Deadlock-free Buffer Configuration for Stream Computing

Peng Li Jonathan Beard Jeremy Buhler
Dept. of Computer Science and Engineering
Washington University in St. Louis
St. Louis, MO 63130, USA
{pengli, jbeard, jbuhler}@wustl.edu

ABSTRACT

Stream computing is a popular paradigm for parallel and distributed computing, which features computing nodes connected by first-in first-out (FIFO) data channels. To increase the efficiency of communication links and boost application throughput, output buffers are often used. However, the connection between the configuration of output buffers and application deadlocks has not been studied. In this paper, we show that bad configuration of output buffers can lead to application deadlock. We prove necessary and sufficient condition for deadlock-free buffer configurations. We also propose an efficient method based on all-pair shortest path algorithms to detect unsafe buffer configurations. We also sketch a method to adjust an unsafe buffer configuration to a safe one.

Categories and Subject Descriptors

F.1.2 [**COMPUTATION BY ABSTRACT DEVICES**]: Modes of Computation; H.3.4 [**SYSTEMS AND SOFTWARE**]: [Distributed Systems]

Keywords

Stream Computing, Buffer Configuration, Deadlock Avoidance

1. INTRODUCTION

Stream computing is a paradigm of parallel and distributed computing featuring computing *nodes* connected by data channels. Each node runs an application *module* and processes data in first-in first-out (FIFO) order. Data channels deliver data, also in FIFO order. The sequence of data items delivered by a data channel is called a *data stream*. Figure 1 is a stream computing system for approximating population variance, which can be calculated with the following formula [30]:

$$\sigma^2 = \overline{z^2} - \overline{z}^2 \quad (1)$$

where $\overline{z}$ is the average of the N values.

The source node u duplicates input data to v and w, which compute $\overline{z}$ and $\overline{z^2}$ respectively for each data set. These quantities are then merged at node x to compute estimated variance values.

PMAM '15, February 7-8, 2015, San Francisco Bay Area, USA.

ACM 978-1-4503-3404-4/15/02 ..$15.00.
http://dx.doi.org/10.1145/2712386.2712403.

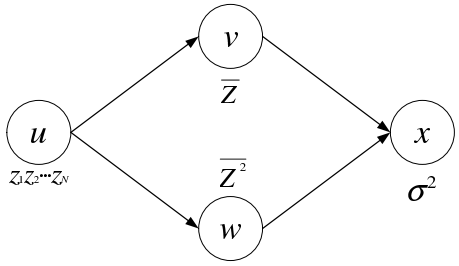

Figure 1: A stream computation for variance.

This simple example demonstrates two benefits of stream computing. First, while benefiting from parallel execution, the application developer can think sequentially when authoring each application module, which is very helpful since most of today's programmers still prefer sequential programming. Second, the FIFO order of data delivery and data processing makes it possible to reason about formal properties of streaming applications, such as the fix-point property [13] and deadlock freedom [4, 18].

While in theory we can think each data channel as a single (bounded) buffer, in practice, it usually consists of an output buffer visible only to the sender, an input buffer visible only to the receiver, and possible transmission buffers in between. The use of output and input buffers are usually for performance considerations since each send or receive operation incurs some fixed overheads. By buffering some data items locally and sending them in one operation, we can amortize overhead per data item and thus improve throughput, which has been observed not only in streaming applications[24, 27] but also in other domains [25, 26]. This "batching" idea has been implemented in some streaming computing systems [11, 31]. Note amortizing communication overhead is just one way of improving throughput of streaming application, which can also be optimized in many other ways [12, 19].

In addition to performance, output buffers might also impact application correctness. More specifically, they can lead to potential application deadlocks. To the best of our knowledge, the deadlock implications of output buffers have not been studied before. In this paper, we will show that in typical streaming applications, if output buffers are not configured appropriately, deadlock can happen during application execution. We will present necessary and sufficient conditions that make a buffer configuration vulnerable to deadlocks. We also provide an algorithm to check whether a buffer configuration is freedom of deadlocks.

