Winter 12-15-2014

The Synchronized Filtering Dataflow

Peng Li
Washington University in St. Louis

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science
Department of Computer Science and Engineering

Dissertation Examination Committee:
Jeremy Buhler, Chair
Kunal Agrawal
Roger D. Chamberlain
Christopher D. Gill
Joseph A. O'Sullivan

The Synchronized Filtering Dataflow

by

Peng Li

A dissertation presented to the
Graduate School of Arts and Sciences
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

December 2014
Saint Louis, Missouri

Table of Contents

List of Figures

List of Tables

Acknowledgments

I lived through the most challenging part of my life so far, thanks to many people.

First, I would like to sincerely thank my advisor, Dr. Jeremy Buhler, for his guidance through the journey. Dr. Buhler has been supportive and encouraging throughout my PhD study. He pointed the research direction to me, fixed my broken proofs, and left mark on me with his affluent knowledge and serious attitude. I could not have finished the dissertation without his advice. I also thank Dr. Roger Chamberlain for being my co-advisor. As a veteran researcher, he knows almost every aspect of computer science and engineering – from hardware to software, and from theory to applications. Thanks to his broad knowledge, I enjoyed incredible research freedom. Whatever ideas I proposed, he was always there to give me insightful advice. Very few students have the luxury of being supervised by three professors – I was lucky being one of them. Dr. Kunal Agrawal, also my co-advisor, guided me through complex parallel computing theories. Without her help, I couldn't have formalized my ideas and published them in peer-reviewed papers. In addition, I would like to thank Dr. Chris Gill and Dr. Joseph O'Sullivan for being on my dissertation committee. Their insightful comments and questions helped further improve my dissertation and research. Dr. Gill also ardently tried to help me find a job.

I am grateful for financial support provided by NIH award R42 HG003225, NSF award CNS-0751212, and NSF award CNS-0905368. This support allowed me to focus on my research without worrying financial situation.

I would like to thank the department staff members Kelli Eckman, Sharon Matlock, Jayme Moehle, Lauren Huffman, and Myrna Harbison, who make WUCSE like a big family. I would like to thank my collegues and friends Lin Ma, Hongtao Sun, Steve Cole, Arpith Jacob, Joe Lancaster,

Michael Hall, Jonathan Beard, and Joe Wingbermuehle – this list could go very long. Conversations with them have helped shape my research.

Finally, I want to thank my family in China, especially my parents. They have been a source of consistent encouragement and support. I am really proud of them.

Peng Li

Washington University in Saint Louis
December 2014

Dedicated to the memory of my grandfather, who passed away on November 17th, 2014.

ABSTRACT OF THE DISSERTATION

The Synchronized Filtering Dataflow
by
Peng Li
Doctor of Philosophy in Computer Science
Washington University in St. Louis, 2014
Professor Jeremy Buhler, Chair

In the past decade, the world has seen the rise of big data, which calls for a paradigm shift in data processing. Streaming processing, where data are processed in their spatial or temporal order, is increasingly common. Meanwhile, parallel computing has become a household term in the computing world. The combination of streaming processing and parallel computing, *streaming computing*, has been playing an important role in data processing.

A streaming computing system is a network of *nodes* connected by unidirectional first-in first-out (FIFO) data *channels*. When a node has multiple input channels, to ensure the deterministic behavior of the whole system, *synchronization* is required on those channels when the node consumes data. After a streaming computing node finishes a computation, it may choose not to produce output on some of its output channels. This behavior, known as *filtering*, is data-dependent and unpredictable. When filtered data streams are synchronized, applications can *deadlock* due to empty and full channel buffers.

To avoid deadlocks and ensure bounded-memory execution, we turn to model-based approaches. In this dissertation, we propose the synchronized filtering dataflow (SFDF) to model synchronization and filtering behaviors. We avoid deadlocks in SFDF applications by augmenting data streams with *dummy messages*. We design *decentralized* algorithms that compute a *dummy interval* for each channel during compilation time and schedule dummy messages according to the dummy intervals during runtime.

The runtime parts of our algorithms are very efficient, adding little overhead to computing nodes, but computing dummy intervals could be very time-consuming on general dataflow graphs.

We design efficient algorithms to compute dummy intervals for streaming applications with special topologies. In particular, we focus on series-parallel directed acyclic graphs (SP-DAGs) and CS4 DAGs, where each *undirected* cycle is single-source and single-sink.

We further extend our work to describe a set of polyhedral constraints that define all sets of safe dummy intervals for any dataflow graphs, which gives us more flexibility to choose dummy intervals. We also provide a polynomial-time algorithm to verify the safety of given dummy intervals for SP-DAGs.

Dummy messages are only one type of *control message* used by streaming applications. We extend our SFDF model to support more types of control message, which are precisely synchronized with data streams. We use two types of control messages, dummy message and *credit message*, to guarantee bounded-memory execution. We demonstrate that the extended model can help improve performance of some applications by adding filtering behavior to non-filtering applications.

Chapter 1

Introduction

For decades, sequential computer programs with random access to input data have prevailed. However, in recent years, the world has increasingly seen hardware built for parallel computing. Even in embedded systems, such as mobile phones, multi-core processors are common. Meanwhile, in data-processing applications, large data sizes push people to use streaming processing. The two trends converge into a new computing paradigm: streaming computing.

1.1 New Trends in Computing

1.1.1 Parallel Computing

In the 1960s, Gordon Moore predicted that the number of transistors that could be placed on a chip would double every two years; the prediction is known as "Moore's Law" [79]. Thanks to the evolution of semiconductor technology, Moore's Law has been continuing for more than four decades, far beyond Moore's expectation. As a result, the computing power of a single chip has been doubling every two years. Before the early 2000s, increases in processor computing power resulted mainly from increases of clock speed, which, however, has plateaued in recent years (see Figure 1.1). In the meantime, more processor cores were integrated into a single chip. Nowadays it is common to see workstations, personal computers, and even mobile phones equipped with multi-core processors [12]. Graphics processing units (GPUs) can have hundreds of cores [84]. While

as a whole chip, processors are more powerful today (see Figure 1.2), single-core performance has not seen a corresponding growth and has even slowed down. But computing demand keeps growing. To meet the requirement for computing power, we need parallel computing[1].

A parallel computing system, or a parallel program, is a collection of sequential modules co-operating with each other, some of which can be executed concurrently. *Parallelism* characterizes the availability of computations that can take place concurrently. Understanding different forms of parallelism is key to understanding parallel computing.

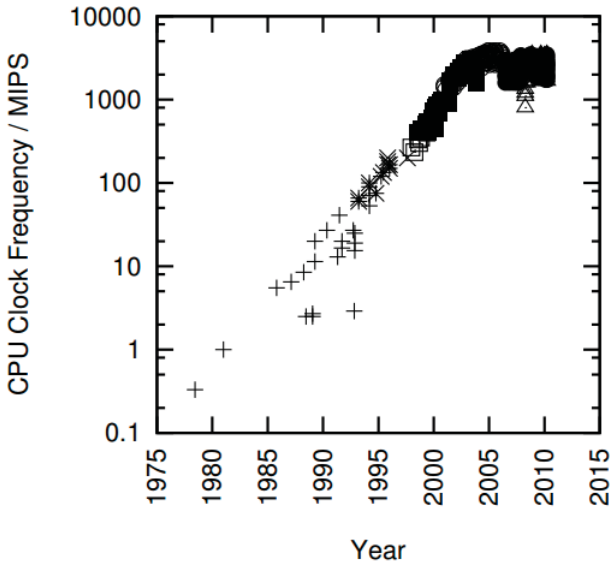

Figure 1.1: Clock-frequency trend for Intel CPUs, adopted from [75].

Understanding Parallelisms

Parallel computing can be boiled down to extracting and implementing parallelism in applications. Depending on the granularity of executing entities, there are bit-level parallelism [19], where a bit is the finest processing granularity, instruction-level parallelism [115], where multiple instructions

[1]Some people might make a distinction between *parallel computing* and *distributed computing*, arguing that processing units are more tightly coupled in parallel computing (e.g. hardware cores in a processor) than in distributed computing (e.g. different computers in a cluster). In this dissertation, we do *not* make such a distinction. We simply use the term *parallel computing* to refer to computations that involve multiple computer programs or modules.

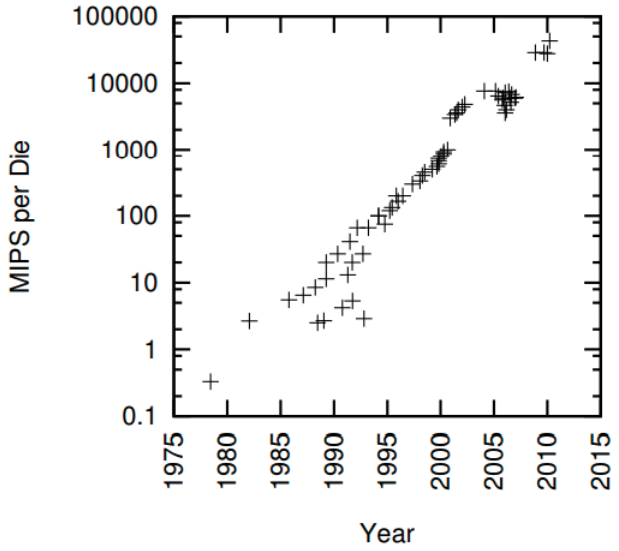

Figure 1.2: MIPS per die trend for Intel CPUs, adopted from [75].

of one program are executed concurrently on different functional units of a processor, and *task-level parallelism* [26, 101], where multiple tasks run in parallel. *Task* here refers to a *sequential* computation module of an application. We discuss task-level parallelism in this dissertation. We define a *computation* as the process of a task computing on an input data item and use X_i to represent the computation of task X on the ith data item. We use Figure 1.3, which shows the dataflow graph of an application with four tasks u, v, w, and x, to help explain different forms of parallelism. Figure 1.4 is a *computation diagram* [2] of Figure 1.3 with three input data items. To differentiate computations from tasks, we use dashed circles to represent computations. We remove task w from the graph in Figure 1.4b to simplify the diagram.

Given two *computations* a and b, if b cannot start before a finishes, we say that b is dependent on a. For example, in Figure 1.4b, both v_1 and u_2 are dependent on u_1. Dependencies are transitive. Note that data transmission implies a dependency (*data dependency*). We use solid lines to represent data dependency and dashed lines to represent other possible dependencies. Even if two *tasks* have data transmission between them, their computations are not necessarily dependent.

[2] We do not use the term *computation graph* because historically it was used to refer to some special systems [58].

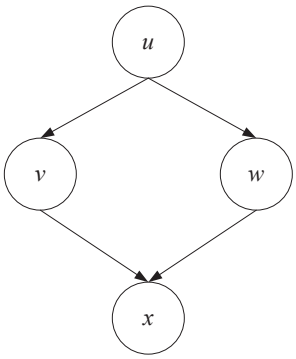

Figure 1.3: An example of the dataflow graph of an application. The circles represent tasks, while the arrows represent unidirectional data transmission.

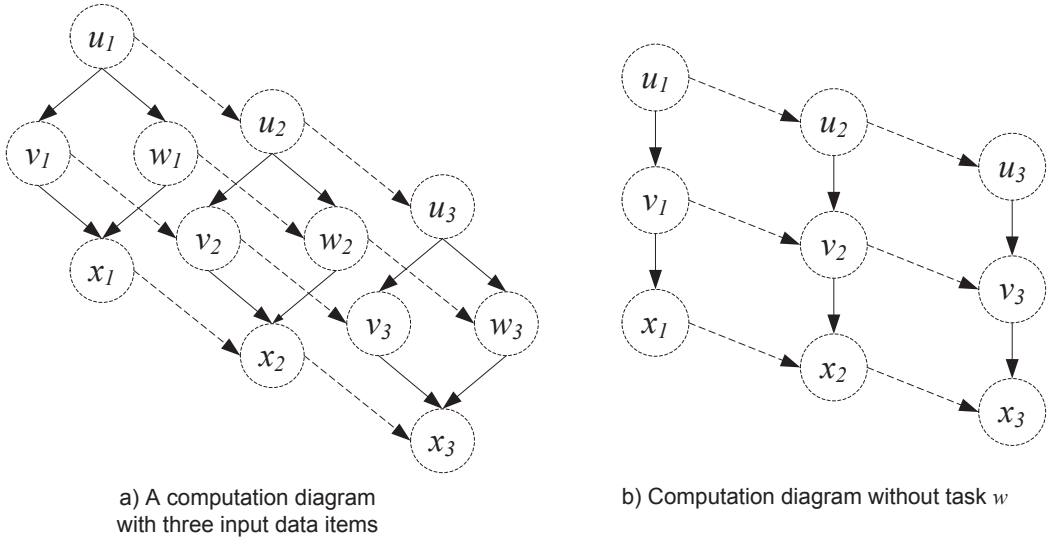

Figure 1.4: Computation diagram for three input data items. Solid lines represent data transmission, while dashed lines represent other dependencies between computations.

For example, tasks v and u are connected by a data transmission channel, but v_1 and u_2 have no dependency between them, so tasks u and v can actually run in parallel on different data items.

Pipeline Parallelism. Computations v_1 and u_2 can be executed in parallel, as can x_1, v_2, and u_3. We can generalize this pattern as follows: x_i, v_j, and u_k, $i < j < k$, have no dependency between them and so can run concurrently. In practice, the tasks u, v, and x can be mapped onto separate executable resources. After the initial setup, u, v, and x can work on different data items concurrently like a workflow pipeline, so this form of parallelism is named *pipeline parallelism*.

Data Parallelism. Whether dependencies exist between v_1, v_2, and v_3 depends on the task v's program. If task v is *stateless*, which means previous input data have no effect on the processing of the current data (v always produces the same output for a given input regardless of input history; otherwise, v is *stateful*), then there is no dependency between input data and v_1, v_2, and v_3 can run in parallel. This form of parallelism is called *data parallelism*.

If v is stateless and processing one data item by v takes much more time than by u and by x, we can improve application throughput by running multiple instances of v in parallel. In contrast, if node v is stateful, it is not easy to extract data parallelism from v. Thies [110] demonstrated a method combining data duplication and batching to extract data parallelism from stateful nodes when node state is based on a sliding window of data history.

Task Parallelism. Regardless of the values of i and j, v_i and w_j have no dependency between them and can always run in parallel. This is because there is no data exchange between v and w in the application's dataflow graph. We name this form of parallelism *task parallelism* if v and w are not replications of the same program, which is just one type of *task-level* parallelism.

To summarize, parallelisms might be available if there are multiple tasks or multiple input data items. Pipeline parallelism is exposed by *different* and *dependent* tasks processing *different* input data; data parallelism is exposed by the *same* task processing *different* input data; task parallelism is exposed by *different* and *independent* tasks processing *any* input data, same or not. Note that the availability of multiple data items is a necessary but not sufficient condition for the availability of data parallelism. Stateful computing could still eliminate data parallelism. Table 1.1 is a summary of the classification.

1.1.2 Streaming Processing

Besides the trend of parallel computing, we are also seeing a change in data processing patterns. Because computer memory is randomly accessible, and many data structures and algorithms require random access (e.g. trees and binary search), random data access has been a commonly used

Table 1.1: Summary of different task-level parallelisms

	Same Task	Different Tasks
Same Input Data	No Parallelism	Task Parallelism
Different Input Data	Data Parallelism	Pipeline Parallelism[a] Task Parallelism[b]

[a] when tasks are dependent
[b] when tasks are independent

pattern. However, random data access exposes no spatial locality and fails to utilize caching and prefetching [27].

We use a microbenchmark to show the time difference between the random access and sequential access. In each test run, we accessed all n elements in an array of size n in sequential order and in random order, respectively. For each element, we performed a sequence of reading, modifying, and writing operations. All test runs started with cold cache on a machine with an Intel Core i5 processor (3MB cache) and 4 GB memory. From Figure 1.5 we can see that as data size grows, sequential access shows more advantage over random access. When the array size is 2^{23}, sequential access took less than half of the time taken by random access.

Random data access requires randomly accessible memory. In the big data era, however, this can be challenging for some applications. For example, graphs for social networks and biological networks, such as protein interaction networks, can have billions of edges [105]. Such data are too big to be entirely loaded into today's computer memory. If we store the data in disks, we should not use random data access because disks are too slow[3].

Streaming processing is a better choice for processing big data. On the one hand, it requires memory constant to the size of a data item; on the other hand, streaming can hide disk latency [38, 39, 121]. Note that not all applications are suitable for streaming processing. Historically, the terms "stream," "stream processing," and "streaming processing" have been used to describe different models and systems. "Stream" in computer science refers to a sequence of data, which can

[3]For a disk with a rotational speed of $7,200$ RPM (revolutions per minute), it can be calculated that the average seek time is 4.2 milliseconds, which is substantial considering clock speeds of today's computers.

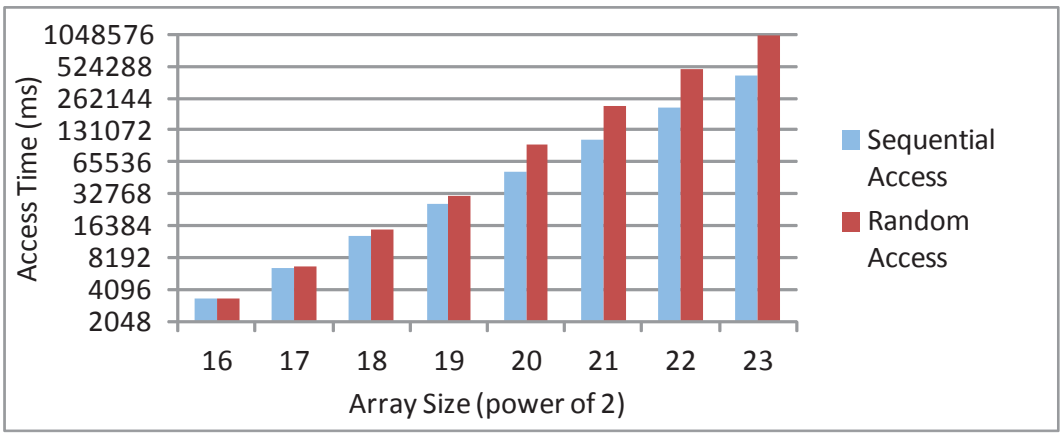

Figure 1.5: Comparison of data access times.

be finite or infinite, such as I/O streams [2]. "Stream" is also a data type in some dataflow languages (e.g. Lucid [7] and Lustre [97, 46]) and functional programming languages (e.g. Scala [86] and Haskell [51]). In data processing, the term "streaming processing" has been used to describe pipelined SIMD (single instruction, multiple data) systems, where a "data stream" is a finite set of data with known length [88]. Each pipeline module computes on an entire stream in one computation. Example languages and systems include Brook [13] and Merrimac [28].

In this dissertation, "streaming processing" is a data access pattern where data are processed sequentially according to their stored locations or arrival times, as opposed to randomly. Here, a "data stream" is an *unbounded* sequence of data [106]. Note that streaming processing is not necessarily parallel computing. A sequential program can also do streaming processing. Some streaming processing systems involve no parallel processing, such as Aurora, a data stream management system [4].

1.2 The Streaming Computing Paradigm

Parallel computing has become an important solution for computation-heavy applications, and streaming processing can handle big data. When we combine the two together, we have a new computing paradigm: *streaming computing*[4].

1.2.1 Integrating Parallel Computing and Streaming Processing

A streaming computing system is a parallel computing system with computing *nodes* connected by *first-in first-out (FIFO)* data channels. Each node runs a *streaming processing* module (i.e. a task); each channel is *unidirectional* and delivers data in order. There are three types of nodes in the system: source nodes, sink nodes, and intermediate nodes. Source nodes read data from external data sources (e.g. sensors, network requests, and database queries) and emit data to their receivers; intermediate nodes receive data, process them, and send intermediate output data to their downstream receivers; sink nodes are responsible for writing final output. Figure 1.6 shows the dataflow graph of a streaming computing system with five nodes and channels between the nodes. u and y the are the source node and the sink node, respectively. Other nodes are intermediate nodes.

Streaming computing is suitable for applications that can be decomposed into multiple tasks and process multiple data, which together expose various forms of parallelism for streaming computing to exploit. For example, in Figure 1.6, nodes u, v, x, and y (or u, w, w, and y) represent pipeline parallelism, as they can process different data concurrently like a workflow pipeline. If v and w are duplicated tasks, they represent data parallelism; otherwise, if v and w are different tasks, they represent task parallelism.

Streaming computing systems can be decentralized, which means they do not need a central authority to manage nodes and channels, such as streaming applications deployed by frameworks

[4]The term *distributed streaming processing* also refers to this paradigm [25, 52].

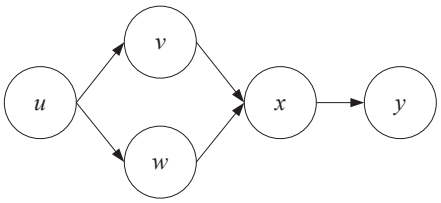

Figure 1.6: Dataflow graph of a streaming computing system.

such as Auto-Pipe [20] and StreamIt [109]. These systems can also be centralized with a master node to provide management over the system, such as applications deployed by frameworks Storm [1] and MillWheel [5]. In this dissertation, we focus on decentralized models for streaming computing.

As it takes advantage of streaming processing, parallel computing, and FIFO communication, streaming computing has the following attractive features:

- **Exploiting parallelisms**. Streaming computing can exploit all three common parallelisms in applications (data parallelism, pipeline parallelism, and task parallelism), as illustrated by Figure 1.6.

- **Exploiting data locality**. Since each node processes data in streaming fashion, the spatial locality of input data is utilized. The FIFO communication also exposes producer-consumer locality [27].

- **Analyzable Properties** Because of FIFO communication and streaming processing, data movement and computations are well-organized. Analytical tools can be used to analyze system properties. For example, queueing theory [45] can be used to predict application performance [82, 90, 92], and linear algebra can be used to analyze memory requirement of some streaming applications [64, 66].

We will review the history of streaming computing and its formal models further in Chapter 2.

1.2.2 Design Concerns in Streaming Computing

Like all computing paradigms, streaming computing is concerned with two fundamental issues: performance and correctness.

Performance is important because it is the reason we turn to parallel computing. Two important metrics for performance are throughput and latency. Throughput is the amount of data processed per unit time, while latency is the time spent in processing one data item. For a single sequential program, reducing latency is equivalent to improving throughput. But when multiple tasks need to coordinate with each other, due to communication overhead, throughput and latency can sometimes go against each other [90]. For example, pipelining improves throughput by exploiting pipeline parallelism, but it also increases latency for each data item. There are numerous performance optimization problems considering different constraints and optimization goals, such as optimizing throughput [68], optimizing latency [114], balancing throughput and latency [107, 92], and minimizing power usage [9]. Hirzel et al. provide a comprehensive summary of performance optimization techniques for streaming computing [48].

While performance is important, *correctness*, which is prerequisite to performance, should be guaranteed first. Correctness of a streaming application means that the application can finish and yield correct results. Even assuming each task of an application executes correctly, there are still factors at network level that can affect correctness, such as deadlocks [67, 94, 102] and failure of hardware or software [52, 8, 120]. In this dissertation, we focus on the correctness of streaming applications, in particular, deadlock avoidance.

1.3 Problem Statement

In streaming computing, some nodes *synchronize* input data and/or *filter* output data in a data-dependent and unpredictable fashion. When a streaming application has both filtering and synchronization, it might require unbounded memory to process unbounded streams, which means

deadlock given bounded memory[5]. Our goal is to execute such applications with bounded memory while avoiding deadlocks.

