

Applications and Execution of Stream Graphs

Nathaniel Duca

Submitted in partial completion of a
Senior Undergraduate Thesis
in the
Department of Computer Science
Johns Hopkins University

May 18, 2004

The main contributions of this thesis are: (1) proposing the stream graph token flow model, a novel high-efficiency and fully general programming model for stream processing problems, (2) providing a modification to queue-based scheduling that allows intelligent and fast stream graph scheduling, and (3) providing a dynamic linking strategy that allows efficient runtime execution of stream graphs.

Contents

1	Introduction	2
2	The Stream Graph Model	4
2.1	Scheduling Order	4
2.2	Signature Types	5
3	Background	6
3.1	Dataflow	6
3.2	Data Stream Processing	7
3.3	General Stream Processing	7
3.3.1	Stream processing in graphics and multimedia	7
3.3.2	Stream Processing Languages	9
3.3.3	General purpose stream processing hardware	9
3.4	Summary	10
4	Using Stream Graphs	11
4.1	Dynamic Data Flow with Network Modification	11
4.1.1	Implementing Uniform Parameters without Dynamic Data Flow	12
4.2	Multi-Pass Stream Graphs	13
4.3	Optimization	14
4.3.1	Low-level Optimizations	15
4.3.2	Optimization using Graph Modification	15
4.3.3	Optimization using Graph Partitioning	16
5	Executing Stream Graphs: Fast-update Schedulers	18
5.1	Background	18
5.2	Graph Traversal and Bindings	19
5.3	Basic Binding Scenarios	20
5.3.1	The Effects of Direct versus Indirect Binding	21
5.3.2	Representing Firing Conditions	21
5.4	Preventing Direct Binding Deadlock	22
5.5	Concurrency Issues	23
5.6	Schedule Caching	24
5.7	Periodic Schedules and Scheduler Thrashing	25
6	Efficient Implementation of Runtime Scheduling Changes	27
6.1	Node Overloading	28
6.2	Automatically Generating Overloaded Nodes using Linking	29
7	Conclusion and Future work	32
7.1	Special Thanks	33

Chapter 1

Introduction

Stream processing has received attention in a variety of fields. In the streaming algorithms field, stream processing is viewed as an algorithmic problem posed originally in [22]: solve a problem by processing inputs in one pass and using poly-logarithmic memory. A wide variety of approximate solutions have been shown within this model for a variety of basic problems. Babcock et al[5] provide an excellent survey of this extensive field. Streaming database research uses the one-pass nature of streaming algorithms as the basis of a simple acyclic token flow network used for continuous query processing [11]. Streaming databases, as they have been called, strongly resemble token-flow models [25, 18, 31] wherein the entire computation is described as a graph of simple operators that communicate exclusively through token passing. Does stream processing introduce new challenges or results into the dataflow problem? Streaming database systems do not, in the sense that (1) they operate only on digraphs and (2) the operators in a streaming database do not have to be processed in the order in which they are received. This makes most dataflow algorithms uninteresting on stream graphs. Two key differences however, make streaming databases unique compared to dataflow systems: the nodes in the processing graph have RAM, and as a result, can perform extensive computations on their input and output streams.

Since the advent of streaming databases (or perhaps in parallel to), a wide variety of other fields have begun research into so-called stream processing models. For example, graphics accelerators are designed to operate on a sequential set of vertex inputs to produce, shade, and commit a large number of fragments (pixels) to the frame buffer. This is believed to be streaming [35] because (1) the process is a pipeline and (2) each node in the pipeline is programmable and (3) the input source and output destination of the pipeline is programmable as well. While the connection between streaming and graphics may appear tenuous, the performance numbers alone justify a certain amount of hype. Whereas graphics hardware executes a two-node stream-processing pipeline, customized hardware has been designed [41, 28, 14] with the execution of token-flow graphs explicitly in mind. Programming these and other cutting-edge systems has become a hot topic: StreamC [28], StreamIT[45], CG [35] and Brook[9] are programming languages that compile a dataflow-like graph to operate on different variations of these systems.

Each of the programming languages mentioned before is tightly coupled with a hardware platform of its own: StreamC to Imagine, StreamIt to MIT RAW[21], CG to the NVidia GeForce product, and Brooke to both graphics cards and the streaming supercomputer project at Stanford[14]. As a result, hardware features reflect the needs of the associated language team, and similarly, the language features reflect the abilities of the underlying hardware. In seeking to be executable on cutting-edge hardware, all stream programming languages have a fundamentally limited worldview about what is, and what is not, streaming.

Let me give a simple example to motivate the rest of this thesis: CG, StreamC, and Brooke all view the stream processing problem as two processes: the data processing part which does the actual work and the controller, which issues commands to the data processor to load, delete and modify the graph being executed at a given moment. This makes sense in this context because these languages operate on client-server hardware: a CPU issues commands to a separate board that contains the streaming hardware. In contrast, StreamIT is meant to execute on an array of superscalar processors. In this environment, the distinction between the “controlling process” and the “worker processes” is arbitrary at best. The point? The designers of stream programming languages have not considered all of the possibilities for stream programming models.

In this thesis, I propose the stream graph as a model for general purpose computation with streams. My thesis focuses on (1) showing ways in which the model is better than past models, (2) showing how it can be efficiently

scheduled and (3) efficiently executed. In Chapter 2, I present the stream graph model and compare it to past stream and dataflow computational models. Chapter 3 provides more extensive background on the different fields and systems that are referenced in discussion of stream graphs. Chapter 4 uses this background to show the many different applications and uses of stream graphs.

In the single-processor context, stream graph execution must be viewed as a dataflow-like scheduling problem. Scheduling algorithms for standard dataflow problems execute too slowly to be used in dataflow graphs. In Chapter 5, I introduce a set of techniques to speed-up the stream-graph scheduling process.

Stream graphs cannot take advantage of compile-time optimizations in order to coax performance out of their code because they change at runtime. This puts stream graphs one step behind their static graph counterparts. To bridge this gap, Chapter 6 proposes an elegant dynamic-linking solution for stream graph code that allows the execution of stream graphs created at runtime to execute as fast as statically compiled code. Finally, Chapter 7 closes this thesis with a discussion of future research that might arise from this thesis with a particular emphasis on my planned stream graph implementation.

Chapter 2

The Stream Graph Model

A stream graph is a directed graph in which each vertex (or node) corresponds to a computation and each edge corresponds to a queue of tokens. A vertex, when it fires, may consume any number of tokens from any subset of its input edges and similarly produce tokens onto its output edges. Because of this, the rate of tokens along an edge is different at each end of the edge. We call each end of an edge a port. With this term, we can clarify our definition: the number of tokens consumed per port per firing is variable. Note that a node is not required to consume or produce a token on every firing.

Communication between vertices happens through token passing. Thus, tokens are not simply impulses that are counted but instead contain meaningful data.

A vertex has a local memory that can be used during the computation. A vertex may, while firing, dynamically modify any portion of the stream graph in its immediate neighborhood. The following operations can be applied to the graph in order to affect a change:

Create a node

Create an edge

Re-bind an edge : Allows movement of an edge from without losing its tokens

Redefine a node : Allows you to change a node function without affecting the graph or the node's memory.

Executing a stream graph is a scheduling problem: one chooses an order in which to execute the vertices in the stream graph such that if the graph is known to be able to execute in bounded memory, then the schedule chosen must execute the graph in bounded memory. Note that although the former problem has no solution, the latter problem is simply one of fairness: a bad scheduler might starve nodes from execution, leading to token buildup along the deadlocked edges.

Consider a node that requires one token on each input edge to fire: if one of its input edges is empty then the node cannot fire. Thus, the arrival of a token on that edge will activate the node. Viewed this way scheduling is the process of detecting node activations in an efficient way. Finding active nodes is difficult in an unrestricted token flow graph. Because a node can fire at will, the only way to find an active node is to fire it. Scheduling of this sort of graph is rarely efficient but always possible. Instead of restricting the firing conditions allowed for nodes as done in past dataflow research[31], stream graphs solve the problem passively using signatures.

In stream graphs, signature information can be added to any part of the graph in order to make the direction and rate of token flow visible to the scheduler. They can be used by the scheduler to anticipate activation and thus efficiently generate a long or short-term schedule for the graph. On paper, signatures can be any unambiguous sentence written next to nodes or vertices in the graph that makes the firing criteria and output behavior of the node more predictable than the plain unsigned node. Like graph topology, stream graph signatures can be changed at runtime.

2.1 Scheduling Order

Kahn's condition for communicating processes [25] is a critical test applied to models like the stream graph. If Kahn's condition is met for a given graph then the order of execution will not affect the computation being performed. This

has important implications for scheduling algorithms: if Kahn's condition is met, then the scheduler is free to execute nodes in whatever order it chooses without fear of affecting the computation.

Stream graphs do not meet Kahn's condition because of their self-modifying abilities. Simple graphs can be created that with self-modifying behavior that will lead to different computations depending on the order in which the nodes are fired. Although this property may seem deadly at first glance, this merely means that synchronization may be necessary when performing graph modifications. Yet like the problem of deadlock in other models, synchronization is not necessary for a well-designed graph. A stream graph programmer must work with the same care as any other parallel programmer: if code is written that modifies the graph, it must be done carefully to guarantee order-independent execution..

2.2 Signature Types

What do signatures for a stream graph look like? Usually, signatures for a graph are placed per-port and per-node. A port signature usually states a token rate for that port. For example, a port signature might say that the node always consumes one token from the port when it fires. Similarly, a node usually maps from the presence of tokens to the activation of the node. For example, a node signature might say that the node will fire only when tokens are present on both ports, or similarly that the node always produces output when it fires.

Unlike other token flow models, the stream graph model does not explicitly define a convention or format for the signatures in a graph. In synchronous data flow all ports are signed with an integer representing the number of tokens consumed per firing over that port. Stream graphs do not do this because this choice precludes a variety of predictable behaviors because they do not reduce to a single integer, for example a node that alternatively reads one and four tokens from an input port on opposing firings. A similar situation arises with node signatures: in synchronous data flow, a node fires only when all of its per-port token requirements are met. This precludes a wide variety of behaviors that are predictable but do not reduce to this all-or-nothing behavior. While this choice makes formal representation and analysis of stream graphs difficult, the result is to provide a system that can cope with a wider variety of data flow problems without corresponding losses in performance.

Chapter 3

Background

A number of systems have been created in the past few decades that have influenced and are implementable in the stream graph model. In the following chapters, I refer to and analyze past work in a wide variety of fields. The purpose of this section is to provide enough background to see the diversity that surrounds stream processing ideas. This chapter exists to point out the salient features of different fields as they relate to stream graphs and thus provide a foundation for later discussion within the paper. The chapter is organized into the following sections:

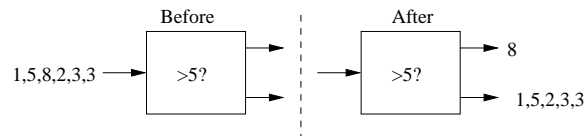
- Dataflow background: what is known about token flow models?
- Data stream processing: analysis and mining of large data sets using token-flow-like models
- General stream processing: use of token networks in different fields where the model differs substantially from the early data flow models of token-flow. The different fields surveyed are:
 - Graphics and multimedia
 - Programming Languages
 - General purpose hardware

In each section, I provide pointers to pertinent literature, a commentary on how the field interacts with graph-like programming models and most importantly, an explanation of how the terminology in this paper relates to the terminology used in the particular area. Thus, useful information should be present in this chapter even to one already familiar with the particular field.

3.1 Dataflow

Dataflow research has usually focused on more restricted graphs than stream graphs. In particular, nodes rarely have memory and port-rates are well defined. Initial dataflow systems [18, 17] were created as alternatives to the von-Neumann architecture. Each node in the graph (“actors” in dataflow literature) corresponds to a simple arithmetic or conditional operation. Later dataflow systems [31, 32] were used for digital signal processing and often compiled the input dataflow graph into machine code that executed on one or multiple processors.