2. MODEL DESCRIPTION

In this section, we present the stream computing model we will

use in this paper. The model is very straightforward and easy to understand. If the reader is familiar with the Synchronous Dataflow (SDF) [15, 16], our model is essentially an acyclic homogeneous SDF, where each port has a data rate of one.

An application is represented as a *directed acyclic graph* (DAG) topology where modules and channels are vertices and edges, respectively. We do not consider feedback channels in this paper, but we may consider it in future work. [28] shows that feedback channels are uncommon in stream computing applications.

As in ordinary streaming applications, data delivery and data processing are in FIFO order. We add a *data rate* restriction that each node consumes *exactly* one data item, or *token*, from each input channel (if the node is not a source node) and produces*exactly* one data item on each output channel (if the node is not a sink node), regardless of data content. Data filtering and dynamic data rates are not allowed.

Each channel q has a *static* buffer size, denoted as $|q|$, which is determined when the application is constructed and stays unchanged during application execution. Each channel has an output buffer, which is part of the channel buffer; hence, the size of the output buffer, denoted as $|q|_o$, is smaller than the total channel buffer. Remember that *data in the output buffer is invisible to the receiver*. The sender can choose to flush the output buffer and make the data visible to the receiver (after some finite time) at any time (e.g. due to some control events [17, 22]); however, if the output buffer becomes full, the sender *must* flush it. If the rest of the channel buffer does not have enough capacity to buffer all data to be flushed from the output buffer, it accepts as much data as it can, and the remaining data remains in the output buffer.

If the input data stream is bounded, then after the source node finishes reading all input data, it sends an end-of-stream (EOS) token to downstream receivers to flush any remaining data in output buffers. All nodes receiving the EOS token must propagate it, so that it eventually reaches the sink node.

If a channel is full, neither the output buffer nor the rest of the buffer can accept any data item, and the sender is blocked; if the receiver does not see input at one of its input channels, the receiver is blocked. Note that even if a channel is not empty, e.g. data are in the output buffer, the receiver could still be blocked because the data in the output buffer is not available to the receiver. Blocking on non-empty channels is a key factor in the deadlocks we study below.

3. DEADLOCK CHARACTERIZATION

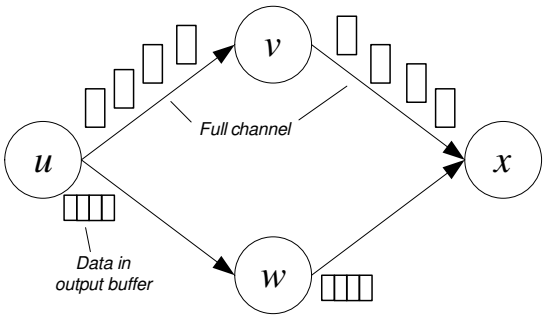

Figure 2: A deadlock due to bad output buffer configuration. Both uw and wx have an output buffer size of 8, so output buffers are not flushed.

We now demonstrate how a deadlock can happen under in the presence of output buffers. As an example, consider Figure 2, which has a topology similar to Figure 1. The buffer configuration is $|uv| = 4$, $|vx| = 4$, $|uw| = 16$, $|uw|_o = 8$, $|wx| = 16$, $|wx|_o = 8$; we ignore $|uv|_o$ and $|vx|_o$ for simplicity. After u sends 4 data items to both v and w, it flushes the output buffer of uw due to some control event although the buffer is not full. It then sends another 4 data items to both v and w, but this time it does not flush uw's output buffer, and wx's output buffer is not flushed, either. Now uv and vx are full, blocking u and v, respectively; the output buffers of uw and wx each holds 4 data items, making w and x see no data on uw and wx, respectively, so w and x are also blocked. None of the four nodes can make any progress, yet there are data unprocessed in the system, so the application is deadlocked.

