothing prompts w to send its stored credit to enable x to use the data. This kind of deadlock is a side effect of the credit protocols. Below, we propose a modified protocol to avoid both causes of deadlock.

4.2 Avoiding Deadlocks with Periodic Communication

To avoid deadlock, we modify our protocol in two ways. First, we periodically flush pending credit from the sender to the receiver, so that data tokens cannot linger indefinitely at the receiver with no credit. Second, we periodically notify the receiver if tokens with consecutive indices have been filtered by upstream senders, using a new kind of control message called a *dummy message* that carries the index of the sender's most recent computation.

The augmented protocol is shown in Algorithm 4. The receiver's protocol is essentially unchanged, except that, instead of a data token with index i, an edge may present a dummy message with index $j \geq i$, which implies that no token with index i will ever be received on that edge, and it should therefore not block

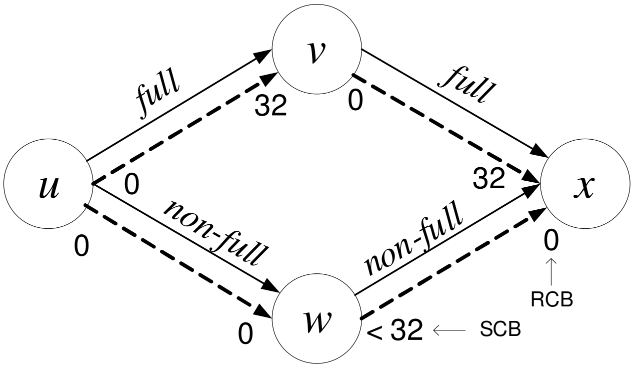

Figure 3: A deadlock example. w filters 46 of 64 consumed data tokens and no other node filters data. Now data channels uv and vx are full, blocking u and v; SCB values for uw and wx are not big enough to prompt credit messages, blocking w and x.

computation i from proceeding.

The sender's protocol is augmented with two state variables: LastSentIdx_e, which tracks the index of the last data token *actually sent* by the sender, and LastRecvIdx_e, which tracks the last index for which the receiver has *permission* to consume inputs with that index from e. If the sender does too much work (as measured by the size of the gap between the index i of the most recent computation and LastRecvIdx_e) without enabling the receiver to proceed, then it either flushes its pending credit for any data tokens sent in this gap, or, if no tokens were sent, transmits a dummy message with index i to tell the receiver not to expect them. The largest permissible gap size for an edge e is called its *heartbeat interval*, denoted in the protocol by $[e]$.

The remaining question is how large to make the heartbeat interval for each edge. It would be trivially safe to set $[e] = 0$ for every e, but doing so would flush credit or send a dummy after every computation, which would add excessive communication overhead. Instead, we utilize the following scheme adapted from [15]. Given a dataflow graph G, for each *undirected cycle* C of G, suppose the set of clockwise edges is H_1 and the set of counterclockwise edges is H_2. We enforce the following inequality constraints for cycle C:

$$\Sigma_{e\in H_1}[e] \quad < \quad \Sigma_{e\in H_2}|q_e| \tag{3}$$

$$\Sigma_{e\in H_2}[e] \quad < \quad \Sigma_{e\in H_1}|q_e|. \tag{4}$$

An application graph may have more than one undirected cycle, in which case each such cycle generates a pair of constraints as described. We also need to avoid local deadlocks, so the following constraint, which we specified for the sender's protocol of Section 3, is added for each edge e:

$$[e] < |q_e|. \tag{5}$$

The union of all these constraints defines a feasible polyhedron of heartbeat intervals for the application, and we select a set of intervals from this feasible region.

Theorem 4.2. *Assuming that control channels never become full, if every node in an SFDF application behaves as Algorithm 4 with heartbeat intervals constrained by Inequalities (3), (4), and (5), then the application cannot deadlock.*

Proof. As noted above, if control channels never become full, the only possible deadlocks involve full data and empty control channels.