Data-dependent flow, the problem of routing a token depending on its contents, is extremely difficult to efficiently schedule in a dataflow network. A classic data-dependent flow example is an “if” node:



This node consumes a single integer per firing, and routes it to a different input depending on a boolean test on the input. With limited exception[10] the only efficient solution to data-dependent flow is to fallback to a round-robin (or slightly more sophisticated) scheduler. Although stream graphs do not solve the dynamic data flow problem, they can provide very efficient solutions to a limited but very commonly occurring set of dynamic data flow problems. I will show this in section 4.1.

3.2 Data Stream Processing

Stream processing was introduced in [22] as the class of algorithms that perform only one sequential pass over their input data and require memory poly-logarithmic in the size of the input stream. This has led to a flurry of research papers on this subject, surveyed in [36, 5].

Stream processing algorithms are used extensively in data stream systems and streaming database management systems[11, 34, 33, 4, 37, 20, 42, 13, 2, 12, 19]. These systems construct a graph where edges contain data tuples and nodes represent filters over the input tuples as shown in figure 3.2. These nodes usually execute streaming algorithms in order to compute their answers. A variety of interesting optimizations have been introduced for these systems that leverage the fact that the nodes in the graph can be rearranged for memory and CPU usage criteria without affecting the overall computation [6, 30, 46, 27, 3, 39]¹

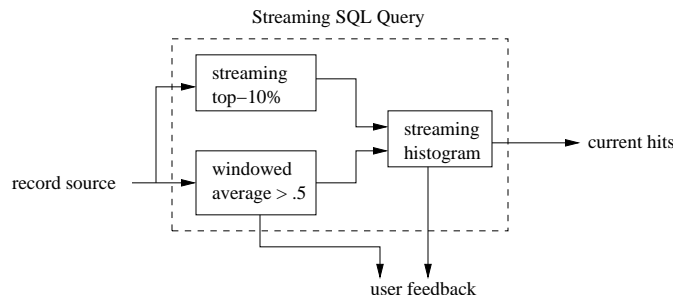


Figure 3.1: Streaming databases combine streaming algorithms with dataflow-like graphs. Token and operator ordering in streaming databases is arbitrary, allowing a wide variety of optimizations not possible in classical dataflow graphs.

3.3 General Stream Processing

The notion of stream processing has been adopted in a number of fields to mean much more than originally intended in the early stream processing literature. As a result, algorithmic stream processing (the original and true stream processing) is better seen as asymptotic stream processing while the general “stream processing” term can be seen as ambiguously referring to any system that processes large quantities of data using a token flow graph. Often times a simple element of runtime decision making — be it dynamic modification, creation or decision making — plays a major role in the computation thus setting the approach apart standard data flow techniques.

3.3.1 Stream processing in graphics and multimedia

The graphics community realized at some point that the graphics processor could be thought of in terms similar to streaming database systems. A simplified graphics card is shown in figure 3.3.1. Although the graph that it executes is fixed, different programs can be loaded into the vertex and fragment processors using CG[35] and similar programming languages. By using CG and OpenGL/DirectX to feed the output of the pipeline back into the pipeline at strategic places, a wide variety of computations can be performed[8, 29, 9].

This abstraction has been carried to a higher level in parallel rendering research with Chromium[23]. This system abstracts the entire OpenGL pipeline as a DAG of stream processing units (SPUs), as shown in figure 3.3.1. A

¹The key detail in these systems is that nodes (operators) are often commutative.

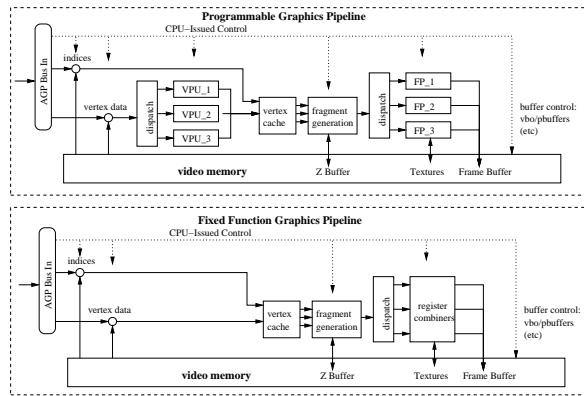


Figure 3.2: Graphics hardware has two or three programmable areas: vertex processing, fragment/pixel processing, and buffer manipulation. Graphics researchers have realized that many complex algorithms can be implemented using this relatively fixed-function pipeline by passing data through the pipeline multiple times with different algorithms and data loaded into the pipeline at each pass. As cards have evolved, it has become possible to route data programmatically from and back into video memory to achieve multiple passes of computation without leaving the card.

stream processing unit contains multiple functions that process an individual OpenGL function call. These units are implemented in C and thus can share memory among one-another and — interestingly — can make calls to external C code. The concept of stream synchronization[24] is introduced in Chromium-related work in order to synchronize OpenGL streams produced by multiple machines. These basic synchronization algorithms turn out to be implicitly used in a variety of stream processing systems.

Note that efficient implementation of stream synchronization primitives is a key feature for stream graphs because synchronization is often necessary in order to create an order-independent graph.

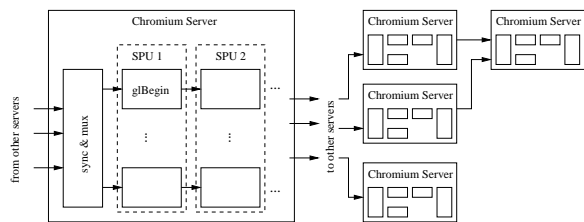


Figure 3.3: Chromium extends the abstraction of the graphics processor as a streaming process to the entire graphics pipeline.

Multimedia programs have long leveraged stream-processing ideas in their program design. Media players, for example the open-source MPlayer shown in figure 3.3.1, implement a simple graph that is used to decode, synchronize, and process audio and video frames. These systems are notable for a few reasons: first, MPlayer dynamically loads an audio and video codec nodes at runtime based on the stream headers. Second, the graph these systems implements has fixed rates and structure — thus easily schedulable — with a few notable and hard-to-solve. First is the audio-video synchronization node, which requires implementation of barrier synchronization. This requires dynamic dataflow ability within the scheduler. Another reason media systems are interesting is because they often create a special effects graph between the decode and the playback nodes of the program. These graphs can vary in complexity from a simple filter as used in commodity media players to a complex dataflow network as used in MSP/Jitter. While these systems are not fundamentally different from early dataflow systems, they do have major differences in token sizes: whereas DSP systems usually execute simple operations at very high frequencies, the video processing systems execute comparatively larger operations on larger data at lower rates. This implicitly shifts the optimization burden from efficient scheduling and buffer management to issues such as memory efficiency and ability to change at runtime.

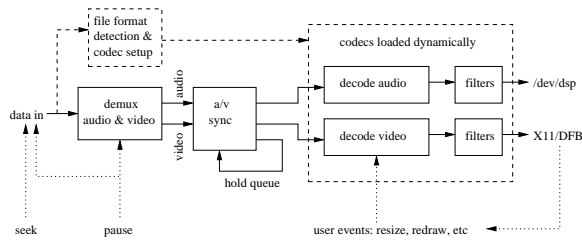


Figure 3.4: Media players are notable because they dynamically load codecs based on the header of the stream, because they have to synchronize audio and video streams dynamically, because they often dynamically load special effects filters into the processing chain and finally because the actual flow of data through the system is controlled by the user: for example, pause, and more dauntingly, fast-forward and rewind. Implementing these features efficiently in a stream processing system is challenging at best.

3.3.2 Stream Processing Languages

Recent research into streaming programming languages has become quite lively. The two leading languages, StreamC[41] and StreamIT[21, 45], have been adopted to execute on standard as well as on customized hardware[41, 44]. The goal of these languages is to balance versatility demands against optimization opportunities: generally speaking, the more versatile the language, the harder it is to optimize.

StreamIT[45] uses a graph formed by hierarchical compositions of simple graph “primitives”: split-joins, loops, and pipelines. Figure 3.3.2 shows this model. The advantage of this model is that the scheduling problem is significantly easier and thus more optimization opportunities are exposed. An interesting feature of this model is the idea of out-of-band parameter changes: in addition to the high-bandwidth stream interface, an external “controlling program” can reach into the stream node and call a control function which changes some parameter that the stream processing function uses on each subsequent record. These out-of-band parameters appear in graphics systems as uniform parameters. One point about StreamIT is that it cannot currently handle variable token rates; thus, problems requiring dynamic data flow are not solvable in the currently published version of the language.

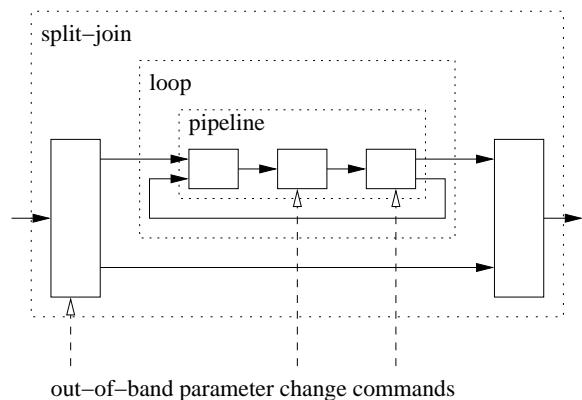


Figure 3.5: The StreamIT network is a hierarchy of simple networks. Current StreamIT implementations have fixed record sizes and rates on each edge. Notably, a StreamIT node can receive out-of-band parameter changes from an external source.

3.3.3 General purpose stream processing hardware

Stanford’s StreamC/KernelC language for the Imagine processor takes a very unique programming approach. In contrast to traditional dataflow where data motion is described using edges in a graph, streams in imagine are explicitly

created and destroyed using the StreamC programming language. This approach allows the stream data to be allocated in co-processor memory very efficiently based both on the size of the stream and the schedule used to execute the corresponding graph of KernelC nodes. This approach strongly resembles the graphics video-memory-management schemes where movement of stream data is managed by procedural code rather than using implicit FIFO-buffers. This approach allows highly optimal memory and register allocation, especially when data is re-used across multiple graph invocations.

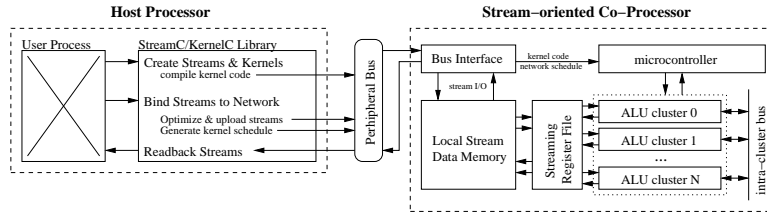


Figure 3.6: The Imagine processor draws a clean distinction between kernel code which executes on the nodes of the dataflow graph and the rest of the dataflow problem. They view the dataflow problem as being driven by buffer creation and movement rather than one of network creation. This allows a procedural description and interaction with the dataflow network.

3.4 Summary

As you can see, there is considerable diversity in the usage of streams. The utility of Stream Graphs is that they can be used to implement all of these systems despite the variation among the different models. The advantage of stream graphs does not stop here: the optimizations and clever tricks used to get performance from each of these application-specific domains are still possible in a stream graph implementation. Interestingly, the use of stream graphs often makes new optimization opportunities possible that were not present in the original application specific area. In the following chapter, I will show a variety of cases where stream graphs are better choices than their application-specific counterparts.

Chapter 4

Using Stream Graphs

At the close of the previous section, I make the claim that all of the cited works can be efficiently implemented with stream graphs. This section will show that this is not a hollow claim and also show the basic power of the stream graph model over previous approaches.

It is difficult to discuss the applications of stream graphs using formal analysis because stream graphs are *overly* Turing complete. Programming for a stream graph is a much broader design space than we are used to: not only can we build an entire applications that run on an individual, we can also string them together into a network. Furthermore, we can modify both the node implementation and the network at runtime as we wish. This is a pretty big design space to work in. One might ask the question, is this design space overly broad? Are stream graphs needlessly complicated?

In response to these questions, I show how stream graphs deal well with the following topics:

1. Certain types of dynamic dataflow are efficiently handled by stream graphs
2. Multi-pass algorithms are easily expressed within the stream graph model and furthermore; their use can be justified as part of stream processing.
3. Optimization algorithms usually applied to streaming databases and stream and dataflow language compilers can be applied to stream graphs.