The buffer configuration for this simple topology is obviously a bit contrived, and it is easy to identify deadlock risk within it. More complex topologies, however, are not so straightforward to analyze. We therefore begin by identifying those topologies in which a deadlock can occur.

Before proceeding to the analysis of properties that lead to potential deadlocks (or freedom of deadlock), let us clarify definitions. Many of the following definitions have been presented in [18].

DEFINITION 3.1. *(**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*, denoted $u \dashv v$. If there exists a sequence of nodes $v_1 \dots v_n$ such that $v_i \dashv v_{i+1}$ for $1 \leq i < n$, we write $v_1 \dashv^+ v_n$.*

DEFINITION 3.2. *(**Deadlock**) A system is said to* deadlock *if no node in the system can make progress, but some channel in the system still retains unprocessed data items (so that the computation is incomplete).*

THEOREM 3.3 (**DEADLOCK THEOREM**). *A system eventually deadlocks if and only if, at some point in the computation, there exists a node u s.t. $u \dashv^+ u$.*

PROOF. ($\leftarrow$) Suppose that at some point in the computation, there is a node u such that $u \dashv^+ u$. Because a blocked node cannot make progress, no node on the cycle involving u can make progress. Hence, once the blocking cycle occurs, it will remain indefinitely. Moreover, not every pair of successive nodes in the cycle can be linked by an empty channel; otherwise, we would have that u is waiting for input from u, which is impossible because the graph of computing nodes is a DAG. Hence, the blocking cycle contains at least one full channel, which means there are unprocessed data items, and so the system is deadlocked.

($\rightarrow$) Suppose that $u \dashv^+ u$ does not hold for any node u at any point in the computation. We show that, as long as there is any data in the system, *some* node is able to make progress; hence, the computation will never halt with unprocessed data on a channel.

At any point in the computation, either every node with input data can make progress, or some such node u is blocked. Let H be the directed graph obtained by tracing all blocking relationships outward from u, such that there is an edge from v to w iff $v \dashv w$. By assumption, H has no cycles and is therefore a DAG. Let v_0 be a topologically minimal node in H, which is not blocked by any node. If v_0 has data items on its input channels, it is able to consume them and so make progress. Otherwise, v_0's input channels are all empty, so that it cannot block any upstream neighbors. Moreover, since v_0 itself is not blocked, either it is a source node that can advance its computation index by spontaneously producing data items, or it must have received the EOS marker and so cannot block any downstream neighbors (which contradicts v_0's presence in H). Conclude that v_0 is able to make progress, as desired. $\square$

DEFINITION 3.4. *(**Blockwise** (*not *clockwise) and **Counterblockwise**) Let C be a cycle of blocked nodes $v_1 \ldots v_n$, such that $v_1 \dashv^+ v_n$ and $v_n \dashv v_1$. The direction of increasing index on C is called* blockwise*, while the opposite direction is* counterblockwise*.*

A channel on C between v_i and v_{i+1} may be oriented either blockwise from v_i to v_{i+1} or counterblockwise from v_{i+1} to v_i. Because $v_i \dashv v_{i+1}$, *a blockwise channel on a blocking cycle is always empty, while a counterblockwise channel is always full*. For example, in Figure 2, uw and wx are blockwise channels while uv and vx are counterblockwise channels.

We notice that not all systems can have deadlocks. For example, a system with just two nodes connected by one channel will never deadlock. However, even quite simple systems, such as one with just two nodes connected by two parallel data channels, can deadlock.

DEFINITION 3.5. *(**Potential Deadlock**) A system with finite buffer sizes on all channels has a* potential deadlock *if, given the node topology and channel buffer configuration, including output buffer configuration, there exist input streams and histories of data flushes at each node such that a deadlock is possible.*

DEFINITION 3.6. *(**Undirected Cycle**) Given a system abstracted as a DAG 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.*

For example, in the graph of Figure 2, $uvxw$ is an undirected cycle that can become blocking. We now show that in a general DAG, *every* undirected cycle can become blocking.