WLOG, the history of control messages that leads to deadlock includes only credit messages and dummy messages; other control messages never cause a node to stop listening on a channel, and control channels are never full, so these other messages do not impact ability to make progress. Also, since control channels never fill, an application that deadlocks will still deadlock if we set every control channel buffer size to an arbitrarily large value.

Algorithm 4: Adding dummy messages to SFDF with control.

```
foreach input edge $e$ do
    while $RCB_e = 0$ do
        wait for a control message on $q'_e$
        let $c$ be credit value carried by message
        if $c = 0$ then
            if message is a dummy then
                break
            consume message
        else
            Detach $c$ credits from message
            $RCB_e \leftarrow RCB_e + c$
    if $RCB_e > 0$ then
        wait for a data token on $q_e$
let $i$ be least index among tokens on edges with
  $RCB > 0$ and dummies on edges with $RCB = 0$
foreach input edge $e$ do
    if $RCB_e > 0$ AND token on $q_e$ has index $i$ then
        consume token
        $RCB_e \leftarrow RCB_e - 1$
    else if dummy on $q'_e$ has index $i$ then
        discard dummy
perform computation for index $i$
foreach output edge $e$ do
    if token is ready on $e$ then
        emit token on $q_e$ with index $i$
        $SCB_e \leftarrow SCB_e + 1$
        $\mathrm{LastSentIdx}_e \leftarrow i$
    if $SCB_e = 0$ AND $i - LastRecvIdx_e > [e]$ then
        emit dummy on $q'_e$ with index $i$
        $\mathrm{LastRecvIdx}_e \leftarrow i$
    while control message is ready on $e$ OR $i - LastRecvIdx_e > [e]$ do
        emit message on $q'_e$ with $SCB_e$ credits
        $SCB_e \leftarrow 0$
        $\mathrm{LastRecvIdx}_e \leftarrow \mathrm{LastSentIdx}_e$
```

In [15, Algorithm 1], we gave a protocol for avoiding deadlock in SFDF networks in which dummy messages are multiplexed with data tokens on the data channel, and no control channels exist. We will leverage this result to show that Algorithm 4 is also deadlock-free. Let Γ be an SFDF application with control channels. We construct a similar SFDF application Φ without control channels and give a mapping from histories of data and control transmission in Γ to histories in Φ. We then argue that (1) every history in Γ that follows Algorithm 4 with heartbeat intervals as described maps to a provably deadlock-free history in Φ, and (2) if the mapped history in Φ does not end in deadlock, neither does the original history in Γ.

To form Φ, clone the dataflow graph of Γ, including nodes and edges. For each channel pair q_e^{γ} and $q_e^{'\gamma}$ of Γ, create in Φ data channels q_e^{ϕ} and $q_e^{'\phi}$ with the same buffer sizes in Γ.

We map histories in Γ to histories in Φ as follows. For *each computation* at a node Γ, the corresponding node of Φ performs a computation with the same index. After the computation, if a data token is emitted on q_e^{γ}, emit a data token with the same index on q_e^{ϕ}; if a credit message is emitted on $q_e^{'\gamma}$, emit a dummy message with the index of $\mathrm{LastRecvIdx}_e$ on $q_e^{'\phi}$; if a dummy message is emitted on $q_e^{'\gamma}$, emit a dummy message with the same index on *both* q_e^{ϕ} and $q_e^{'\phi}$.

With the mapping from Γ to Φ, we make the following claims.