1.3.1 Filtering in Streaming Computing

According to how output streams are produced from input streams, the *producing behaviors* of streaming computing nodes with input data items x_i can be roughly classified as mapping (e.g. $x_i \to x_i+1$), reduction or aggregation (e.g. $x_1, x_2, \ldots, x_N \to \sum_{i=1}^{N} x_i$), and filtering (e.g. $x_i \to x_i$ *if* x_i is even *else* nothing). In this dissertation, we focus on the *filtering* behavior. Filtering here means that a node consumes input but does not produce output. It is different from the filtering in signal processing [56, 72], which is actually selective mapping by the definitions above.

Many applications expose filtering behaviors: in computer networking, a packet filter drops packets that fail to meet firewall rules [31]; in machine learning, a classifier filters datasets that do not have the required feature [113]; in a gamma-ray observation system, a data processing module discards images that do not indicate any gamma-ray events [111]. Figure 1.7 shows a filtering module f that receives data from s and sends data that pass its filtering rule to t for further processing.

Figure 1.7: A streaming pipeline with a filtering node.

1.3.2 Synchronization for Determinism

According to how input streams are consumed, the *consuming behavior* of a streaming computing node can be synchronized or unsynchronized. Synchronized consumption at a node means that if the node has multiple input channels, it decides the number of data items consumed from each

[5]Some people call it "artificial deadlock" to distinguish from deadlocks caused by all empty channels [49, 43].

channel during a computation based on the available data at all input channels. Synchronized consumption or *synchronization* is usually adopted because of its importance to determinism.

To understand synchronization, consider the scenario that f in Figure 1.7 takes a lot of time to process a data item and hence is a performance bottleneck. To speed up data processing, we can add data parallelism to the application by replicating f in multiple copies to process data concurrently, as Figure 1.8 shows. Now t faces a question: when there are data at its input channels, from which channel it should choose to consume data? It can randomly choose a channel, which, however, leads to nondeterminism as the same sequence of input data may result in different output sequence in different runs. Another option is assigning priorities to input channels. For example, if both f_1t and f_2t have input data, t chooses to consume data from f_1t. But this still does not guarantee determinism in the presence of unpredictable delays in data transmission on these channels. Filtering makes the problem more complicated as an anticipated data item may never arrive.

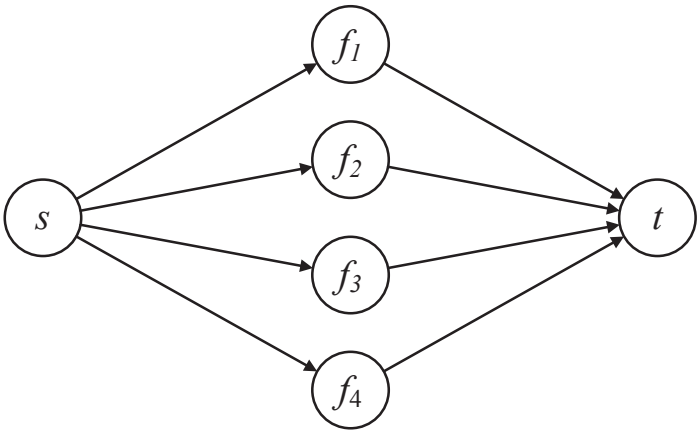

Figure 1.8: A streaming application with multiple filtering nodes.

To counter transmission delay and filtering, we can add a data index to each data item. The data indices at each channel should be strictly increasing so that t can synchronize input data accordingly. If an input channel has no data available, t might have to wait, not consuming data from other channels, because a data item with a lower index might eventually arrive at the empty input channel. However, waiting can cause problems. Imagine that, in a rare situation, f_1 filters all incoming data, but t does not know this and waits for data from f_1; in the meantime, both channel

sf_2 and channel f_2t fill up, causing backpressure to s. Now s and f_2 are blocked by full output channels while f_1 and t are waiting on empty input channels, which means a *deadlock*.

If t can poll f_1, asking whether f_1 has filtered data with indices up to some i, such deadlocks might be avoided. However, we avoid this approach for two reasons. First, it requires a communication channel going the opposite direction from the data channel, which adds listening overhead to the sender and makes it harder to analyze application properties. Second and more importantly, it is difficult for t to decide when it should poll upstream senders; t could end up sending many unnecessary polling requests, only to hurt performance. As a result, instead of letting receivers poll senders, we try to let senders notify receivers about filtering *when it is necessary*.

1.3.3 Synchronization Is a Natural Application Behavior

In the previous example, synchronization is added for determinism when data parallelism is exploited. We may avoid synchronization in Figure 1.8 by not replicating f if we are satisfied with the performance. However, in some applications, synchronization is a natural behavior that cannot be eliminated. Below, we describe an example of such a system, Mercury BLAST [17].

Mercury BLAST 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 1.9. 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.

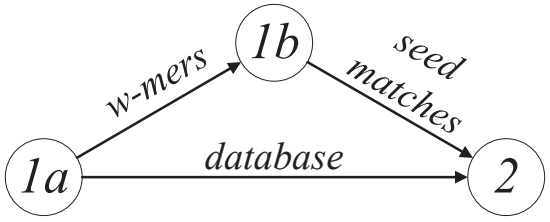

Figure 1.9: The first two stages of Mercury BLAST

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. The filtering ratio at module 1b is usually higher than 98%, making the application very vulnerable to deadlocks.

In Figure 1.8, all filters f_i use the same filtering rule. In contrast, in Figure 1.9, multiple data streams from the same node (1a) are processed with different filtering rules (think of an imaginary

node between module 1a and module 2 that simply forwards database locations), which inherently requires synchronization.

1.3.4 The Deadlock Issue

As demonstrated by the two example applications above, when filtering and synchronization are both featured in an application, they pose deadlock risk. Bounded memory is part of the reason for deadlocks. If channels have unbounded memory, such deadlocks cannot happen because no node is ever blocked by a full channel. Unfortunately, in real systems, memory is limited, so deadlock is a real threat to streaming applications. For example, Mercury BLAST runs with limited memory resources and does not resize channel buffers during runtime, as is typical in embedded systems. In the rest of the dissertation, the terms *bounded-memory execution* and *deadlock avoidance* are used interchangeably.

Because streaming applications have strong performance requirements, potential deadlock solutions should be lightweight, not causing significant overhead. Considering some applications' inability to resize channel buffers during runtime, potential solutions should not require dynamic buffer resizing.

1.4 Summary of Our Approach

For a specific application, *ad hoc* solutions may be used to avoid deadlocks, but we prefer generic solutions that can be applied to a broad range of applications. To find generic solutions, we turn to designing *dataflow models*. If we can model the range of possible filtering and synchronization behaviors, we can then work on bounded-memory scheduling in such models to provide model-based solutions.

We propose a new model named *synchronized filtering dataflow* (SFDF) featuring both data filtering and synchronization. To prevent deadlocks in SFDF, we augment data streams with special

messages called *dummy messages*. The augmentation is based on both the application's dataflow graph and nodes' runtime behavior. We try to add as few dummy messages as possible to reduce computation and communication overhead. For applications with special dataflow graphs, we can can improve the efficiency of our generic solutions without sacrificing correctness.

Our work involves algorithm design and extensive theorem proving because correctness of execution is our top priority. We also provide experimental evaluation when necessary.

1.5 Contributions

In this dissertation, we make the following contributions:

- In Chapter 3, we propose the deterministic *synchronized filtering dataflow* (SFDF) model, which features both filtering and synchronization behaviors. We characterize under what circumstances an SFDF application can deadlock with bounded memory.
- In Chapter 4, we design decentralized algorithms to guarantee bounded-memory execution of SFDF applications. Our algorithms augment data streams with dummy messages. Each algorithm has two parts, a compile-time part and a runtime part. The compile-time part computes a *dummy interval* for each channel; the runtime part schedules dummy messages according dummy intervals. The runtime algorithm adds negligible overhead to applications, but the compile-time algorithm could have high asymptotic complexity.
- In Chapter 5, for applications with special dataflow topologies, we provide efficient compile-time algorithms to compute dummy intervals. In particular, we focus on topologies that are series-parallel DAGs or CS4 DAGs, which are a new category of DAGs defined by us. In a CS4 DAG, each *undirected cycle* has only one source node and one sink node.
- In Chapter 6, we use polyhedral theory to develop safe dummy interval polyhedra to provide multiple sets of safe dummy intervals for application developers. We provide a polynomial-time algorithm to verify the safeness of dummy intervals for SP-DAGs.

- In Chapter 7, we extend our SFDF model to support precise synchronization of data streams and *control messages*. Dummy messages are only one type of control messages that are used by streaming applications. There is also other control information, such as data boundaries and node configurations, that needs to be passed from upstream nodes to downstream nodes. Control messages including dummy message are integrated into the model and precisely synchronized with data streams. Bounded-memory execution of application is still guaranteed. The extended model can help improve performance of some applications by facilitating the conversion of a non-filtering application to a filtering application.

Chapter 2

Background and Related Work

2.1 A Brief History of Streaming Computing

Streaming computing can date back to the 1960s, when the computation graph [58, 76] was proposed as a model for parallel computing. In the early 1970s, another model, Kahn's process network (KPN) [55], was also proposed. Both models feature computing nodes connected by FIFO data channels, which are conceptually similar to today's streaming computing systems. In the 1980s, an influential dataflow model, synchronous dataflow (SDF) [64, 66], was invented by Lee et al. In an SDF application, computing behaviors are static and defined before computations start, which makes static schedules possible. SDF has found success in many areas, especially signal-processing applications [63, 29, 93]. Following SDF, dataflow models such as boolean dataflow and dynamic dataflow were proposed to allow data-dependent node behaviors [14, 15, 65]. Based on those dataflow models, dataflow programming languages, which describe a program as a dataflow system, were designed to explore non-von Neumann programming, such as KPN-based Lucid [7] in the 1970s and SDF-based Lustre [46, 97] in the 1980s. Those models and languages can handle unbounded streams.

Pipelined SIMD processing, which is also a form of streaming computing, began to draw attention in 1990s, when Streaming SIMD Extensions (SSE) were added to x86 architectures for streaming processing [108, 98]. In the 2000s, specialized stream processors were used as co-processors to take advantage of the power of streaming computing [57, 60, 28, 119] . Meanwhile,

the use of GPU for streaming computing was also studied [13, 47, 103]. The use of GPU for pipelined SIMD processing has taken off since then [85]. In recent years, embedded systems such as the field-programmable gate array (FPGA) [44, 80, 54, 100] and multiprocessor System-on-Chip (MPSoC) [50, 99, 116, 122, 81] have also been popular platforms for streaming computing.

Numerous languages and frameworks for processing unbounded streams have also been designed since the early 2000s. StreamIt [109] is a language and a compiler for developing SDF-based applications. Auto-Pipe [20] is a framework for developing streaming applications on architecturally diverse systems. Database management systems (DBMS) are another area in which streaming computing has thrived. Compared with traditional DBMSes, which manage stored finite data, a stream-based DBMS is designed for continuous queries on real-time data streams. Aurora [4] and Borealis [3] are examples of such systems. As big data prevail, streaming computing has become an important choice for data processing in many companies. In recent years, many frameworks have been implemented and used in processing real-time data, such as Storm [1], S4 [83], Kafka [61], and MillWheel [5].

As to applications, streaming computing has found success in many application areas, such as digital signal processing [30, 64, 111, 29], computational biology [37, 53, 71], multimedia [34, 59, 89], database management systems [6, 3, 22], and web data analysis [83, 5].

2.2 Models of Streaming Computing

Though streaming computing is a relatively new trend, many formal models of this paradigm have historically been proposed. As this dissertation foci on model-based solutions, we now review some influential models and discuss their ability to model the filtering and synchronization behaviors described in Chapter 1.

Kahn Process Networks

In the early 1970s, Gilles Kahn introduced a computation model where sequential processes communicate through first-in first-out channels [55], which was later referred to as the Kahn Process Network (KPN). In KPNs, processes are determinate, and channels have unlimited buffering capacity. Each process can be associated with multiple input and output channels, and the data consumption is synchronized for determinism. Filtering is not prohibited in KPNs; however, since channels are assumed to be unbounded, deadlocks due to full data channels do not exist in the model, though they can be a problem in real-world systems.

Computation Graphs

To represent task and pipeline parallelisms in program loops, Karp and Miller formulated *computation graphs* [58], which are structurally similar to KPNs. A computation graph is also a network of processes connected by FIFO queues (or channels). Different from a KPN, each queue is parameterized by a tuple (A, U, W, T), where A is the initial number of data items in the queue, U is the number of data items produced on the queue each time the producing process is fired, W is number of data items removed from the queue each time the consuming process is fired, and T is the least number of data items required for the consuming process to fire. In [58], the authors provided necessary and sufficient conditions to decide the termination and memory boundedness of computation graph networks. Since data consuming and producing rates are fixed, the model prohibits data-dependent filtering computations.

Synchronous Dataflow

Synchronous dataflow is a restricted version of KPN or computation graph. Like these models, an SDF network is also a network of computing nodes connected by FIFO channels. Each channel has known and static data consuming and producing rates, which are called *sample rates* [64,

66]. Homogeneous dataflow (HDF) [64] is a special case of SDF where all sample rates are 1. While KPNs and computation graphs are focused on computability issues such as determinacy and termination, the inventors of SDF provided scheduling strategies for bounded-memory execution (if possible), which makes SDF more attractive than the other two models.

A *periodic schedule* of an SDF application clears all channels and return to its initial status after each node repeats execution a specified finite number of times. With a periodic schedule, the application can process unbounded data with bounded memory. However, not all SDF applications permit a periodic schedule. The sample rates of an SDF application are *consistent* if a periodic schedule exists; otherwise, they are inconsistent. For example, given SDF graphs with indicated sample rates in Figure 2.1, graph 2.1a is inconsistent, and no periodic schedule can be found; graph 2.1b is consistent as node A, B, and C can be executed for 1, 1, and 2 times in a periodic schedule.

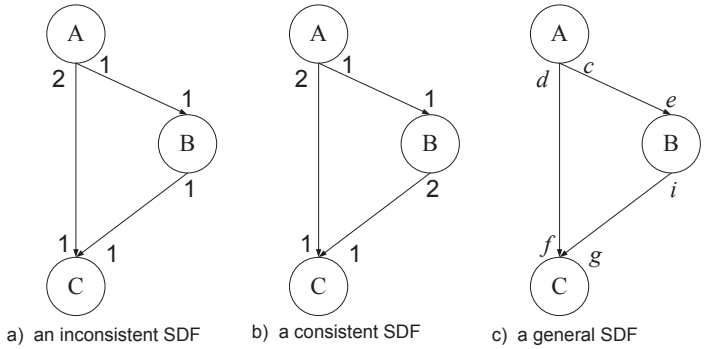

Figure 2.1: Examples of SDFs [66]

To compute a periodic schedule, a *topology matrix* is defined in [66], where each row represents an arc and each column represents a node. The (i, j)th element in the matrix is the number of data items placed on i after each invocation of j. If i is an input channel for j, element (i, j) is negative. The topology matrix for Figure 2.1c is as follows:

$$\begin{pmatrix} c & -e & 0 \\ d & 0 & -f \\ 0 & i & -g \end{pmatrix}.$$

Lee et al. proved that given a connected SDF graph, a necessary condition for the existence of a periodic schedule is that the rank of the topology matrix is $s - 1$, where s is the number of nodes. This condition is sufficient if the SDF topology is a directed acyclic graph. A vector q with positive integers in the nullspace of the topology matrix is a valid repetition vector for the SDF, which can be used to construct a periodic schedule.

Since its invention, SDF has been popular for many applications, especially digital signal processing [64, 10]. The static schedule property is very helpful for orchestrating deadlock-free execution, and it has become a widely used dataflow model. A number of frameworks have been implemented based on it, such as Ptolemy [16] from Berkeley and StreamIt [109] from MIT. However, SDF prohibits data-dependent filtering.

Cyclo-Static Dataflow

Cyclo-Static Dataflow (CSDF) is a variant of SDF [36, 11]. The difference is that instead of static sample rates for each channel, the sample rates change cyclically according to *statically* defined cyclic values. Parks et al. pointed out that CSDF has advantages over SDF in eliminating dead code and exposing additional parallelism [96]. They also provided a method to transform a CSDF graph into an SDF graph using higher-order functions. Like SDF, CSDF also prohibits data-dependent filtering computations.

Boolean Dataflow and Dynamic Dataflow

SDF and CSDF cannot handle dynamic node behaviors, e.g. conditional execution. Boolean dataflow (BDF) and dynamic dataflow (DDF) were proposed to support dynamic behaviors [14, 15, 65]. Compared to SDF, BDF adds two kinds of *switch nodes*, which are like multiplexers or demultiplexers. A control token tells the switch from which channel a data item should be consumed (for input cases) or to which channel a data item should be produced (for output cases). DDF extends BDF to support control switches to consume multiple control tokens and allow computing nodes

execute conditionally based on input data. BDF and DDF are Turing-complete; however, whether an arbitrary BDF or DDF application can be scheduled with bounded memory is undecidable [14].

Summary of Related Models

KPNs allow filtering and synchronization, but they are assumed to have unbounded channel buffers, so they have no deadlock associated with full channels. Computation graph, SDF, and variants like CSDF all prohibit filtering, so they cannot model streaming applications with filtering computations. BDF and DDF can model applications with dynamic data rates, but we cannot check whether an arbitrary BDF/DDF application can execute with bounded memory or not. Moreover, although BDFs and DDFs are Turing-complete, implementing filtering behavior on them is not straightforward. In short, none of these existing models can guarantee bounded-memory execution while allowing data-dependent filtering computations.

2.3 Deadlock Avoidance Approaches

Deadlocks in computer systems can be divided into two categories: resource deadlocks and communication deadlocks [23]. Resource deadlocks are caused by multiple processes trying to access a resource that must be accessed exclusively, such as the Dining Philosophers' Problem [32, 33]. Communication deadlocks, which usually happen in distributed systems, are caused by multiple processes waiting for communication activity from each other, such as the deadlocks described in Chapter 1.3.4.

Communication deadlocks have been well studied. Chandy et al. developed algorithms to detect distributed deadlocks based on probes [23, 24]. Mitchell and Merritt designed a deadlock detection algorithm using public and private labels [78], which are similar to the notion of Chandy's probes. After raising the issue of artificial deadlock in bounded KPNs, Parks tried to avoid such deadlocks by dynamically increasing channel capacity [95]. Geilen and Basten improved Parks' idea and

proposed a new scheduling algorithm which guarantees fairness and behaves correctly for bounded and effective KPNs [43]. Here "effective" means all tokens produced are ultimately consumed. This algorithm also requires dynamic changes to channel capacity. Olson and Evans improved Mitchell's algorithm to detect local deadlocks in bounded KPNs [87]. All these deadlock avoidance and resolution algorithms require runtime change to channel capacities, while we seek algorithms that do not.

We avoid deadlocks by augmenting data streams with special dummy messages, which are inspired by the null messages [42, 77] in parallel discrete-event simulation (PDES). In a discrete-event simulation, a system operates on a ordered sequence of events with time stamps. PDES utilizes parallel systems and runs multiple processes concurrently during simulation. Figure 2.2 shows a PDES system with multiple processes. At merge point M, events from $Proc\ 1$ and $Proc\ 2$ need to be synchronized according to their time stamps. After finite time, a deadlock can occur if $Proc\ 1$ routes every message to M while $Proc\ 2$ does not route any message, which is very similar to the deadlocks we described in streaming computing applications.

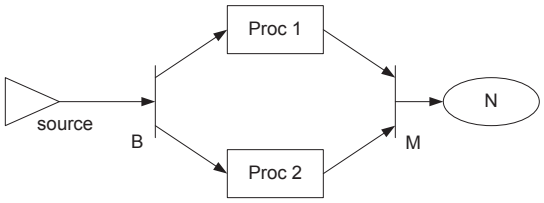

Figure 2.2: A parallel discrete-event simulation system from [77].

Chandy and Misra proposed null message-based methods to resolve deadlocks in PDES [23, 77]. A null message *(t, null)* received by a process p means that no message will show up until time t. Null messages mean the absence of messages and allow receivers to advance their clocks safely to avoid deadlocks. While we use a similar idea in dummy messages, we make contributions in providing provably correct schedules for sending dummy messages.

Chapter 3

The Synchronized Filtering Dataflow

In streaming computing, filtering and synchronization can cause deadlocks, as we explained in Chapter 1. To prevent such deadlocks and ensure bounded-memory execution, we prefer model-based solutions. In this chapter, we introduce our *synchronized filtering dataflow* (SFDF) model and explain the conditions for deadlocks.

3.1 General Description

Besides the basic features of streaming computing described in Chapter 1.2, the SFDF model assumes that dataflow graphs are directed acyclic multigraphs (DAMGs), which are most common in streaming applications. A *multigraph* differs from an ordinary graph in that there can be multiple edges between two vertices. The dataflow graph of a streaming application can be a multigraph when some senders send multiple data streams to their receivers, each of those data streams with its dedicated channel. Figure 3.1 illustrates the notion of multigraph by connecting the sender and the receiver with two channels, which may deliver streams of the same data type (e.g. two streams of integers) or streams of different data types (e.g. a stream of integers and a stream of floats).

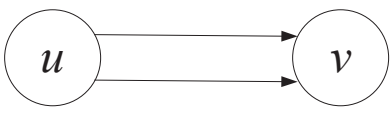

Figure 3.1: A pair of nodes connected by two channels.

Each node computes *sequentially* and spends a finite but *unpredictable* amount of time to process an input item, which we call a *token*. There are two types of tokens: tokens that are from the original input data and tokens that are generated by nodes during the computing process to carry control information. When we need to distinguish the two types, we use *data token* to refer to the first type and *control message* or simply *message* (e.g. dummy message) to refer to the second type.

Each channel in the model has a finite buffer capacity, which is known at compile time and does not change during runtime. In practice, it might be possible to shrink or expand the channel buffers of software applications, but for hardware modules (e.g. FPGA applications), it is very hard to change buffer size at runtime, so as a general rule, we assume channel buffers cannot be resized during runtime. We denote the buffer size of channel q as $|q|$, which means q can hold up to $|q|$ tokens. Channels in our model are reliable and guarantee ordered transmission, so tokens on a channel are received in the same order as they are emitted. However, a system provides no timing guarantee. There may be an arbitrary finite delay before a token emitted into a channel is received. Streams can be bounded or unbounded. If a stream is bounded, there should be an *End-of-Stream* message with index ∞ at the end of the stream.

Each token is associated with an integer, which we call its *data index*. A token emitted into a channel should have a strictly larger index than the ones emitted earlier. Note that those indices might be implicit. For example, in Mercury BLAST, the database stream consists of strictly increasing database locations, which are also indices. Note that indices are not necessarily consecutive; for example, filtering can cause index gaps.

3.1.1 Notations

In this dissertation, we will be consistent with notations for SFDF. Given a dataflow graph, we use letters from $\{s, t, u, v, w, x, y, z\}$ to represent nodes, and s and t are usually used to represent the sole source node and the sole sink node, respectively. To refer to an *edge* in a graph, we use

e or the connected vertices (e.g. uv), while to refer to a *channel* in an application, we use q. In most chapters, e and q can be used interchangeably with the exception of Chapter 7, where an graph edge is associated with two application channels, so we have to distinguish graph edges from application channels. We use p to denote a path in a dataflow graph. $|a|$ denotes the length or the buffer size of a, which could be an edge, a path, or an application channel. We use C to refer to an *undirected cycle* in dataflow graphs. For other notations that are not used frequently, we will explain them when first using them.

