Deadlock Avoidance for Streaming Applications with Split-Join Structure: Two Case Studies

Anonymous for Review

Abstract

Streaming is a highly effective paradigm for expressing parallelism in high-throughput applictions. A streaming computation *is a network of compute nodes connected by unidirectional FIFO channels. However, when these computations are mapped on to real parallel platforms, some computations, especially ones in which some nodes act as* filters*, can* deadlock *the system due to finite buffering on channels. In this paper, we focus on streaming computations which contain a commonly used structure called* split-join*. Based on our previous work, we propose two correct* deadlock-avoidance *algorithms for applications with the split-join structure. These algorithms are based on* dummy messages*, messages that the system generates in order to propagate information about the status of the computation. In one of these algorithms, only the split nodes generate dummy messages, but these messages are propagated by other nodes. In the other algorith, all nodes may potentially generate dummy messages, but these messages are not propagated. Theoretically, these algorithms are incomparable in terms of performance since there are conditions under which each outperforms the other. We focus on two representative applications, biological sequence alignment and random number generation, which can be efficiently parallelized with the split-join streaming structure. Our evaluation of these applications shows that the non-propagating algorithm generates much fewer dummy messages (and thus has a smaller overhead) than the propagating algorithm. In particular, even for systems which are very vulnerable to deadlocks (buffering $\leq$ 10 messages and filtering $\geq$ 95% of input data in one path), our non-propagating algorithm causes less than 10% message overhead. For systems with large buffer sizes or a low filtering ratio, the message overhead of the Non-propagating Algorithm is negligible.*

1 Introduction

Streaming is becoming an increasingly popular computing paradigm since it can be used to conveniently express throughput-oriented parallel computation. A streaming application is essentially a network of compute nodes connected by unidirectional communication channels. Each compute node reads data items from its input channels and places data items on its output channels. From the programmer's viewpoint, the compute nodes can perform arbitrary computation and channels provide FIFO guarantees. Here, we assume that computation on each node is bounded and deterministic for a particular set of input values.

In this paper, we consider streaming applications with two interesting features. First, they include a *split-join structure* (shown in Figure 1), in which several channels split from the source node to feed independent computations, the results of which are joined at the sink node. Each path from source to sink has a linear chain of zero or more intermediate nodes, but there are no links between paths except at the source and sink.

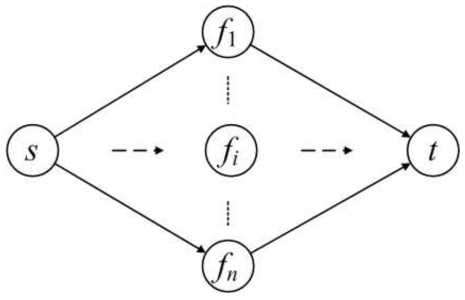

Figure 1. A split-join streaming example

We focus on split-join topologies in which asynchronous computations on each path must synchronize their results at the sink node. For example, it may be important that the system generate exactly the same stream of results every time it is run, or the sink node may implement some non-commutative reduction that must process results in the order that the original inputs were generated. To permit synchronization at the sink, we add a non-negative timestamp (also called an "index") to each data item.

The second feature of the streaming computations considered in this paper is that some nodes are *filters*. If a computation at node v does not result in an output data d being emitted on an outgoing channel q, we say that v *filters* data d. In our applications, filtering is a data-dependent behavior that cannot be predicted at compilation time. There are many real world applications that have both filtering and split-join structures. For example, many algorithms for generating nonuniform random deviates, such as Marsaglia's polar optimization [10] to Box-Muller [2] or the ziggurat algorithm [11], use rejection sampling, which discards a portion of the input stream of uniform deviates.

As a practical matter, the streaming computation is mapped onto a physical parallel computating platform. The platforms we consider are different from an idealized machine in three important ways: First, we consider platforms that may be distributed over several devices. Therefore, the compute nodes operate *asynchronously*. Second, due to the asynchrony, while the platform guarantees that each datum on a channel will be eventually delivered to its destination, there are no guarantees about how long it might take for an item to traverse a channel. Finally, even though the programmers view of the machine is that each channel has infinite buffering, as a practical matter, buffer sizes are bounded and finite and this bound is decided at compile time when the computation is mapped on to the platform. Therefore, if a channel buffer from node u to node v is full, then u must block and wait for v to consume some data before it can put any more data on the channel.