Claim 4.3. *For any computation history in Γ that follows Algorithm 4 with heartbeat intervals as indicated, the mapped computation history in Φ completes without deadlock.*

Proof. Due to space constraints, we only sketch the proof. It may be verified that, given Algorithm 4 and our mapping, each node of Φ issues dummies in exactly the way directed by the deadlock avoidance protocol for SFDF in [15, Algorithm 1]. Moreover, the *dummy intervals* (i.e. maximum number of data tokens that can be filtered without sending a dummy message) for each edge in Φ can be computed based on the corresponding heartbeat intervals in Γ. It may then be verified that these intervals meet the inequality criteria given in [15, Section III.C] (which are essentially identical to Inequalities 3 and 4) that guarantee that Φ's computation can never deadlock. ■

Claim 4.4. *If node v^{ϕ} of Φ can advance its computation index (CI) to i_v, so can the corresponding node v^{γ} of Γ.*

Proof. We prove by induction on tuple (v, i^v).
Bas. v is a source node and $i = 1$, trivially true.
Ind. Suppose $u_1, u_2, \ldots, u_k$ are topological predecessors of v. In order for v^{ϕ} to advance its CI to i_v, their CI's first have to be advanced to at least $i_{u_1}, i_{u_2}, \ldots, i_{u_k}$, respectively. According to the IH, u_j^{γ} has also advanced its CI's to at least $i_{u_j}, j = \{1, 2, \ldots, k\}$.

During v^{ϕ}'s computation on index i_v, on each channel pair q_e^{ϕ} and $q_e^{'\phi}$, according to the construction of Φ, v^{ϕ} either (1) consumes a data token with index i_v on q_e^{ϕ} and sees a dummy message with index $\geq i_v$ on q_e' (it also discards the dummy message if its index is i_v) or (2) discards a dummy message with index i_v on both q_e^{ϕ} and $q_e^{'\phi}$.

According to the mapping from Γ to Φ, if case (1) happens, v^{γ} receives a data token with index i and has credit to consume it; if case (2) happens, v^{γ} receives a dummy message on $q_e^{'\gamma}$. In either case, v^{γ} can finish processing input on edge e. After v^{γ} finishes processing all input edges, it computes and sends output, including data tokens, dummy messages, and credit messages. Data tokens and dummy messages are one-to-one mapped to those in Φ, so they will be sent successfully. A credit message is mapped to a dummy message, so there is no problem in sending credit messages. Hence, v^{γ} can advance to i_v. ■

Conclude that, since a mapped history of computations in Φ is always able to make progress, and a computation in Γ makes progress whenever the mapped computation in Φ does, it must be that an arbitrary computation in Γ can always make progress, and hence Γ is deadlock-free. □

4.3 Deadlocks Due to Full Control Channels

We assumed previously that no node is ever blocked due to a full message channel. If the number of control messages generated per data token is *a priori* bounded, this assumption can be enforced by statically allocating a large enough buffer for each control channel. In particular, for each edge e, if we set $|q_e'| > m|q_e|$, then the edge's sender can safely emit up to m control messages per data token. The larger control buffer guarantees that q_e will fill before q_e', so that the sender blocks on the data channel, not the control channel. In practice, it might be possible to derive weaker bounds, e.g. that a node never sends more than b control messages for each d data tokens, in which case we could set $|q_e'| > b/d \cdot |q_e|$.

If we do not *a priori* bound the number of control messages sent per data token, a new type of deadlock involving full control channels is possible, as the following example shows. Consider the same four nodes of Figure 1. For computation index 1, u sends a data token to v, which in turns sends a token to x, but u sends nothing to w. v then attempts to send $|q_{vx}'| + 1$ control messages to x. After the first control message, x has credit for edge vx and then blocks waiting for credit on edge q_{wx}'. Hence, v eventually blocks on the full control channel q_{vx}'. If u then attempts to send $|q_{uv}'| + 1$ control messages on edge uv, that edge's control buffer will fill as well. At this point, (1) u is blocked by v on the full q_{uv}'; (2) v is blocked by x on the full q_{vx}'; (3) x is blocked by w on the empty q_{wx}'; and (4) w is blocked by u on the empty q_{uw}'. Hence, the system is deadlocked.

One way to avoid deadlocks on full control channels is to utilize the protocol of Algorithm 4, setting the heartbeat interval $[e]$ to 0 for *every* edge e. This causes a node to schedule a dummy message for *every* filtered data token and to always send a credit immediately after sending a data token.