3.2 Synchronizing and Filtering Behaviors

When a node has multiple input channels, it needs to decide how many tokens to consume from each channel during a computation. In SFDF, a computation at a node *consumes only input tokens with the same index*, which is called the *computation index*. At any time, a node's current computation index is the index of the last set of inputs that it consumed. Computation on data with index i does not require that all input channels contain tokens with that index; it is well-defined even if only a subset of input channels ever receive tokens with index i. However, a node may not proceed to compute for index i unless it knows that no further tokens with this index will ever arrive at its inputs. In other words, input tokens are synchronized by data indices, and all input tokens with the same index must be consumed in one computation. Since no two tokens in a channel have the same index, at most one token can be consumed from a channel during one computation.

A computation may output tokens with the same index as its inputs on any subset of a node's outputs, including the empty set. We say that a computation *filters* a data token on a channel q if it does not result in an output token on q. Filtering is a data-dependent behavior, performed independently by each node, that cannot be predicted at the time that a system is constructed. For example, a filtering node may decide whether to pass a data token depending on the result of a predicate.

Detailed behavior of a single *intermediate* node is described in Algorithm 3.1. For conciseness, we do not explicitly describe single-node behaviors of source node and sink node, which are similar to the behavior of intermediate node except source nodes do not consume tokens and sink nodes do not send tokens. Note that, in this and all following protocols, all **emit** operations block until the output channel is not full.

Algorithm 3.1: Behavior of a single intermediate node in SFDF.

```
1 ComputeIndex ← 0
2 while ComputeIndex ≠ Index of EOS do
3     wait until every input channel has a pending token
4     let i be minimum index of any pending token
5     consume pending tokens with index i from input channels ComputeIndex ← i
6     compute on data tokens with index i
7     emit output tokens with index i
```

3.3 Deadlock Concerns

3.3.1 Deadlock Example

Due to the filtering and synchronizing behaviors, even if each node runs on an independent computing resource, deadlocks are still a potential threat to the execution of SFDF applications. Figure 3.2 illustrates a deadlock in an SFDF application with four nodes. u and v are blocked due to full channels, while w and x are blocked due to empty channels. If there is unbounded memory, deadlocks like this would not have happened, because there would not be any full channels and the cycle of blocking relations is hence broken. For real-world applications, however, memory and channel buffers are bounded. For applications deployed on embedded platforms such as FPGAs, memory resources can be scarce, and resizing channel buffers during application runtime can be difficult. Even if runtime memory resizing is possible, there is no guarantee that memory would be sufficient to prevent deadlock.

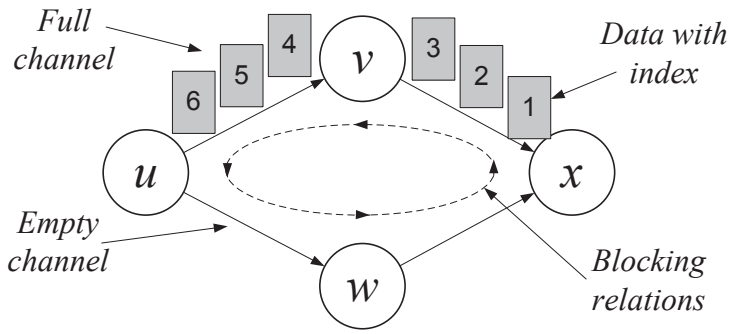

Figure 3.2: A deadlock example in SFDF. Both uv and vx are assumed to have a buffer size of 3.

3.3.2 Conditions for Deadlock

During a computing process, one node may be temporarily blocked by another due to an empty input or full output channel. However, not every blocking situation is a deadlock. In this section, we derive the conditions under which blocking can lead to deadlock in SFDF.

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 *(**Liveness**) If a node can increase its compute index in finite time, we say the node is* live*, or equivalently that it* makes progress*.*

Definition 3.3 *(**Deadlock**) A system is said to* deadlock *if no node in the system is live, but some channel in the system still retains unprocessed tokens (so that the computation is incomplete).*

We now prove that a cycle of blocking relations is a sufficient and necessary condition of deadlock.

Theorem 3.1 (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 DAMG. Hence, the blocking cycle contains at least one full channel, which means there are unprocessed tokens, 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 $v \dashv w$. (H is also called a "waiting-for graph" [23, 78].) 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 tokens 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 tokens, or it must have received the EOS message 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. □

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 3.2, uw and wx are blockwise channels, while uv and vx are counterblockwise channels.

We note that not all systems have deadlocks. For example, a system with just two nodes connected by one channel will never deadlock, even with filtering; the sender can block the receiver because the channel is empty, or the receiver can block the sender because the channel is full, but they cannot block each other at the same time. However, even quite simple systems, such as one with just two nodes connected by two parallel data channels, can deadlock.

We claim that filtering and synchronizing behaviors are two necessary (but not sufficient) conditions for deadlocks in this model.

Theorem 3.2 *(**Filtering Theorem**) If no node ever filters any input, then the system cannot deadlock.*

Proof.

The proof is by contradiction. Suppose there is a deadlock; then by the Deadlock Theorem, the computation reaches a state in which some node $y \dashv^{+} y$. Let C be the cycle of blocked nodes that includes y. Each node z on cycle C may be labeled with one of four types, depending on the directions of the channels that link z to its two neighbors in C:

1. Both channels are oriented blockwise, as for node w in Figure 3.2;
2. Both channels are oriented counterblockwise, as for node v in Figure 3.2;
3. The channel located to blockwise of z is oriented blockwise, while that to counterblockwise of z is oriented counterblockwise, as for node u in Figure 3.2;
4. The channel located to blockwise of z is oriented counterblockwise, while that to counterblockwise of z is oriented blockwise, as for node x in Figure 3.2.

We now argue that, in the absence of filtering, the minval of a channel on C is always $\geq$ that of its counterblockwise neighbor. Let z be a node between two channels on the cycle.

- If z has type 1, both channels are empty, with one pointing into z and one pointing out. Because z does not filter, every token input to z causes a token to be emitted; hence, the two channels have the same minval.

- If z has type 2, both channels are full, with the blockwise channel pointing into z and other pointing out. Any value output by z has a strictly smaller index than a value waiting to be input to it, so the blockwise channel has the larger minval.

- If z has type 3, then both channels are outputs from z, and the blockwise channel is empty while the other is full. Because z does not filter, it always emits tokens with a given index on both channels at once. Hence, the minval of the blockwise channel is at least the index of the most recently emitted value on the other channel, which is $\geq$ the latter's minval.

- If z has type 4, then both channels are inputs to z, and the blockwise channel is full while the other is empty. The minval of the full channel must be strictly greater than that of the empty channel; otherwise, z could consume a value from the full channel.

Hence, the minvals of the channels in C increase monotonically to blockwise. Moreover, because there are no directed cycles in the original network, there is always a node of type 4 in C, and so the minvals of all channels in C cannot be identical. But this is impossible, because traversing the entire cycle implies that the minval of some channel is strictly greater than itself. Conclude that no blocking cycle can exist in the absence of filtering. □

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 sizes, there exist input streams and histories of filtering at each node that cause a deadlock.*

Definition 3.6 *(**Undirected Cycle**) Given a system abstracted as a DAMG G, an undirected cycle of G is a cycle in the undirected graph G' that is the same as G, except that all edge directions have been removed.*

For example, in the graph of Figure 3.2, $uvxw$ is an undirected cycle that can become blocking. We now show that in a DAMG, *every* undirected cycle can become blocking.

Theorem 3.3 (Potential Deadlock Theorem) *Given a system S abstracted as a DAMG G, S has a potential deadlock if and only if G has an undirected cycle.*

Proof. ($\rightarrow$) By definition, if S has a potential deadlock, then a deadlock can happen given the right pattern of inputs and filtering. By the Deadlock Theorem, such a deadlock implies the presence of a blocking cycle of nodes, which implies an undirected cycle of channels in G.

($\leftarrow$) Suppose that there is an undirected cycle C of channels in G. We will construct a set of tokens and a filtering history that causes C to become a blocking cycle, implying a deadlock.

First, we arbitrarily choose a direction on C to be the blockwise direction. We then topologically sort the nodes of the DAMG. We mark each node u and channel uv with M_u and M_{uv} values calculated as follows. For each node u, if u is a sink node, $M_u = 0$; otherwise, $M_u = \max_{uv} M_{uv}$, where uv is any outbound channel from u. For each outbound channel uv, if uv is a counterblockwise channel in C, $M_{uv} = M_v + |uv| + 1$; otherwise, $M_{uv} = M_v$.

The filtering history for each channel out of each node is as follows. Each input token consumed by a node u results in an output token (i.e. no filtering) on any output channel of u that is not on cycle C or is oriented counterblockwise on C. For an output channel uv that is oriented blockwise on C, u emits tokens on uv until its computation index reaches M_{uv}, then filters (i.e. emits no output on uv) for any larger index.

The above construction ensures that:

- For a blockwise channel uv in C, $u \dashv v$ because v will consume all M_{uv} inputs sent to it by u, leaving the channel empty.
- For a counterblockwise channel uv in C, $v \dashv u$ because u tries to send $|uv| + 1$ tokens to v after v becomes unable to consume tokens, and so uv becomes full and blocks further output by u.

Since each node in C now blocks its blockwise neighbor, it follows that for any node u in C, $u \dashv^{+} u$, which implies a deadlock. □

The above proof shows that given *enough* input tokens and*arbitrary* filtering behavior, *any* undirected cycle of G could cause a deadlock.

3.4 Summary

In this chapter, we described the synchronized filtering dataflow (SFDF) model, which has data-dependent filtering and synchronized consuming behavior. The filtering and synchronization behavior can cause SFDF applications to require unbounded memory and hence deadlock. We characterized deadlocks in the SFDF model and revealed that any undirected cycle could lead to deadlocks given enough input data and arbitrary filtering behaviors. In the next chapter, we will discuss algorithms to avoid deadlocks and guarantee bounded-memory execution of SFDF applications.

Chapter 4

Bounded-memory Execution of SFDF Applications

In this chapter, we discuss a general method and algorithms based on it to avoid deadlocks. The method uses a special token called a *dummy message*. During compile time, a dynamic schedule is computed for sending dummy messages; during runtime, dummy messages are scheduled according to the dynamic schedule and computations. The algorithms and results were previously published in [67] and [69].

4.1 Dummy Messages for Deadlock Avoidance

According to the Fitering Theorem in Chapter 3, if no node filters data, the system never deadlocks. Inspired by this fact, we may avoid deadlocks by mimicking non-filtering applications. To mimic non-filtering applications, we use *dummy messages* (or *dummy tokens*), which are a distinguished class of token with an index but no content of its own. A dummy message may be emitted as a standalone token, or it may be combined with a data token with the same index (as we will see later in the Propagation Algorithm). The purpose of dummy messages is to communicate a node's current computation index to its successors.

By sending a dummy message in place of every filtered data token (referred to hereafter as the *Naive Algorithm*), we can effectively avoid deadlocks with a trivial data-driven schedule: a node

is able to execute as long as all input channels have data. However, this approach is likely to send many unnecessary dummy messages. Real distributed systems have limited channel bandwidth, so that communication costs can become a bottleneck. For many applications, such as Mercury BLAST, the primary purpose of some nodes is to filter the data stream. Using this approach for such applications would negate the communication bandwidth savings achieved by their natural filtering. Hence, we next give algorithms that reduce the number of dummy messages while still ensuring that the resulting system is free from deadlock.

4.2 Limiting the Frequency of Dummy Messages

We now consider how to avoid emitting dummy messages for every data token filtered by a node. Our approach includes two parts. We first extend the behavior of each compute node u to include *propagation* of received dummy messages, as well as *generation* of dummy messages on each output channel q of u at a *statically* defined *dummy interval* $[q]$. If $[q] = \infty$, then u never generates new dummy messages on output q; otherwise, it is guaranteed to emit a dummy message each time its computation index advances by more than $[q]$. Using this extended behavior with the specified dummy intervals, we obtain a system that is deadlock-free yet sends many fewer dummy messages than the Naive Algorithm when some nodes filter their inputs.

Algorithm 4.1 describes how we extend the behavior of a computation node to include generation and propagation of dummy messages. Generator nodes are guaranteed to emit a dummy message on channel q whenever the computation index has advanced by more than $[q]$ since the last dummy message was sent, regardless of whether any data tokens has been sent. *All* nodes propagate any incoming dummy message to all their output channels, combining it if needed with any data token with the same index to be emitted on each channel. Hence, even with dummy messages, *no node ever emits two tokens with the same index on the same channel*. This approach is referred to as the "Propagation Algorithm."

Algorithm 4.1: Single-node behavior with propagation of dummy message.

```
1  ComputeIndex ← 0
2  foreach output channel q do
3      LastSentIdx_q ← 0
4  while ComputeIndex ≠ Index of EOS do
5      wait until every input channel has a pending token
6      let i be minimum index of any pending token
7      consume pending tokens with index i from input channels
8      compute on data tokens with index i
9      foreach output channel q do
10         if i − LastSentIdx_q > [q] OR some pending token with index i is a dummy message then
11             schedule a dummy message with index i for output q
12             LastSentIdx_q ← i
13     ComputeIndex ← i
14     emit output tokens with index i, including any scheduled dummy messages
```

Algorithm 4.2 computes dummy intervals $[q]$ for every channel q of an application graph G. In the algorithm description and subsequently, $|p|$ denotes the length of a directed path p, which is the sum of all channel buffer sizes on p. A *maximal* directed path is one that is not a proper prefix of a longer directed path.

Algorithm 4.2: Dummy interval calculation with propagation of dummy message.

Input: A system abstracted as graph $G = \{V, E\}$
Output: Dummy intervals for each channel

```
1  foreach edge uv ∈ E do [uv] ← ∞
2  foreach undirected cycle C of G do
3      foreach node u with two output channels uv_1, uw_1 on C do
4          let p_1 = uv_1 ... v_m be maximal directed path on C starting with uv_1
5          let p_2 = uw_1 ... w_n be maximal directed path on C starting with uw_1
6          [uv_1] ← min([uv_1], |p_2| − 1)
7          [uw_1] ← min([uw_1], |p_1| − 1)
```

The algorithm first finds all undirected cycles in G; then for each undirected cycle C and for each node $u \in C$ that has at least two outgoing channels uv and uw, suppose the maximal path beginning with uw (uv) is p_w (p_v), $[uv]$ ($[uw]$) is *less than* $|p_w|(|p_v|)$. Algorithm 4.2 iterates over all undirected cycles of the system, which may in general require time exponential in the system size; we will improve the algorithm to reduce the cost of calculating dummy intervals in future chapters. For each node with two output channels on the same undirected cycle, the algorithm

calculates a dummy interval for each channel that (as we will prove) is small enough to guarantee that the cycle can never become blocking. Channels that are not the *first channel on a directed path on some undirected cycle*, including those not on a cycle at all, receive intervals of ∞.

Theorem 4.1 *If all nodes behave according to Algorithm 4.1, using the intervals calculated by Algorithm 4.2, then the system is deadlock-free.*

Proof. Suppose not; that is, suppose that the system as constructed above experiences a deadlock. According to the Deadlock Theorem, the system must at some point contain a blocking cycle C. We will show by contradiction that C cannot exist.

Let C be given. Divide C into alternating maximal directed paths of blockwise and counterblockwise edges, as shown in Figure 4.1. Choose an arbitrary node with two output channels on C (as s_1 in Figure 4.1) and, proceeding to blockwise from this node, label these *paths* in blockwise order as $p_{e1}, p_{f1}, \ldots p_{ek}, p_{fk}$ ("e" means "empty" while "f" means "full"). By the Deadlock Theorem, each path p_{ei} consists entirely of empty channels, while each path p_{fi} consists entirely of full channels.

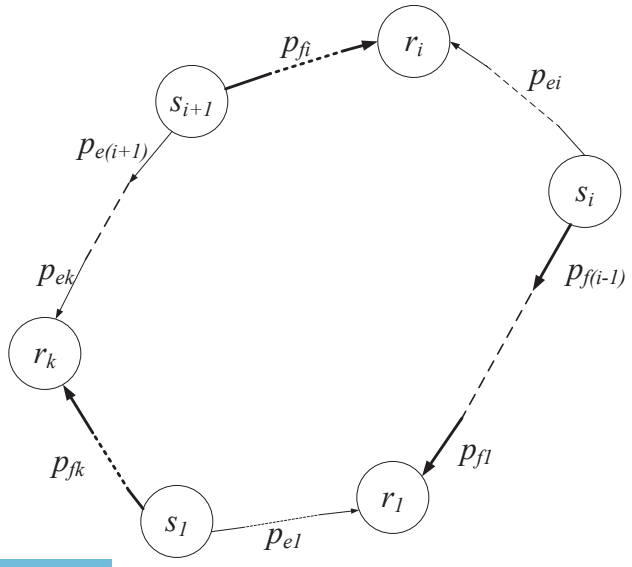

Figure 4.1: The division of a blocking cycle for Theorem 4.1.

For convenience, let $p_{f0} = p_{fk}$. Label each node between p_{ei} and p_{fi} as the *receiver* r_i, and label each node between $p_{f(i-1)}$ and p_{ei} as the *sender* s_i. Each sender node has two output channels on C, both of which receive finite dummy intervals according to Algorithm 4.2.

The key observation is that, given the rules for assigning dummy intervals, *node s_i cannot emit more than $|p_{f(i-1)}|$ tokens along path $p_{f(i-1)}$ without also sending a dummy message τ along path p_{ei}*. Because path $p_{f(i-1)}$ is entirely full, while path p_{ei} is entirely empty, the dummy message τ must have already been emitted by s_i and been propagated to receiver r_i by the time the blocking cycle C formed.

The rest of proof introduces the important concepts of *minval* and *maxval*, which will also be used in later proofs.

Definition 4.1 ***(minval** and **maxval**) For any full channel or path q, minval(q) is defined to be* lowest *index of any token queued on q, while maxval(q) is defined to be the* highest *such index. For an empty channel or path q', minval(q') is defined to be the index of the token that has most recently traversed q'.*

(Theorem 4.1's proof continues.) Algorithm 4.1 and Algorithm 4.2 above imply that

$$\text{minval}(p_{ei}) \geq \text{maxval}(p_{f(i-1)}) - |p_{f(i-1)}|. \tag{4.1}$$

Because each channel receives at most one token with a given index, we have that, since p_{fi} is full,

$$\text{maxval}(p_{f(i-1)}) - |p_{f(i-1)}| \geq \text{minval}(p_{f(i-1)}). \tag{4.2}$$

Finally, because the cycle C is a blocking cycle, r_i remains blocked by its counterblockwise neighbor even after receiving dummy message τ. Hence, we have that

$$\text{minval}(p_{fi}) > \text{minval}(p_{ei}). \tag{4.3}$$

Combining these three inequalities for a given i yields $\text{minval}(p_{fi}) > \text{minval}(p_{f(i-1)})$. But this inequality holds for every i, and so we have transitively that $\text{minval}(p_{fk}) > \text{minval}(p_{f0})$, which is impossible because these two paths are the same. Hence, blocking cycle C cannot exist, and so deadlock is impossible. □

We note that in Algorithm 4.1, one cannot suppress a dummy message on a channel q even if a data token has been sent within the last $[q]$ indices. Suppose that a data token was sent along p_{ei} from node s_i in the proof above; it could be filtered by any node on p_{ei} before reaching r_i, thereby invalidating Inequality 4.3. Similarly, one cannot permit both a data token and a dummy message with the same index to be sent separately, as doing so would invalidate Inequality 4.2.

This scheme for deadlock avoidance can greatly reduce the frequency of dummy messages on some channels in a system. In particular, a source node with two output channels q_1 and q_2 that emits a series of n tokens only on q_1 would have to emit n dummy messages under the Naive Algorithm but only about $n/[q_2]$ tokens with the revised approach. Unfortunately, propagation of dummy messages ensures that a node receives all tokens (with distinct indices) emitted by any of its ancestors, even if the node is not on any of the cycles that required emitting the dummy messages in the first place! Hence, nodes with many ancestors that participate in undirected cycles may be flooded with useless dummy messages.

4.3 Eliminating Propagation of Dummy Message

In this section, we propose another deadlock avoidance scheme that uses a method similar to the Propagation Algorithm to assign dummy intervals to output channels. The key difference between the new scheme and the Propagation Algorithm is that dummy messages no longer propagate. Since propagation is not required, we no longer need to send a dummy message if we can send a data token with the same index; rather, the behavior at each node ensures only that *some* token is sent on channel q at least once each time the computation index increases by more than $[q]$. By increasing the frequency of dummy message generation on some channels, we can guarantee

freedom from deadlock without the need for propagation of dummy message. Hence, this approach is referred to as the "Non-Propagation Algorithm."

Algorithm 4.3 describes node behavior in which dummy messages are *never* propagated beyond the channel on which they first appear, while Algorithm 4.4 gives a revised procedure to assign dummy intervals to channels. To avoid propagation, the new dummy interval computation assigns finite dummy intervals to *all* channels on the directed paths found by the previous algorithm, rather than just the first node. The assigned intervals are smaller than before for paths with two or more channels. As in the previous section, this algorithm may take exponential time in the graph size, but it executes at compilation time and has no effect on the runtime of a computation.

Algorithm 4.3: Single-node behavior without propagation of dummy message.

```
1  ComputeIndex ← 0
2  foreach output channel q do
3      LastSentIdx_q ← 0
4  while ComputeIndex ≠ Index of EOS do
5      wait until every input channel has a pending token
6      let i be minimum index of any pending token
7      consume pending tokens with index i from input channels
8      ComputeIndex ← i
9      compute on data tokens with index i
10     foreach output channel q do
11         if a data token with index i will be emitted on q then
12             schedule a token with index i for output q
13             LastSentIdx_q ← i
14         else if i − LastSentIdx_q > [q] then
15             schedule a dummy message with index i for output q
16             LastSentIdx_q ← i
17     emit output tokens with index i, including any dummy messages
```

Theorem 4.2 *If all nodes behave according to Algorithm 4.3, using the intervals calculated by Algorithm 4.4, then the system cannot deadlock.*

Proof. As before, suppose that a blocking cycle C occurs in a system using this deadlock avoidance scheme. Divide cycle C into paths, senders, and receivers as before. Label the nodes on path p_{ei} $v_0, \ldots v_n$, with $v_0 = s_i$ and $v_n = r_i$.

Algorithm 4.4: Dummy interval calculation without propagation of dummy message.

Input: A system abstracted as graph $G = \{V, E\}$
Output: Dummy intervals for each channel

```
1  foreach edge uv ∈ E do [uv] ← ∞
2  foreach undirected cycle C of G do
3      foreach node u with two output channels uv_1, uw_1 on C do
4          let p_1 = uv_1 ... v_m be maximal directed path on C starting with uv_1
5          let p_2 = uw_1 ... w_n be maximal directed path on C starting with uw_1
6          [uv_1] ← min([uv_1], ⌊(|p_2| − 1)/m⌋)
7          for i in 2 ... m do
8              [v_{i−1}v_i] ← min([v_{i−1}v_i], ⌊(|p_2| − 1)/m⌋)
9          [uw_1] ← min([uw_1], ⌊(|p_1| − 1)/n⌋)
10         for i in 2 ... n do
11             [w_{i−1}w_i] ← min([w_{i−1}w_i], ⌊(|p_1| − 1)/n⌋)
```

Let $\gamma = \lfloor(|p_{f(i-1)} - 1)|/n\rfloor$, the dummy interval defined for the channels on p_{ei} by Algorithm 4.4.

We first prove that if r_i has received no token with index minval$(p_{f(i-1)})$, then the last token received by node v_j of p_{ei} must have index at most minval$(p_{f(i-1)}) - 1 + \gamma \cdot (n - j)$. The proof is by induction on i in decreasing order. In the base case, when $j = n$, the theorem is trivially true, since $v_n = r_i$.

