# Modeling the Scalability of Acyclic Stream Programs

by

Jeremy Ng Wong

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 2004

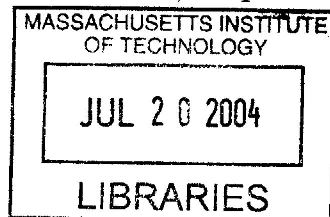
[February 2004]

© Massachusetts Institute of Technology 2004. All rights reserved.

Author ....................
Department of Electrical Engineering and Computer Science
January 30, 2004

Certified by ........................................
Saman P. Amarasinghe
Associate Professor
Thesis Supervisor

Accepted by ...........
Arthur C. Smith
Chairman, Department Committee on Graduate Students


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 2 0 2004
LIBRARIES

BARKER

# Modeling the Scalability of Acyclic Stream Programs

by

Jeremy Ng Wong

Submitted to the Department of Electrical Engineering and Computer Science
on January 30, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

Despite the fact that the streaming application domain is becoming increasingly widespread, few studies have focused specifically on the performance characteristics of stream programs. We introduce two models by which the scalability of stream programs can be predicted to some degree of accuracy. This is accomplished by testing a series of stream benchmarks on our numerical representations of the two models. These numbers are then compared to actual speedups obtained by running the benchmarks through the Raw machine and a Magic network. Using the metrics, we show that stateless acyclic stream programs benefit considerably from data parallelization. In particular, programs with low communication datarates experience up to a tenfold speedup increase when parallelized to a reasonable margin. Those with high communication datarates also experience approximately a twofold speedup. We find that the model that takes synchronization communication overhead into account, in addition to a cost proportional to the communication rate of the stream, provides the highest predictive accuracy.

Thesis Supervisor: Saman P. Amarasinghe
Title: Associate Professor

# Acknowledgments

I would like to thank Saman Amarasinghe for his patience and guidance throughout this thesis project, in helping me formulate the original ideas as well as refining them throughout the entire process. I'd also like to thank the entire StreamIt group at CSAIL, for providing me with the tools to learn more about streaming programs than I ever wanted to know!

In particular, I'm eternally grateful to Bill Thies, whose guidance in all aspects was both instrumental and inspirational. I've spent countless hours badgering him about any and all things related to Computer Science, and have learned more from him than I could have ever imagined.

Without my friends, both from MIT and from before college, there's no way I would have been able to survive five and a half years of this school. From lending an ear upon which I could vent, a pair of eyes to proofread my writing, or a similarly procrastinating soul with whom I could play Starcraft until the morning hours, they have been essential in maintaining my sanity.

Lastly, I would like to thank my parents, Joseph and Mary June, and my siblings, Jonathan and Adrienne. I feel truly blessed to have had their love and support in the past, and feel extremely lucky in knowing I'll have it for the rest of my life.

Whoohoo!

# Contents

**1 Introduction** **15**

1.1 Problem Description . . . . . . . . . . . . . . . . . . . . . . . . . . 15

1.2 Prior Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

1.3 General Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

**2 Background and Methodology** **21**

2.1 The Raw Machine . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

2.2 Magic Network . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

2.3 The StreamIt Language and Compiler . . . . . . . . . . . . . . . . . 23

2.3.1 The StreamIt to Raw Compiler Path . . . . . . . . . . . . . . 26

2.4 Description of Steady State . . . . . . . . . . . . . . . . . . . . . . 28

2.5 Filter Fission . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

2.6 Partitioning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

**3 Benchmarks** **31**

3.1 Pipeline Benchmarks . . . . . . . . . . . . . . . . . . . . . . . . . . 34

3.2 Splitjoin Benchmarks . . . . . . . . . . . . . . . . . . . . . . . . . . 36

3.3 Roundrobin Benchmarks . . . . . . . . . . . . . . . . . . . . . . . . 39

3.4 Testing the Benchmarks . . . . . . . . . . . . . . . . . . . . . . . . 39

**4 Basic Cost Metric** **41**

4.1 The Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

4.2 Results and Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . 43

4.2.1 Cost Type Comparison . . . . . . . . . . . . . . . . . . . . . . . 43
4.2.2 Metric Comparison . . . . . . . . . . . . . . . . . . . . . . . . . 46

**5 Synchronization Cost Metric** **57**
5.1 The Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
5.2 Results and Analysis . . . . . . . . . . . . . . . . . . . . . . . . 59