The main messages of this section are twofold. First, stream graphs make optimization opportunities clearer and more exploitable than possible in von-Neumann models, especially with regard to memory access. Secondly, stream graphs are more versatile than dataflow models because they can handle dynamic data flow problems more gracefully and because they provide greater limits on concurrency than offered in data flow.

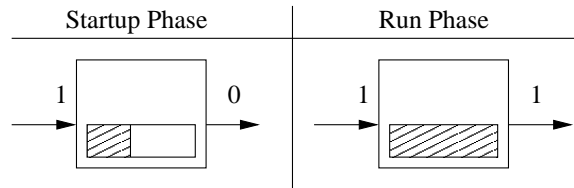
4.1 Dynamic Data Flow with Network Modification

In this section, I show how certain problems posed as dynamic data flow problems can be better implemented in a stream graph model using graph modifications. As I show in chapter 5, efficient solutions to dynamic network modification are often more easily obtained than equivalent solutions for dynamic data flow.

A classic problem encountered in media and graphics processing applications is stream synchronization as mentioned in the background chapter. For example, although a media player receives a single stream of interleaved audio and video frames, the audio stream might need to play slightly behind the video in order to compensate for decoding time. This is an instance of the barrier synchronization problem. Implementing barrier synchronization on a dataflow node causes token production/consumption on each port to alternate between one and zero depending on which streams have and have not issued and matched barrier tokens. Dynamic network modification helps us make this change without resorting to full-fledged dynamic dataflow: when a node receives an unmatched barrier on a given port, we simply disconnect its input and output edges until a matching barrier is received. At this point, we reconnect the edges and resume operation. We can use this approach to implement a variety of synchronization operations, for example stream semaphores[24], without resorting to dynamic dataflow.

Sliding window algorithms [7, 16] have become popular because they give stream processing algorithms a boost in their computational ability. Implementing sliding windows can be done by modifying the queue algorithm for

each edge to buffer a window of tokens instead of a single token. However, a simpler approach can be followed that uses the node’s RAM to store the window. Unfortunately, the latter solution is impossible to implement without a data-dependent node because of a sliding window’s “startup phase”.



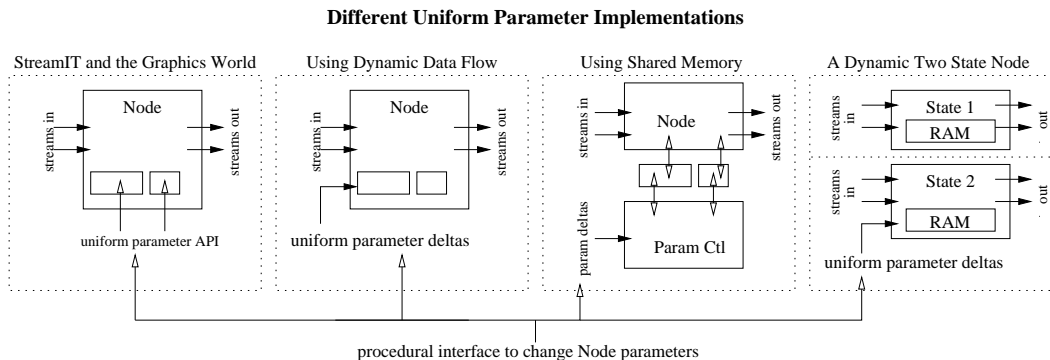
A node must fire in order to initialize its RAM-backed sliding window. However, the sliding-window computation cannot begin until the entire window buffer is filled and therefore the node may not produce output tokens. In dataflow, this means that our operation is done using dynamic data flow: the flow of tokens depends on the number of tokens received by the node. This brings a variety of associated negative implications that we do not necessarily want. Dynamic network modification makes implementing this two phase startup easy: start the network with a node that consumes one token and whose output edge is temporarily signed to produce zero tokens on firing. When the window buffer is full, re-bind the node’s output edge, thus switching us to a full-flow mode. If we were really clever, we could even change the node’s implementation after its window buffer has been filled thus avoiding the step where we check to see whether the window buffer has been filled. Generally speaking, The ability of stream graphs to self-modify at runtime allows them to better-represent data-dependent data flow problems as compared to their dataflow counterparts.

4.1.1 Implementing Uniform Parameters without Dynamic Data Flow

Uniform parameters have been used in a variety of previous stream systems as a bandwidth-saving measure or equivalently as a way to communicate asynchronously with a data flow graph. Although Computer Graphics’ CG language[35] provides the name we use here, their utility is best motivated by the volume-control-knob argument used in StreamIT[45]: a radio’s volume setting does not need to be continually transmitted to the amplifier, only its changes in value. One way to look at the uniform parameter is that it is a “cache” at the node for the current volume knob setting.

A more powerful way to view a uniform parameter is as a point where two processes with different rates interact. Depending on the nature of the problem the interaction might be synchronous or asynchronous, meaning that the interaction stops or does not temporarily suspend firing of the nod. The previous example is asynchronous. CG implements uniform parameters synchronously.

Implementing uniform parameters in a stream graph model is difficult because, by design, uniform parameters are supposed to be supported by the model without a corresponding performance hit. If the uniform parameter is unchanging, then the node’s performance should be the same as the equivalent node without the uniform parameter. Achieving this sort of feature without adding a feature to the stream graph model requires some clever solutions.



An explicit language construct was used in graphics[35] and StreamIT[45]. Implementation of uniform parameters

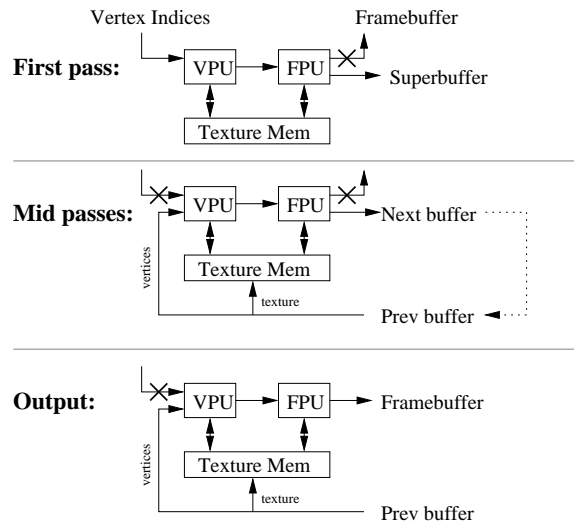
in this way, or using shared memory as in Chromium[23] requires explicit modification of our Stream Graph model. These solutions, though elegant from a programming perspective, require subtle modifications to stream graphs, thus complicating the model.¹

A simple strategy for uniform parameter implementation that fits within the stream graph model is to use dynamic data flow. Let a node consist of two sets of input edges: the data stream edges, and the uniform parameter edges. Presence of tokens on the data stream edges can be used to trigger regular token processing; tokens on the uniform edges will also trigger token processing but in addition contain data that is used to update a local state variable. Though this is conceptually simple, efficiently scheduling this sort of node is extremely difficult. This is because its token-rates are variable at each of the uniform parameter ports². Although chapter 5 introduces techniques that improve the efficiency of this node, this design precludes the use of the fastest static scheduling algorithms and thus fundamentally limits nodes that use uniform parameters.

In light of this limitation, a good implementation of uniform parameters should maintain fixed rates at each input edge. We can achieve this using network modification. Refer to the fourth variation in figure 4.1.1 for this discussion. Let a node with uniform parameters have two actual node implementations, one that does the stream processing, and one that takes an input token that is used to change the uniform parameters. When the system wants to change a uniform parameter, it temporarily replaces the first implementation with the second for the duration of the parameter change. This allows full control over whether the change is done synchronously or asynchronously. This solution effectively offloads the burden of execution from the scheduling algorithm to the algorithms for handling network changes. This is analogous to the trade-offs described in the dynamic dataflow section (section 4.1).

4.2 Multi-Pass Stream Graphs

Graphics cards implement a relatively simple pipeline with feedback capability [1]. On the initial pass a card usually accepts vertex input and rasterizes it into a frame buffer. On more recent hardware, the frame buffer can be fed back through the pipeline as new geometry or texture for subsequent passes. At some point this framebuffer can be displayed or read back to the CPU for further processing. The following diagram illustrates this process:



Even though a graphics card represents very primitive hardware, many complex algorithms including matrix solvers[8] and various graphics algorithms³ have been implemented to work on these basic systems. These are imple-

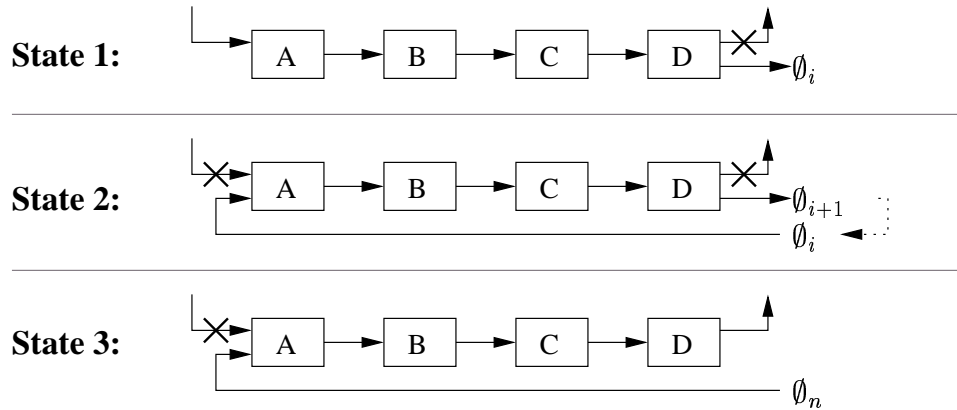
¹One could argue that the shared-RAM model does not present any major complications to the stream model. On the surface, this indeed seems correct. At what level, however, does a stream graph implementation using Shared Memory turn completely into a PRAM machine with some graph tweaks? The threshold for this case is not clear.

²To get a feel for why this is the case, realize that a node is usually eligible for firing whenever it has tokens on its stream inputs. Thus, if the node fires but there are no uniform-parameter tokens, the rate at the uniform parameter ports is zero. In contrast, if uniform tokens are available, the rate on the uniform ports will be one. Thus, the rate on these edges is variable over time

³See the extensive proceedings of NVidiaU 2004 or Graphics Hardware 2004 for in-depth text

mented by modifying a stream graph at runtime and routing/re-routing edges full of tokens between the computation passes.

A stream graph formulation of this process applies to any data flow graph containing a cycle that meets an important condition: the loop entry may not interleave tokens from the entry point with the tokens already in the loop. This effectively flushes the loop after each “pass”, giving rise to a multi-pass-like computation model. When this criteria is met, then the whole loop can be “unrolled” to look like this:



I show in this figure how these loops can be easily implemented as a stream graph. A loop begins with the entry edge being bound to some data source, and, exit edge being unbound, and the feedback edge being bound to a vertex. The (NULL) vertex is shorthand for a node that never produces and never consumes tokens. Used in this context, it causes tokens pass through the loop body and accumulate, without any conditional scheduling, on the feedback edge, shown as state 1 below. When some predetermined amount of data has passed through the entry edge it is disabled. The feedback edge is then bound to the feedback port on node A and a new edge bound from the feedback port of the graph to a non-consuming-non-producing vertex. This corresponds to state 2 in the diagram below. When all of the tokens in the previous edge have been exhausted, the old feedback edge is discarded and the process repeats as much as desired. When all of the tokens have been suitably processed, we can switch to the third termination configuration shown by State 3 thus producing a final output.

As I pointed out before, the graphics pipeline has a fixed function version of this unwinding. In comparison, the StreamC programming environment used by the Imagine processor allows any data flow graph to be implemented in each “pass”. StreamC can be seen as a simpler version of a stream graph. The explicit constructs for managing stream buffers are analogous to the re-binding of edges described above. While an edge can be routed to or from a non-producing-non-consuming vertex, network changes are not incremental: the entire network must be reloaded en-masse. The result of this design is that large graphs and programs can be decomposed into a set of smaller overlay graphs that change as the program runs. As has been demonstrated in the literature, this approach is very versatile [38].