For the inductive step, by the inductive hypothesis, the last token received by v_{j+1} had index at most $M_{j+1} =$ minval$(p_{f(i-1)}) - 1 + \gamma \cdot (n - j - 1)$, and so v_j's last token sent to v_{j+1} had index at most M_{j+1}. Now suppose that v_j has received a token with an index, say M', greater than

$$M_j = \text{minval}(p_{f(i-1)}) - 1 + \gamma \cdot (n - j).$$

We have that $M_j - M_{j+1} = \gamma$, and so $M' - M_{j+1} > \gamma$, which means the interval between v_j's last received and last sent tokens is greater than γ. Algorithms 4.3 and 4.4 therefore ensure that v_j must have sent a token, either data or dummy, to v_{j+1} with index $> M_{j+1}$. But this contradicts our IH. Thus, we conclude that the last token received by v_j has index at most M_j, as desired.

Next, we observe a special case of the fact proved above: if r_i has not received a token with index at least minval$(p_{f(i-1)})$, then s_i's most recently received token has some index t, where

$$\begin{aligned} t &\leq \text{minval}(p_{f(i-1)}) - 1 + \gamma \cdot n \\ &< \text{minval}(p_{f(i-1)}) + |p_{f(i-1)}| \\ &\leq \text{maxval}(p_{f(i-1)}). \end{aligned}$$

But this is impossible because s_i has already emitted a token with index maxval$(p_{f(i-1)})$, so it must have received such a token.

Conclude that minval$(p_{ei}) \geq$ minval$(p_{f(i-1)})$. As in Theorem 4.1, we also have minval$(p_{fi}) >$ minval(p_{ei}) because cycle C is blocking, and so a contradiction follows using the cycle-following argument of that theorem. Hence, blocking cycle C cannot exist, and no deadlock occurs. □

4.4 Comparison of Algorithms

The runtime protocols for dummy message scheduling are fairly simple, causing very little computational overhead, so our comparisons pay attention to the number of dummy messages generated and sent by nodes across communications links.

4.4.1 A Paper-and-pencil Comparison

Since the Naive Algorithm does not take advantage of channel buffers, it always sends more dummy messages than the other two algorithms. However, the Propagation and the Non-Propagation algorithms are incomparable; each may outperform the other based on the graph topology and buffer sizes.

In most cases, we expect the Non-Propagation Algorithm to perform better. The Propagation Algorithm has two inherent disadvantages over the Non-Propagation Algorithm. First, it sends

dummy messages at specific intervals regardless of whether the node is actually filtering any inputs. (Here we assume that the nodes are capable of filtering data, they just happen not to for that particular set of inputs.) Therefore, if no node ever filters any data, the Propagation Algorithm will still send dummy messages, while the Non-Propagation Algorithm will never send any dummy messages. Second, all downstream nodes propagate the dummy messages they receive. Therefore, in some cases, the dummy messages will be sent downstream even if they are no longer required. Due to these reasons, in most cases, we expect the Non-Propagation Algorithm to be more efficient in terms of the number of dummy messages sent.

However, there are in theory circumstances for which the Propagation Algorithm will generate fewer dummy messages. This situation arises when the nodes filter a very large number of tokens, sending virtually no *data tokens* on some channels. Consider the following case. Let $u_1u_2 \ldots u_{k+1}$ be some maximal path on an undirected cycle, and let the dummy interval for u_1u_2 be $[u_1u_2]_p$ in the Propagation Algorithm. When the computation index increases from 0 to m at node u_1, in case of the Propagation Algorithm, u_1 will send $\lfloor m/[u_1u_2]_p \rfloor$ dummy messages, which are then propagated by u_i $(1 < i \leq k)$, so the total number of dummy messages is $k \times \lfloor m/[u_1u_2]_p \rfloor$. According to Algorithm 4.4, in the Non-Propagation Algorithm, the dummy interval for every channel on this path is at most $\lfloor [u_1u_2]_p/k \rfloor$. If all tokens are filtered by u_1, the total number of dummy messages sent by the Non-Propagation algorithm is about $k \times (m/\lfloor [u_1u_2]_p/k \rfloor)$, which is about k times larger than the number of messages sent by the Propagation Algorithm.

4.4.2 Experimental Evaluation

We evaluated the overhead associated with the deadlock avoidance algorithms on two applications: Mercury BLAST and a pseudo-random number generator (PRNG) that is a part of the application of financial Monte-Carlo simulation.

The number of dummy messages sent by the Naive Algorithm equals the quantity of filtered data, no matter the buffer size. The number of dummy messages generated by the Propagation

Algorithm is decided by buffer sizes, independent of the runtime filtering ratio. Hence the number of dummy messages sent by the Propagation Algorithm can be statically computed given the input volume and system topology. The number of dummy messages sent by the Non-Propagation Algorithm is decided by the runtime filtering trace and cannot be statically calculated.

Mercury BLAST

To acquire the number of dummy messages sent in Mercury BLAST, we ran Mercury BLASTN to search a set of 1000 queries sampled from human messenger RNA (mRNA) sequences against all other vertebrate mRNA as the database. This represents 787 billion input tokens. We monitored the number of dummy messages out of module 1a. We ran the Non-Propagation Algorithm and used a hardware monitor (described in [62]) to count the actual dummy messages. The number of dummy messages generated by the Naive Algorithm can be estimated by multiplying the input data volume and the filtering rate, while dummy messages generated by the Propagation Algorithm can be calculated by dividing the number of input tokens by the dummy interval, which is a fixed value. We set the buffer size of the database channel to 32, 256, and 2048. The corresponding dummy intervals are 32, 256, and 2048 for the Propagation Algorithm and 16, 128, and 1024 for the Non-Propagation Algorithm. Our results (shown in Figure 4.2 and Table 4.1) indicate that the Non-Propagation Algorithm has, by far, the smallest message overhead.

Table 4.1: Measured dummy message counts from module 1a for Mercury BLASTN

	Dummy message count		
Total Buffer Size (msgs)	32	256	2048
Naive Algorithm	787×10^9	787×10^9	787×10^9
Propagation Algorithm	25×10^9	3×10^9	0.4×10^9
Non-Propagation	36×10^9	36×10^6	$72,000$

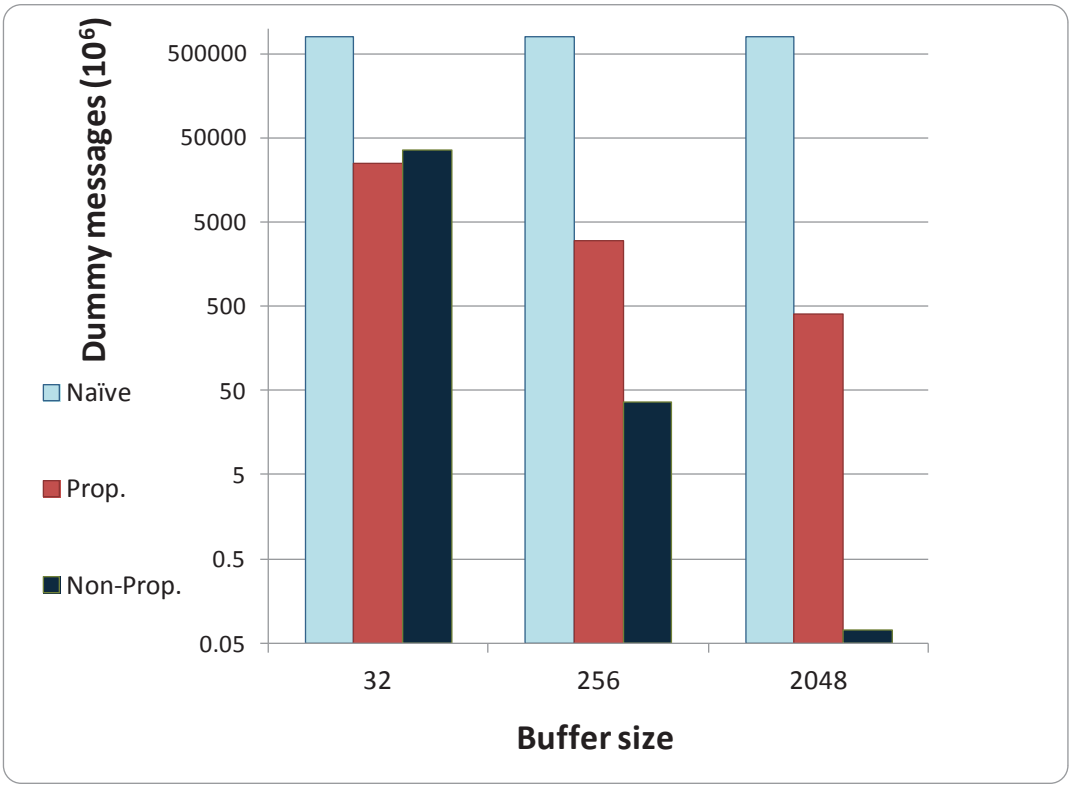

Figure 4.2: Dummy message counts for Mercury BLASTN.

Pseudo-random Number Generator

Pseudorandom number generators (PRNGs) are widely used in applications, such as Monte Carlo simulation [104], 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 [73] and the ziggurat algorithm [74], 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 the pipelined application topology shown in Figure 1.7. 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 1.7'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 speed up the bottleneck, we may replicate the filter f, as Figure 1.8 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 1.7, t must implement some form of predictable synchronization over all filters.

To assess the performance impact of our algorithms on PRNGs, we simulated the Marsaglia polar method, which has a rejection rate of 21.46%. We chose this method rather than the ziggurat algorithm because the former has a higher filtering ratio, which means it is more vulnerable to deadlocks. We replicated four filters between the source and the sink. In three different runs, the total buffer size of each path was set to 10, 100, and 1,000, which determines the total number of elements, including tokens 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 applied three deadlock avoidance algorithms and counted the total number of dummy messages each of them generates.

The results are shown in Table 4.2. In the Propagation 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 those generated by the source. From the data in Table 4.2, the Non-Propagation 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. In Table 4.3 and Table 4.4, we show results for the same experiment except filtering ratios were set to 95% and 5%, respectively. Even with a high filtering ratio (95%) and small channel buffer size (only buffering 10 tokens), the communication overhead is less than 10%. In low filtering ratio (5%) or large buffer size (1000 tokens) cases, the overhead is negligible.

Table 4.2: Simulation results for Marsaglia polar algorithm (filtering ratio = 21.46%).

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

Table 4.3: Simulation results for 4 replicated filters and filtering ratio = 95%.

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

Table 4.4: Simulation results for 4 replicated filters and filtering ratio = 5%.

	Dummy message count		
Total Buffer Size (msgs)	10	100	1000
Naive Algorithm	50,172	50,172	50,172
Propagation Algorithm	200,000	20,000	2,000
Non-Propagation Algorithm	0	0	0

4.5 Summary

In this chapter, we proposed three algorithms to avoid deadlocks in SFDF systems to ensure bounded-memory execution. Our algorithms rely on sending extra tokens called "dummy messages." The three algorithms differ in the scheduling of dummy messages. The Naive Algorithm generates a dummy message for every filtered data token. The Propagation Algorithm schedules dummy messages according to precomputed intervals. Once a dummy message is generated, it needs to be propagated to the sink node. The Non-Propagation Algorithm also schedules dummy messages according to dummy intervals, but both the computation of dummy intervals and scheduling of dummy messages are different from the Propagation Algorithm, and dummy messages are not required to be propagated beyond their direct receivers. Experimental results show that the Non-Propagation Algorithm generates the fewest dummy messages.

In order to compute dummy intervals, we need to enumerate all undirected cycles, which can be time-consuming on some DAGs because of the number of undirected cycles can be exponential in the graph size. In the next chapter, we will propose algorithms to reduce the time complexity of computing dummy intervals on DAGs with special structure.

Chapter 5

Efficient Deadlock Avoidance for Applications with Structured Topologies

In the previous chapter, we designed algorithms to avoid deadlocks (or ensure bounded-memory execution) for SFDF applications. The basic strategy is that application nodes send dummy messages at pre-defined intervals, which are computed at compile time for the whole application. Depending on whether dummy messages should be propagated, we gave two algorithms: the Propagation Algorithm and the Non-Propagation Algorithm. In both algorithms, we sought to choose maximal dummy intervals to minimize the total number of dummies sent. Unfortunately, maximizing dummy intervals is challenging. In Chapter 4, our algorithms for computing a safe set of such intervals run in worst-case time exponential in the size of the application's topology, raising the question of whether deadlock-free filtering can be implemented efficiently as part of compiling a streaming application.

In this chapter, we show that for a large class of intuitive and useful DAG topologies, dummy intervals can be computed efficiently. We first present a new method where each dummy message is tagged with a destination, so as to reduce the number of dummy messages sent over the network in the Propagation Algorithm. We then give efficient algorithms for dummy interval computation in series-parallel DAGs [112]. We finally generalize our results to a larger graph family, the *CS4 DAGs*, in which every undirected Cycle is Single-Source and Single-Sink (CS^4). The results in this chapter have previously been published in [18].

5.1 Destination-Tagged Propagation Algorithm

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

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

Because each source can generate dummy messages for multiple destinations, each channel can have more than one dummy interval associated with it. Formally, we represent the *dummy message schedule* of a channel q as a set $[q] = \{d_1, d_2, ..., d_k\}$, where each pair $d_i = (\tau_i, z_i)$ is a *dummy interval-destination pair*. τ_i represents an interval at which a dummy message must be sent, while z_i represents its destination node. In addition, each dummy message pair d_i has a counter c_i associated with it, and the maximum value of the counter is τ_i. A source node uses the dummy message schedule and the counters to decide when to send dummy messages along q. In Sections 5.2 and 5.4, we show how to efficiently compute the dummy message schedules for SP-DAGs and CS4 DAGs respectively, and also how nodes must behave at runtime in order to correctly propagate tagged dummy messages.

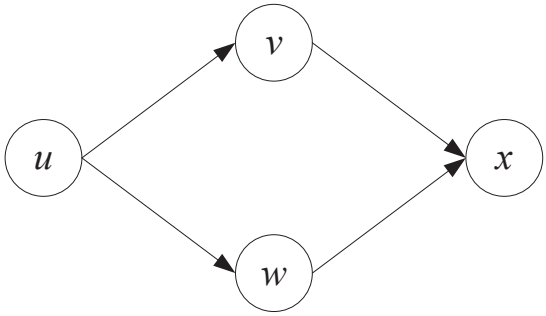

Figure 5.1: A simple split/join streaming topology.

5.2 Efficient Deadlock Avoidance for SP-DAGs

Series-parallel (SP) DAGs, which were defined by Valdes et al. [112], intuitively describe a large class of natural streaming topologies that can be built up recursively via pipelining and parallel splits and joins.

Definition 5.1 (Series-parallel DAG) *A series-parallel DAG (SP-DAG) is a connected, directed acyclic multigraph with two distinguished terminals, a source and a sink. The set of all SP-DAGs is defined recursively as follows:*

***Base**: a source and sink connected by any non-zero multiplicity of edges*[6] *is an SP-DAG.*

***Ind. 1** (Serial composition, Sc): if H_1 and H_2 are SP-DAGs, connecting them by merging the sink of H_1 and the source of H_2 yields an SP-DAG $Sc(H_1, H_2)$.*

***Ind. 2** (Parallel composition, Pc): if H_1 and H_2 are SP-DAGs, connecting them by merging the sources of H_1 and H_2, and the sinks of H_1 and H_2, yields an SP-DAG $Pc(H_1, H_2)$.*

For example, in Figure 5.1, each of the four edges uv, vx, uw, and wx is a base-case SP-DAG; we have $uvx = Sc(uv, vx)$, $uwx = Sc(uw, wx)$, and $uvwx = Pc(uvx, uwx)$. We sometimes refer to subgraphs H_1 and H_2 in the composition operations as *components* of the composed graph.

[6] Since this chapter involves intensive graph theory, we will use the term *edge* more frequently than *channel*.

5.2.1 SP-DAG Preliminaries

The next few lemmas elucidate the undirected cycle structure of SP-DAGs, which we will exploit later to build efficient deadlock avoidance algorithms. In particular, we use the property, verified in Lemma 5.5, that every undirected cycle on an SP-DAG has a single source and a single sink. We also use the hierarchical decomposition structure of SP-DAGs to efficiently compute dummy message schedules.

Definition 5.2 *(**Dominator** and **Postdominator**) Given two nodes u and v in a DAG, if any directed path from a source node to v goes through u, we say u is a* dominator *of v; if any directed path from u to a sink node goes through v, we say v is a* postdominator *of u.*

Fact 5.1 *In an SP-DAG, every node has an immediate postdominator (follows trivially from single-sink property).*

Lemma 5.2 *In an SP-DAG G, let x be a node with at least two outgoing edges. Let y be the immediate postdominator of x. Then for any directed path p from x to y, x dominates all nodes of p other than y.*

Proof. By induction on the structure of G.

Base: in an SP-DAG with a single multi-edge, p is a single edge from x to y. x trivially dominates itself.

Ind.: Otherwise, G is either $Sc(H_1, H_2)$ or $Pc(H_1, H_2)$ for SP-DAGs H_1, H_2. If x is the source of G, then x trivially dominates all of G, since SP-DAGs have a single source. x cannot be the sink of G since the sink has no outgoing edges.

Now x lies either in $H_1 - H_2$ or in $H_2 - H_1$, or $G = Sc(H_1, H_2)$ and x is the sink of H_1 and the source of H_2. If x is in $H_1 - H_2$, then H_1's sink always postdominates x, so y, the immediate postdominator of x, is a node in H_1. Applying the IH to subgraph H_1, the Lemma holds for x and

y. Analogous reasoning holds if x is in $H_2 - H_1$. Finally, if x is the source of H_2 and the sink of H_1, then y is in H_2 and x dominates all of H_2. □

Lemma 5.3 *Let $G = Pc(H_1, H_2)$ be an SP-DAG, where s is its source and t is its sink. Let x be a node of $H_1 - \{s, t\}$ that has at least two outgoing edges e and e' in G. Let C be an undirected simple cycle that contains both e and e'. Then C contains no edge $e'' \in H_2$.*

Proof. Suppose not. WLOG, let the counterexample simple cycle C leave x via edge $e = x \to u$ and return via edge $e' = x \to v$. Since C passes through an edge in H_2, it must also pass through both s and t, since those are the only two nodes that connect H_1 and H_2. So there must be two vertex-disjoint undirected paths in H_1: p_1 goes from x to u to t, and p_2 (entirely in H_1) goes from x to v to s.

Let y be the immediate postdominator of x, which lies in H_1. We claim that both paths p_1 and p_2 must pass through y.

Suppose path p_1 does not pass through y. Now U is a predecessor of y, while t is not, so there is some first edge in p_1 that connects a predecessor A of y to a non-predecessor B. We have two cases.

1. If the edge is oriented $A \to B$, then there is a directed path from x to A to B to t that bypasses y, which contradicts y's postdomination of x.

2. If the edge is oriented $B \to A$, then B is not a successor of y, since G is acyclic. There is then a directed path from s to B to A that bypasses x, which contracts x's domination of A by Lemma 5.2.

Conclude that p_1 must indeed pass through y.

Suppose p_2 does not pass through y. Now v is a successor of x, while s is not; hence, there is some first edge on path p_2 that connects a successor A of x to a non-successor B. This edge must be oriented $B \to A$, else B would be a successor of x.

Now A cannot be a predecessor of y; otherwise, there would be a directed path from s to B to A that bypasses x, contradicting x's dominance of A by Lemma 5.2. Hence, A is a successor of y. The subpath of p_2 from V to A therefore contains some first edge connecting a predecessor C of y to a successor D of y. This edge must be oriented $C \to D$, since G is acyclic. But then there is a directed path from x to C to D to t that bypasses y, which contradicts y's postdomination of x. Conclude that p_2 must indeed pass through y.

Since p_1 and p_2 both contain y, they are not vertex disjoint, leading us to a contradiction. □

Lemma 5.4 *For an SP-DAG $G = Pc(H_1, H_2)$, any undirected simple cycle C in G that has edges in both H_1 and H_2 consists of a pair of directed paths p_1 through H_1 and p_2 through H_2 that connect the source s of G to its sink t.*

Proof. We know from Lemma 5.3 that undirected simple cycles in G that traverse edges of both H_1 and H_2 do not pass through two outgoing edges of any node other than s. Moreover, each such cycle passes through two incoming edges of node t, since t does not have any outgoing edges.

Let p_1 be the directed path on C that exits s in (WLOG) H_1. If this path were to terminate at some node x prior to t, then the portion of cycle following p_1 would traverse two adjacent incoming edges of x. But if the cycle leaves x via an edge that points into x and eventually reaches t via an edge that points into t, it must at some point "change direction" by passing through two outgoing edges of a node w other than s, which is impossible by Lemma 5.3.

Conclude that C must be fully directed from s to t in both components. □

Lemma 5.5 *Each undirected simple cycle in an SP-DAG G has a single source and a single sink.*

Proof. By induction on the structure of G.

Base: Trivially true for a single multi-edge.

Ind.: If $G = Sc(H_1, H_2)$, then the property holds for H_1 and H_2, and their serial composition creates no new cycles. Hence the property holds for every cycle of G.

If $G = Pc(H_1, H_2)$, then every new cycle created by their parallel composition connects the common source s of G to its common sink t by directed paths passing through H_1 and H_2, respectively. All such cycles therefore have one source s and one sink t. □

Lemma 5.6 *If s is the source for two components with sinks t and x, and these components share a common edge, then either t is a successor of x in G or vice versa.*

Proof. Suppose not, then st and sx are subgraphs of two parallel components, which share the common source node s. According to Lemma5.4, any undirected cycle connecting t and x must go through s. However, since st and sx share a common edge, say uv, so there exists an undirected cycle containing both t to x with v as the source and some common postdominator as the sink, which contradicts Lemma 5.4. □

We now show that restricting filtering application topologies to SP-DAGs permits efficient implementations of both the Destination-Tagged Propagation Algorithm and the Non-Propagation Algorithm for deadlock avoidance. We next describe how to compute dummy schedules for both avoidance algorithms in small polynomial time.

5.2.2 The Destination-Tagged Propagation Algorithm for SP-DAGs

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

In our approach, the source node of each component H of an SP-DAG is responsible for preventing deadlock on undirected cycles of H that cross more than one of its sub-components. Since a node can be a source for multiple distinct components, it may need to send dummy messages that target multiple sinks. Therefore, a channel q from source u has a dummy message schedule $[q] = \{d_1, d_2, ..., d_k\}$, where in each pair $p_i = (\tau_i, z_i)$, z_i is a sink of some component for which

u is the source. τ_i is the interval at which a dummy message must be sent to sink z_i. We keep this list of pairs sorted by τ_i. In addition, for each edge, we have at most one pair for a particular destination.

Computing Dummy Message Schedules

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

1. We first recursively decompose G according to the construction rules for SP-DAGs, using e.g. the linear-time recognition algorithm of Valdes, Tarjan, and Lawler [112]. The decomposition results in a tree T whose leaves are single (multi-)edge graphs and whose internal nodes are labeled with the composition operators *Sc* or *Pc*, such that applying the composition operations in post-order results in graph G. The size of this tree is $O(|G|)$.

2. For every component H of G, we compute $L(H)$, which is the length of a shortest directed path (with buffer lengths as edge weights) from the source of H to its sink. This calculation can be done bottom-up on the tree T in $O(|G|)$ time.

3. We then compute schedules for all edges in total time $O(|G|^2)$ as follows.

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

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

Case 2: Say $H = Sc(H_1, H_2)$. Since H_1 and H_2 are joined by a single articulation point, their composition creates no new simple cycles. The schedules for edges in H_1 and H_2 do not change.