CLAIM 3.7. *Given a system S abstracted as a DAG G, S has potential deadlocks only if G has an undirected cycle.*

PROOF. Note that the claim says an undirected cycle is only a necessary condition for deadlocks. Indeed, if there is no undirected cycle, there cannot be a blocking cycle, hence deadlocks cannot happen. □

THEOREM 3.8. *If every channel has an output buffer of size* zero*, which means every output data item is guaranteed to be visible to the receiver after finite time, then the system cannot deadlock.*

PROOF. The system is equivalent to a non-filtering system described in [18]. According to Theorem 3.1 in [18], the system is freedom of deadlocks. □

According to Theorem 3.8, if we do not use an output buffer, the system cannot deadlock. But this "write-through" style of data delivery is not good for throughput because of the delivery overhead per data item. Generally speaking, a large output buffer can improve data throughput (though it might increase data latency), so how large should we set the output buffer so that the system is still deadlock-free? In other words, given a buffer configuration, can we tell if the system is deadlock-free? If so, how can we change the buffer configuration so that the system is deadlock-free? We will answer these questions in the next couple of sections.

4. DEADLOCK AVOIDANCE

To avoid the deadlocks, one solution is using a timer for each node (or each channel). When the timer expires, all data in the corresponding buffer(s) must be flushed. This solution works because it avoids buffering data indefinitely. However, choosing appropriate length for timers is non-trivial. Too long or too short timers can degrade application performance.

We avoid using timers by setting safe buffer sizes. We argue that by setting appropriate total channel buffer sizes and output buffer sizes, a streaming computing system under the conditions we have specified can never deadlock.

4.1 Conditions for Deadlock-free Buffer Configuration

We will prove that the space of safe buffer configurations for a given application graph G is precisely defined by a set of linear constraints on these total buffer sizes and output buffer sizes. 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.

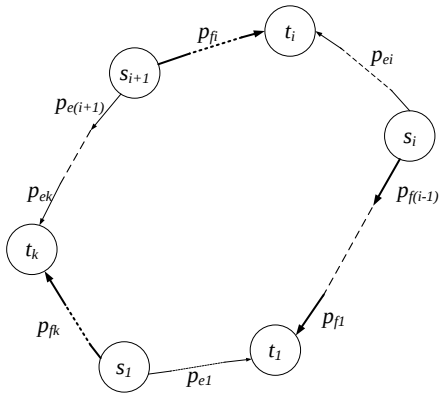

Figure 3: The division of an undirected cycle.

To describe the necessary constraints, consider Figure 3, which illustrates the division of an undirected cycle C in an application. Channels on this cycle are directed either clockwise or counterclockwise. Given such an undirected cycle C, suppose the set of clockwise channels is H_1 and the set of counterclockwise channels is H_2. Let $|q|_o$ be the size of the output buffer for channel q, and set $|q|'_o = |q|_o - 1$. Let $|q|$ be the total buffer size of q.

We establish the following inequality constraints for cycle C:

$$\Sigma_{q\in H_1}|q|'_o < \Sigma_{q\in H_2}|q| \quad (2)$$

$$\Sigma_{q\in H_2}|q|'_o < \Sigma_{q\in H_1}|q|. \quad (3)$$

Besides the above constraints, for each channel q, the following constraint is also naturally enforced:

$$0 \leq |q|'_o \quad (4)$$

$$|q|'_o < |q|. \quad (5)$$

Note that if $|q|_o = 0$, which means no output buffer is associated with q, we let $|q|'_o$ be 0 rather than -1. The reason is that the effect of $|q|_o = 0$ is same as that of $|q|_o = 1$ as no data item will ever stay in the output buffer.

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 the space of safe buffer configurations.