The utility of a multi-pass formulation of a program lies not in its algorithmic accomplishment but its ability to simplify a complex scheduling process. In the case of both graphics cards and the StreamC/Imagine systems, network modification can be used to implement complex algorithms on hardware capable of only the simplest schedules. The stream graph model is useful in these contexts because it can capture both of these models without losing the optimization opportunities leveraged by both. By formalizing these multi-pass systems in terms of stream graphs, a whole realm of unexplored opportunities becomes visible — the use of incremental network modifications in implementing multi-pass algorithms.

4.3 Optimization

Stream processing systems are popular in part because they are heavily optimizable. First, they make much more efficient use of memory compared to equivalent RAM implementations. This is done through aggressive I/O optimization. The second reason why stream processing is popular is because application-level knowledge can be used to heavily

optimize the stream processing graph to maximize its throughput and efficiency. Converting an application-specific stream processing model to a stream graph should not disable all of the optimization opportunities that were possible in the original implementation.

There are optimization techniques that *absolutely must* be built into the stream graph implementation in order to work efficiently. The smaller this set of techniques is, the better, because it makes stream graph implementations correspondingly easier. The ability of a stream graph to self-modify at runtime works in favor optimization algorithms: unlike static systems like dataflow where the optimizer has to be built into the compiler, an optimization program can easily link into the stream graph and perform modifications external from the stream graph runtime. Thus, there are only a very small number of algorithms that must be built into a stream graph implementation in order to maximize the stream graph's execution optimality.

4.3.1 Low-level Optimizations

Stream graphs can make efficient use of memory in a variety of different contexts. First, like its dataflow counterparts, stream graph scheduling can be extremely resilient to high memory latencies. This is usually eclipsed by the second gain: streaming I/O optimization. In addition to basic I/O optimizations, stream graph execution can be optimized so that token transport along edges is done using compression. Finally, mechanisms can be provided in a stream graph to allow efficient conversion from a stream into a RAM and vice versa.

A variety of optimizations exist for hardware and software platforms that allow sequential memory references to execute dramatically faster and more simply than their corresponding RAM equivalents. Some of these techniques, for example prefetching, can occur without explicit knowledge of graph's structure. We can apply these types of optimizations to the algorithms that implement edges in a stream graph.

More complex optimizations can be made when the data flow rate across an edge relative to flow across other edges is known. Streaming register files [40] and stream scheduling [26] use this information to efficiently bring stream data into and back out of the CPU with as little spilling as possible. These types of optimizations have been implemented for the Imagine architecture (see figure 3.3.3). Adapting this approach to work with Stream graphs primarily involves adapting from the StreamC interface to the dynamic network change interface: the big difference between StreamC and stream graphs is that network changes can occur at a finer granularity than supported by StreamC. Of course, another problem is that vertices in StreamC (which are programmed in KernelC) do not have access to RAM. Similarly, while most standard hardware has lots of RAM access, it often will not have a streaming register file on which these optimizations are based. Thus, while it is possible to perform hardware-specific optimizations using a stream graph as an input, the process is prone to the usual hardware mismatches that occur with any programming model.

Streaming algorithms[22, 36] are seen as memory efficient for an entirely different reason than discussed above. Streaming algorithms require that per-vertex memory must be poly-logarithmic to the length of their input, which must be processed sequentially. From the perspective of stream graphs, this distinction is arbitrary: while there are clear benefits to having nodes with small local memories, there are no major optimizations to be made based on the restriction. Note that optimizations based on static allocation are not feasible because of the dynamic nature of the graph. Even if they were possible, chapter 6 presents a technique that makes usage of dynamically allocated regions performance competitive to statically compiled regions.

A variety of systems have experimented with stream compression. An extension to Chromium was proposed called Stream Caches(XXX: cite) that provided dictionary-style compression of the OpenGL stream (geometry and commands) when crossing network connections. Similar systems have been proposed for compression of the X protocol using byte-level compression [15]. These sorts of ideas can be easily adapted to stream graphs without modifying the underlying edge implementation by creating a pair of compress-decompress nodes that perform automatic compression.

4.3.2 Optimization using Graph Modification

A variety of optimization techniques can be applied to dataflow and stream processing graphs to dynamically improve their performance based on the resources available at runtime. There are three broad types of optimizations: (1) automatic spreading-out of tokens across multiple processors, (2) automatic partitioning of the graph across multiple processors and (3) automatic reordering of the graph to achieve higher throughput or lower latency. Anyone researching these techniques falls prey to a basic problem: they need to create a system capable of some primitive sort of

network modification in order to test their system. As I will show, stream graphs provide an excellent platform on top of which any of these algorithms can be implemented.

Automatic parallel processing of tokens on dataflow graphs has been extensively studied. More recent streaming databases research has renewed these ideas for parallel query processing [5]. In either case, this is a hard problem to solve, but possible when heavy restrictions are made on graph types and per-port token rates. These algorithms can be applied to stream graphs without much overall modification; the basic problems caused by these algorithms are already solved, namely the fact that these optimization algorithms want to dynamically create and delete nodes at runtime or compile time.

A few adjustments and practical considerations must be taken into account when adopting parallel token processing strategies to stream graphs. First, the order of tokens in a stream graph is assumed to be strictly FIFO unless otherwise relaxed by the application programmer. Thus, parallelization will often be forced into appearing to process tokens in the same order as would be done serially. This shows a further complication: even though this constraint is present in most dataflow systems, one could always count on the simplifying assumption that dataflow nodes did not have state. This assumption cannot carry through to stream graphs. Even so, these parallelization techniques can be applied to stream graphs with some work. It is worth noting that the signature mechanism in stream graphs can be used to label sections of a graph so that they can be parallelized. For example, we might label an output port of a node we wanted parallelized as being allowed to produce reordered tokens, or similarly mark the input port of a vertex after a parallel section as allowing reordered token inputs.

Optimization is possible when the ordering of the tokens is unimportant despite the fact that node ordering is fixed. This scenario, which is plausible in CG fragment programs[35], allows more aggressive parallelization than is possible when ordering is important. While graphics cards implicitly implement this sort of process, study of this on a formal level seems not to have taken place.

As mentioned in the background chapter, streaming database systems and optimizing compilers for network processors often leverage the fact that the ordering of nodes in a graph may sometimes be switched without affecting the computation's outcome. This is particularly easy in streaming databases because the condition applies to all of the nodes in the query DAG. Implementing algorithms based on these types of re-orderings is easy in stream graphs provided that the information about order-independence is somehow available. As usual, this might be conveyed using signatures on a per-vertex basis or globally for the entire graph for cases resembling the streaming database scenario.

4.3.3 Optimization using Graph Partitioning

Another popular parallelization technique is partitioning (or clustering) in which one large graph gets partitioned across multiple systems. This technique is used in streaming databases, StreamIT, and in dataflow systems. At the core of any of these systems is a set of algorithms that create and manage the partitioning of nodes between processors. Surrounding this problem are a variety of nuisances that stream graphs help solve:

1. Communication must be provided between the partitions
2. A “control program” must run at each node to cause new partitions to go into effect
3. The control program needs to be able to modify the graph partition without losing any buffered state
4. Code for each node must be dynamically loadable (and efficiently accessible) for each of the nodes

Stream graphs do not simplify the challenge of partitioning. In fact, if the goal is to partition a stream graph across multiple systems, the partitioning problem becomes more difficult. Ignoring this issue, stream graphs provide solutions to the nuisance problems listed above.

In synchronous data flow models, implementation of communication between partitions is complicated by the fact that the entire graph can block waiting for input from a remote process. As a result, the scheduler must have and handle the notion of special “input edges” that form a boundary between the internal synchronously scheduled graph and the external world. This is not necessary in stream graphs: a link to an external process can be implemented in a stream graph as a node that requires no tokens to fire and produces tokens variably. Importantly, this node can be scheduled without causing over-polling and busy-waiting using backward-chained activation as described in the next chapter. As a result, stream graphs allow efficiently and clean handling of asynchronous connections to the outside world without special casing required for other graph models.

Because vertices can create and delete nodes in the stream graph model, the entire control program for a partitioned graph can be implemented as a node just like the rest of the program. This is shown in the figure 4.3.3.

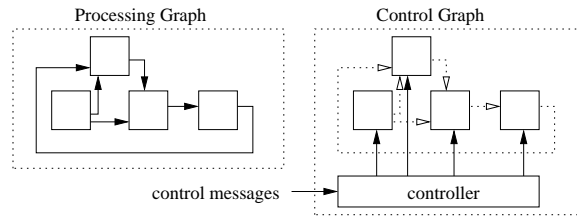


Figure 4.1: A stream graph can be constructed from a base controller node: the node receives control messages from an external source and modifies the network accordingly. The control node would maintain a connection to each node it instantiated: this causes the graph to have two distinct but connected parts: the control network and the actual processing network, shown separately here for convenience.

Because stream graphs are designed to support modification at runtime, it is not necessary to reload the entire graph in order to change it. While this does not solve synchronization problems that arise from changing the partitioning mid-stream, it solves many of the state-loss problems that would arise in a static graph implementation.

Stream graphs provide assistance in one of the major challenges in a partitioning system: code transport. The principal challenge in this sort of problem is the challenge of picking up a node — its state and its implementation — and moving it to a new processor. State is easily handled within a stream graph model because a node's local memory is finite and well defined. The larger challenge (from a programming perspective) is moving the code that implements the node from one processor to another. Stream graphs already provide a local mechanism for this because they must be able to create and delete nodes at runtime. While the stream graph is not obligated to deal with issues such as cross-endianness and hardware-portability, it is clear that the stream graph implementation is an elegant place in which to place this functionality. If the stream graph took care of all of these problems, the task of partitioning a graph across multiple processors is made considerably simpler.

In summary, stream graphs do not make the partitioning problem simpler from the theoretical perspective. However, stream graphs do provide a variety of base services that make the construction of a partitioned system simpler and considerably more elegant.

Chapter 5

Executing Stream Graphs: Fast-update Schedulers

A variety of trivial algorithms exist that can schedule dataflow graphs containing no signature information. The advent of algorithms that compute very efficient schedules for restricted graphs made research into the more robust algorithms less important. In this chapter, I show a modification to a standard queue-based scheduling technique that precomputes special “bindings” based on signature information in the stream graph in order to incrementally generate and update a conditional-free schedule for a stream graph.

This chapter is organized into three parts: first, I present background on the token-flow graph scheduling problem. Next, section 5.2 presents the basic idea of binding as a way to compute a temporary schedule for a graph in response to a specific token creation or consumption event. Section 5.3 then shows specific scenarios for which bindings can be computed. The idea of this section is to show a set of rules used to incrementally build the schedule for a stream graph. I then turn to issues of implementation. As it turns out, a binding can be implemented in two different ways that produce computationally equivalent behavior. I compare and contrast the two approaches. Next, I point out (section 5.4) how special care must be taken with cycles to prevent deadlock and over-enthusiastic execution in a binding-driven model. Because graphs often oscillate between a small set of configurations, I introduce a technique called schedule caching which allows complex scheduling decisions to be reused when a stream graph repeatedly switches between a small set of configurations. Finally, although it is possible using stream caches to make periodic schedulers viable for use with stream graphs, section 5.7 points out a critical problem with them that may preclude their use in all but very-slowly-changing contexts.

5.1 Background

Queue based scheduling algorithms used in dataflow hardware have fast update times because their schedule computation is done online. The waiting-matching algorithms[17] that they use are driven by a queue of active nodes that is continually evaluated and added to. When a node fires, its downstream neighbors are placed onto the active queue. After firing a node, the scheduler pops a nodes from the front of the active queue and checks to see the node is (active) (whether it can fire). If the node is active, it is fired. In either case, the node *is not* added back to the active queue.

These queue-based algorithms have never been particularly competitive: first, they perform a great deal of extra enqueue-dequeue work that is unnecessary for simple graphs. Second, because these behave like a breadth-first-search through the graph, they tend to work well only on graphs whose token actually flow in a breadth or depth first pattern. Graphs that require tokens to accumulate at edges before firing cause these queue-based schedulers to perform far more extra work than is actually necessary. It has been observed [31, 32] that this is a greedy algorithm prone to standard local-minima problems. Despite this, active-queue algorithms are the most resilient to both (a) execution of nodes with little signature information and more importantly (b) execution of graphs that dynamically change. As a result, they are an ideal last-resort algorithm for use in stream graph execution.