Due to these limitations of physical platforms, streaming computations with split-join structures and filtering nodes may *deadlock* during execution. Consider, for example, the system of Figure 2a, in which a uniform random number generator s sends pairs of random deviates to nodes f_1 and f_2, each of which implements a transform with rejection to produce nonuniform random deviates. Nodes f_1 and f_2 are not synchronized, and either may filter an arbitrary subset of its inputs from s. To ensure repeatable output, we index the uniform deviates sent from source s and demand that t emits the combined output streams from f_1 and f_2 in index order. Suppose that node f_2 rejects a number of successive inputs, and so emits nothing on channel f_2t, while node f_1 continues to emit outputs on f_1t. Since channels may have arbitrary delays, node t cannot immediately consume data from f_1t because it does not know whether data with an earlier index might subsequently appear on f_2t. If channels sf_2 and f_2t become empty due to filtering while sf_1 and f_2t fill, the system may reach the situation shown in Figure 2b, where node t is unable to consume data since it is waiting to receive data that must be sent from s, but s is unable to generate any more output because it is blocked trying to emit onto the full path from s to f_1. This situation is a deadlock since no node can make progress for an infinite amount of time.

In previous work [9], we showed that in asynchronous streaming systems with a variety of topologies, including

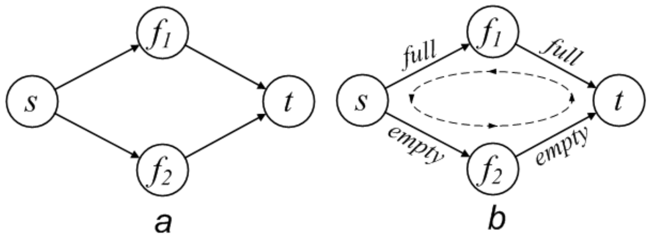

Figure 2. A deadlock example

split-join systems, the combination of synchronization and filtering can cause deadlock. That work proposes algorithms for deadlock avoidance in such systems. In this paper, we adapt those algorithms for the special case of split-join structiures. In addition, we empirically evaluate these algorithms on two real-world streaming applications that fit the split-join model: nonuniform random number generation as described above, and the Mercury BLAST [3] system for fast biological sequence comparison.

2 Application Descriptions

This section describes our two representative applications which both have a split-join structure with filtering nodes. Therefore, both of these applications are vulnerable to deadlocks. Mercury BLAST, our biological application is much more vulnerable than the random number generator and in fact, this work was motivated by our observation of actual deadlocks while running Mercury BLAST. However, it is much easier to analytically compute the probability of deadlocks with random number generator application, and we show this computation at the end of this section.

2.1 Mercury BLAST

Mercury BLAST [3] is an FPGA-accelerated implementation of the Basic Local Alignment Search Tool (BLAST), a bioinformatics tool for comparing DNA or protein sequences, which is one of the most widely used computational tools in molecular biology. It compares a short query sequence to a large sequence database to discover regions of biologically meaningful similarity between them.

Detailed comparison of a query to any region of a sequence database requires an expensive edit distance computation. To avoid this expensive computation whenever possible, BLAST uses filtering heuristics to quickly discard large portions of the database that are unlikely to match the query sequence. The principal heuristic, *seed matching*, divides the database into overlapping sequences of some short, fixed length w, then tests whether each such w-*mer* appears in the query. If a w-mer is present at position x in the database and position y in the query, this test generates a *seed match* (x, y). The portions of the database and query near these coordinates are then subjected to further testing to confirm or reject the presence of biologically meaningful similarity. In BLASTN, the variant of BLAST used for DNA sequences, w is on the order of 10 characters, and only about one in 100 database positions generates a seed match even for a query tens of thousands of characters in length.

Mercury BLASTN implements BLASTN's filters as a streaming computation network, with a split-join topology as shown in Figure 3. The query is preprocessed into a lookup table stored in seed matching module 1b. The database is then streamed into module 1a, which both divides it into w-mers that are sent to 1b for matching and forwards it unmodified to later stages of the application (represented in the diagram by module 2). Seed matches discovered by 1b are forwarded to module 2 for further testing.