Case 3: Say $H = Pc(H_1, H_2)$, where s is H's source and t is H's sink. Now we add new pairs for each edge e out of s in H_1 as follows:

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

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

$$[e'] \leftarrow [e'] \cup \{(L(H_1), t)\}.$$

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

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

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

Lemma 5.7 *In any edge's dummy schedule $[e]$, there is at most one dummy interval per destination, and the dummy messages are sorted by increasing τ.*

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

Runtime Node Behavior

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

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

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

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

Proof of Freedom from Deadlock

We now argue that the Destination-Tagged Propagation Algorithm ensures freedom from deadlock for SP-DAGs. As noted by Theorem 3.1 in Section 3.3, deadlock can arise in a DAG G only through the creation of a blocking cycle. Since SP-DAGs have exactly one source and one sink on each cycle, a blocking cycle consists of one path from the source to the sink with full buffers and another path from the source to the sink with empty buffers.

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

Lemma 5.8 *Let H be a component of G with source s and sink t. If s propagates an incoming dummy message, then that message will reach t.*

Proof. A dummy message arriving at s was generated by the source of some super-component H' of H with sink x. By the properties of SP-DAGs, x must be either t or a successor of t. In either case, all paths from s to x lead through t, so t will eventually receive the dummy message. □

Lemma 5.9 *If an edge's schedule includes pairs $d_i = (\tau_i, z_i)$ and $d_j = (\tau_j, z_j)$, and $\tau_i < \tau_j$, then z_j is a successor of z_i.*

Proof. Step 1 of postprocessing ensures that $z_i \neq z_j$. By Lemma 5.6, one of these nodes is a successor of the other. If z_i were a successor of z_j, then step 2 of postprocessing would have removed d_j. □

Lemma 5.10 *Suppose that, for edge e out of node s, pair $(\tau_i, z_i) \in [e]$. For each τ_i tokens that s receives, it sends at least one dummy message along e that will reach z_i.*

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

1. If one of the tokens is a dummy that does not target s, then by Lemma 5.8, the dummy will reach z_i.

2. If all the tokens either are non-dummies or target s, then either counter c_i will increase until it reaches τ_i, triggering a dummy message to z_i, or some other counter c_j, $j > i$, will reach τ_j, triggering a dummy message to z_j. By Lemma 5.9, we know that z_j is a successor of z_i, and so this message will pass through z_i.

□

Lemma 5.11 *Consider a parallel component $H = Pc(H_1, H_2)$ with source s and sink t. Let $L(H_1)$ be the length of a shortest path from s to t through H_1. Consider any edge $e \in H_2$ that starts at s. In any time period during which s receives $L(H_1)$ tokens, it sends (or forwards) at least one dummy message on e with destination either t or a successor of t.*

Proof. When the schedule-setting algorithm first processes H, it adds the pair $(L(H_1), t)$ to $[e]$. Postprocessing will remove this pair only if s is also scheduled to send a more frequent dummy message to t or to one of its successors. Hence, Lemma 5.10 guarantees that s will send at least one dummy message along e that reaches t for each $L(H_1)$ tokens it receives. □

Theorem 5.12 *If dummy messages are sent as described in Section 5.2.2, using the interval-destination pairs computed as described in Section 5.2.2, then deadlock cannot occur in G.*

Proof. Suppose a deadlock does occur in G. Then there must be a blocking cycle C in G. Since G is an SP-DAG, C lies in some smallest parallel component H and consists of two directed paths p_1 and p_2 joining H's source s to its sink t.

Suppose WLOG that p_1 is full and p_2 is empty. We can decompose H into parallel sub-components H_1 and H_2 such that $p_1 \subseteq H_1$ and $p_2 \subseteq H_2$. By construction, the total length of all edges' buffers along path p_1 is $\geq L(H_1)$, while that along p_2 is $\geq L(H_2)$.

Now consider the first edge e on path p_2, which leaves source s. This edge lies in component H_2. For p_1 to fill, s must have received and passed on at least $L(H_1)$ tokens. But then Lemma 5.11 guarantees that s has sent a dummy message along e within its last $L(H_1)$ received messages. This dummy will eventually propagate to t, where it will allow t to consume at least one of the buffered tokens from p_1. Since p_1 remains full, we conclude that the dummy must still be somewhere on path p_2, and so p_2 cannot be empty. This contradicts our assumption that cycle C is blocking. □

5.2.3 The Non-Propagation Algorithm for SP-DAGs

We now show how to efficiently calculate dummy intervals for the Non-Propagation Algorithm when the graph topology is restricted to be an SP-DAG. The approach is broadly similar to that for the Destination-Tagged Propagation Algorithm, except that the schedule $[e]$ for an edge e now consists of only a single pair whose destination is the node at the end of the edge. For this section, we therefore adopt the convention, as in Chapter 4, that $[e]$ is a single number, the dummy interval for e. In addition, *all* nodes, not just sources, may generate dummy messages on their outgoing edges.

Dummy interval calculation

Our algorithm for dummy interval computation is as follows.

1. Decompose the graph into a tree of components.

2. Compute $L(H)$ for each component H, where $L(H)$ is the shortest path from H's source to H's sink, with buffer lengths as edge weights.

3. Compute $h(H)$ for each component H, where $h(H)$ is the longest path (in terms of the number of hops) from the source of H to its sink.

   - For a single multi-edge, $h(H) = 1$.
   - If $H = Sc(H_1, H_2)$, $h(H) = h(H_1) + h(H_2)$.
   - If $H = Pc(H_1, H_2)$, $h(H) = \max(h(H_1), h(H_2))$.

4. Compute $h(H, e)$ for each edge $e \in H$, where $h(H)$ is the longest path (in terms of the number of hops) from the source of H to its sink that passes through e. For a single multi-edge, $h(H, e) = 1$. For a series composition, for all $e \in H_1$, $h(H, e) = h(H_1, e) + h(H_2)$. Similarly for $e \in H_2$, $h(H, e) = h(H_2, e) + h(H_1)$. For parallel composition, if $e \in H_2$,

$h(H, e) = h(H_1, e)$. Similarly for $e \in H_2$. All these computations can be done in $O(|G|^2)$ time.

5. Compute the dummy interval $[e]$ for each edge e in a bottom-up fashion.

The first four steps in the above procedure are straightforward. For the fifth step, we visit the components of T in post-order. When considering component H, we update $[e]$ for all the edges in H considering only cycles internal to H.

Case 1: If H is a multi-edge from $s \to t$, let e be an edge from s to t. If we consider only cycles internal to H, $L(H, e)$ is the minimum buffer size over all edges other than e between s and t, and $h(H, e) = 1$. Therefore, the calculation in this case is identical to the that for the Dummy-Tagged Propagation Algorithm.

Case 2: If $H = Sc(H_1, H_2)$, serial composition introduces no new simple cycles through e, so $[e]$ is unchanged.

Case 3: If $H = Pc(H_1, H_2)$, suppose WLOG that e is in H_1. Let s be the source of H, and let t be its sink. Every new cycle created by the parallel composition consists of two confluent paths from s to t, one in each of H_1 and H_2. Let C be the newly created cycle that traverses a longest (in hop count) directed path in H_1 that includes e and returns via a shortest (in buffer length) path in H_2. Then the ratio $L(C, e)/h(C, e)$ for C is minimum among all new cycles created by the composition. Since, $L(C, e) = L(H_2)$ and $h(C, e) = h(H_1, e)$, we have $[e] = \min([e], L(H_2)/h(H_1, e))$. The symmetric computation applies if e is in H_2.

Each case above takes constant time per edge in the component H, or $O(|G|)$ time per component. Conclude that the entire tree traversal is $O(|G|^2)$.

Runtime node behavior and correctness

We previously described the runtime behavior of nodes for the Non-Propagation Algorithm in a general graph in Section 4.3. Briefly, a node sends a dummy message along an edge e if it filters

$[e]$ continuous tokens on edge e. This behavior applies unchanged to SP-DAGs. The dummy intervals $[e]$ of the previous section minimize a ratio between the length of a component-dependent shortest path and the number of hops in an edge-dependent longest path, as for the computation we previously gave for general graphs. Correctness for SP-DAGs therefore follows by the proof given for the algorithm on general graphs, as described in Section 4.3.

5.3 CS4 DAGs: a Larger Set of Simple Streaming Topologies

We have shown how to efficiently prevent deadlock in SP-DAGs, a large, practically useful class of DAG topologies that can be constructed with simple composition operations. A natural question at this point is, do there exist "natural" topologies that are not SP-DAGs? Might these topologies also have efficient algorithms for deadlock avoidance?

Figure 5.2 shows two simple two-terminal DAGs that are not SP-DAGs. The topology on the left augments a trivial split/join with a one-way communication channel linking its two sides; it is perhaps the simplest DAG that is not series-parallel. The topology on the right adds slightly more complexity, creating a "butterfly" structure like that commonly used to decompose large FFT computations. A key feature distinguishing the two graphs is that, in the left-hand example, every undirected simple cycle has only one source and one sink. This property is true for SP-DAGs, and we exploited it implicitly in the algorithms of the previous section. On the other hand, the butterfly graph contains a cycle *wyxzw* with two sources and two sinks.

In this section, we characterize the set of all DAGs whose undirected cycles each contain one source and one sink. The next section shows that all such DAGs are amenable to efficient deadlock avoidance using generalizations of our algorithms from Sections 5.2.2 and 5.2.3.

Definition 5.3 *Let G be a DAG with a single source and sink. We say that G is "CS4" if every undirected simple **C**ycle in G has a **S**ingle **S**ource and a **S**ingle **S**ink (for short, CS^4).*

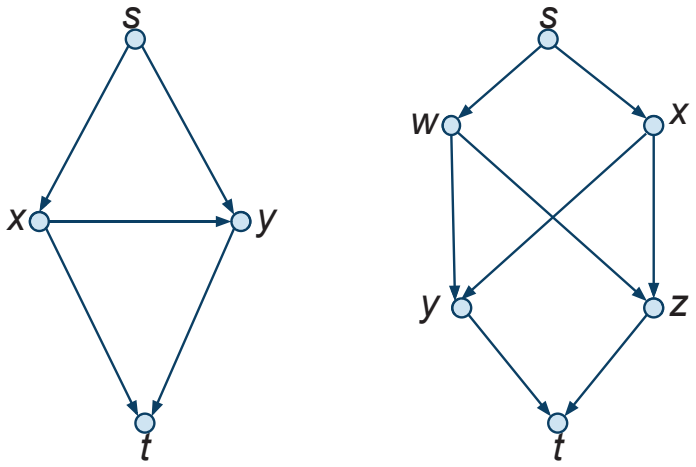

Figure 5.2: Two simple non-SP-DAGs.

A streaming application with the butterfly topology of Figure 5.2B is neither an SP-DAG nor even a CS4 DAG. However, it can be transformed to topologies with these properties by removing and redirecting certain graph edges. To transform this topology to a CS4 DAG without adding or removing nodes, we remove edge wz and add a directed edge from y to z. All tokens passed from w to z directly in the original topology would then be routed via node y. However, if we are limited to using only SP-DAGs, besides removing wz and adding yz, we would also need to remove edge xz and route tokens from x to z via node y, as Figure 5.3 shows. Hence, we can realize the original topology as a CS4 DAG with fewer changes than are needed to realize it as an SP-DAG.

A practical consequence of the difference between the CS4 and SP-DAG realizations of Figure 5.2B is that the CS4 DAG requires removing fewer edges, and hence less forwarding of tokens that were delivered directly in the original topology. Moreover, the total number of tokens sent is greater for the SP-DAG than for the CS4 DAG. As our experiments illustrate, reducing the total number of tokens sent by a given node can significantly improve its real-world performance.

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

Lemma 5.13 *G is CS4 only if no subgraph of G is homeomorphic to K_4, the complete graph on 4 vertices.*

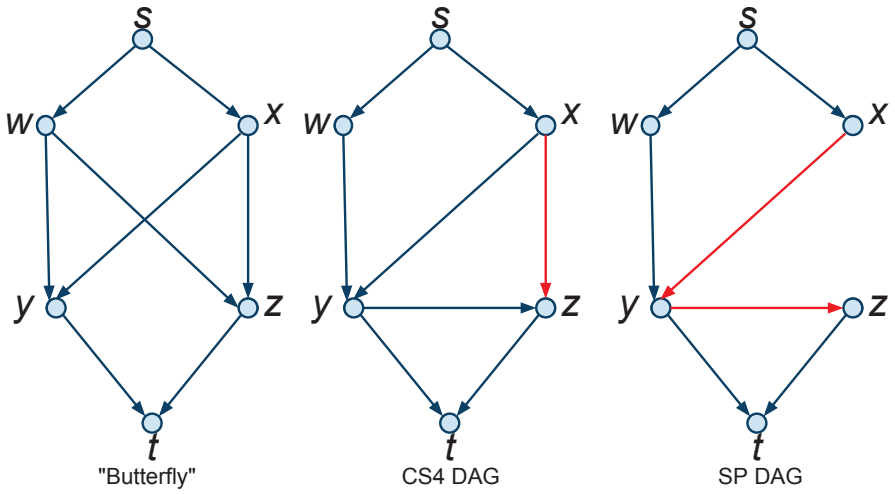

Figure 5.3: transforming butterfly to CS4 DAG and SP-DAG.

Proof. Suppose G has a subgraph H homeomorphic to K_4. H has 4 "corner" vertices and 6 connections (which may in general be paths rather than single edges) connecting them in the pattern of K_4. There are therefore 12 incidences of connections on corner vertices in H. WLOG, suppose that at least 6 of these are incoming. Now we have two cases.

1. Two vertices s and t of H have exactly two incoming edges apiece.

2. One vertex has 3 incoming edges.

Consider case 1. If the (unique) shared connection between s and t is oriented identically w/r to s and t (either into both or out of both), then it is possible to find a cycle through s and t with two sinks. Now consider the case when the connection $s - t$ is directed out of one vertex and into the other. Suppose WLOG that connection $x - y$ is directed out of s and into t. Let y and x be the other two corner vertices of H.

Exactly one of the connections $t - y$ and $t - x$ must be directed out of t. Suppose WLOG that $t - x$ is directed out of t. Because each of s and t have exactly two incoming edges, we know the following: (1) $s - x$ must be directed into s; (2) $y - s$ must be directed into s; (3) $y - t$ must be directed into t. Now $t - x$ must be directed into x; otherwise, there must be a sink on this

connection, and the cycle $sytx$ would contain two sinks. It follows that $s - x$ is directed out of x; otherwise, s and x would constitute the forbidden case (1).

Now we established above that $t - x$ may not contain a sink. Similarly, $s - x$ may not contain a sink because of cycle syx, and $s - t$ may not contain a sink because of cycle syt. Hence, cycle stx must be a directed cycle, which is forbidden because G is a DAG.

Consider case 2 above, where one corner vertex v of H has three incoming edges. Then no other corner vertex of H can have two incoming edges without creating a cycle with two sinks. Since H has at least six incoming edges on its corner vertices, it follows that the other three corner vertices of H each have exactly one incoming, and hence two outgoing, edges. Repeat the argument of Case (1) for any two of these vertices, swapping "in" and "out."

Conclude that there is no way to direct the edges of H so as to ensure that all its cycles have one source and one sink. □

Now absence of K_4 is a characteristic property of *undirected* series-parallel graphs [35]. Hence, we may expect that CS4 DAGs have an undirected series-parallel structure. However, this does not imply that a CS4 DAG is an SP-DAG; our simple four-node graph above provides a counterexample. Fortunately, as we now show, it turns out that just a small amount of extra complexity is needed to capture all CS4 DAGs.

Definition 5.4 *A* 2-path cycle *is a DAG consisting of a single source s, a single sink t, and two directed paths connecting s to t that are disjoint except at their endpoints.*

Definition 5.5 *Let C be a cycle. A* chord graph *H is a DAG with a single source and sink that connects two vertices of C, such that H's source and sink lie on C.*

Definition 5.6 *Let C be a 2-path cycle with paths p_1 and p_2. A* cross-link *is a chord graph that connects a vertex of p_1 to a vertex of p_2, where neither endpoint of the connection is C's source or sink. A* down-link *is a chord graph that is not a cross-link.*

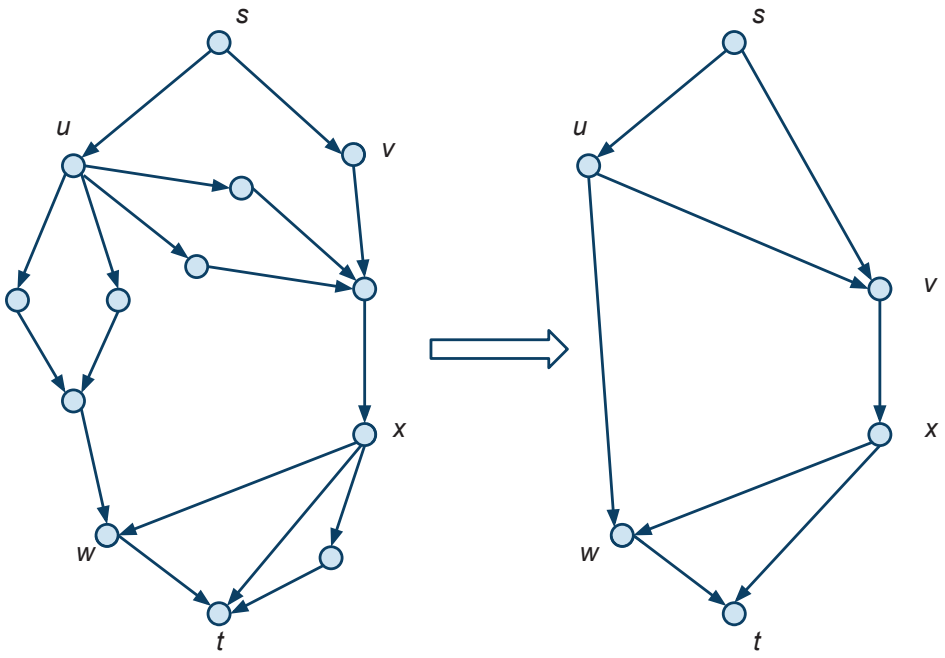

Figure 5.4: Decomposition of an SP-ladder graph.

Definition 5.7 *An* SP-ladder G *is a DAG consisting of a 2-path cycle with paths* p_1 *and* p_2*, called the outer cycle of* G*, and one or more chord graphs* $H_1 \ldots H_k$*, such that:*

- *Each* H_i *is an SP-DAG;*
- *At least one* H_i *is a cross-link;*
- *If* G *contains two chord graphs with endpoints* (u_1, v_1) *and* (u_2, v_2)*, then these chord graphs do not cross; that is, in tracing the outer cycle around* G*, we never encounter both* u_2 *and* v_2 *between* u_1 *and* v_1*.*

Intuitively, we call G an SP-ladder because it can be viewed as a 2-path cycle "decorated" with non-cross-link chord graphs, plus one or more cross-links connecting the paths, none of which cross each other. The cross-links are similar to the rungs of a ladder. Examples of simple and complex SP-ladders are given in Figure 5.4.

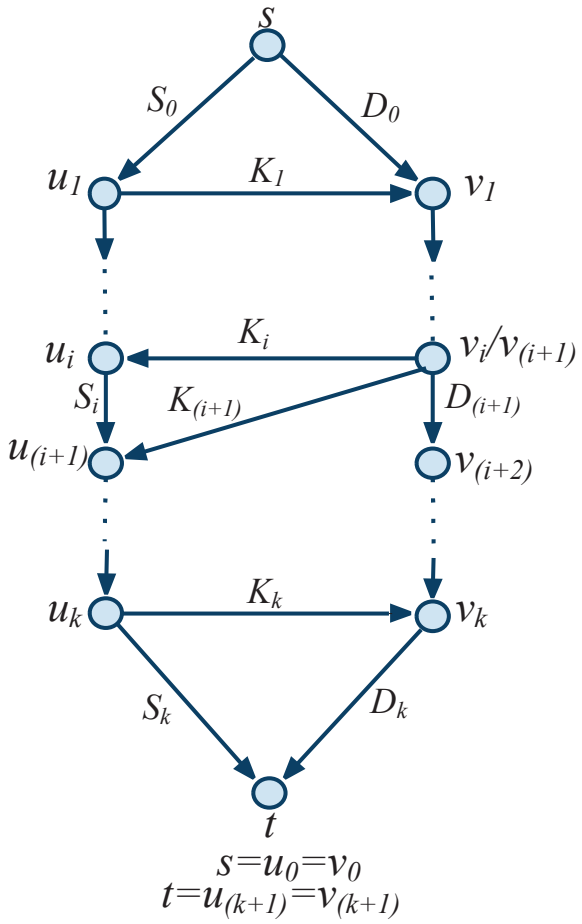

Figure 5.5: General structure of a decomposed SP-ladder graph, including an example of cross-links sharing an endpoint.

Definition 5.8 *Say that a cycle C of SP-ladder G* traverses *a chord graph H if C passes through a node of H other than its source or sink but is not confined to H.*

Lemma 5.14 *If an undirected simple cycle C in G traverses a chord graph H, then C contains a directed path in H from its source u to its sink v.*

Proof. C reaches an internal vertex of H from outside, so it must consist of a simple path p in H that connects u to v, plus a path to return from v to u outside H. We claim that path p is directed. Suppose not; p enters and leaves H through edges directed out of its source and into its sink, so p must contain an internal source at some node x. But Lemma 5.3 showed that there is no simple path connecting the source and sink of H that contains an internal source. □

Lemma 5.15 *Suppose that C traverses $k \geq 0$ cross-links of G. Then there is a cycle C' in G with at least as many sources/sinks as C that does not traverse any cross-link of G.*

Proof. By induction on k.

Base: Trivially true if $k = 0$; set $C' = C$.

Ind.: Suppose that C traverses k cross-links of G. Order these links as $H_1 \ldots H_k$ in topologically increasing order of their endpoints (which is possible, because they cannot cross). Let $u_i < v_i$ be the endpoints of H_i in G.

We claim that either C does not pass through any strict predecessor of u_1 or v_1, or that it does not pass through any strict successor of u_k or v_k. Since C traverses H_1, it contains a directed path from u_1 to v_1. Starting from v_1, C must return by some undirected path p to u_1. Now if the first edge on this path touches a predecessor of v_1, then C must return to u_1 without touching any successor w of u_1 or v_1; indeed, to reach w without passing through u_1 or v_1 itself, the path would have to traverse a chord graph that crosses H_1, which cannot exist. If, on the other hand, p's first edge touches a successor of v_1, then C must return to u_1 without touching any predecessor w of u_1 or v_1, for the same reason.

Suppose that C does not touch a predecessor of u_1 or v_1. Construct C' from C by removing the path through H_1 and replacing it with the path on G's outer cycle that connects u_1 and v_1, passing through G's source s. C' does not contain the source that lies at endpoint u_1 of H_1 in C, but it does contain a new source at s. Removing H_1 cannot eliminate any other source or sink of C, so C' has as many sources/sinks as C.

If instead C does not touch a successor of u_k or v_k, construct C' from C by removing the path through H_k and replacing it with the path on G's outer cycle that directly connects u_k and v_k, passing through G's sink t. C' does not contain the sink that lies at endpoint v_k of H_k in C, but it does contain a new sink at t. Removing H_k cannot eliminate any other source or sink of C, so C' has as many sources/sinks as C.

By the IH, there is a cycle C'' in G with at least as many sources/sinks as C' that does not pass through any cross-link of G. □

Corollary 5.16 *Every SP-ladder is CS4.*

Proof. Let C be any cycle in an SP-ladder G. If C traverses $k > 0$ cross-links of G, Lemma 5.15 guarantees that there is a cycle C' that does not traverse any cross-links of G with at least as many sources/sinks as C. Now either C' is confined to some chord graph H of G, or C' lies in the graph G' obtained by removing all cross-links from G. H and G' are both SP-DAGs, which are CS4 by Lemma 5.5. Hence, C' has only one source and one sink. Conclude that C has only one source and one sink, and so G is CS4. □

Lemma 5.17 *Let G be a DAG with a single source and sink that is CS4. Then G is a serial composition of one or more graphs $G_1 \ldots G_k$, s.t. each G_i is either an SP-DAG or an SP-ladder.*

Proof. Divide G into subgraphs $G_1 \ldots G_k$ at its articulation points, so that G is the serial composition of $G_1 \ldots G_k$. If every G_i is an SP-DAG, we are done. Otherwise, let G^* be a component of G that is not an SP-DAG. Now G^* has no internal articulation points, so it is composed of a 2-path outer cycle cut by one or more chord graphs.