A variety of global analysis techniques have been devised to compute static periodic schedules for data flow graphs. A periodic schedule for a graph is a list of node invocations that guarantees tokens to flow through the graph at a fixed

rate. A schedule is often represented with a repetition vector, where the i 'th value in the vector corresponds to the number of times to fire the i 'th node. These periodic schedules are favored because they provide guaranteed buffer sizes and also because, in stark contrast to active schedulers, they are guaranteed to provide optimal solutions where greedy approaches would fail.

This is in contrast to the active-queue schedulers discussed above which are prone to the same local minima problems as any other greedy algorithm. These techniques work by creating a topology matrix [31, 32] that has one row for each node in the graph and one column for each edge. The cell at row i and column j corresponds to the number of tokens added to edge j by node i . A null vector is then extracted from this matrix is the optimal repetition vector for scheduling this graph.

While computing this solution is fast for small graphs, recomputing the repetition vector for large graphs with comparatively simple structure can become extremely costly.

Another issue is that this method *only works for graphs whose per-port token flow rates are constant*: while this does not prevent cycles in the graph, it does prevent data-dependent decision making and iteration. Techniques that find formal solutions to this problem either (1) make assumptions about token distributions[10], or (2) use loop unwinding techniques to convert the iterations into a graph with known flow rates. A more common solution to this problem is to use clustering to break the graph to produce multiple independently, more easily schedulable subgraphs that get linked together with an active-queue scheduler. The problem with all of these approaches, however, is that they work only when the graph is static: when the graph is dynamically changing, these techniques are no longer practical.

5.2 Graph Traversal and Bindings

One way to implement stream graph scheduling is to view it as type of graph traversal problem: when a token arrives into a graph, we traverse the graph recursively looking for and executing nodes that have been activated by its arrival. As with any graph traversal, the breadth and depth of our traversal will affect the overall efficiency of our search.

Graph traversal in this context is complicated by the fuzzy notion of node activation in stream graphs. In most traversal algorithms, it is assumed that arrival at a node will always lead to traversal of its outgoing edges. This is not the case for stream graphs: node activation from incoming edges and similarly, activation of outgoing edges is unspecified by default, but indicated through signature if known in advance. A token may trigger execution only when it has a certain value, when there are multiple tokens already on the queue, when other tokens are present on other edges of the nodes, or only with a mix of all of the above exists on the input queues of a node. Furthermore, token traversal, like graph traversal, can be reversible: backward traversal can occur driven by the need for a token rather than the production of a token.

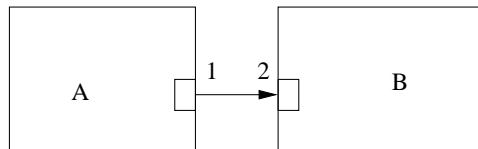


Figure 5.1: Bindings are used to implement recursive graph traversals that occur in response to a node receiving or producing tokens.

A “binding” is a command that gets executed when a token is placed onto or removed from a specific edge. Bindings allow us to implement a wide variety of traversals within a token flow graph. Bindings are edge specific. For example, we might bind edge A-B in figure 5.2 to the FSM in figure 5.2:

Executing this graph involves firing node A repeatedly. The precomputed bindings will take care of firing node B. Alternate implementations exist for this: for example, we could reverse the direction of the traversal. When node B fires, it requests a token from the A-B binding, causing two firings of node A. Execution here would involve continually firing node B.

Scheduling of a stream graph involves finding good bindings for all the edges in the graph and constructing when necessary a controlling loop that periodically re-starts traversals when new tokens arrive or the traversal stops.

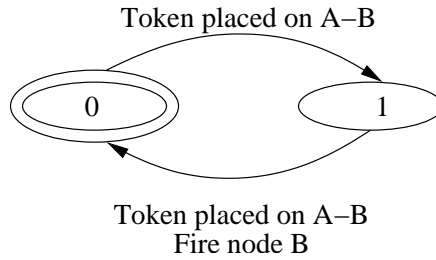


Figure 5.2: A FSM that performs token traversal in response to placement of tokens onto node A.

The next section shows a variety of common binding scenarios for edges in a stream graph. These binding decisions can be made very quickly, making them ideal for stream graph use. It is my belief that, for most hardware, simple bindings can be combined with a multi-state queue-based scheduler that causes initial traversals to efficiently execute stream graphs.

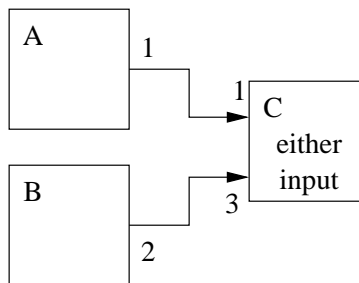
5.3 Basic Binding Scenarios

Binding is done for each edge in a stream graph by analyzing signature information on the edge. Whenever an edge changes or a signature is updated, only the affected node-edge-node pair is re-bound. While this technique is inherently greedy, it allows extremely fast updates and produce extremely fast schedules for the most common graph topologies.

I draw a distinction between indirect and direct bindings. A direct binding is one which causes immediate execution of the downstream node. Direct bindings recurse like a depth first search, causing more traversals whenever possible. An indirect binding, in contrast, is one that adds the downstream node to an active queue.

Bindings can be unconditional or conditional. An unconditional binding is one that always activates the downstream node. A conditional binding, in contrast, requires the state of the downstream node to be checked at runtime. Conditional bindings can be direct or indirect: a direct conditional binding checks the node and directly fires it if it is active. An indirect conditional binding performs a similar check, but enqueues the node instead of directly firing the node when it is found to be active. Queue based schedulers can be tweaked to allow more efficient execution of nodes that are known to be active when they are enqueued, as I show in a later section.

The following graph illustrates a very simple direct unconditional binding scenario:



Using the signature information provided with this graph, we can determine a number of firing events that will occur during execution. Notice that Node C is signed as firing on either input. Therefore, if a single token arrives at C, it can fire. We know that node A produces one token per firing; therefore when node A places a token onto its single output edge, we can directly fire node C. In order to achieve this, we set the binding on edge (A, C) to directly execute C. This is a direct binding of the simplest type.

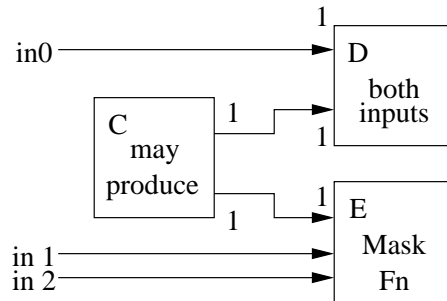
As I manually demonstrated in the previous section for a similar graph, we can perform a simple analysis on edge (B, C) that uses a small amount of state to obtain a direct binding for edges whose ports have unequal token rates. The formal procedure for this is acyclic precedence graph construction, which is described in [32, 43]. For our purposes,

however, this procedure is overcomplex: We can derive a simple direct binding template for edges with known output rate α and input rate β :

- If $\alpha = \beta = 1$: fire always
- If $\beta = k * \alpha$ for some integer $k \geq 1$: fire downstream node k times.
- If $\alpha = k * \beta$ for some integer $k \geq 1$: fire downstream node every k times.

There are two more complicated rules for arbitrary values of α and β that involve more complex state machines that basically maintain $\max(\alpha, \beta)$ states where all but one state leads to a firing event.

The situation is slightly more complex for nodes with multiple output ports and/or multiple input ports and more restrictive rules about when the node will fire or produce tokens because binding activation not longer guarantees downstream activation. The following diagram will be used to understand the state space for these sorts of problems. You will note that in the following figure, node C is labeled as “maybe producing” tokens; although its rates on its output ports are fixed, tokens may not always be placed on its output ports. This is not a particular problem for a binding-based solution to this problem, however: remember that bindings are triggered by token movement. Binding does not, therefore, have a particular problem with distinguishing between node C and the simpler node A from the previous example.



Computing a direct binding for edge (C, D) is more complex than our previous examples because node D requires input on both edges in order to fire. The easiest way to handle this edge is to use an unconditional indirect binding. This would cause, at some later time, the scheduler to check for token presence on both ports of node D before it is fired. We can leverage our signature knowledge to speed things up a bit: if there is a token sitting on node D 's other port when the (C, D) binding fires, then we know that we have just enabled D . The question now is whether this conditionally discovered node should be directly or indirectly fired.

5.3.1 The Effects of Direct versus Indirect Binding

The two different approaches to dealing with a newly-activated node will give us different types of behaviors: direct firing of this node will give us fast depth-first execution of the graphs, minimizing the buffering requirements while risking extra enqueues if the network has blocking components. Indirect firing of the node will push tokens breadth-first through the dataflow network, giving rise to more buffering and higher levels of concurrency. The choice of which policy to use depends upon the graph being executed and the hardware on which execution takes place.

5.3.2 Representing Firing Conditions

An interesting challenge is efficient representation of a firing condition. Some firing conditions allow unconditional bindings to occur, for example, a “fire on either input” condition. Thus, it is important that our firing conditions are represented to make this obvious to the binding algorithm.

A simple representation of a firing conditions is a boolean function over token presence on each input-edge. For example Node E might have the a function that says, “fire when tokens are on port 1 and port 3, or port 2 and port 3”. These functions can be easily stored and analyzed, and can be used to express a wide variety of common firing conditions. Note that certain functions — for example, XORS — can lead to deadlock. As a design point but not a

fundamental rule, a functional firing condition should never let token presence be inhibitory to execution. The reason that this is not outright forbidden is because network modification can be used to drop edges with tokens that inhibit firing.

Boolean representations do not work for nodes whose firing is dependent on the data contained within a token. For example, let us assume that node E from figure 5.3 is a multiplexer that consumes the token produced by edge (C, E) only when there is a token on edge $(in2, E)$ with the value 1. A clean way to handle this node is to provide a lightweight polling function in the node signature that answers a simple question: given the current token inputs, can this node fire? Having a separate function from the actual node has the advantage that binding can retain control over the actual firing of the node.

We can use the presence of both a boolean and polling functions to improve the efficiency of this process. By checking the boolean function first, we allow rapid rejection of activation events where there simply aren't enough tokens to proceed. Once these criteria are met, we can use the polling function to actually determine whether the node will execute.

Finally, as should be clear, direct binding is an endlessly interesting problem. I have provided direct binding cases for a variety of simple scenarios, but clearly there is more to be done. In some senses, it is the bottom-up equivalent of the StreamIT scheduling technique: StreamIT starts with the network decomposed into primitive topology types that have their own "best schedule" and computes a schedule from it. Here, because we are faced with the need for fast update times, we effectively compute the decomposition on the fly. Doing so may produce a sub-optimal schedule because we make fundamentally greedy decisions during the binding.

5.4 Preventing Direct Binding Deadlock

Dealing with binding in the face of cycles is difficult. Consider the example in figure 5.4.

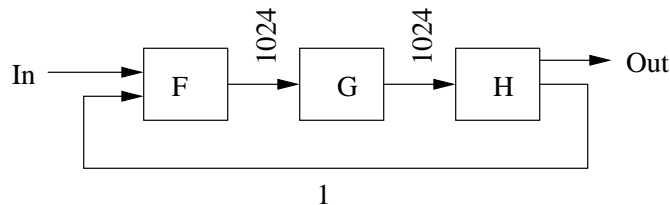


Figure 5.3: A graph with a cycle. The numbers in the center of edges (F, G) , (G, H) and (H, F) represent the token size being passed between the nodes.

We must choose three bindings for this graph. It should be clear that the best choice would be to set direct unconditional bindings for both F-G and G-H that immediately call the downstream edge. Depending on the behavior of node H and F, applying a similar "immediate" binding to edge (H, F) could be:

1. The correct decision, if this cycle is just a tight iteration that will shortly end
2. A correct but greedy decision, if this is a cycle that will iterate for a very long time before ending
3. A decision that could result in deadlock, for example if this cycle is a feedback loop rather than an actual iteration

The safest policy to handle cycles is to never let a cycle of immediate calls occur. When an "immediate" cycle is created by a bind, switch that edge binding to perform an enqueue instead. This approach effectively rate limits all cycles, forcing a check for other work in between each iteration.