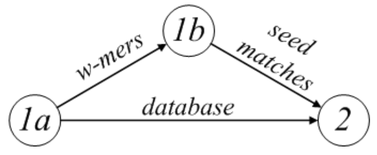

Figure 3. The first two stages of Mercury BLAST

Testing a seed (x, y) requires module 2 to inspect a window of the database centered at position x; hence, module 2 cannot discard a given chunk of database sequence until it is sure that no seed match has been found in it. Module 2 must therefore synchronize its two input streams to ensure correctness. Moreover, the rate at which module 1b generates seed matches is highly data-dependent: some database regions may generate many matches, while others may generate none over thousands of positions. Because the database input channel to module 2 has a finite buffer (on the order of 64 Kchars), there is a risk of deadlock if 1b happens not to find any seed matches in a long enough piece of the database.

2.2 Pseudorandom Number Generation

Pseudorandom number generators (PRNGs) are widely used in applications, such as Monte Carlo simulation, that require a long stream of input values that appear "random" but can be generated repeatably. Most techniques for directly generating pseudorandom numbers produce uniform random deviates, but some applications need numbers that follow some other distribution, such as a Gaussian or exponential. For these applications, the output of a uniform PRNG is typically transformed by some computation to produce random deviates with the desired distribution.

A common strategy employed by nonuniform PRNGs is *rejection sampling*. Rejection-based PRNGs use k-tuples of uniform deviates, for some fixed $k \geq 1$, to drive a second sampling process that sometimes produces a sample from the desired target distribution and sometimes produces nothing. Classic examples include the Marsaglia polar method [10] and the ziggurat algorithm [11], each of which have $k = 2$, but the same technique is also used in more complex approaches such as Markov-chain Monte Carlo.

When an application has a high demand for pseudorandom numbers, and the necessary transform is computationally demanding, the generator may be parallelized using a pipeline stream shown in Figure 4. Node s generates a sequence of uniform deviates, which are transformed by the filter f. The outputs of f are passed to t for further usage. In Figure 4's architecture, the stage f tends to be the bottleneck of pipeline because sampling takes more computation than the random number generation in s. To speedup the bottleneck, one method is replicating the filter f, as Figure 5 shows. Node s feeds generated numbers in round-robin fashion to multiple replicated filters f_i (four in this example) that run the same rejection-based transform. The filters' results are merged at the sink t. To ensure that we can produce the same stream of values given the same seed as the pipeline of Figure 4, t must implement some form of predictable synchronization over all filters.

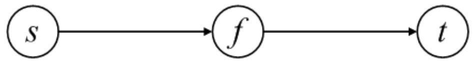

Figure 4. A pipeline for PRNG

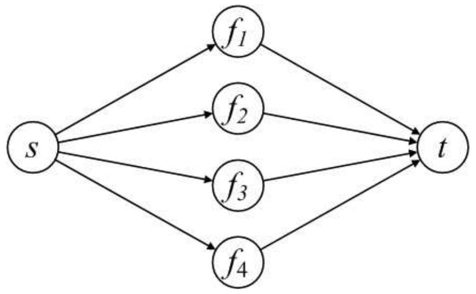

Figure 5. A split-join pipeline for PRNG

One approach to synchronization is for s to assign a monotonically increasing index to each k-tuple sent from s to a filter. Filters emit samples with the same index as the k-tuples that produced them, and the sink t emits samples in order of their indices. Alternatively, s may emit k-tuples with the same index i to every filter at once, and t must emit all samples with index i before any with index $i + 1$; within a given i, samples are emitted in some fixed order assigned to the filters *a priori*. For either approach, deadlock can occur if any filter happens to discard a long sequence of inputs while another continues to produce outputs. We choose the latter because the span of indices is smaller, hence it is more efficient for our deadlock algorithms discussed later.

2.3 Deadlock Probability

In this section, we develop expressions to quantify the probability of deadlock in split-join systems where all filtering nodes filter data independently. We begin with the system that has two paths between the source and the sink, like the system in Figure 2a. There are two possible deadlock situations: (1) the path sf_1t is full but sf_2t is empty; (2) the path sf_2t is full but sf_1t is empty. Suppose the total buffer sizes of sf_1t and sf_2t are b_1 and b_2, the accumulative filtering rates of sf_1t and sf_2t are p_1 and p_2, respectively.