Let H_1, H_2 be two chord graphs in G^*, with endpoints u_1/v_1 and u_2/v_2. If these subgraphs cross, then there exist paths p_1 connecting u_1 and v_1 in H_1 and p_2 connecting u_2 and v_2 in H_2. Moreover, G^*'s outer cycle contains u_1, v_1, u_2, and v_2 in some alternating order. Hence, the union of p_1, p_2, and this cycle is homeomorphic to K_4, and so G^* (and hence G) cannot be CS4. Conclude that no two chord graphs of G^* cross.

Now suppose that some chord graph H is not an SP-DAG. Let H^* be a smallest subgraph of H that is not an SP-DAG. H^* cannot be a serial composition of multiple subgraphs, so it is a 2-path outer cycle with one or more chord graphs, all of which are SP-DAGs. If H^* had no cross-link, we could decompose it as an SP-DAG via repeated parallel compositions to extract all of its chord graphs. Hence, some chord graph J of H^* is a cross-link.

Let u, v be the endpoints of J, and let x, t be its source and sink. The outer cycle of H^* connects these vertices in the order $x - u - y - v$. Moreover, there is a path from u to v bypassing s and t (through the cross-link) and a path from s to t bypassing u and v (from s outwards to the source of H, then via the outer cycle of G^* to the sink of H, and finally inwards to y). The union of these two paths and the outer cycle of H^* is therefore homeomorphic to K_4 , and so H^* (and hence G) cannot be CS4. Conclude that H^*, and therefore H, cannot exist, and so every chord graph of G^* is indeed an SP-DAG.

Finally, if no chord graph of G^* is a cross-link, G^* can be decomposed via repeated parallel compositions to expose all its chord graphs and so is an SP-DAG. Otherwise, it is an SP-ladder. Conclude that every component of G is either an SP-DAG or an SP-ladder. □

Theorem 5.18 *The set of single-source, single-sink CS4 DAGs is exactly the family of graphs of which each one is a serial composition of one or more graphs $G_1 \ldots G_k$, s.t. each G_i is either an SP-DAG or an SP-ladder.*

Proof. Lemma 5.17 shows that every single-source, single-sink CS4 DAG is in the claimed family. Conversely, Lemma 5.13 and Corollary 5.16 show that SP-DAGs and SP-ladders respectively are CS4. Serial composition of such graphs cannot introduce new cycles, so all such compositions remain CS4. □

5.4 Efficient Deadlock Avoidance for CS4 DAGs

We now present algorithms to compute optimal dummy message schedules for deadlock avoidance on CS4 graphs. Since a CS4 graph is serial composition of SP-DAGs and SP-ladders, edges on different SP-DAGs and SP-ladders cannot be on the same simple cycle. Hence, we can first decompose a CS4 graph into SP-DAGs and SP-ladders, then compute schedules for edges in each of these subgraphs separately. We have already described algorithms for SP-DAGs, so here we focus on SP-ladders.

An SP-ladder can be decomposed into its constituent SP-DAGs as shown in Figure 5.4, where each edge represents an SP-DAG directed the same way as the edge. This simplified representation of an SP-ladder has two paths from the source s to the sink t. For convenience, we assume the two paths go from top to the bottom and distinguish them as the "left path" and the "right path." We call the vertices that connect these paths to cross-links *corner vertices* and mark them from top to bottom, with the vertices on the left labeled $u_0, u_1, u_2, \ldots, u_{k+1}$ and the vertices on the right path from top to bottom labeled $v_0, v_1, v_2, \ldots, v_{k+1}$. The source $s = u_0 = v_0$ and the sink $t = u_{k+1} = v_{k+1}$. All other nodes are called *internal nodes*. This graph has k cross-links, which are numbered from top to bottom as K_1 through K_k, and the SP-DAGs on the outer cycle are numbered S_0 through S_k on the left and D_0 through D_k on the right. Note that in some cases, $u_i = u_{i+1}$, in which case S_k is a graph with a single node. Figure 5.5 illustrates the general decomposition and this special case.

Definition 5.9 *We say that an undirected simple cycle is* external *if it traverses at least two of the constituent SP-DAGs.*

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

Fact 5.19 *Any external cycle with source $s = u_0 = v_0$ has a path through S_0 and another path through D_0. Any external cycle with source u_i ($i \neq 0$) has one path going through S_i and another path going through K_i. Similarly for source v_i ($i \neq 0$). All external cycles have corner nodes as sources and sinks.*

Fact 5.20 *Consider any external cycle C with source u_i. There are three possibilities:*

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

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

- *The sink of the cycle is $t = u_m = v_m$, the sink of the ladder. One path on the cycle crosses K_i and passes through all v_j where $i \leq j$. The other path traverses S_i, goes through all u_j where $i \leq j$.*

We call the sinks defined in Fact 5.20 the *potential sinks* of u_i. We can similarly define potential sinks for an internal source v_i.

5.4.1 Destination-Tagged Propagation Algorithm for SP-ladders

We now give an efficient version of the Destination-Tagged Propagation Algorithm specialized for SP-ladders. Again, only sources send dummy messages. An SP-ladder has two types of cycle sources: *internal sources* and *corner sources*. The algorithms for internal sources are similar to those described in Section 5.2. We will concentrate on describing the algorithms for the corner sources. We will describe all the algorithms for some u_i, where u_i is a corner node on the left path of the ladder. Analogous algorithms can be derived for nodes on the right path.

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

1. $[e]_i$ consists of pairs for messages that stay within the chord for which u_i is a source (S_i for down-link, and K_i for cross-link). These are kept sorted by increasing τ as in the case of SP-DAGs.

2. $[e]_s$ consists of pairs for nodes v_k where $k > i$, i.e. corner nodes on the opposite side of the ladder from u_i

3. $[e]_y$ consists of pairs for nodes u_i where $k > i$, i.e. corner nodes on the same side of the ladder as u_i

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

Computing Dummy Message Schedules

We calculate the dummy message schedules for edges as follows:

1. Decompose the SP-ladder into the component SP-DAGs, identifying the u_i's, v_i's, S_i's, D_i's and K_i's. In addition, mark each edge as either belonging to a cross-link or a down-link. This can be done in $O(|G|)$ time.

2. Compute $[e]_i$, schedules for all edges due to cycles internal to each chord graph , using the algorithm of Section 5.2.2, which has a time complexity of $O(|G|^2)$.

3. For all $H \in \bigcup_{0 \leq i \leq k} S_i \cup D_i \cup K_i$, compute $L(H)$, which is the length of a shortest path from H's source to its sink (in terms of buffer sizes). Again, this is done as shown in Section 5.2.2 with a time complexity of $O(|G|)$.

4. Starting at the bottom of the SP-ladder, for each u_i, and for each potential sink t of u_i, compute $L_s(u_i, t)$, which is defined as the shortest directed path starting at u_i, going through S_i and ending at t. Similarly, define $L_k(u_i, t)$ as the shortest directed path starting at u_i, going through K_i and ending at t. If u_i is not the source of K_i, then just set $L_k(u_i, t) = 0$. Define and compute $L_d(v_i, t)$ and $L_k(v_i, t)$ in a similar manner. This step can be done in $O(|G|)$.

5. Using these L values, update the set of dummy interval pairs for all edges that start at internal sources and at source s. No other sets change.

For step 1 above, we decompose an SP-ladder into its constituent SP-DAGs in $O(|G|)$ time as follows: Identify an outer cycle C for G with left and right sides, using DFS in linear time.

For each vertex u on the left side of C, determine (via DFS) whether any directed path leaving u encounters the right side of C at some vertex v before it encounters the left side again. If so, the nodes and edges on all such paths from u to v form a cross link. Repeat for the right side of C to identify cross-links directed from right to left. Now that we have identified all u_i's and v_i's, we can easily compute S_i's, D_i's and K_i's.

For step 4 above, we compute $L_s(u_i, t)$ and $L_k(u_i, t)$, where t is a potential sink u_k or v_k of u_i. We consider u_i's in decreasing order of i. In order to compute $[e]_s$ and $[e]_y$ in sorted order, for a particular u_i, we consider t in increasing order of k.

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

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

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

- If any edge e has an internal pair $d_a = (\tau_a, z_a)$ and an external pair $d_b = (\tau_b, z_b)$, where $\tau_a \geq \tau_b$, then d_a is removed.

- If a particular edge e has more than one interval with the same destination, we keep only the one with the smallest τ.

The most costly step above can be finished in $O(|G|^2)$, so the overall time complexity is $O(|G|^2)$.

Runtime Node Behavior

The behavior of all nodes except the corner source remains the same as in the corresponding algorithm for SP-DAGs. As mentioned above, a corner source u_i has 3 lists of dummy message pairs, $[e]_i$, $[e]_s$ and $[e]_y$, where $[e]_i$ is sorted by increasing τ and $[e]_s$ and $[e]_y$ are sorted by increasing k, where destination is a corner sink v_k or u_k respectively. Each dummy pair $d_a = (\tau_a, z_a)$ has counter c_a associated with it, and the maximum value of the counter is τ_a. One other difference from SP-dags is that in some cases, a dummy message can have more than one destination. If that is the case, the dummy message carries the list of destinations with it. There are two cases in the runtime behavior of a corner source u_i

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

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

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

- Say d_a is some u_k, or v_k, $k > i$, [7] then the dummy message is scheduled on all the down-link edges, and the counter for the pairs going to this destination are zeroed out. For a down-link edge e, all the counters in $[e]_i$ (for all the internal dummy messages) on these down-links are zeroed out. All the counters on $[e]_y$ with destination u_j, $j \leq k$ are zeroed out. All the counters (on down-links and cross-links) that are not zeroed out are incremented.

- If d_a is some v_k, $k = i$, [8] then the dummy message is scheduled on along all the cross-link edges and all the counters in $[e]_i$ are zeroed out. All the other counters are incremented.

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

Proof of Correctness

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

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

Lemma 5.21 *The following claims are true.*

1. *If a corner source u_i forwards a dummy message along an edge of a chord graph, it will go through all the nodes within that chord.*

[7] If there are two cross-links out of u_i, then we use the larger index i to make this decision.
[8] If there are two cross-links from i, we forward along the one that is equal.

2. *If a corner source u_i sends or forwards a dummy message along a down-link to some sink u_k or v_k, where $k \geq i$, this message will go through all the sinks u_j, $i \leq j \leq k$.*

3. *If a corner source u_i sends or forwards a dummy message along a cross link K_i intended for v_k or u_k, where $k \geq i$, it reaches all the nodes v_j, $i \leq j \leq k$.*

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

Claims 2 and 3 are true due to Lemma 5.20. □

The following lemmas are analogous to Lemmas 5.10 and 5.11 for SP-dags.

Lemma 5.22 *Suppose that, for edge e out of node s, pair $(\tau_i, z_i) \in [e]$. For each τ_i messages that s receives, it sends at least one dummy message along e that will reach z_i.*

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

1. The counter will be incremented until it reaches τ_i, triggering a dummy message to z_i.

2. The counter will be zeroed out because some other dummy message is sent or forwarded. From node behavior and Lemma 5.21, the counter is zeroed out only if the dummy message sent or forwarded will pass through z_i.

□

Lemma 5.23 *Suppose that an external cycle in G starts at u_i and ends at t. Every time u_i receives $L_S(u_i, t)$ messages, it sends at least one dummy message with destination t along all its cross-link*

edges. Every time u receives $L_K(u_i, t)$ messages, it sends at least one dummy message along all its down-link edges.

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

Therefore, Lemma 5.22 implies the proof. □

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

Theorem 5.24 *If dummy messages are sent as described in Section 5.2.2, using the interval-destination pairs computed by the above procedure, then deadlock cannot occur in G.*

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

We know that $\mathit{length}(s_1) \geq L_K(u_i, t)$. If we consider the first edge of path s_2, it leaves u_i through its cross-link. From Lemma 5.23, u_i sends a dummy message along this edge every time it gets $L_K(u_i, t)$ messages. Since this message is propagated all the way to t, s_2 cannot be completely empty, which contradicts our assumption that cycle C is blocking. □

5.4.2 Non-Propagation Algorithm

Computing the dummy intervals for the Non-Propagation Algorithm takes longer than for the Destination-Tagged Propagation Algorithm on SP-ladders. Here we give an $O(|G|^3)$ algorithm.

Again, we decompose into constituent SP-DAGs. As in the Non-Propagation Algorithm for SP-DAGs, for each constituent SP-DAG H, we precompute $h(H)$ as the length of the longest path (in terms of the number of hops) from H's source to its sink. In addition, for each edge e in H, compute $h(H, e)$ as the longest path from H's source to its sink that passes through e. In addition, we compute the initial estimate of the dummy intervals considering only the cycles internal to the constituent SP-DAGs.

Now consider every source u_i in the SP-ladder. We can enumerate all the potential sinks t for that source using Lemma 5.20. As we defined $L_s(u_i, t)$ and $L_K(u_i, t)$ we define $h_s(u_i, t)$ is the length of the longest directed path (in terms of hop count) from u_i to t that goes along S_i and $h_k(u_i, t)$ as the length of the longest directed path from u_i to t that goes along K_i.

Now consider an edge e in some constituent SP-DAG H along the path from u_i to t. We can update the dummy interval for e as follows: If e lies along some path from u_i to t that goes across K_i, then $[e] = L_s(u_i, t)/(h_k(u_i, t) - h(H) + h(H, e))$. If on the other hand, e lies along some path from u_i to t that goes across S_i, then $[e] = L_k(u_i, t)/(h_s(u_i, t) - h(H) + h(H, e))$. We can do the analogous procedure for each potential source v_i.

Running time: There are $O(|G|^2)$ source-sink pairs. For a given pair u_i and t, we can calculate $L_s(u_i, t)$, $L_k(u_i, t)$, $h_s(u_i, t)$ and $h_k(u_i, t)$ using L and h values of the constituent SP-DAGs in $O(|G|)$ time. We can also update all dummy intervals for edges on some path from u_i to t in $O(|G|)$ time. Therefore, the overall algorithm takes $O(|G|^3)$ time.

5.5 Summary

Computing dummy intervals for general DAGs can be very time-consuming. In this chapter, we discussed properties of the series-parallel DAGs (SP-DAGs) and proposed a new class of DAGs, CS4 DAGs, where every undirected cycle has only one source node and one sink node. We have shown that, if the allowed streaming topologies are restricted to the SP-DAGs or the CS4 DAGs, then we can efficiently (in time polynomial to graph size) compute dummy message intervals for

all edges. In addition, we have extended the Propagation Algorithm to reduce the amount of propagation, thereby potentially reducing overheads.

Chapter 6

Polyhedral Constraints for Dummy Message Scheduling

In the previous chapters, we proposed three different algorithms to ensure bounded-memory execution for SFDF applications: the Naive Algorithm, the Propagation Algorithm (and its variants), and the Non-Propagation Algorithm. We also proposed efficient algorithms to compute a deadlock-free set of dummy intervals for special dataflow graphs. In this chapter, we try to provide more choices for dummy interval selection. We show that the set of safe dummy intervals for the Non-Propagation Algorithm can be defined by a set of linear constraints. The number of such constraints, however, may be exponential to the size of a DAMG. For SP-DAGs, we reduce the number of constraints to a number polynomial in the graph size. This chapter has been published in [70].

6.1 Polyhedral Characterization of Safe Dummy Intervals

Algorithm 4.3 in Section 4.3 describes the node behavior for the Non-Propagation Algorithm. Just to review, each node maintains a *compute index*, which is the index of the last set of input tokens it consumed, as well as a *last sent index* for each output channel, indicating when it last sent a token on that channel. Each channel q has a (finite) buffer length $|q|$ and a dummy interval $[q]$, which is statically determined given only the graph topology and buffer lengths. A node emits a

dummy message on channel q only if it has no output data to send *and* the node's compute index has increased by more than $[q]$ since it last sent a token on q. If $[q] = 0$ for every channel, the application behavior reverts to the HDF [64]; if $[q] = \infty$, then a node need never send any dummy messages on channel q.

Algorithm 4.4, which computes dummy intervals at compilation time, provides one set of safe dummy intervals. Setting each dummy interval to a value no *greater than* the one computed by Algorithm 4.4 is also safe (though doing so might reduce throughput), but are those all possible sets of safe dummy intervals? If we increase the dummy interval for a channel computed by Algorithm 4.4 but decrease the dummy interval for another channel at the same time, is the new set of dummy intervals still safe?

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

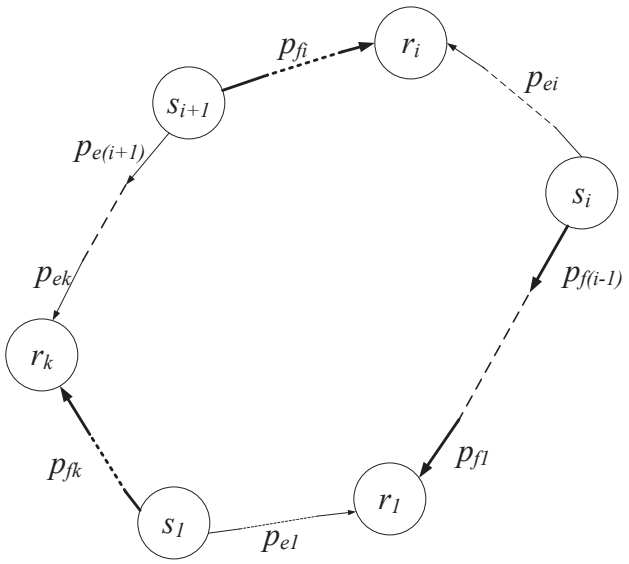

Figure 6.1: The division of a blocking cycle, previously used as Figure 4.1. Node and channel labels are used in the proofs of Theorem 6.1 and Theorem 6.2.

To describe the necessary constraints, consider Figure 6.1, which illustrates a blocking cycle C in an application. Channels on this cycle are directed either clockwise or counterclockwise. If C is blocking, then either all its clockwise channels are full and all its counterclockwise channels are empty, or vice versa.

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

$$\Sigma_{q \in H_1}[q] \quad < \quad \Sigma_{q \in H_2}|q| \tag{6.1}$$

$$\Sigma_{q \in H_2}[q] \quad < \quad \Sigma_{q \in H_1}|q|. \tag{6.2}$$

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

Figure 6.3 visualizes safe dummy intervals for the topology in Figure 6.2 on 2D planes. We use two coordinate systems since safe dummy intervals for uv/vx and uw/wx are independent. The blue-shaded areas define the safe dummy intervals, but the red lines are excluded.

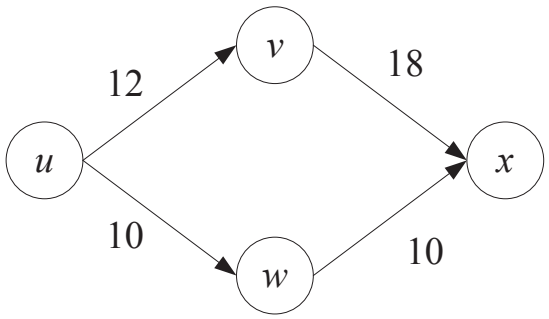

Figure 6.2: A simple streaming topology with buffer sizes labeled.

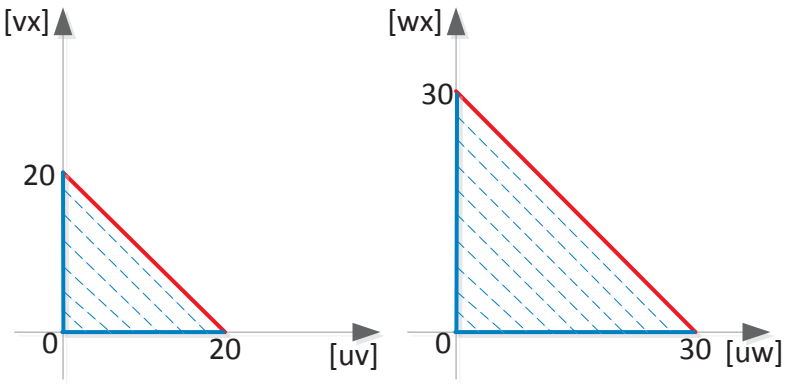

Figure 6.3: Visualization of safe dummy intervals for the topology shown in Figure 6.2.

Proof for Sufficiency

Theorem 6.1 *If all nodes behave according to Algorithm 4.3, using dummy intervals that satisfy constraints as defined by Inequalities 6.1 and 6.2, then the application cannot deadlock.*

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

Divide cycle C into paths p of consecutive, similarly directed channels connecting *senders* s and *receivers* r. In particular, label the nodes on path p_{ei} $v_0, \ldots v_n$, with $v_0 = s_i$ and $v_n = r_i$, as in Figure 6.1.

We continue to use the concepts of *minval* and *maxval* defined in the proof for the Non-Propagation Algorithm in Section 4.3.

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

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

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

We have that $M_j - M_{j+1} = [v_j v_{j+1}]$, and so $M' - M_{j+1} > [v_j v_{j+1}]$, which means the interval between v_j's last received and last sent tokens is at least $[v_j v_{j+1}]$, the dummy interval of $v_j v_{j+1}$. Hence, Algorithm 4.3 implies that v_j must have sent a token, either real or dummy, to v_{j+1} with index $> M_{j+1}$. But this contradicts our IH. Conclude that the last token received by v_j has index at most M_j, as desired.

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

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

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

$$\text{minval}(p_{ei}) \geq \text{minval}(p_{f(i-1)}) + |p_{f(i-1)}| - [p_{ei}] - 1, \tag{6.3}$$

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

$$\text{minval}(p_{fi}) \geq \text{minval}(p_{ei}) + 1. \tag{6.4}$$

Starting from a receiver r_i, traversing the cycle clockwise, and applying inequalities 6.3 and 6.4 alternately, we have

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

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

Proof for Necessity

Theorem 6.2 *If all nodes behave according to Algorithm 4.3, but dummy intervals do* not *satisfy the constraints defined by Inequalities 6.1 and 6.2, then the application can potentially deadlock.*

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

Let C be an undirected cycle in the application, again as shown in Figure 6.1, for which the chosen dummy intervals violate a constraint. WLOG, suppose in particular that $\Sigma_{q \in H}[q] \geq \Sigma_{q \in C \backslash H}|q|$, where H is the set of counterclockwise channels on C.

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

yz for every input received. Hence, the only channels on which outputs may be filtered are the counterclockwise channels of C.

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

$$N(v) = N(u) + [uv] \tag{6.5}$$

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

$$N(v) = N(u) - |uv| \tag{6.6}$$

if uv is a clockwise channel.

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

Lemma 6.3 *For any node u, suppose its clockwise neighbor is v. The event $I(u) = N(u) + 1$ happens after the event $I(v) = N(v) + 1$.*

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

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

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

(Theorem 6.2's proof continues.) According to Lemma 6.3, we have a temporal contradiction if any $I(u)$ advances to $N(u)+1$. Hence for any node u on C, it can only advance its compute index to $N(u)$. But there are still unprocessed tokens. According to the definition of deadlock, the system is deadlocked. □

6.2 Constraints for Series-parallel DAGs

For some graphs with particular structures, it is possible to enumerate a polynomial-sized set of constraints that is equivalent to the feasible polyhedron, so verifying safety of dummy intervals and finding extrema can be simpler. Consider, for example, series-parallel DAGs (SP-DAGs), which can be constructed via a sequence of series compositions and/or parallel compositions starting from single edges. We provide an algorithm for defining the polyhedron of deadlock-free dummy intervals for an SP-DAG G.