Theorem 4.5. *If every node in an SFDF application behaves as in Algorithm 4 using heartbeat interval* 0 *for every edge, the application cannot deadlock, even if the control messages sent per data token are unbounded.*

Proof. We first prove that all nodes can advance to the same computation index as source nodes do.

Claim 4.6. *If all source nodes can advance their computation indices (CI) to I, so can all nodes.*

Proof. We prove it in two steps. First, we prove that all nodes can advance their CI to 1, the lowest computing index by induction on the topology of the dataflow graph. Let $u_0, u_1, \ldots, u_n$ be a fixed topological order of the application graph.

Bas. Source nodes can advance CI to 1.

Ind. Suppose $u_0, u_1, \ldots, u_k$ can advance CI to 1, which means for node u_{k+1}, all its predecessors have advanced their CI to 1. For each incoming edge e of u_{k+1}, u can consume control messages on q'_e until seeing a credit message, if the upstream node does not filter the data token on q_e, or a dummy message, if the upstream nodes filters the data token. In either case, u_{k+1} is able to finish waiting on e to proceed to the computation. After finishing computation, u_{k+1} sends control messages and possible data tokens on output channels. It may send multiple control messages on a output channel, but all control messages can be consumed by the receiver asynchronously, so it won't be blocked indefinitely by an output message channel; it sends at most one data token, so it won't be blocked indefinitely by an output data channel.

Hence all nodes can advance their CI to 1.

Now we prove the claim with an induction on CI. Suppose all nodes can advance their computing index to i, we show that if all source nodes can advance their CI to $i+1$, so can all nodes. Indeed, we know all nodes can finish computing on index i and clear channel buffers for computing on index $i+1$, with a similar induction on a fixed topological order, as we just did for index 1, we can prove all nodes can advance CI to $i+1$. ■

We can see that when source nodes start a new computation index, all nodes of the application can advance to that computation index, which means no deadlock can happen. □

4.4 Adding Output Buffers

In distributed computing, nodes often use local *output buffers* to reduce amortized communication overhead. Outgoing data tokens are simply queued in the node's output buffer until they can be sent in a batch. An output buffer is simply a special, sender-controlled case of the abstract channel buffers in SFDF, so provided these buffers are of size at least $[e]$ for every edge, and sending credits causes the output buffer to be flushed, we can still describe the application's behavior by Algorithm 4 and so can guarantee deadlock freedom.

5 Experimental Evaluation

We have implemented support for precisely ordered control messages in filtering SFDF on top of Auto-Pipe, a framework for streaming applications [8]. To evaluate the performance impact of filtering and control messages, we implemented the streaming application for computing variance described in Figure 1.

In our experiments, node u generates simulated VERITAS images, each consisting of $32 \times 32 = 1024$ integer-valued pixels. Nodes v and w compute the mean and mean-of-squared-values for each image, respectively, and x receives these and computes standard deviations. We tested images with 10%, 30%, 50%, 70%, and 90% random zeros. We set heartbeat intervals appropriately to ensure deadlock freedom, as described above. To simulate the case in which the application is implemented without filtering, we tested the filtering implementation with a heartbeat interval of 0. We implemented the application both with and without per-node output buffers to compare throughputs. We ran experiments on a 2.6-GHz, six-core AMD Opteron processor. Each node of the application was mapped onto a separate physical processor core. Communication channels were implemented in shared memory.

Figure 4 illustrates observed throughput (in images/second) for increasingly sparse images when the heartbeat interval is set to 16 for each edge. (Qualitatively similar results were observed for intervals of 32, 64, and 128.) For sparse images, filtering greatly improves application throughput. Profiling reveals that node w, which computes the mean of squares, is the bottleneck in the pipeline. Node w's workload decreases linearly as the filtering ratio increases.