If a deadlock of situation 1 happens, which means sf_1t is full but sf_2t is empty, let the index of the latest data accumulated in sf_1t be n ($n \geq b_1$) and the index of the earliest data accumulated be i, so i ranges from 1 to $n - b_1 + 1$. We can infer that path sf_2t filters all data items with indices in $[i, n]$, while path sf_1t filters $n - i + 1 - b_1$ data items with indices in $[i + 1, n - 1]$, so the probability of such deadlock is

$$\begin{aligned} P_1 &= \Sigma_{i=1}^{n-b_1+1} p_2^{n-i+1} \binom{m}{b_1 - 2} (1 - p_1)^{b_1} p_1^l \\ &= \Sigma_{j=b_1}^{n} p_2^j \binom{j-2}{b_1 - 2} (1 - p_1)^{b_1} p_1^{j-b_1} \end{aligned}$$

where $m = (n - 1) - (i + 1) + 1$, $l = n - i + 1 - b_1$. We are interested in the case when n is very large, for example $n \to \infty$, which is common in data-intensive applications. After some algebraic manipulation, we get the following probability:

$$\begin{aligned} P_1 &= \Sigma_{j=b_1}^{\infty} p_2^j \binom{j-2}{b_1 - 2} (1 - p_1)^{b_1} p_1^{j-b_1} \\ &= \frac{(1 - p_1)^{b_1} p_2^{b_1}}{(1 - p_1 p_2)^{b_1 - 1}}. \end{aligned}$$

Similarly, we can calculate the probability for the situation that sf_1t is empty but sf_2t is full. When $n \to \infty$,

$$P_2 = \frac{(1 - p_2)^{b_2} p_1^{b_2}}{(1 - p_1 p_2)^{b_2 - 1}}.$$

Since the two deadlock situations are mutually exclusive, the total probability of deadlock is $P = P_1 + P_2$, which is

$$P = \frac{(1 - p_1)^{b_1} p_2^{b_1}}{(1 - p_1 p_2)^{b_1 - 1}} + \frac{(1 - p_2)^{b_2} p_1^{b_2}}{(1 - p_1 p_2)^{b_2 - 1}}.$$

As a special case, when $p_1 = p_2 = p$, $b_1 = b_2 = b$, which is the case of PRNG, the above result is simplified as

$$P' = (1 - p^2) \times (\frac{p}{1 + p})^b.$$

From the above equation we can learn that given a fixed filtering ratio p, increasing the buffer size will reduce the chance of deadlock, which is in accord with our intuition. When there are more than two paths between the source and the sink, the situation becomes much more complex to compute.

We note that precise probability estimation for deadlocks is practical only if the input stream is statistically well-behaved. This is not the case for, e.g., the Mercury BLAST application, which must deal with the irregularities of real biological sequences. Moreover, the probability of deadlock is relatively low in a split-join application if it is unlikely that one path from source to sink filters many more messages than another; this is the case for PRNGs with identical, independent computations on each path. In contrast, Mercury BLAST by design produces one stream (seed matches) with a much higher filtering rate than the other (database), making it much more prone to deadlock. Indeed, our work on deadlock avoidance was initially motivated by the need to address real-world deadlocks observed in Mercury BLAST.

3 Deadlock Avoidance Algorithms

As mentioned earlier, filtering nodes can lead to deadlocks in streaming applications. In our previous work [9], we designed algorithms that (provably) avoid deadlocks. The general idea behind our algorithms is the concept of dummy messages. A *dummy* is a distinguished class of messages with an index but no content of its own. Its purpose is to notify receivers that a data item in the stream with this index has been filtered by an upstream sender. The algorithms described in [9] work for general network topologies including, but not limited to, split-join constructs. In this section, we describe these algorithms, slightly simplified to only handle split-join structures.

Since we concentrate on parallel platforms that may be distributed over different types of compute engines, our primary goal in designing deadlock avoidance algorithms was to avoid any global communication between various nodes. Due to this limitation, our algorithms are necessarily conservative and in theory, may send may unnecessary dummy messages adversely impacting system performance. In this paper, we evaluate the performance impact of these algorithms for the BLAST and the PRNG applications discussed in Section 2.

3.1 Semantics of a Single-node

Before introducing deadlock avoidance algorithms, we must explain the behavior of a single node (without any deadlock avoidance mechanisms) to help understand how a join node performs synchronization naturally in our model.

The precide behavior is shown in Algorithm 1. A node maintains a *computation index* and only consumes input tokens with indices the same as its computation index. The source node increases the *computation index* by 1 before each computation. A non-source node waits until all input channels have data, then it chooses the smallest index of all input data as its new computing index. Therefore, a sink node can not proceed unless it has recieved the data with current computation index or larger on all of its input channels. End-Of-Stream (EOS) is a marker at the end of each stream and never filtered. It has a computing index of ∞.