One problem with the previous approach is that the order in which a cycle is created affects which edge gets bound as an enqueue. Imagine that the edges in this graph were created in the following two orders:

- Bind F-G; Bind G-H; Bind H-F
- Bind H-F; Bind F-G; Bind G-H;

In the first bind ordering, we will discover a cycle during the bind of edge H-F and thus prevent immediate binding from occurring along this edge. In the second case, we will discover the at edge G-H and prevent binding here instead. Whether this is desirable depends on the application. Arbitrary record sizes have been added to this graph to show one possible downside to this approach: in the first binding scenario, we end up temporarily suspending graph execution on the H-A edge, resulting in one byte enqueued on edge H-F waiting to be processed. In contrast, the second scenario leaves 1024 bytes enqueued: if very little else is happening in the graph, then this may not matter; however, if a lot is happening, then this may be quite wasteful. There are ways to fix this for this scenario: we can direct bind the entire direct invocation loop, switching the binding method of the edge that enqueues the least data per execution. Obviously, this is a simple minimum-search problem that can be easily generalized to a user-specified measure of edge priority if necessary.

It may be possible to perform more elaborate loop-unrolling-like operations stream graphs based on signature analysis. For example, if the node that contains the input edge fires only in the presence of tokens in the input and feedback edges, then the cycle is self-regulating. Therefore, a loop executed with direct calls will be able to sustain itself for only a limited amount of time before it exhausts all input tokens and needs to defer to the rest of the graph for more input. While these sorts of unrollings are probably implicitly predicted by loop-unrolling techniques, the advantage of doing an explicit analysis as we have done here is simple: it can be done with limited global analysis.

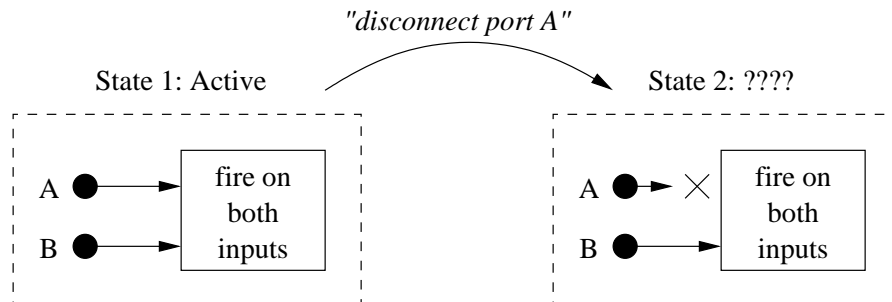
Direct binding is not topology-blind: it has to look at network topology to perform a variety of unpleasant analyses to verify that its choices will not introduce deadlock conditions. The difference between this and explicit clustering techniques that perform global analysis is that these techniques provide faster update times. We can learn from these systems: clustering can definitely be used to accelerate the topology verification process that signature analysis requires. It may even be possible to use our knowledge of the graph changes to dynamically update the clustering on the fly.

5.5 Concurrency Issues

Indirect bindings can be made to give rise to two types of enqueue-events for our catch-all scheduler. The maybe-active type is self-explanatory: this node has been guessed to be executable. This arises for nodes whose input ports are unsigned or whose overall behavior is unsigned, or for nodes where dynamic binding has decided for whatever reason that deferring the execution check is a better policy than direct execution. Executing these types of nodes requires first checking to see if they can execute. The other type of node that gets added to the the queue is one that was active when it was enqueued.

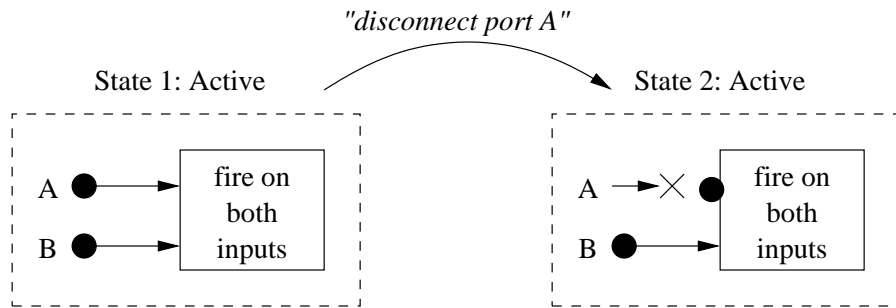
As a first note, it is important to execute these items in the order that they are received. While performance gains can be obtained from executing all active nodes at once, the urge to do so must be resisted because it might lead to unfair execution and ultimately, node starvation.

A number of timing issues arise in dynamically changing networks when we have a delayed execution process such as the one described above. Problems arise when an edge gets moved, disabled, or deleted after an edge has been classified as active but before it executes and consumes the token. The following figure shows this scenario:



The explicit interpretation of this scenario tells us that the node is not active in State 2 because examination of its input edges shows us that there is no token on its first input. The ambiguity arises in how we interpret an enqueue-as-active event: the simple way of interpreting the enqueue event is to say that early-classification is simply a guess and that if the network changes in the interim, the guess was wrong. If we follow this policy, then every time the network

changes, we need to change the status of the affected nodes from active to maybe-active in order to prevent false execution. In order to avoid this edge-verification step, we could try to treat the enqueue-as-active event as consuming tokens but delaying the actual computation. In this policy, the graph above looks like the following:



While this policy works for scenarios where the node is guaranteed to consume the tokens in question, I am unsure whether data-dependent token consumption affects the consistency of this policy. Consider the multiplexer example from before: if the port A is only conditionally consumed, then once we discover that the token was not consumed, we have to repatriate it to the front of the moved edge. We get a problem, however, if the moved edge has activated a node in the graph in the meantime. In effect, tokens have been processed out of order. Therefore, this policy is feasible only if we can guarantee that the moved edge will not enable any other nodes before we can execute the original nodes. I have not studied this problem extensively, and clearly, it needs to be studied further to see if it can be avoided.

Changes to node signatures and functions suffer from similar timing considerations that might arise from any sort of phased execution mechanism. Because concurrency issues are not my primary point of concern for this project and because the alternate strategy I present in the previous paragraph may be unstable, I am going to advocate the simplest policy for concurrency: when a node or its bindings change and it has been labeled for execution, switch its labeling in the queue. While this may cause efficiency degradations for changing networks, I see no better solution until further study has been made.

We could easily get ourselves tied up in knots over multi-CPU concurrency issues for dynamically changing graphs. I maintain that this is a relatively unimportant issue: we must simply let the application programmer know that changing edges that have tokens on them is dangerous. As long as there is a clear way to prevent data loss, then there is no reason to prevent it for all cases: after all, their application domain might not care that a few tokens have been lost.

5.6 Schedule Caching

Many of our scheduling techniques, especially the so-called global analysis techniques, have overheads that make rapid dynamic schedule changes undesirable. We introduce here two schemes for rapid recall of a dataflow schedule in the face of network changes.

We can speed up topology-matrix approaches to data flow analysis by caching the topology matrix for each computed periodic schedule. When an change is made to the dataflow graph, we can look up the new topology matrix and, if found, skip the schedule computation process. While this saves us the effort of computing the null-space of the dataflow graph, it still leaves us with a costly search through some arbitrary number of relatively-large topology matrices.

A slightly better way to cache state schedules is to keep a directed schedule change graph as shown in figure 5.6. Vertices in this graph correspond to a particular scheduling configuration, and edges correspond to a graph change operation applied to the graph. We make this into a cache by realizing that each graph operator has an inverse: the inverse of an edge create is a delete, or similarly, the inverse of an edge move restores the previous location of the vertex. Similarly, we know that there are certain operators that do not affect schedules: node creation, for example, does not affect the schedule. Obviously, in order to compute the inverse of an operation we must know the original precise parameters that it used. We store these on the edges of the graph.

Schedule caching works as follows: we track our current position in the schedule change graph. When presented with a graph operator, we check to see if any of the edges incident to the current vertex in the schedule graph correspond

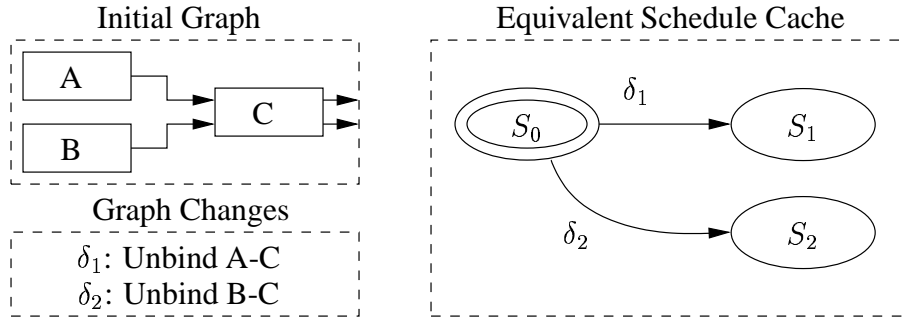


Figure 5.4: This scheduling cache example demonstrates schedule caching for a barrier-scheduler described in [24] and earlier in this paper: nodes A and B produce a data stream that might occasionally contain a “barrier” token: when no barrier token has been received, C simply reads tokens from A and B and passes them to a matching output port. When a barrier token on one input is received, C stops reading from that port until a matching barrier is received on the opposite token. In this example, S_0 corresponds to the initial state where no barrier tokens have arrived and δ_1, S_1 to the state where node A has issued a barrier and the network is waiting for B to do the same.

to the this operator or its inverse. If they do, then we move to this vertex and activate its schedule directly. If no match exists, then we compute the schedule and update our schedule change graph accordingly. Note that this technique, unlike the other, is scheduler neutral.

5.7 Periodic Schedules and Scheduler Thrashing

Periodic-schedule based algorithms will have the scheduling equivalent of a nervous breakdown if the network changes mid-period. While a new periodic schedule can be computed for the new network, they assume that the network is flushed (the internal edges of the network are empty) when the schedule starts. Because a network change can happen mid period, there may be residual tokens in the newly changed network. Only allowing the network to change between scheduling periods is not an acceptable solution to this problem.

The usual solution to this problem is to execute all of the nodes in a round-robin fashion until the network is flushed. As long as a another network change hasn’t taken place in the intervening period, and the network isn’t fundamentally deadlocked, this will ready the network for its new periodic schedule.

The problem with this approach, which I call scheduler-backoff, is obvious: the round robin schedule is not an optimal schedule and could waste time. In effect, when a network change occurs mid schedule period, we must compute a dynamic schedule using any of the techniques discussed previously and hold this schedule until the network is flushed.

The need for scheduler backoff arises only for periodic schedules. However, if the rate of network change is sufficiently high, the need for scheduler backoff is effectively a stake-through-the-heart of the periodic scheduler. To understand why this is the case, imagine that we a network that generates a network change at some uniform rate: call the time between changes the network change period. When the network changes, the network will spend a certain amount of time executing the scheduler-backoff phase. Figure 5.7 demonstrates this process.

The efficiency of the periodic scheduler depends ultimately on the difference between the backoff phase and change period. If this distance is large, as it was for synchronous dataflow graphs where this problem was originally mentioned ([31]), then the backoff phase and the corresponding algorithm is unimportant. The use of stream caches and a periodic scheduler may be sufficient in these cases to execute a stream graph. On the other hand, when the difference between backoff time and change time is small, we end up in a thrashing situation where we spend most of our time executing the backoff algorithm. For networks where this is the case — which occur often (see chapter 4) — making the aperiodic scheduling algorithms work as well as possible becomes a tantamount concern.

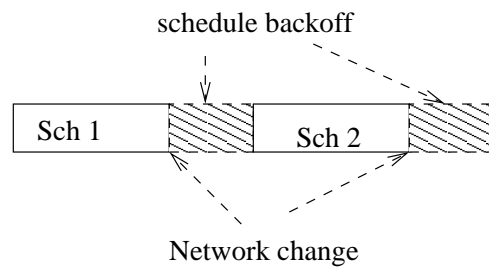


Figure 5.5: Execution of two periodic schedules results in backoff. If the network change period is too small in comparison to backoff time, then thrashing results and we end up executing a dynamic schedule all the time.