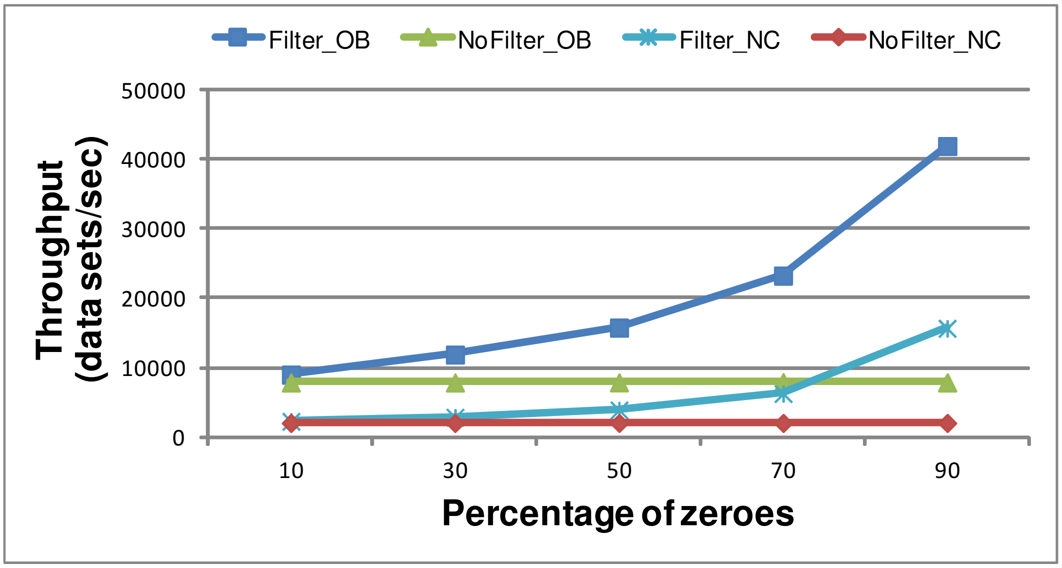

Figure 4: Throughput of variance application vs. rate of filtering (heartbeat interval = 16). Filter_OB, NoFilter_OB, Filter_NB, NoFilter_NB represent: filtering w/ output buffer, non-filtering w/ output buffer, filtering w/o output buffer, and non-filtering w/o output buffer.

This experiment provides an example of how filtering unnecessary data in streaming applications can boost throughput, even with the overhead needed to implement precise control and avoid deadlock. We further investigated the impact of local output buffers as a strategy to limit the overhead of copying data through shared memory buffers. With output buffers, observed throughput increased by 3-4x.

6 Related Work

The control messaging system proposed in this paper is based on synchronized filtering dataflow (SFDF), which can be viewed as Homogeneous Synchronous Dataflow (HSDF) [12] with the addition of node filtering. HSDF is a special type of Synchronous Dataflow (SDF) [11] where the data rate (the number of items a node reads/writes from its input/output channels) is 1 for all channels. SFDF applications are vulnerable to deadlocks with finite buffer capacity due to filtering and synchronization. We previously described the use of dummy messages, a special type of control message, sent in-band with data streams to avoid deadlock [13, 14, 5, 15]. The system in this paper incorporates dummy messages but instead delivers them through dedicated control channels.

Synchronizing data streams and control messages is also common in network protocols. For example, the Internet Control Message Protocol (ICMP) is designed to exchange control messages between two Internet devices during data transmission [16]. Those protocols are too complex for streaming computing, where nodes are usually tightly coupled and a simpler, higher-performance protocol is preferred. While tagging-based data serialization procotols (e.g. Thrift [1] and Protocol Buffers [2]) can be an option, they are inefficient in supporting infrequent control messages. Moreover, serialization breaks the regularity of data streams, making it hard to avoid deadlocks.