Algorithm 1: Single-node behavior in a system.

```
ComputeIndex ← 0 ;
while ComputeIndex ≠ Index of EOS do
    if not source node then
        wait until every input channel has a pending
            token ;
        let $T$ be minimum index of any pending token ;
        remove pending tokens with index $T$ from
            input channels ;
    else
        $T \leftarrow$ ComputeIndex + 1 ;
    ComputeIndex ← $T$ ;
    perform computation on data tokens with index $T$ ;
    if not sink node then
        emit output tokens with index $T$ ;
```

3.2 Algorithms

We trimmed the deadlock algorithms proposed in [9] to adjust to applications with split-join structure. The first algorithm is a simple algorithm that sends a dummy message each time a data item is filtered, referred to as *Naive Algorithm* later. In effect, this approach converts a filtering node to a non-filtering node. While this approach guarantees no deadlocks, it can potentially increase the communication requirements of the application by quite a bit.

The other two algorithms try to send fewer dummy messages than the simple naive algorithm. [1] The first of these two algorithms, called *Propagating Algorithm* (shown in Algorithm 2 and 3), is a static algorithm in which only the source (split) node sends dummy messages and these messages are propagated until they get to the sink (join) node. That is, if a node receives a dummy message with a particular index, it must propagate it on its output links. The source node sends a dummy to output channel q if the computing index has increased by $[q]$ since the last dummy sent to this channel. $[q]$ is computed by Algorithm 3 as the minimum total buffer size of the other paths between the source and the sink.

[1]For simplicity in the description, we assume that the split node is the source and the join node is the sink. These algorithms work, however, even if the application other nodes before and after the split-join structure. In addition, with minor modifications, they work in the case when the application has nested split-join structures.

Algorithm 3: Dummy interval calculation with dummy propagation.

```
Input: A system abstracted as graph $G = \{V, E\}$
Output: Dummy intervals for each channel
foreach edge $q \in E$ do $[q] \leftarrow \infty$ ;
foreach path $p$ between the source $s$ and the sink $t$ do
    let $b$ be the minimum total buffer size of other
        paths between $s$ and $t$;
    let $q$ be $s$'s output channel on $p$;
    $[q] = b$;
```

The last algorithm is called *Non-propagating Algorithm* and is shown in Algorithm 4 and 5. Here, each node that filters data can generate dummy messages, but there is no requirement to propagate them. In this algorithm, a node sends a dummy message on channel q if its computing index has increased by $[q]$ since the last data item or dummy message was sent on this channel. $[q]$ is computed by Algorithm 5 as the minimum total buffer size of other paths divided by the number of filtering nodes on this path. All paths here start from the source and end at the sink.

Algorithm 5: Dummy interval calculation without dummy propagation.

```
Input: A system abstracted as graph $G = \{V, E\}$
Output: Dummy intervals for each channel
foreach edge $q \in E$ do $[q] \leftarrow \infty$ ;
foreach path $p$ between the source $s$ and the sink $t$ do
    let $b$ be the minimum total buffer size of other
        paths between $s$ and $t$;
    let $m$ be the number of filtering channels on $p$;
    foreach filtering channel $q$ on $p$ do
        $[q] = \lceil b/m \rceil$;
```

4 Performance Analysis

In this section, we evaluate the overhead associated with the deadlock avoidance algorithms. The dummy message scheduling is fairly simple, causing very little computational overhead, so our performance evaluation pays attention to the number of dummy messages generated and sent by nodes across communications links.

According to the behavior of algorithms, the number of dummy messages sent by the Naive Algorithm equals the quantity of filtered data, no matter the buffer size. The

Algorithm 2: Node behavior with dummy propagation.

```
ComputeIndex ← 0 ;
**foreach** *output port q* **do**
    LastOutputIndex_q ← 0 ;
**while** *ComputeIndex ≠ Index of EOS* **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 $T$ from input channels ;
    **else**
        $T \leftarrow$ ComputeIndex $+ 1$ ;
    **foreach** *output channel q* **do**
        **if** $T -$ *LastOutputIndex*$_q \geq [q]$ ***OR*** *some pending token with index T is a dummy* **then**
            schedule a dummy token with index $T$ for output $q$ ;
            LastOutputIndex$_q \leftarrow T$ ;
    ComputeIndex $\leftarrow T$ ;
    perform computation on data tokens with index $T$ ;
    **if** *not sink node* **then**
        emit output tokens with index $T$, combined with any scheduled dummies ;
```

number of dummy messages generated by the Propagating Algorithm is decided by the total buffer sizes, and is not affected by the filtering ratio. Hence the number of dummy messages sent by the Propagating Algorithm can be statically analyzed given the input data volume and system topology. The number of dummy messages sent by the Non-propagating Algorithm is decided by filtering trace and cannot be statically calculated.