1. Decompose the graph G into a tree of components.

2. Compute $L(H)$ for each component H, where $L(H)$ is the shortest path from H's source to H's sink, with buffer sizes as edge weights.

3. Introduce a variable $d(H)$ for each component H.

   - For a single edge H, add constraint

   $$L(H) = |H|. \tag{6.7}$$

   - If $H = Sc(H_1, H_2)$, add constraints

   $$L(H) = L(H_1) + L(H_2) \tag{6.8}$$

   $$d(H) = d(H_1) + d(H_2). \tag{6.9}$$

- If $H = Pc(H_1, H_2)$, add constraints

$$L(H) \leq L(H1) \tag{6.10}$$

$$L(H) \leq L(H2) \tag{6.11}$$

$$d(H) \geq d(H_1) \tag{6.12}$$

$$d(H) \geq d(H_2) \tag{6.13}$$

$$d(H_1) < L(H_2) \tag{6.14}$$

$$d(H_2) < L(H_1). \tag{6.15}$$

Given a directed path p, we define $[p] = \Sigma_{q \in p}[q]$. We make the following claims.

Claim 6.4 *Let γ be the set of all directed source-to-sink paths in an SP-DAG component H, $L(H) = min_{p \in \gamma}|p|$.*

Proof. We prove by induction.

Bas. H is a single edge. It is trivially true, as indicated by by Equality 6.7.

Ind. If $H = Sc(H_1, H_2)$, since the min operator is additive, Equality 6.8 ensures $L(H)$ to be the minimum total weight of all source-to-sink paths in H; if $H = Pc(H_1, H_2)$, Inequalities 6.10 and 6.11 together guarantee $L(H) = min(L(H_1), L(H_2)) = min_{p \in \gamma}|p|$. □

Claim 6.5 *Let γ be the set of all directed source-to-sink paths in an SP-DAG component H, $d(H) = max_{p \in \gamma}[p]$.*

Proof. We prove by induction.

Bas. H is a single edge, trivially true.

Ind. If $H = Sc(H_1, H_2)$, since the max operator is additive, our claim holds because of Equality 6.9; if $H = Pc(H_1, H_2)$, Inequalities 6.12 and 6.13 guarantee $d(H) = max_{p \in \gamma}[p]$. □

Claim 6.6 *The set of constraints defined by Inequalities 6.7 to 6.15 defines the polyhedron of deadlock-free dummy intervals for the SP-DAG G.*

Proof. We argue that the given inequalities enforce exactly the constraints of the formulation for general graphs. Undirected cycles are created only by parallel compositions, and according to Claim 5.5, every undirected cycle is single-source and single-sink. If $H = Pc(H_1, H_2)$, let γ_1 and γ_2 be the set of all directed source-to-sink paths in H_1 and H_2, respectively. $\forall p_1 \in \gamma_1, \forall p_2 \in \gamma_2$, according to Claim 6.4 and Claim 6.5, we have

$$\begin{aligned} L(H_1) &= min_{p\in\gamma_1}|p| \\ L(H_2) &= min_{p\in\gamma_2}|p| \\ d(H_1) &= max_{p\in\gamma_1}[p] \\ d(H_2) &= max_{p\in\gamma_2}[p], \end{aligned}$$

which, combined with Inequalities 6.14 and 6.15, are at least as constrained as Inequalities 6.1 and 6.2. Since each parallel composition creates at least one cycle, so Inequalities 6.14 and 6.15 never induce unnecessary constraints. Conclude that Inequalities 6.7 to 6.15 define the polyhedron of deadlock-free dummy intervals for G. □ For each serial or parallel composition, we add constant number of constraints. Since the number of compositions is polynomial to graph size, we have polynomial-size constraints for SP-DAGs. We can verify constraint violations in polynomial time by checking each constraint directly.

6.3 Selection of Dummy Intervals for Performance

We have now defined a polyhedral space of feasible dummy intervals for preventing deadlock. Given a plethora of possible feasible solutions, which one should we choose? The answer depends on the application designer's performance goals.

Latency and throughput are two common performance measures for parallel applications. In streaming computing, long-running analytical applications may have throughput but no latency requirements (e.g. [53]). Others, such as computational finance [104] and face recognition [113],

have a real-time component and so may have both latency and throughput requirements. Balancing latency and throughput requires challenging optimization [91], which is not the focus of this dissertation; here, we focus on selecting dummy intervals purely to optimize throughput. Generally speaking, larger dummy intervals introduce less communication overhead and so favor throughput; this qualitative statement has been verified by our previous experiments [69]. As a result, we are interested in sets of *maximal* dummy intervals, i.e. those at the boundaries of the feasible polyhedron. In the previous example, we might investigate sets of dummy intervals that satisfy $[uv] + [vx] = 15$ and $[uw] + [wx] = 15$.

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

Experimental Results

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

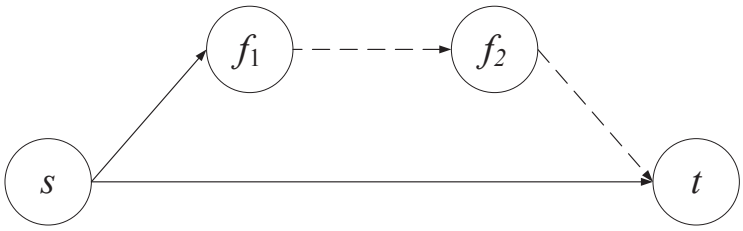

Figure 6.4: Dataflow graph of a synthetic application.

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

Figure 6.4 shows the dataflow graph of our synthetic application. The two dashed channels filter data. We assign a buffer size of 32 for channel st, so $[f_1 f_2] + [f_2 t] < 32$ is the constraint for avoiding deadlocks. Nodes f_1 and f_2 each filter 45% of the data arriving from the start node s. Applications with such high filtering rates are typically among the most vulnerable to deadlock, since some channels may go a long time without seeing any real data.

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

Table 6.1 and Table 6.2 show our experimental results. All sets of dummy intervals studied are maximal with respect to the polyhedron of feasible solutions, so no one set dominates another *a priori*. As expected, correlated filtering, which tends to produce more persistently empty channels, results in a larger number of dummy messages overall.

Table 6.1: Measured dummy message counts for correlated filtering

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

Table 6.2: Measured dummy message counts for uncorrelated filtering

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

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

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

6.4 Summary

The Propagation Algorithm and the Non-Propagation Algorithm (and their variations) only find *one* set of maximal safe dummy intervals. In this chapter, we used polyhedral constraints to precisely characterize the *complete* set of safe dummy intervals. For a general DAG, the number of polyhedral constraints can be exponential in the graph size. For applications with an SP-DAG topology, we showed that the number of linear constraints is polynomial, so verifying safety of dummy intervals and finding extrema can both be done in time polynomial to the size of the graph size.

Chapter 7

Support for General Control Messages in SFDF Applications

In the previous chapters, we have discussed using dummy messages to ensure correctness of SFDF applications. The dummy messages differ from data tokens in that they deliver control information generated by computing nodes rather than data from the data source. Dummy messages are just one type of *control message*, which carry control information between compute nodes. In this chapter, we will extend the SFDF model to support*general* control messages correctly and efficiently. Bounded-memory execution is still guaranteed in the extended model. We will show that control messages can help improve application performance.

7.1 Control Messages and Their Uses

A control message has one of a finite set of types and can contain arbitrary content. Besides the dummy message, there are other kinds of control messages that help maintain correctness or improve performance of streaming applications. We have used dummy message for guaranteeing correctness of streaming applications; the following application example shows the use of control messages for improving application performance.

7.1.1 An Application Example

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

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(z_i - \overline{z})^2, \tag{7.1}$$

where $\overline{z}$ is the average of the N values. Equation 7.1 seems to require a two-pass calculation process: one pass to compute the mean, and a second to compute the variance using the mean. However, we can convert this computation to a one-pass algorithm [118, 21] that is more streaming-friendly:

$$\sigma^2 = \overline{z^2} - \overline{z}^2. \tag{7.2}$$

We can implement Equation 7.2 as a streaming computation as in Figure 7.1. The source node u duplicates input data to v and w, which compute $\overline{z}$ and $\overline{z^2}$ respectively. These quantities are then merged at node x to compute variance values.

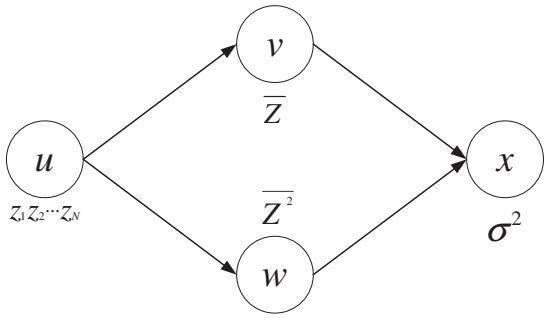

Figure 7.1: A streaming computation for variance. It occurs as part of large streaming computing systems, including the next generation of VERITAS [111], a ground-based gamma-ray observatory system.

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. Node u does not need 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 message* 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.

7.1.2 Other Potential Uses of Control Message

The variance example is a case of using control messages to communicate boundaries between finite-length datasets. There are other application scenarios that control messages are necessary:

Application finishing. All nodes should stop computing after the input stream (if bounded) has been processed. The source node knows when there is no more input data, so it can send a control message to indicate "no more input data."

On-the-fly configuration. A node might need to change its algorithmic behavior during computation, and the change might be determined by the source node. For example, the streaming graph matching algorithms proposed in [39] require multiple passes over the input data. The algorithm for each pass is different. The source node needs to tell other nodes to switch algorithm when a new pass begins.

Fault tolerance. When a large application runs on many machines, node failure can be common. Typical fault-tolerance implementations require each node to report its status periodically [52]. The status information can be delivered via control messages.

Not every type of control message needs to be precisely ordered with respect to a data stream. For example, fault tolerance information [1, 5] is not required to be synchronized with data streams.

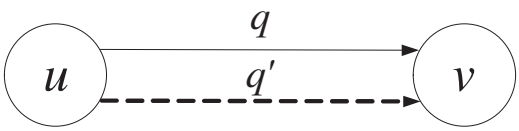

Figure 7.2: An edge with paired data and control channels q and q'.

However, it is important to support precise ordering semantics when required. In this chapter, we focus on control messages that need to be precisely ordered with respect to data streams.

7.2 Precise Control-Data Ordering for SFDF

7.2.1 Delivery of Control Messages

To deliver control messages, for each data channel q connecting two nodes, we create a parallel control channel q', as shown in Figure 7.2. We refer to this pair of channels as an *edge* between the nodes as the two channels are corresponding to an edge in the dataflow graph. All control messages between the two nodes are delivered through q'. A node can choose to listen for input on at most one channel at a time; to guarantee determinism, once a channel is chosen for listening, the node can take no further action until input appears on that channel.

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.

7.2.2 A Credit-based Protocol

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.

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 7.1.

Algorithm 7.1: Receiver Credit Balance Protocol

```
1  while RCB = 0 do
2      wait for a control message on q'
3      let c be credit value carried by message
4      if c = 0 then
5          consume message
6      else
7          Detach c credits from message
8          RCB ← RCB + c
9  wait for a data token on q
10 consume token
11 RCB ← RCB − 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. Algorithm 7.2 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.

Algorithm 7.2: Sender Credit Balance Protocol

```
1  if token is ready then
2      emit token on q
3      SCB ← SCB + 1
4  while control message is ready OR SCB > T do
5      emit message on q' with SCB credits
6      SCB ← 0
```

7.2.3 Correctness and Safety

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

Claim 7.1 *If a receiver and sender are connected by an edge and behave as in Algorithms 7.1 and 7.2, 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 7.2 *If a receiver and sender are connected by an edge and behave as in Algorithms 7.1 and 7.2, 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 Chapter 4.)

- *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. □

7.3 Extending SFDF with Precise Control

We now explore how to extend 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 7.3 describes how to combine SFDF with control channels. To ensure precise data and control ordering, each node implements Algorithm 7.1 on each of its input edges and Algorithm 7.2 on each of its output edges. A new type of control message called *credit message* is used in the protocol, which carries credit value. 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. Since data channels are synchronized, and each control channel is synchronized with its paired data channel, control channels are implicitly synchronized. Therefore, no attempt is made to explicitly synchronize control messages across edges.

As in ordinary SFDF applications, 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.

Algorithm 7.3: Single-node behavior in SFDF with control messages.

```
1  foreach input edge e do
2      while RCB_e = 0 do
3          wait for a control message on q'_e
4          let c be credit value carried by message
5          if c = 0 then
6              consume message
7          else
8              Detach c credits from message
9              RCB_e ← RCB_e + c
10     wait for a data token on q_e
11 let i be least index among data tokens on all edges
12 foreach input edge e do
13     if token on q_e has index i then
14         consume token
15         RCB_e ← RCB_e − 1
16 perform computation for index i
17 foreach output edge e do
18     if token is ready on e then
19         emit token on q_e with index i
20         SCB_e ← SCB_e + 1
21     while control message is ready on e OR SCB_e > T_e do
22         emit message with SCB_e credits
23         SCB_e ← 0
```

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

7.3.1 Deadlocks Due to Full Data Channels

We know by Theorem 3.1 that 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. With control channels added to the model, 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 7.1 above, with edges uv, vx, uw, and wx. Every computation of u produces 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 7.3.

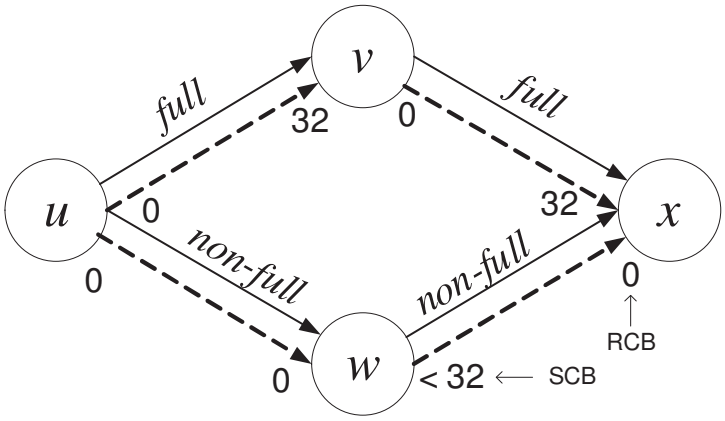

Figure 7.3: A deadlock example. w filters 46 of 64 consumed data tokens, and no other node filters data. Now data channels uv and vx are full, blocking u and v; SCB values for uw and wx are not big enough to prompt credit messages, blocking w and x.

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.

In this example, 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, as we have discussed in the previous chapters. 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.

7.3.2 Avoiding Deadlocks for the Extended SFDF Model

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 schedule dummy messages like we do in the Non-Propagation Algorithm for the basic SFDF model.

The augmented protocol is shown in Algorithm 7.4. 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 $j \geq i$. 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 $[e]$. We do not use *dummy interval* here because the interval is not just for scheduling dummy messages.

Algorithm 7.4: Adding dummy messages to SFDF with control.

```
1  foreach input edge e do
2      while RCB_e = 0 do
3          wait for a control message on q'_e
4          let c be credit value carried by message
5          if c = 0 then
6              if message is a dummy then
7                  break
8              consume message
9          else
10             Detach c credits from message
11             RCB_e ← RCB_e + c
12     if RCB_e > 0 then
13         wait for a data token on q_e
14 let i be least index among tokens on edges with RCB > 0 and dummies on edges with RCB = 0
15 foreach input edge e do
16     if RCB_e > 0 AND token on q_e has index i then
17         consume token
18         RCB_e ← RCB_e − 1
19     else if dummy on q'_e has index i then
20         discard dummy
21 perform computation for index i
22 foreach output edge e do
23     if token is ready on e then
24         emit token on q_e with index i
25         SCB_e ← SCB_e + 1
26         LastSentIdx_e ← i
27     if SCB_e = 0 AND i − LastRecvIdx_e > [e] then
28         emit dummy on q'_e with index i
```

The remaining question is how large to make the heartbeat interval for each edge. We utilize constraints similar to those presented in Chapter 6. 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:

$$\Sigma_{e \in H_1}[e] < \Sigma_{e \in H_2}|q_e| \tag{7.3}$$

$$\Sigma_{e \in H_2}[e] < \Sigma_{e \in H_1}|q_e|. \tag{7.4}$$

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, which do not exist in the basic SFDF model, so the following constraint, which we specified for the sender's protocol of Section 7.2.2, is added for each edge e:

$$[e] < |q_e|. \tag{7.5}$$

Note that this last constraint is not presented in Chapter 6. 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 7.3 *Assuming that control channels never become full, if every node in an SFDF application behaves as Algorithm 7.4 with heartbeat intervals constrained by Inequalities (7.3), (7.4), and (7.5), 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.

In Algorithm 4.3, 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 7.4 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 7.4 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 index 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 7.4 *For any computation history in Γ that follows Algorithm 7.4 with heartbeat intervals as indicated, the mapped computation history in Φ completes without deadlock.*

Proof. We give a proof sketch here. It may be verified that, given Algorithm 7.4 and our mapping, each node of Φ issues dummies in exactly the way directed by the deadlock avoidance protocol for SFDF in 4.3. Moreover, the *dummy intervals* for each edge in Φ are the same as the corresponding *heartbeat intervals* in Γ. It may then be verified that these intervals meet the inequality criteria given by Inequalities 7.3 and 7.4 in Chapter 6 (which are essentially identical to Inequalities 7.3 and 7.4) that guarantee that Φ's computation can never deadlock. ■

Claim 7.5 *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. ■

(Theorem 7.3's proof continues.) Since the ordinary SFDF Γ does not deadlock, Φ does not deadlock, either. □

7.3.3 Verifying Safety of Heartbeat Intervals

We have formalized safe heartbeat intervals, but there are issues remaining. For some DAGs, the number of undirected cycles is exponential in the graph size, so the number of linear constraints is also exponential. For example, an undirected complete graph can be turned into a DAG, which would have 2^N undirected cycles, where N is the number of vertices. Given a set of heartbeat

intervals, verifying them against all those constraints can be very time consuming. Can we verify them in a more efficient way? Another related question is whether we can find extrema of the polyhedron defined by those constraints efficiently. "Efficient" means that the complexity of the algorithm should be polynomial in the graph size.

The issues are similar to those we have in Chapter 6, for which we do not have efficient algorithms. But luckily, Inequality 7.5, which does not exist in the constraints for safe dummy intervals in Chapter 6, precludes any negative cycle in the later defined *HI-graph*, and with that we can design efficient algorithms to detect any violations to Inequality 7.3, 7.4, and 7.5.

We propose an efficient method to verify the safety of a set of heartbeat intervals based on the classic all-pair shortest path (APSP) algorithms. Before we can apply any APSP algorithm, we need to create an auxiliary graph called *HI-graph*.

Definition 7.1 *(**HI-graph** and **Mirror Edge**) Given a dataflow graph $G = (V, E)$, we create a new graph $G' = (V, E')$. For each edge $e = uv \in E$, we create two edges on G': e and $e' = vu$. The weight of e' is the* negative value *of $[e]$, the heartbeat interval assigned for e. G' is the HI-graph (short for* heartbeat interval graph*) for G, and e' and e are* mirror edges *of each other.*

Claim 7.6 *Given a DAG G and its corresponding HI-graph G', Inequalities 7.3, 7.4 and 7.5 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 either 1) an undirected cycle C in G or 2) a single edge with its mirror edge. Suppose all inequalities hold. If case 1) is true, C' has a positive weight according to Inequality 7.3 and 7.4, since the absolute value of the sum of negative edges is less than the sum of the positive edges; if Case 2) is true, Inequality 7.5 guarantees a non-positive cycle is impossible.

($\rightarrow$) Suppose one of the inequalities fails to hold for some undirected cycle C in G. If Inequality 7.5 is violated, an edge and its mirror edge would form a non-positive cycle; for Inequality 7.3 and 7.4, WLOG, suppose Inequality 7.3 is violated, which means the sum of heartbeat interval 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 sum of negative edges on C' is at least the sum of its positive edges, so C' has a non-positive weight. □

To check whether there is a non-positive cycle, we can run an APSP algorithm (e.g. the Floyd-Warshall algorithm [40, 117]) on G'. A non-positive distance from a vertex to itself indicates the existence of a non-positive cycle. Algorithm 7.5 returns `True` if there is a non-positive cycle in a directed graph. The time complexity of the algorithm is $O(|V|^3)$, where $|V|$ is the number of vertices.

Algorithm 7.5: Checking for Non-positive cycle.

```
1  for i ← 1 to n do
2      for j ← 1 to n do
3          if v_i v_j ∈ E then
4              d_ij ← |v_i v_j|
5          else
6              d_ij ← ∞
7  for k ← 1 to n do
8      for i ← 1 to n do
9          for j ← 1 to n do
10             if d_ij < d_ik + d_kj then
11                 d_ij ← d_ik + d_kj
12         if d_ii ≤ 0 then
13             return True
14 return False
```

7.3.4 Finding Extrema of Heartbeat Interval

Given a set of heartbeat interval, if it is safe and incrementing any heartbeat interval would make it unsafe, we say it is an *extremum* of heartbeat interval. To find extrema of the polyhedron defined by the linear constraints, we can start from a set of safe heartbeat intervals (e.g. every heartbeat interval is 0), augmenting the heartbeat interval for each channel in order. We keep the heartbeat intervals safe during the augmentation process.

To augment heartbeat interval $[e]$ for $e = uv$, we create the HI-graph G', then compute the shortest simple path from u to v on $G' - e$ with any shortest path algorithm and set $[e]$ as the weight of this shortest path *minus one*.

Algorithm 7.6: Computing an extremum of a heartbeat interval polyhedron.