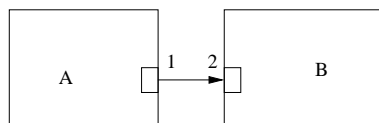
Chapter 6

Efficient Implementation of Runtime Scheduling Changes

Statically computed schedules give the fastest execution rates because the static schedule can be compiled to remove the layers of indirection that come from the more generic scheduling approaches. These optimizations can effectively flatten the entire graph into a single piece of inlined, pointer free code. In contrast, stream graphs must use a layer of indirection in node functions and in the scheduler loop to hide the fact that scheduling is constantly changing at runtime. This layer puts stream graphs at a major disadvantage as compared to static counterparts. This chapter presents a novel dynamic linking technique that allows this layer of abstraction to be removed in order to make stream graph execution competitive.

Optimization opportunities arise in two places within a static scheduler: first, optimizing the code that represents the node, and second, optimizing the scheduling loop that calls the node functions. In a static schedule, the buffer allocations can be constant. This allows the code that a node uses to obtain tokens and produce tokens to refer to a statically allocated memory location. At the scheduler loop level, once a static schedule is known, we can use inlining, loop unwinding and program-counter-relative jumps to call functions. The first two approaches linearize control flow, while the third allows jumps to take place without first referencing memory.

Stream graph implementations have to add a layer of abstraction into each node and scheduler to compensate for the lack of (a) statically allocated buffers and (b) a statically computed schedule. The following diagram is useful in discussing these optimizations:



A generic way to implement the firing functions for each of these nodes is:

```
fireA(Node* me) {
    static int tmp = 1;
    tmp++;
    me->out[0].push(&tmp); /* in[0] is the input port */
}

fireB(Node* me) {
    int t;
    me->in[0].shift(&t); /* in[0] is the input port */
    printf("%i\n", t);
}
```

The problem with this implementation is indirection: we work too hard in this implementation just to push a token from one node to another. In stream graphs, if $\alpha = 1$ and $\beta = 1$, then a direct-bind solution to this network would probably be chosen. In this case, we would like to rewrite the functions above such that the push and shift functions are inlined into the function body.

```

fireA(Node* me) {
    static int tmp = 1;
    tmp++;
    fire(B, &tmp);    // was me->out[0].push(&tmp)
}

fireB(Node* me, ...) {
    int t;
    *dst = *(var_args_get(0)); // was me->in[0].shift(&t);
    printf("%i\n", t);
}

```

Notice how we are using the stack to pass the token from one call to another. This is clever stuff! If we were even more clever, we could pass the parameters directly using a register as is done with inlining compilers.

Unfortunately, the network might change, causing us to have to switch from a direct firing to an indirect firing. We would have to rewrite the functions to use some sort of persistent buffer:

```

fireA(Node* me) {
    static int tmp = 1;
    tmp++;
    *me->out[0].buffer_tail = &tmp;    // was me->out[0].push(&tmp)
    me->out[0].buffer_tail++;           // kludge, should be a ring buffer
    /* check to see if the buffer needs to be resized */
}

fireB(Node* me, ...) {
    int t;
    *dst = *(me->in[0].buffer_tail); // was me->in[0].shift(&t);
    me->in[0].buffer_tail++;           // kludge, should be a ring buffer
    printf("%i\n", t);
}

```

Unlike before, we must set up a buffer for this call when the edge is bound. Importantly, this buffer must be variably sized, because though this graph is simple and we know that the nodes will be called by the queue scheduler one after the other, more complex graphs might not guarantee such a predictable ordering.

In a stream graph, there will be multiple efficient implementations of a node (and scheduler loop) that will be brought into existence as the graph changes. In the next section, I show how this is just an instance of overloading, and in the following section, I show how to dynamically generate new functions from a base machine code implementation without the need for recompilation. These two concepts combined allow very efficient execution of stream graphs.

6.1 Node Overloading

Stream graph nodes can have multiple implementations based on the current signature on the node *and* its current bindings. This amounts to having multiple functionally equivalent implementations of the same node with slightly different signature information. We call this node overloading because of its obvious analogy to function overloading.

A node might have several overloaded functions as shown in the following example:

```

/* code to efficiently fire B when A binds directly to B
   and when ALPHA==BETA */
fireA(Node* me) < me->out[0] = B->in[0],
                me->outRecs[0] == B->inRecs[0] == 1 > {
}

/* code to efficiently fire when we have done a direct binding
   to some unknown node */
fireA(Node* me) < me->out[0] is DIRECT > {
}

/* code for when we're not bound in the previous two ways */
fireA(Node* me) {
}

```

Overloading allows creation of several plausible node functions. When an edge's binding is set or changed at runtime, the list of available functions for each affected node is searched for one matching the node's current bindings, as noted in the angle brackets in the above example. If no match is found, the edge change has to be rejected. If a match is found, a certain amount of setup must occur: buffers and state information are allocated and the firing functions on each node are set to point at the newly chosen functions.

It may be possible, as I show in a later section, to automatically generate overloaded functions at runtime.

This approach allows us to remove the set of function pointers for I/O to and from the node. We can apply a similar technique to schedulers: if a schedule is generated for a graph that will contain a set of known function calls, we can compile a version of that schedule with a matching signature and use it whenever we reach that state.

6.2 Automatically Generating Overloaded Nodes using Linking

As we saw in the overloading discussion, it is conceptually easy to generate a set of node functions with different underlying implementations for different scheduling scenarios. While it might be possible to anticipate *some* node configurations at design time, we cannot anticipate all configurations, especially when networks are dynamically created at runtime. Here, we create a black-box system that allows us to modify (annotated) machine code so that retrieval of tokens from a queue, pushing tokens to a queue, and determination/invoke of the next schedule action can all be inlined into a function.

In order to create new permutations of a base function at runtime, it is convenient to agree upon the basic operations that we are trying to support. We model token transfer as a queue with push and shift operations to add and remove tokens from the queue. FIFO ordering is assumed, but as I will show in the example section at the end of the thesis, not necessary.

To help guide this discussion, keep in mind the basic structure of the C-linkage problem, off of which this scheme is based. Note that I am generalizing so that we stick to the point: true C linkage is more involved than I suggest here. In ELF or COFF-style C linkage, the compilation process generates a segmented file that consists of (minimally) a `.text` and `.data` segment. The `.text` section contains assembly code for each function, while the `.data` section contains any globals defined by the source. When a function references a global in the data section or an external symbol (`extern`), a relocation table is appended to this file which specifies:

1. The name of the symbol referenced
2. The location in the machine code (`.text`) that references this symbol

When a symbol is specified in the source, it is left as a null value in the machine code. During the linkage step, the linker maps all of the input sections to one linear address space and then processes the relocation records, rewriting the machine code to use absolute locations instead of nulls. As we all know, this process can be elegantly extended to defer certain linkages to runtime. Furthermore, this process has been extended to support overloading by changing symbol names: the functions `foo(int)` and `foo()` have different "mangled" names in the object file. When a call is made

to a specific function, it is mangled and then indexed in the symbol table. We try to extend these basic elegant ideas to work for stream graph functions.

We propose a model in which every node function can be decomposed into three parts: (1) code that brings tokens from edges into the local frame, (2) code that pushes tokens from the current frame back into a queue for further scheduling, and (3) everything else (the “function code”). Importantly, we insist that code to load a token from a port will not be interrupted in the middle by other code. This insistence does not extend to the overall input process: input, output and computation can be mixed together in any order. This insistence allows us to break the function into the following components:

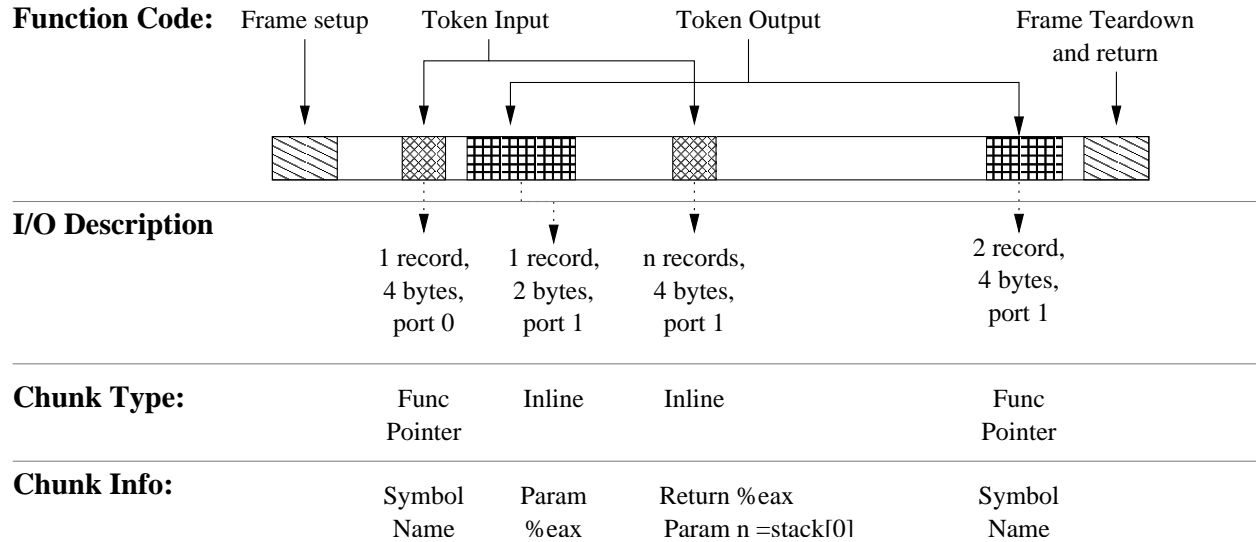


Figure 6.1: A stream processing function’s anatomy. Note how input and output chunks are interleaved with regular computation (white). The internal representation of the function affects how easy it is to modify these chunks.

For this discussion, we make the assumption that the function is implemented in machine code because it is by far the hardest to analyze. The assumption is that for more readable representations, the following analysis procedure will be considerably simplified. Our choice of machine language is further motivated by the fact that, if machine language can be re-written, then we do not have to deal with any intermediate steps necessary to re-compile the node.

In reality, C linkage does not worry about calling conventions nor does it change the overall length of a function. As a result, a variety of simplifying assumptions can be made in the object file format to simplify its representation. We can make no such assumptions. One nice assumption, however is possible: all of our linkage takes place at runtime. While there ways to extend the idea back to compile time, they cost a great deal of pain for proportionally little consequence, because the runtime linkage should be just as fast as the compile-time equivalent.

As mentioned previously, the first optimization replaces calls to function pointers with absolute jumps. This is possible, according to the Chunk-Type field, for the first input chunk and similarly the last input chunk. This corresponds to pieces of code that do the following:

```
void Fire(Node* me) {
    ...
    me->in[0].shift(1 record, 4 bytes);
    ...
}
```

This would translate in machine code to a series of operations that look up the address of the `shift` field in the `me->in[0]` structure, some frame setup, and then a jump. At runtime, we know the full binding, that is, the address of the appropriate `shift` function and even its parameter. Using the I/O information for the chunk, we can replace the chunk with our own parameter setup and absolute jump calls. You are probably wondering how we find the chunk: ignore this for a minute... we’ll come back to it.

The big issue with this is that we might change the length of this function. This is of consequence to any machine code instruction that is program-counter-relative: for example, we might have the following code:

```
Byte 0x00: void Fire() {
    ...
    <chunk 0>
    ...
}

Byte 0x14: void SomethingElse() {
    ...
    call -14 <addressof Fire>
}
```

Because of this transformation, we must rewrite all of the affected downstream segments that perform backward to anywhere past the start address of the changed chunk. While this requires some unpleasant machine code analysis, it is conceptually possible to do this. If this proves impossible for certain architectures, a fallback strategy is to abuse the NOOP principle by padding a function at compile time. However, this limits the amount of code that can be generated at runtime and may lead to code bloat.

The next issue is inlining push/shift code during linkage. To do this, we need several pieces of information: for pushes, we need to know where our record is located and related size information. For shifts, we need to know where to place the record. At a machine level, this means interfacing the compiler's mapping of the function code to registers or memory becomes necessary. If we can figure out where our parameters and output goes, we already have the ability to grow or shrink the method's code, and thus, the ability to inline the function.

The method code that we inline, of course, has to be well-behaved: it should leave the stack and registers in a predictable state. Because of this, inlined functions need to be manually created rather than generated automatically from a compiler.