4.1 Mercury BLAST

To acquire the number of dummy messages sent in Mercury BLAST, we run Mercury BLASTN to perform a query of mRNA_non[FIXME] against the Mammal mRNA database. We only monitor the number of dummy messages out of stage 1a. We run the Non-propagating Algorithm and use hardware monitor described in [8] to count the actual dummy messages. The amount of dummy messages generated by the Naive Algorithm can be estimated by the multiplication of input data volume and the filtering rate and dummy messages generated by the Propagating Algorithm can be calculated by dividing the number of input data items by the dummy interval, which is a fixed value. We set the buffer size of the database channel to 128, 512, and 2048, which stand for the database items the channel can hold. The corresponding dummy intervals are 128, 512, and 2048 for the Propagating Algorithm and 64, 256, and 1024 for the Non-propagating Algorithm. Our results (shown in Table 1 indicate that Non-Propagating Algorithm has, by far, the smallest overhead.

Table 1. Measured dummy message counts for Mercury BLASTN

	Dummy message count		
Total Buffer Size (msgs)	128	512	2048
Naive Algo.			
Prop. Algo.			
Non-prop.			

4.2 PRNG

To assess the performance impact of our algorithms on PRNGs, we simulate the Marsaglia polar method, which has a rejection rate of 21.46%. We choose Marsaglia polar method rather than the ziggurat algorithm because the former has a higher filtering ratio, which means it is more vulnerable to deadlocks. We replicate four filters between the source and the sink. In three different runs, the total buffer size of each path is set to 10, 100, and 1,000, which determines the total number of elements, including data items and dummy messages, that can be buffered. The source generates 1 million uniformly distributed random numbers and distributes them evenly to four replicated filters, each of which runs the Marsaglia polar method independently. We apply the three deadlock avoidance algorithms and count the total number of dummy messages each of them generates.

Algorithm 4: Single-node behavior without dummy propagation.

```
ComputeIndex ← 0 ;
**foreach** *output port q* **do**
    LastOutputIndex$_q$ ← 0 ;
**while** *ComputeIndex ≠ Index of EOS* **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 $T$ from input channels ;
    **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** $T - \mathit{LastOutputIndex}_q \geq [q]$ **then**
            schedule a dummy token with index $T$ for output $q$ ;
            LastOutputIndex$_q$ ← $T$ ;
    **if** *not sink node* **then**
        emit output tokens with index $T$, including any dummies
```

The results are shown in Table 2. In the Propagating Algorithm, the dummy messages are generated by the source and propagated by intermediate nodes to the sink, so the total dummy messages transmitted among nodes are twice of those generated by the source. From the data in Table 2, the Non-propagating Algorithm is also the most efficient, as it sends only one dummy message (from a run with one million true messages) even when the total buffer size is as small as 10.

Table 2. Simulation results for Marsaglia polar algorithm (filtering ratio = 21.46%)

	Dummy message count		
Total Buffer Size (msgs)	10	100	1000
Naive Algo.	215,030	215,030	215,030
Prop. Algo.	200,000	20,000	2,000
Non-prop.	1	0	0

4.3 Potential Applications

The Marsaglia polar method represents a category of applications in which an arbitrary number of filters can be replicated between the source and the sink if the filter is the performance bottleneck of the system. Due to the conservative nature of the three algorithms (explained below), the number of dummy messages is decided by the channel buffer size configuration and data filtering history rather than the actual instances of deadlock. Hence we can use alternative buffer size and filtering ratio configurations to estimate the performance of our deadlock avoidance algorithms in other, alternative applications. We executed another two simulations with the filtering ratio set as high as 95% and as low as 5%. The buffer sizes of each path are set to 10, 100, and 1,000 in different runs. The total data volume is 1 million true data messages for each run. The results of these simulations are reported in Tables 3 and 4.