```
1 create G' with a mirror edge e' for each edge e
2 set the heartbeat interval of all mirror edges to 0
3 foreach edge e = uv ∈ G do
4     Let d be the length of shortest path from u to v in G' − e
5     [e] ← d − 1
6     update |e'| ← −[e] in G'
```

Theorem 7.7 *Algorithm 7.6 finds an extremum of the polyhedron that defines safe heartbeat intervals.*

Proof. We first prove by induction that after finishing assigning heartbeat intervals, all *directed* cycles (except edge loops) on G' have a positive total length.

Bas. Initially, all heartbeat intervals are set to zero, so all cycles are positive.

Ind. Suppose before assigning the heartbeat interval for $e = uv$, all cycles have a positive total length, and d_{uv} on G'_e is $\alpha + 1$. After setting $[e] = -\alpha$, if there is a non-positive cycle, the cycle must involve e'. Suppose one such cycle is $u, w_1, w_2, \ldots, w_k, v, u$; we can infer that the path $u, w_1, w_2, \ldots, w_k, v$ has a length $\leq \alpha$, which contradicts the fact that the shortest directed path between u and v on G'_e is $\alpha + 1$.

Since all directed cycles are positive, we know the constraints for assigning heartbeat intervals are not violated.

Next we prove the set of heartbeat intervals chosen by Algorithm 7.6 is an extremum of the polyhedron. Indeed, say we set $[e]$ based on some cycle C, increasing any heartbeat interval in

C would lead to a cycle of zero or negative total length, which means a violation of the linear constraints. □

Since Algorithm 7.6 calls a shortest path algorithm for every edge, with the classic Dijkstra's algorithm, the algorithm can run in $O(|E|^2 + |V||E|log|V|)$, where E and V are the number of edges and the number of vertices, respectively.

Note that the augmentation process has only one pass. We make a linear constraint tight immediately after incrementing one heartbeat interval, which may cause a large variance of the computed heartbeat intervals. We can reduce the variance by using a multi-pass augmentation process. For example, when a heartbeat interval needs to be increased, we increase it by half the of maximum value that it may be increased by. A similar strategy also applies to decrementing steps until all constraints are tight.

Besides augmenting from a set of *safe* heartbeat intervals, we can also find extrema starting from *any* set of heartbeat intervals. The idea is similar, except that if we find a non-positive cycle, we reduce the heartbeat intervals of involved edges until all cycles are positive; we can then apply Algorithm 7.6.

With the ability to check for constraint violations and compute extrema, we can assign heartbeat intervals that are "large enough." For example, for each channel, we can set a minimum value of heartbeat interval. If the minimum values constitute a safe set of heartbeat intervals, we can further augment them to reduce heartbeat messages during runtime.

7.3.5 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 7.1. 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 7.4, 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, similar to the behavior of the Naive Algorithm for the ordinary SFDF. The downside of this approach is the overhead caused by frequent dummy messages and credit messages, but it can guarantee the freedom of deadlock.

Claim 7.8 *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 will not be blocked indefinitely by an output message channel; it sends at most one data token, so it will not 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 $i+1$. □

Theorem 7.9 *If every node in an SFDF application behaves as in Algorithm 7.4 using heartbeat interval* 0 *for every edge, the application cannot deadlock, even if the control messages sent per data token are unbounded.*

Proof. According to Claim 7.8, all nodes can advance to the same computation index as source nodes do, which means no deadlock can happen. □

7.4 Experimental Evaluation

We have implemented support for precisely ordered control messages in filtering SFDF on top of Auto-Pipe [41]. To evaluate the performance impact of filtering and control messages, we implemented the streaming application for computing variance described in Figure 7.1.

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

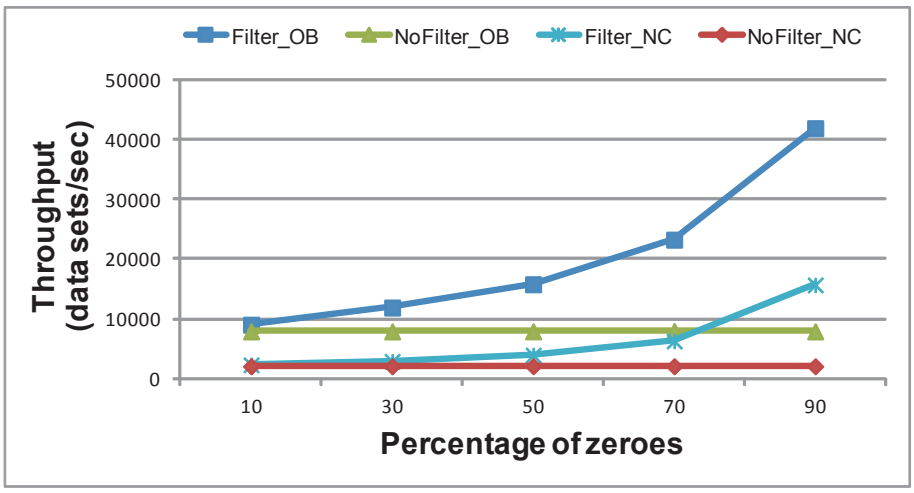

Figure 7.4: Throughput of variance application vs. rate of filtering (heartbeat interval = 16). Filter_-OB, NoFilter_OB, Filter_NB, NoFilter_NB represent: filtering w/ output buffer, non-filtering w/ output buffer, filtering w/o output buffer, and non-filtering w/o output buffer.

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 7.4 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.

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.

7.5 Summary

In this chapter, we extended our SFDF model to include precise synchronization between data streams and general control messages. In the extended model, each data channel is associated with a control channel. We designed a credit-based protocol, which features credit messages, to support the synchronization. To avoid deadlocks, we modified the node behavior to schedule both dummy messages and credit messages appropriately.

With the support of general control messages, we can convert some non-filtering applications to filtering ones for performance improvement. We demonstrated that by filtering unnecessary data in a variance application, the throughput can be improved in proportion to the filtering ratio.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this dissertation, we proposed a new dataflow model: synchronized filtering dataflow (SFDF), where nodes must synchronize input data and might filter output data. In SFDF, each data token is associated with a data index, which is strictly increasing in a data stream. To synchronize input, during a computation, a node consumes only data tokens with the same index and all tokens with the same input must be consumed during one computation. A node can filter output data by producing no token on some of its output channels. The dataflow graph of an SFDF application is a directed acyclic multigraph (DAMG), which has no directed cycle. However, a dataflow graph can have *undirected* cycles, which, together with the synchronization and filtering behaviors, can cause deadlocks in the application.

To avoid deadlocks, we augment data streams with *dummy messages*, which are a kind of special tokens. A dummy message carries an index but no data of its own. It is purpose is to notify the receiver that the sender has finished computing on tokens with the index. By augmenting data streams with dummy messages, we can successfully avoid deadlocks. But questions remain as to when dummy messages should be inserted into data streams. By inserting a dummy message for every filtered data token, we convert an SFDF application into a homogeneous dataflow application, which is deadlock free, but it would send many unnecessary dummy messages and incur performance overhead. On the other hand, if too few dummy messages are inserted into

data streams, we might not be able to avoid deadlocks. It is challenging to insert as few dummy messages as possible while still avoiding deadlocks.

Trying to minimizing the number of dummy messages inserted into data streams, we designed two *decentralized* algorithms: the Propagation Algorithm and the Non-Propagation Algorithm. Both algorithms have a compile-time part and a runtime part. During compilation time, both algorithms compute a *dummy interval* for each data channel; during runtime, both algorithms insert dummy messages according to the computed dummy intervals. The Propagation Algorithm requires every dummy message to be propagated all the way to sink nodes, while the Non-Propagation Algorithm never mandates propagation beyond the immediate receiver, so the dummy intervals computed by the two algorithms are different. Although the two algorithms are not directly comparable, in most cases we expect the Non-Propagation Algorithm to insert many fewer dummy messages than the Propagation Algorithm because the Non-Propagation Algorithm utilizes filtering history while the Propagation Algorithm does not. This hypothesis is supported by our experiments.

The runtime parts of our algorithms are efficient, adding little overhead to computing nodes, but computing dummy intervals involves enumerating all undirected cycles in the dataflow graph, which could be time-consuming for general DAMGs because the number of undirected cycles could be exponential in the graph size. We have observed that practical streaming applications have special topologies, for which we can design efficient algorithms to compute dummy intervals. In particular, we focus on two topologies: series-parallel DAGs (SP-DAGs) and CS4 DAGs. SP-DAG is a well-studied topology that is constructed by repeatedly applying two kinds of compositions: serial composition and parallel composition. CS4 DAG is a superclass of SP-DAGs discovered by us, where each undirected cycle is single-source and single-sink. We designed efficient algorithms to compute dummy intervals on SP-DAGs and CS4 DAGs without changing the runtime behavior of nodes.

We further extended our work on selecting dummy intervals. Rather than providing one set of safe dummy intervals, we proposed a set of polyhedral constraints to define all sets of safe

dummy intervals, which gives us more flexibility to choose dummy intervals. We showed that for SP-DAGs, the number of constraints is polynomial to the graph size.

In our deadlock avoidance algorithms, we rely on the usage of dummy messages, which is only one type of *control messages* that are used by streaming applications. To support more general control messages, we extended the SFDF model to synchronize data streams and control messages precisely with a carefully designed protocol. We showed that the extended model can not only incorporate dummy messages to guarantee application correctness but also improve performance of some applications by facilitating the conversion of a non-filtering application to a filtering application.

8.2 Future Work

There are several promising future directions for this dissertation; we list some interesting ones.

Extend SFDF to support *directed* cycles. The SFDF model does not support directed cycles in dataflow graphs. To support directed cycles, we first need to extend node behavior to support the synchronization between *ordinary* channels and *feedback* channels during the first few computations when feedback channels do not have data. There are several possible ways. We can place some initial dummy messages in feedback channels, or the node may ignore feedback channels for the first few computations. We need to evaluate pros and cons of those approaches. We also need to avoid potential deadlocks cause by directed cycles, which do not exist in ordinary SFDF dataflow graphs. A directed cycle of all full channels or all empty channels could lead to such deadlocks. Potential deadlock avoidance approaches could use dummy messages, but the schedule is expected to be different from the ones for DAGs.

Study the impact of output buffers. Output buffers can improve throughput of streaming applications. When a data token in placed in an output buffer, it cannot reach the receiver unless the output buffer is flushed. If output buffers are flushed timely, they do not impact the correctness of streaming applications. However, if a node gets blocked before flushing its output buffers, the

buffered tokens might never reach their receivers, and there is the risk of potential deadlock. For example, in Figure 3.2, if channel uw does not filter data tokens but has a very large output buffer, then when uv and vx are full, u still has not flushed output buffer uw, and a deadlock happens. We would like to extend the SFDF model to include output buffers and study bounded-memory execution under the extended model.

Use dummy messages to improve latency. SFDF applications with small buffer sizes face deadlock threat. If buffer sizes are sufficiently large for an application, while deadlock is unlikely to happen in this scenario, the application might suffer from long data latency. For example, in an application with split-join structure like Figure 1.8, if one path from the source to the sink filters most of the data, even there is no deadlock, the sink node has frequent long waits at the empty channels while other channels have accumulated many data tokens. For this application scenario, we would like to study how dummy messages can be used to reduce latency without sacrificing much throughput.

Communication compression. In Chapter 7, we have demonstrated how control messages help filter unnecessary data and reduce communication and computation for a particular application. A further interesting question is whether we can design a general scheme to compress data tokens between nodes. The compression scheme should be in the domain of streaming processing so it can align well with streaming computing. A preliminary study of the classic LZ77 [123] and LZ78 [124] algorithms has found some related streaming features. We will further investigate the problem to look for a compression scheme, which could be based on LZ77 and LZ78.

References

[1] Apache Storm. http://storm.incubator.apache.org/. Accessed: June 30, 2014.

[2] Java I/O streams. http://docs.oracle.com/javase/tutorial/essential/io/streams.html. Accessed: June 30, 2014.

[3] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the Borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[4] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, August 2003.

[5] 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.

[6] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. STREAM: The stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 665–665. ACM, 2003.

[7] Edward A. Ashcroft and William W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20(7):519–526, 1977.

[8] Magdalena Balazinska, Hari Balakrishnan, Samuel R. Madden, and Michael Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. *ACM Transactions on Database Systems (TODS)*, 33(1):3, 2008.

[9] Anne Benoit, Paul Renaud-Goud, and Yves Robert. Performance and energy optimization of concurrent pipelined applications. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[10] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *The Journal of VLSI Signal Processing*, 21(2):151–166, 1999.

[11] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cycle-static dataflow. *Signal Processing, IEEE Transactions on*, 44(2):397–408, 1996.

[12] Geoffrey Blake, Ronald G Dreslinski, and Trevor Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26–37, 2009.

[13] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. Graphics*, 23(3):777–786, 2004.

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

[15] 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.

[16] Joseph T. Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *Int. Journal in Computer Simulation*, 4(2):0–, 1994.

[17] Jeremy Buhler, Joseph M. Lancaster, Arpith C. Jacob, and Roger D. Chamberlain. Mercury BLASTN: Faster DNA sequence comparison using a streaming hardware architecture. In *Proc. Reconfigurable Systems Summer Institute*, Urbana, IL, July 2007.

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

[19] Rob Cameron, Tom Shermer, Arrvindh Shriraman, Ken Herdy, Dan Lin, Ben Hull, and Meng Lin. Bitwise data parallelism in regular expression matching. In *The 23rd International Conference on Parallel Architectures and Compilation Techniques*, August 2014.

[20] Roger D. Chamberlain, Mark A. Franklin, Eric J. Tyson, James H. Buckley, Jeremy Buhler, Greg Galloway, Saurabh Gayen, Michael Hall, E. F. Berkley Shands, and Naveen Singla. Auto-Pipe: Streaming applications on architecturally diverse systems. *Computer*, 43(3):42–49, 2010.

[21] 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.

[22] Sirish Chandrasekaran and Michael J Franklin. Streaming queries over streaming data. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 203–214. VLDB Endowment, 2002.

[23] 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.

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

[25] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stanley B Zdonik. Scalable distributed stream processing. In *CIDR*, volume 3, pages 257–268, 2003.

[26] Ron Cytron, Michael Hind, and Wilson Hsieh. Automatic generation of DAG parallelism. In *ACM SIGPLAN Notices*, volume 24, pages 54–68. ACM, 1989.

[27] William J. Dally, Ujval J. Kapasi, Brucek Khailany, Jung Ho Ahn, and Abhishek Das. Stream processors: Progammability and efficiency. *Queue*, 2(1):52, 2004.

[28] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J Knight, et al. Merrimac: Supercomputing with streams. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35. ACM, 2003.

[29] Erwin A. de Kock, WJM Smits, Pieter van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, Paul Lieverse, Kees A. Vissers, and Gerben Essink. YAPI: Application modeling for signal processing systems. In *Proceedings of the 37th Annual Design Automation Conference*, pages 402–405. ACM, 2000.

[30] Marco de Vos, Andre W. Gunst, and Ronald Nijboer. The LOFAR telescope: System architecture and signal processing. *Proceedings of the IEEE*, 97(8):1431–1437, 2009.

[31] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John Lockwood. Deep packet inspection using parallel Bloom filters. In *High Performance Interconnects, 2003. Proceedings. 11th Symposium on*, pages 44–51. IEEE, 2003.

[32] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1:115–138, 1971.

[33] Edsger W. Dijkstra. Two starvation-free solutions of a general exclusion problem. 1977.

[34] Matthew Drake, Henry Hoffmann, Rodric Rabbah, and Saman Amarasinghe. MPEG-2 decoding in a stream programming language. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE, 2006.

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

[36] Marc Engels, Greet Bilson, Rudy Lauwereins, and Jean Peperstraete. Cycle-static dataflow: Model and implementation. In *Signals, Systems and Computers, 1994. 1994 Conference Record of the Twenty-Eighth Asilomar Conference on*, volume 1, pages 503–507. IEEE, 1994.

[37] 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.

[38] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Automata, Languages and Programming*, pages 17–44, 2004.

[39] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2):207–216, 2005.

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

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

[42] Richard Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, 1990.

[43] Marc Geilen and Twan Basten. Requirements on the execution of Kahn process networks. In *12th European Symposium on Programming*, pages 319–334, 2003.

[44] Maya Gokhale, Jan Stone, Jeff Arnold, and Mirek Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pages 49–56. IEEE, 2000.

[45] Donald Gross and Carl Harris. Fundamentals of queueing theory. 1998.

[46] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[47] Mark Harris. Mapping computational concepts to GPUs. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[48] 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.

[49] Richard C. Holt. Comments on prevention of system deadlocks. *Communications of the ACM*, 14(1):36–38, 1971.

[50] Philip KF Hölzenspies, Johann L Hurink, Jan Kuper, and Gerard JM Smit. Run-time spatial mapping of streaming applications to a heterogeneous multi-processor system-on-chip (mpsoc). In *Proceedings of the conference on Design, automation and test in Europe*, pages 212–217. ACM, 2008.

[51] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, et al. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *ACM SigPlan notices*, 27(5):1–164, 1992.

[52] Jeong-Hyon Hwang, Magdalena Balazinska, Alex Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 779–790. IEEE, 2005.

[53] 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.

[54] Christopher T. Johnston and Donald G. Bailey. FPGA implementation of a single pass connected components algorithm. In *Electronic Design, Test and Applications, 2008. DELTA 2008. 4th IEEE International Symposium on*, pages 228–231. IEEE, 2008.

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

[56] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Journal of Fluids Engineering*, 82(1):35–45, 1960.

[57] Ujval J Kapasi, William J Dally, Scott Rixner, John D Owens, and Brucek Khailany. The Imagine stream processor. In *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*, pages 282–288. IEEE, 2002.

[58] Richard M. Karp and Rayamond E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.

[59] 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.

[60] B.K. Khailany, T. Williams, J. Lin, E.P. Long, M. Rygh, D.W. Tovey, and W.J. Dally. A programmable 512 GOPS stream processor for signal, image, and video processing. *Solid-State Circuits, IEEE Journal of*, 43(1):202–213, Jan 2008.

[61] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, 2011.

[62] Joseph M. Lancaster, Jeremy Buhler, and Roger D. Chamberlain. Efficient runtime performance monitoring of FPGA-based applications. In *Proc. of 22nd IEEE Int'l System-on-Chip Conf. (SoCC)*, pages 23–28, Belfast, Northern Ireland, UK, Semptember 2009.

[63] Paul Le Guernic, Albert Benveniste, Patricia Bournai, and Thierry Gautier. Signal–a data flow-oriented language for signal processing. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 34(2):362–374, 1986.

[64] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[65] Edward A Lee and Thomas M Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[66] 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.

[67] Peng Li, Kunal Agrawal, Jeremy Buhler, and Roger D. Chamberlain. Deadlock avoidance for streaming computations with filtering. In *ACM Symp. on Parallelism in Algorithms and Architectures*, 2010.

[68] 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.

[69] 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.

[70] 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.

[71] Weiguo Liu, Bertil Schmidt, Gerrit Voss, and Wolfgang Muller-Wittig. Streaming algorithms for biological sequence alignment on GPUs. *Parallel and Distributed Systems, IEEE Transactions on*, 18(9):1270–1281, 2007.

[72] Vijay Madisetti. *Digital signal processing fundamentals*. CRC press, 2010.

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

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

[75] Paul E. McKenney. Is parallel programming hard, and, if so, what can you do about it? *Linux Technology Center, IBM Beaverton*, 2014.

[76] Raymond E. Miller. A comparison of some theoretical models of parallel computation. *Computers, IEEE Transactions on*, 100(8):710–717, 1973.

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

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

[79] Gordon E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *Solid-State Circuits Newsletter, IEEE*, 11(5):33 –35, sept. 2006.

[80] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data processing on FPGAs. *Proceedings of the VLDB Endowment*, 2(1):910–921, 2009.

[81] Fabrizio Mulas, David Atienza, Andrea Acquaviva, Salvatore Carta, Luca Benini, and Giovanni De Micheli. Thermal balancing policy for multiprocessor stream computing platforms. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(12):1870–1882, 2009.

[82] Angeles Navarro, Rafael Asenjo, Siham Tabik, and Calin Cascaval. Analytical modeling of pipeline parallelism. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 281–290. IEEE, 2009.

[83] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

[84] John Nickolls and William J Dally. The gpu computing era. *IEEE micro*, 30(2):56–69, 2010.

[85] CUDA Nvidia. Compute unified device architecture programming guide. 2007.

[86] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical report, 2004.

[87] Alex G. Olson and Brian L. Evans. Deadlock detection for distributed process networks. In *IEEE Int'l Conf. Acoustics, Speech, Signal Processing*, volume 5, pages v/73–v/76 Vol. 5, March 2005.

[88] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[89] John D. Owens, Scott Rixner, Ujval J. Kapasi, Peter Mattson, Brian Towles, Ben Serebrin, and William J. Dally. Media processing applications on the Imagine stream processor. In *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*, pages 295–302. IEEE, 2002.

[90] Shobana Padmanabhan, Yixin Chen, and Roger D Chamberlain. Optimal design-space exploration of streaming applications. In *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on*, pages 227–230. IEEE, 2011.

[91] Shobana Padmanabhan, Yixin Chen, and Roger D Chamberlain. Convexity in non-convex optimizations of streaming applications. In *ICPADS*. IEEE, 2012.

[92] Shobana Padmanabhan, Yixin Chen, and Roger D Chamberlain. Unchaining in design-space optimization of streaming applications. *Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM)*, 2013.

[93] Chanik Park, Jaewoong Jung, and Soonhoi Ha. Extended synchronous dataflow for efficient dsp system prototyping. *Design Automation for Embedded Systems*, 6(3):295–322, 2002.

[94] Jongsoo Park and William J. Dally. Buffer-space efficient and deadlock-free scheduling of stream applications on multi-core architectures. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 1–10. ACM, 2010.

[95] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, Berkeley, Dec 1995.

[96] Thomas M. Parks, José Luis Pino, and Edward A. Lee. A comparison of synchronous and cycle-static dataflow. In *Signals, Systems and Computers, 1995. 1995 Conference Record of the Twenty-Ninth Asilomar Conference on*, volume 1, pages 204–210. IEEE, 1995.

[97] Daniel Pilaud, N Halbwachs, and JA Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*, volume 178, page 188, 1987.

[98] Srinivas K Raman, Vladimir Pentkovski, and Jagannath Keshava. Implementing streaming SIMD extensions on the Pentium III processor. *IEEE Micro*, 20(4):47–57, 2000.

[99] Martino Ruggiero, Alessio Guerri, Davide Bertozzi, Michela Milano, and Luca Benini. A fast and accurate technique for mapping parallel applications on stream-oriented mpsoc platforms with communication awareness. *International Journal of Parallel Programming*, 36(1):3–36, 2008.

[100] Kentaro Sano, Oliver Pell, Wayne Luk, and Satoru Yamamoto. FPGA-based streaming computation for lattice boltzmann method. In *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*, pages 233–236. IEEE, 2007.

[101] Vivek Sarkar. *Partitioning and scheduling parallel programs for multiprocessors*. MIT press, 1989.

[102] S. Schneider, M. Hirzel, B. Gedik, and K. Wu. Safe data parallelism for general streaming. *Computers, IEEE Transactions on*, PP(99):1–1, 2013.

[103] G. C. Sharp, N. Kandasamy, H. Singh, and M. Folkert. GPU-based streaming architectures for fast cone-beam CT image reconstruction and demons deformable registration. *Physics in medicine and biology*, 52(19):5771, 2007.

[104] N. Singla, M. Hall, B. Shands, and R.D. Chamberlain. Financial monte carlo simulation on architecturally diverse systems. In *High Performance Computational Finance, 2008. WHPCF 2008. Workshop on*, pages 1 –7, nov. 2008.

[105] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230. ACM, 2012.

[106] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[107] Jaspar Subhlok and Gary Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 62–71. ACM, 1996.

[108] S. Thakkur and Thomas Huff. Internet streaming SIMD extensions. *Computer*, 32(12):26–34, 1999.

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

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

[111] 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 AutoPipe design system. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 595(2):474–479, 2008.

[112] Jacobo Valdes, Robert E. Tarjan, and Eugene L. Lawler. The recognition of series parallel digraphs. In *ACM Symposium on Theory of Computing*, 1979.

[113] P. Viola and M. Jones. Robust real-time object detection. *International Journal of Computer Vision*, 57(2), 2004.

[114] Nagavijayalakshmi Vydyanathan, Umit V Catalyurek, Tahsin M Kurc, Ponnuswamy Sadayappan, and Joel H Saltz. Toward optimizing latency under throughput constraints for application workflows on clusters. In *Euro-Par 2007 Parallel Processing*, pages 173–183. Springer, 2007.

[115] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 176–188, New York, NY, USA, 1991. ACM.

[116] Yi Wang, Duo Liu, Meng Wang, Zhiwei Qin, and Zili Shao. Optimal task scheduling by removing inter-core communication overhead for streaming applications on mpsoc. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, pages 195–204. IEEE, 2010.

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

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

[119] Xuejun Yang, Xiaobo Yan, Zuocheng Xing, Yu Deng, Jiang Jiang, and Ying Zhang. A 64-bit stream processor architecture for scientific applications. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 210–219, New York, NY, USA, 2007. ACM.

[120] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 10–10. USENIX Association, 2012.

[121] Jian Zhang. A survey on streaming algorithms for massive graphs. *Managing and Mining Graph Data*, pages 393–420, 2010.

[122] Jun Zhu, Ingo Sander, and Axel Jantsch. Energy efficient streaming applications with guaranteed throughput on mpsocs. In *Proceedings of the 8th ACM international conference on Embedded software*, pages 119–128. ACM, 2008.

[123] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[124] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536, 1978.