This discussion assumes that chunks can be located in the machine code and that variable to register mappings can be conveyed in the machine code. Chunk location can be done by including in the original source code jumps to special undefined symbols that, at runtime, allow us to identify the location of the port where the jump occurs. Mapping program variables to registers and stack locations, for both input and output purposes, is done by using inline-assembly functionality present in modern C compilers: most compilers allow "escaping" into assembly that, optionally, allows the use of C-variables in the actual assembly. By writing macros that convert to assembly-with-variable-references in the original C code, we can leave enough information to recover register mappings. A fallback strategy to this process is, of course, to use embedded debugging information.

Even though this idea takes scarcely a whole page to present, it is best to implement. I have only implemented basic components of this scheme for this thesis: detecting a chunk, loading it at runtime, and so on. The power of the approach, however, is that we can create code at runtime that can produce code that is nearly equivalent to static code created by regular dataflow systems.

Chapter 7

Conclusion and Future work

In this paper, I have presented the stream graph as a way to model a variety of problems, some practical and some esoteric. I have shown how this model expands upon the basic set of problems that are already solvable in other stream programming languages. Furthermore, I have shown a class of dynamic data flow problems that can be solved efficiently in the stream graph model that have no efficient solution elsewhere.

As I pointed out, this comes at the cost of a more complex scheduling problem. I have improved on basic blind dataflow scheduling using the binding approach of chapter 5. Binding causes a set of instantaneous node traversals to ripple outward from a single node activation to minimize the amount of time spent in scheduling logic and thus maximize the amount of time spent executing real code.

Because stream graphs require the majority of the compilation process to occur at runtime, I have proposed a dynamic linking technique for machine code that minimizes the number of indirect buffer references and function pointer calls made when executing a stream graph.

This thesis explores stream graphs in a breadth-first rather than depth-first manner. A variety of problems have been posed here each of which should be independently studied:

- Implementation:
 - Implement the dynamic linker
 - Implement an entire stream graph system
- Research:
 - How can stream graphs be reconciled with Kahn's ordering requirements so that computation is always independent of scheduling order?
 - Alternative models to stream graphs with similar goals in mind:
 - * Stream graphs without graph modification but with signature modification
 - * Techniques for scheduling static token flow graphs that use signatures
 - * Study of graphs that self-modify with state but without worrying about signature robustness
 - How do object orientation and data typing relate to the idea of signatures? Are signatures the analog of data types for token flow.
 - Compilation/execution of stream graphs onto existing hardware designs
 - Idealized hardware designs for stream graph execution

The dynamic linkage system I propose is useful for a variety of purposes outside of this thesis: current techniques for dynamic linking are out of date and few C-compatible alternatives exist that are state-of-the-art save the approach proposed here.

I would like to close by pointing at the utility of any stream graph implementation. Chapter 4 is devoted to pointing out the applications of stream graphs in a neutral fashion. Here, let me sell the idea more aggressively: a

wide variety of systems already implement lightweight versions of stream graphs every time they try to solve a token-flow-like problem. Some of these projects are research oriented and some are practical in nature: both can benefit from the availability of stream graphs. They can be used as the base mechanism on top of which can be built a more sophisticated data processing system. This applies to database management systems, where stream graphs can provide the base services not only to perform scheduling but also to implement dynamic optimization and partitioning. Stream graphs can be used in the graphics and multimedia area as a front-end description of high and low-level graphics processes and in the multimedia field as the base framework for a wide variety of media applications. All of the fields extolled by recent hardware research — for example, vision, numeric simulation, signal processing — can use stream graphs as a base on which to build applications. I recognize that there is a large gap between the fact that (a) stream graphs can represent all these problems and (b) the choice to use stream graphs for doing so. Yet if the first half of this can be done, implementing a stream graph, then the rest of these dreams can become a distant yet possible reality.

7.1 Special Thanks

My friends and family have been instrumental in me surviving this project: from the beginning, I have taken on more than permitted by time and reason; these people have been instrumental in getting me through it without being consumed in the process. Thank you Richard and Meridith, Dave, Crag, Allison, Laura and Alice, Josh and Josh, Jack, Mike, Mike and Mike, and anyone else who has dared to read this pile of paper or listen to me rant about it.

On the professional side of things, I would like to thank Dr. Kosaraju at Hopkins for his comments, mentorship, and for pushing me hard throughout the process. I also wish to thank Dr. Jonathan Cohen, also of Hopkins, for his support on early phases of this project and many long discussions since then in the connection between this thesis and graphics. My former colleagues at IBM Research, Drs. Thomas Jackman, Peter Kirchner, James Klosowsky provided much of the inspiration for work by pointing out many of the streaming aspects of graphics hardware to me in the first place. Additionally, I want to thank Dr. Suresh Venkatasubramanian for making me aware of the larger applications of stream processing outside of graphics and dataflow fields that really got this project going.

Bibliography

- [1] The opengl 1.5 specification. (online) <http://www.opengl.org/documentation/spec.html>, 2004.
- [2] ABADI, D., ET AL. Aurora: A data stream management system. In *SIGMOD* (2003), p. 666.
- [3] ARASU, A., BABCOCK, B., BABU, S., MCALISTER, J., AND WIDOM, J. Characterizing memory requirements for queries over continuous data streams. In *PODS* (2002), p. 221.
- [4] AVNUR, R., AND HELLERSTEIN, J. M. Eddies: Continuously adaptive query processing. In *SIGMOD* (2000), p. 261.
- [5] BABCOCK, B., BABU, S., DATAR, M., MOTWANI, R., AND WIDOM, J. Models and issues in data stream systems. In *PODS* (2002), p. 1.
- [6] BABCOCK, B., DATAR, M., AND MOTWANI, R. Sampling from a moving window over streaming data. In *SODA* (2002), p. 633.
- [7] BABCOCK, B., DATAR, M., MOTWANI, R., AND O'CALLAGHAN, L. Maintaining variance and k-medians over data stream windows. In *PODS* (2003), p. 234.
- [8] BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRODER, P. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. In *SIGGRAPH* (2003), p. 917.
- [9] BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., AND HANRAHAN, P. Brook for gpus: Stream computing on graphics hardware. In *SIGGRAPH* (2004).
- [10] BUCK, J. T. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, U.C. Berkeley, 1993.
- [11] CARNEY, D., CETINTEMEL, U., MITCH CHERNIACK, S. L., SEIDMAN, G., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. Cs-02-01: Monitoring streams - a new class of dbms applications. Tech. rep., Department of Computer Science, Brown University, 2002.
- [12] CHANDRASEKARAN, S., ET AL. Telegraphcq: Continuous dataflow processing. In *SIGMOD* (2003), p. 668.
- [13] CHEN, J., DEWITT, D., TIAN, F., AND WANG, Y. Niagracq: A scalable continuous query system for internet databases. In *SIGMOD* (2002), p. 379.
- [14] DALLY, B., KAPASI, U., EREZ, M., SEREBRIN, B., KNIGHT, T., AND AHN, J. H. Streaming supercomputer project: Strawman architecture document. [online] <http://merrimac.stanford.edu/resources.html>, 2003.
- [15] DANSKIN, J., AND HANRAHAN, P. Compression performance of the xremote protocol. In *Proc. Data Compression Conference* (1994).
- [16] DATAR, M., GIONIS, A., INDYK, P., AND MOTWANI, R. Maintaining stream statistics over sliding windows (extended abstract). In *SODA* (2002), p. 635.
- [17] DENNIS, J. Data flow supercomputers. In *IEEE Computer* (November 1980), vol. 13.

- [18] DENNIS, J. B. A preliminary architecture for a basic dataflow processor. In *2nd Annual Symposium on Computer Architecture* (May 1975).
- [19] DOBRA, A., GAROFALAKIS, M., GEHRKE, J., AND RASTOGI, R. Processing complex aggregate queries over data streams. In *SIGMOD* (2002), p. 61.
- [20] GOLAB, L., AND OZSU, M. T. Issues in data stream management. In *SIGMOD* (2003), p. 5.
- [21] GORDON, M. I., THIES, W., ET AL. A stream compiler for communication-exposed architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2002), vol. 10.
- [22] HENZINGER, M., RAGHAVAN, P., AND RAJAGOPALAN, S. Computing on data streams. In *External Memory Algorithms* (1998), vol. 50 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*.
- [23] HUMPHREYS, G., HOUSTON, M., NG, R., FRANK, R., AHERN, S., KIRCHNER, P., AND KLOSOWSKI, J. Chromium: A stream processing framework for interactive rendering on clusters. In *SIGGRAPH* (2002).
- [24] IGEHY, H., STOLL, G., AND HANRAHAN, P. The design of a parallel graphics interface. In *Proceedings of SIGGRAPH 1998* (1998), pp. 141–150.
- [25] KAHN, G. The semantics of a simple language for parallel programming. In *IFIPS 74* (1974), pp. 471–475.
- [26] KAPASI, U., MATTSON, P., DALLY, W. J., OWENS, J. D., AND TOWLES, B. Stream scheduling. In *The 3rd Workshop on Media and Stream Processors* (2001).
- [27] KARZMAREK, M., THIES, W., AND AMARASINGHE, S. Phased scheduling of stream programs. In *SIGPLAN* (2003).
- [28] KHAILANY, B., DALLY, W., KAPASI, U., MATTSON, P., NAMKOONG, J., OWENS, J., TOWLES, B., CHANG, A., AND RIXNER, S. Imagine media processing with streams. 35–47.
- [29] KRUGER, J., AND WESTERMANN, R. Linear algebra operators for gpu implementation of numerical algorithms. In *SIGGRAPH* (2003), p. 908.
- [30] LAMB, A. A., THIES, W., AND AMARASINGHE, S. Linear analysis and optimization of stream programs. In *Programming Language Design and Implementation* (2003).
- [31] LEE, E., AND MESSERSCHMITT, D. Static scheduling of synchronous dataflow graphs for digital signal processing. In *IEEE Trans. on Computers* (January 1987).
- [32] LEE, E., AND MESSERSCHMITT, D. Synchronous data flow. In *IEEE Proceedings* (September 1987).
- [33] MADDEN, S., AND FRANKLIN, M. J. Fjording the stream: An architecture for queries over streaming sensor data. In *18th International Conference on Data Engineering* (2002).
- [34] MADDEN, S., SHAH, M., HELLERSTEIN, J. M., AND RAMAN, V. Continuously adaptive continuous queries over streams. In *International Conference on Data Engineering* (2002), p. 49.
- [35] MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. Cg: A system for programming graphics hardware in a c-like language. In *SIGGRAPH* (2003), p. 896.
- [36] MUTHUKRISHNAN, S. M. Data streams: Algorithms and applications. [online] <http://athos.rutgers.edu/muthu/>, 2003.
- [37] NAVAS, J., AND WYNBLATT, M. The network is the database: Data management for highly distributed systems. In *SIGMOD* (2001), p. 544.
- [38] OWENS, J., DALLY, W., KAPASI, U., RIXNER, S., MATTSON, P., AND MOWERY, B. Polygon rendering on a stream architecture. In *SIGGRAPH* (2000).

- [39] RAMAN, V., RAMAN, B., AND HELLERSTEIN, J. M. Online dynamic reordering for interactive data processing. In *Proceedings of the 25th VLDB Conference* (1999).
- [40] RIXNER, S., AND ALL. Register organization for media processing. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture* (2000).
- [41] RIXNER, S., DALLY, W. J., ET AL. A bandwidth-efficient architecture for media processing. In *Micro-31* (1998), p. 1.
- [42] SHANMUGASUNDARAM, J., ET AL. Architecting a network query engine for producing partial results. In *Proc. 2000 International Workshop on the Web and Databases* (May 2000), pp. 17–22.
- [43] SIH, G. *Multiprocessor Scheduling to Account for Interprocessor Communication*. PhD thesis, U.C. Berkeley, 1991.
- [44] TAYLOR, M., AND ALL. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *International Symposium on Computer Architecture (to appear)* (June 2004).
- [45] THIES, W., KARCZMAREK, M., AND AMARASINGHE, S. Streamit: A language for streaming applications. In *International Conference on Compiler Construction* (2002).
- [46] VIGLAS, S. D., AND NAUGHTON, J. F. Rate-based query optimization for streaming information sources. In *SIGMOD* (2002), p. 37.