THEOREM 4.1. *Inequalities 2, 3, 4, and 5 together are both necessary and sufficient to guarantee deadlock freedom the given stream computing system.*

PROOF. "Necessary" means that if any of the constraints are violated, the system is at risk of deadlock; "sufficient" means that by following the constraints, the system is guaranteed to be freedom of deadlocks.

Instead of proving the theorem from scratch, we map the system in this paper, denoted as Γ, to the one described in Section III.C of [21], denoted as Φ, which proves that a dummy message schedule constrained by a set of linear inequalities can avoid deadlocks caused by data filtering. We set $|q|'_o$ for q in Γ as the dummy interval $[q]$ (defined in [21]) in Φ. Note that Inequalities 4 and 5 cannot be violated, otherwise the buffer configuration is impossible.

Suppose Γ deadlocks; then there must be a blocking cycle with some full channels and other channels with unflushed output buffers. We construct a data history in Φ such that all node receive the same history of data, all channels corresponding to full channels in Γ do not filter, and other channels on the cycle filter data items corresponding to the ones in output buffers. In Γ, we get a blocking cycle with alternate full paths and empty paths, which means a deadlock.

Suppose Φ deadlocks; then there is a blocking cycle with alternate full paths and empty paths. WLOG, we assume that full channels have not filtered any data. We let each node in Γ receive the same sequence of data as the corresponding node in Φ does. No output buffer is flushed unless it is full. When the deadlock happens in Φ, the filtered data items are corresponding to the data items in the output buffer in that they are invisible to the receiver. Since the filtering history causes a deadlock in Φ, the invisibility caused by output buffers causes a deadlock in Γ.

Because the dummy intervals constrained by Inequalities 2 and 3 are sufficient for avoiding deadlocks, inequalities 2, 3, 4 and 5 are both sufficient and necessary for avoiding deadlocks caused by bad buffer configurations. □

To verify whether a set of buffer configurations is freedom of deadlock or not, we can enumerate all undirected cycles and check whether any of the inequalities is violated. However, the number of undirected cycles could be exponential to the graph size. For example, by turning an undirected complete graph into a DAG, we can have 2^N undirected cycles, where N is the number of vertices. Verifying the inequalities by enumerating all undirected cycles could be very expensive, so we next present an efficient algorithm to verify the safety of buffer configurations.

4.2 Verifying Safety of Buffer Configuration

We sketch a method to verify the safety of a given set of buffer configurations, which involves checking for *non-positive* cycles on a specially defined graph.

DEFINITION 4.2. *(**OB-graph** and **Mirror Edge**) Given a DAG $G = (V, E)$ for a streaming system and its output buffer configurations, we create a new graph $G' = (V, E')$. For each edge $e = uv \in E$, we create two edges on G': e and $e' = vu$ (Note the direction of e'). The weight of e' is $-|uv|'_o$ G' is the OB-graph (short for* output-buffer graph*) for G, and e' and e are* mirror edges *of each other.*

The careful reader will notice that the assignment of the weight $|e'|$ is related to the inequalities defined in the previous section.

CLAIM 4.3. *Given a dataflow graph G for a streaming system and its OB-graph G', Inequalities 2, 3, 4 and 5 hold for every simple undirected cycle in G iff every cycle in G' has a positive total weight.*

PROOF. ($\leftarrow$) A directed cycle C' in G' is created from an undirected cycle C in G. If the inequalities hold for C, C' has a positive weight, since the absolute value of the sum of negative edges is less than the sum of the positive edges.

($\rightarrow$) Suppose one of the inequalities fails to hold for some undirected cycle C in G. WLOG, suppose Inequality 2 is violated, which means the sum of $|q|'_o$ of clockwise channels is at least the sum of buffer size of counterclockwise channels. Let C' be the directed cycle created with clockwise negative edges and counterclockwise positive edges based on C. The absolute value of the total weight of negative edges on C' is at least the total weight of its positive edges, so C' has a non-positive total weight. □

To check whether there is a non-positive cycle, we can run an all-pairs shortest path algorithm (e.g. the Floyd-Warshall algorithm [10, 29]) on G', as described in Algorithm 1. A non-positive distance from a vertex to itself indicates the existence of a non-positive cycle. With the classic Floyd-Warshall algorithm, we can check for non-positive cycle in $O(N^3)$, where N is the number of total nodes in the stream computing system.