Table 3. Simulation results for 4 replicated filters and filtering ratio = 95%

	Dummy message count		
Total Buffer Size (msgs)	10	100	1000
Naive Algo.	950,090	950,090	950,090
Prop. Algo.	200,000	20,000	2,000
Non-prop.	74,633	333	0

We can see that the non-propagating deadlock avoidance algorithm has the least overhead. Even with a high filtering

Table 4. Simulation results for 4 replicated filters and filtering ratio = 5%

	Dummy message count		
Total Buffer Size (msgs)	10	100	1000
Naive Algo.	50,172	50,172	50,172
Prop. Algo.	200,000	20,000	2,000
Non-prop.	0	0	0

ratio (95%) and small channel buffer size (only buffering 10 data items), the communication overhead is less than 10%. In low filtering ratio (5%) or large buffer size (1000 data items) cases, the overhead is negligible.

5 Related Work

In [9], we formalized a streaming computation model based on directed acyclic multigraph, which is more general than the split-join structure. For that model, we proposed three deadlock avoidance algorithms and proved the correctness of them, which are the basis of the algorithms adopted in this work.

There has been considerable prior research on dealing with deadlocks in distributed systems. Chandy et al. developed algorithms to detect resource deadlocks based on probes [4, 5]. Mitchell and Merritt designed a deadlock detection algorithm using public and private labels [12]. There has been some recent work on deadlock avoidance using Kahn Process Networks (KPNs) [7], which is a theoretical model close to streaming computation. Geilen and Basten proposed a new scheduling algorithm for KPNs which avoided deadlocks by increase buffer size during runtime [6]. Allen et al. proposed algorithms using private and public sets to detect all deadlocks in bounded KPNs and to resolve them if possible [1]. However, not all detected deadlocks can be resolved. These deadlock avoidance and resolution algorithms require runtime changes to channel capacities, while our algorithms do not. Furthermore, some applications may have difficulty in dynamic resource allocation, such as those deployed on FPGAs.

6 Conclusion and Future Work

In this work, we have analyzed the split-join streaming computation model and demonstrated how applications like BLAST and PRNG can be efficiently parallelized within this model. The PRNG application actually represents a category of applications in which some pipeline stages can be arbitrarily replicated to satisfy performance requirements. Based on the split-join model, we characterized the deadlock risk of applications with filtering at non-sink nodes and data synchronization at the sink node. For the application with independent and statistically well-behaved filtering, we provided a quantitative calculation of deadlock probability. We proposed three deadlock avoidance algorithms, which are based on previous theoretical work and trimmed for the split-join model. Among them, the Non-propagating Algorithm is the most efficient, adding negligible message overhead while correctly avoiding deadlocks.

The deadlocks discussed in this work are mostly due to application behavior. In our practice, we have identified deadlocks that arise due to the sharing of physical communication links. Two channels sharing the same physical link such as PCI-X can cause deadlock even without data filtering. This type of deadlock requires a different avoidance strategy. In our future work, we will explore methods to avoid deadlock due to these causes.

References

[1] G. Allen, P. Zucknick, and B. Evans. A distributed deadlock detection and resolution algorithm for process networks. In *Proc. IEEE Int'l Conf. Acoustics, Speech, and Signal Processing*, pages 33–36, Apr. 2007.

[2] G. E. P. Box and M. E. Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 29(2):610–611, 1958.

[3] Mercury BLASTN: Faster DNA sequence comparison using a streaming hardware architecture. In [*anonymous*].

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

[5] K. M. Chandy, J. Misra, and L. M. Haas. Distributed deadlock detection. *ACM Trans. Comput. Syst.*, 1(2):144–156, 1983.

[6] M. Geilen and T. Basten. Requirements on the execution of Kahn process networks. In *Proc. of 12th European Symposium on Programming*, pages 319–334, 2003.

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

[8] Efficient runtime performance monitoring of FPGA-based applications. In [*anonymous*].

[9] Deadlock avoidance for streaming computations with filtering. In [*anonymous*].

[10] G. Marsaglia. Improving the polar method for generating a pair of normal random variables. Technical Report D1-82-0203, Boeing Sci. Res. Labs., Seattle, WA, Sept. 1964.

[11] G. Marsaglia and W. W. Tsang. The ziggurat method for generating random variables. *Journal of Statistical Software*, 5(8):1–7, 10 2000.

[12] D. P. Mitchell and M. J. Merritt. A distributed algorithm for deadlock detection and resolution. In *Proc. of ACM Symp. on Principles of Distributed Computing*, pages 282–284, 1984.