The work most closely related to ours is StreamIt's Teleport Messaging system [18, 19]. Both their work and our work address the problem of sending infrequent and irregular control messages for streaming applications computing on regular data streams. The key difference is that Teleport Messaging is based on the SDF model and uses dependence analysis for precise event handling, while our precise control mechanism, the Credit Balance Protocol, does not rely on any specific model. We also provide remedy for deadlocks when the messaging system works with a synchronized filtering dataflow model, which is not discussed in Teleport Messaging.

7 Conclusion and Future Work

Precisely ordered control messages are important to the correctness and performance of streaming applications. In this work, we have designed and implemented a messaging system that works for streaming computations in which nodes can filter their inputs. It can work even in streaming pipelines that require global synchronization, as in SFDF. We have given protocols and sufficient design constraints to avoid deadlocks while delivering precise control even in the presence of filtering. Experimental results show that filtering with our protocol can substantially improve a sparse-data streaming computation's throughput.

In the future, we will further investigate the performance impact of precise control messages. While we have increased application throughput, increasing buffer sizes will also increase end-to-end latency. Choosing static buffer sizes to balance throughput and latency considerations is a problem for future work. Another open problem is how to efficiently find the highest-throughput set of heartbeat intervals that satisfy the constraints of Inequalities (3), (4), and (5). Currently, computing a satisfactory (not necessarily maximal) set of intervals given the buffer sizes requires superpolynomial time in the application size. We will investigate faster interval selection algorithms with stronger guarantees.

References

[1] Apache thrift. Accessed: June 30, 2014.

[2] Protocol buffers. Accessed: June 30, 2014.

[3] Storm. Accessed: June 30, 2014.

[4] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proc. VLDB Endowment*, 6(11):1033–1044, 2013.

[5] Jeremy D Buhler, Kunal Agrawal, Peng Li, and Roger D Chamberlain. Efficient deadlock avoidance for streaming computation with filtering. In *Proc. 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 235–246. ACM, 2012.

[6] Tony F Chan, Gene H Golub, and Randall J LeVeque. Algorithms for computing the sample variance: Analysis and recommendations. *The American Statistician*, 37(3):242–247, 1983.

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

[8] Mark A. Franklin, Eric J. Tyson, James H. Buckley, Patrick Crowley, and John Maschmeyer. Autopipe and the X language: A pipeline design tool and description language. In *IEEE Int'l Parallel and Distributed Processing Symp.*, 2006.

[9] Arpith C. Jacob, Joseph M. Lancaster, Jeremy Buhler, Brandon Harris, and Roger D. Chamberlain. Mercury BLASTP: Accelerating protein sequence alignment. *ACM Transactions on Reconfigurable Technology and Systems*, 1(2), 2008.

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

[11] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proc. IEEE*, 75(9), 1987.

[12] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, January 1987.

[13] Peng Li, Kunal Agrawal, Jeremy Buhler, and Roger D. Chamberlain. Deadlock avoidance for streaming computations with filtering. In *Proc. 22nd ACM Symp. on Parallelism in Algorithms and Architectures*, pages 243–252, 2010.

[14] Peng Li, Kunal Agrawal, Jeremy Buhler, Roger D. Chamberlain, and Joseph 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.

[15] Peng Li and Jeremy Buhler. Polyhedral constraints for bounded-memory execution of synchronized filtering dataflow. *Workshop on Data-Flow Execution Models for Extreme Scale Computing*, September 2013.

[16] Jon Postel. Internet control message protocol. *RFC-792*.

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

[18] William Thies. *Language and Compiler Support for Stream Programs.* PhD thesis, Massachusetts Institute of Technology, Feb 2009.

[19] William Thies, Michal Karczmarek, Janis Sermulins, Rodric Rabbah, and Saman Amarasinghe. Teleport messaging for distributed stream programs. In *Proc. 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 224–235. ACM, 2005.

[20] Eric J Tyson, James Buckley, Mark A Franklin, and Roger 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 Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 595(2):474–479, 2008.

[21] P. Viola and M. Jones. Robust real-time object detection. *Int'l J. Computer Vision*, 57(2), 2002.

[22] BP Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.