Algorithm 1: Checking for Non-positive cycle.

```
for i ← 1 to n do
    for j ← 1 to n do
        if v_i v_j ∈ E then
            d_ij ← |v_i v_j|
        else
            d_ij ← ∞
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            if d_ij < d_ik + d_kj then
                d_ij ← d_ik + d_kj
        if d_ii ≤ 0 then
            return True
return False
```

If a set of buffer configuration is found to be unsafe, we can adjust the configuration to make it safe with some additional steps. To be specific, if a non-positive path from a vertex to itself is discovered, we pick some negative edges and increment its value (e.g. from -8 to -4), which means shrinking the corresponding output buffers, until the configuration is safe.

5. RELATED WORK

Some streaming computing models, such as the Kahn's Process Networks [13], assume infinite buffer capacity for each channel, which is impractical. With bounded buffering capacity, setting appropriate buffer sizes is important to both correctness and performance of streaming applications. For models with static data rates, such as the synchronous dataflow (SDF) [15, 16], it is possible to compute a bounded-memory schedule (if there exists one) and assign buffer sizes accordingly to guarantee freedom of deadlocks. But for models with fully dynamic data rates, whether a bounded-memory schedule exists is unknown [2, 3]. If the dynamic data rate is limited to data-dependent filtering, it is possible to schedule the application in bounded memory with the use of special control messages [4, 18, 20, 21]. The flushing behavior is similar to the control message in that they both make the sender's output history visible to the receiver. The method for determining buffer configuration in this paper is actually inspired by the scheduling of those special messages.

Deadlocks in other distributed systems have also been studied intensively. Chandy et al. classified deadlocks in distributed system as communication deadlocks and resource deadlocks and proposed detection algorithms for each type of deadlock [5, 6], but no prevention mechanism is introduced. In packet-switched networks, deadlock is an issue for routing algorithms. The network could deadlock if the "waiting-for" relations form a blocking cycle, and

many routing algorithms have been designed to guarantee freedom of deadlock while trying to maximize performance [7, 8, 9]. While those deadlocks are related with blocking queues, their models did not feature output buffers and how they could relate with deadlocks. Deadlock avoidance has also been studied in queueing networks, where small queues could lead to deadlocks [14, 23]. In contrast, small channel buffers alone without large output buffers in our model do not cause deadlocks. To conclude, the deadlock problem we study in this paper is a new and interesting one, which has not been studied before.

6. CONCLUSION AND FUTURE WORK

In this paper, we showed the influence of output buffers on the correctness of streaming applications. If output buffers are not configured appropriately, the streaming application could have potential deadlocks. We proved the sufficient and necessary condition of deadlock-free buffer configurations. We also proposed an efficient method for verifying if a set of buffer configuration is deadlock-free or not by using classic all-pair shortest path algorithms. If a set of buffer configuration is not deadlock-free, we also provided methods to change it to a deadlock-free one.

In future, there are several direction we plan to take. First, we want to add directed cycles to our model. By allowing directed cycles, deadlocks can be caused by all-full cycles or all-empty cycles. Secondly, we plan to extend our model to general SDFs, where the data rates are not necessarily 1 at each port. It would be more promising if we can solve the output buffer configuration problem for general SDFs. A third direction is comparing our approach with the one that uses timers. Our approach does not need any timer, but it prohibits certain buffer configurations; while the timer approach allows arbitrary buffer configurations but the length of timer needs to be carefully chosen to avoid performance degradation. We would like to see which approach has better performance in applications deployed with frameworks such as RaftLib [1].

7. ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their devoted time and insightful comments. This work was supported by NIH award R42 HG003225, NSF award CNS-0751212, and NSF award CNS-0905368.

8. REFERENCES

[1] Jonathan C. Beard, Peng Li, and Roger D. Chamberlain. Raftlib: A C++ template library for high performance stream parallel processing. In *Proceedings of Programming Models and Applications on Multicores and Manycores*, PMAM 2015, New York, NY, USA, February 2015. ACM. to be published.

[2] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, University of California, Berkeley, 1993.

[3] Joseph T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. In *Asilomar Conf. on Signals, Systems, and Computers*, pages 508–513, November 1994.

[4] 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.

[5] K. M. Chandy and J. Misra. A distributed algorithm for detecting resource deadlocks in distributed systems. In *ACM Symp. on Principles of Distributed Computing*, pages 157–164, 1982.

[6] K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Trans. Comput. Syst.*, 1(2):144–156, 1983.

[7] William J. Dally and Hiromichi Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. *Parallel and Distributed Systems, IEEE Transactions on*, 4(4):466–475, 1993.

[8] William J Dally and Charles L Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *Computers, IEEE Transactions on*, 100(5):547–553, 1987.

[9] José Duato. A new theory of deadlock-free adaptive routing in wormhole networks. *Parallel and Distributed Systems, IEEE Transactions on*, 4(12):1320–1331, 1993.

[10] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

[11] Bingsheng He, Mao Yang, Zhenyu Guo, Rishan Chen, Bing Su, Wei Lin, and Lidong Zhou. Comet: batched stream processing for data intensive distributed computing. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 63–74. ACM, 2010.

[12] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46, 2014.

[13] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[14] S Kundu and Ian F. Akyildiz. Deadlock free buffer allocation in closed queueing networks. *Queueing Systems*, 4(1):47–56, 1989.

[15] Edward A. 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.

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

[17] Peng Li, Kunal Agrawal, Jeremy Buhler, and Roger Chamberlain. Orchestrating safe streaming computations with precise control. In *International Workshop on Extreme Scale Computing Application Enablement - Modeling and Tools, in conjunction with ICPADS'14*, December 2014.

[18] 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.

[19] Peng Li, Kunal Agrawal, Jeremy Buhler, and Roger D Chamberlain. Adding data parallelism to streaming pipelines for throughput optimization. In *High Performance Computing (HiPC), 2013 20th International Conference on*, pages 20–29. IEEE, 2013.

[20] 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.

[21] Peng Li and J. Buhler. Polyhedral constraints for bounded-memory execution of synchronized filtering dataflow. In *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2013*, pages 29–37, Sept 2013.

[22] Peng Li and Jeremy Buhler. Improving performance of streaming applications with filtering and control messages. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, PACT '14, August 2014.

[23] Jörg Liebeherr and Ian F Akyildiz. Deadlock properties of queueing networks with finite capacities and multiple routing chains. *Queueing systems*, 20(3-4):409–431, 1995.

[24] Björn Lohrmann, Daniel Warneke, and Odej Kao. Massively-parallel stream processing under qos constraints with nephele. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 271–282. ACM, 2012.

[25] Lin Ma and Roger D. Chamberlain. A performance model for memory bandwidth constrained applications on graphics engines. In *Proc. of Int'l Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, 2012.

[26] Lin Ma, Roger D. Chamberlain, Jeremy D. Buhler, and Mark A. Franklin. Bloom filter performance on graphics engines. In *Proc. of Int'l Conf. on Parallel Processing*, pages 522–531, 2011.

[27] Alexander Maxiaguine, Simon Künzli, Samarjit Chakraborty, and Lothar Thiele. Rate analysis for streaming applications with on-chip buffer constraints. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, pages 131–136. IEEE Press, 2004.

[28] William Thies and Saman Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 365–376, 2010.

[29] Stephen Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, January 1962.

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

[31] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.