**6 Conclusion** **69**
6.1 Results Summary . . . . . . . . . . . . . . . . . . . . . . . . . . 69
6.1.1 Metric Validation . . . . . . . . . . . . . . . . . . . . . . . . . 69
6.1.2 Scalability Analysis . . . . . . . . . . . . . . . . . . . . . . . . 73
6.2 Current State . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
6.3 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75

**A Metric Implementation File** **77**
A.1 ParallelizationGathering.java . . . . . . . . . . . . . . . . . . . . 77

**B Benchmark Source Files** **97**

**C Simulation Graphs for Round Robin Splitting** **103**

# List of Figures

2-1 Raw Tile Block Diagram . . . . . . . . . . . . . . . . . . . . . . . . 22
2-2 A StreamIt Pipeline . . . . . . . . . . . . . . . . . . . . . . . . . 24
2-3 A Splitjoin Pipeline . . . . . . . . . . . . . . . . . . . . . . . . . 25
2-4 A Sample Raw Layout . . . . . . . . . . . . . . . . . . . . . . . . 27
2-5 Steady State Example . . . . . . . . . . . . . . . . . . . . . . . . 28
2-6 Pictoral Representation of Fissed Filters' cost characteristics Pre-fissing 30
2-7 Pictoral Representation of Fissed Filters' Cost Characteristics Post-fissing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

3-1 Stream Graphs of Communication Heavy Pipelines . . . . . . . . . . . 35
3-2 Stream Graphs of Computation Heavy Splitjoins . . . . . . . . . . . . 37
3-3 Stream Graphs of Communication Heavy Splitjoins . . . . . . . . . . 38
3-4 Stream Graphs of Round Robin Splitjoins . . . . . . . . . . . . . . . 39

4-1 Cost Distribution for Computation Dominated Pipelines . . . . . . . 44
4-2 Cost Distribution for Communication Dominated Pipelines . . . . . . 45
4-3 Graphical comparison of Basic Metric to Raw simulator for Computation Heavy Pipelines . . . . . . . . . . . . . . . . . . . . . . . . . 48
4-4 Graphical comparison of Basic Metric to Raw simulator for Communication Heavy Pipelines . . . . . . . . . . . . . . . . . . . . . . . . 49
4-5 Graphical comparison of Basic Metric to Raw simulator for Computation Heavy splitjoins . . . . . . . . . . . . . . . . . . . . . . . . . 51
4-6 Graphical comparison of Basic Metric to Raw simulator for Communication Heavy splitjoins . . . . . . . . . . . . . . . . . . . . . . . . 52

4-7 Graphical comparison of Basic Metric to Raw simulator for Round Robin splitjoins . . . 53

4-8 Synchronization metric accuracy vs. Magic Network and Raw simulations for pipelines . . . 55

4-9 Synchronization metric accuracy vs. Magic Network and Raw simulations for splitjoins . . . 55

5-1 Graphical comparison of Synchronization Metric to Raw simulator for Computation Heavy Pipelines . . . 60

5-2 Graphical comparison of Synchronization Metric to Raw simulator for Communication Heavy Pipelines . . . 61

5-3 Graphical comparison of Synchronization Metric to Raw simulator for Computation Heavy splitjoins . . . 63

5-4 Graphical comparison of Synchronization Metric to Raw simulator for Communication Heavy splitjoins . . . 64

5-5 Graphical comparison of Synchronization Metric to Raw simulator for Round Robin splitjoins . . . 65

5-6 Synchronization metric accuracy vs. Magic Network and Raw simulations for pipelines . . . 66

5-7 Synchronization metric accuracy vs. Magic Network and Raw simulations for splitjoins . . . 66

6-1 Metric Comparison of Pipelines with Respect to Magic Network . . . 70

6-2 Metric Comparison of Pipelines with Respect to Raw . . . 70

6-3 Metric Comparison of Splitjoins with Respect to Magic Network . . . 71

6-4 Metric Comparison of Splitjoins with Respect to Raw . . . 72

C-1 Graphical comparison of Synchronization Metric to Raw simulator for Computation Heavy Pipelines . . . 104

C-2 Graphical comparison of Synchronization Metric to Raw simulator for Communication Heavy Pipelines . . . 105

C-3 Graphical comparison of Synchronization Metric to Raw simulator for Computation Heavy Splitjoins . . . . . . . . . . . . . . . . . . . . . . . 106
C-4 Graphical comparison of Synchronization Metric to Raw simulator for Communication Heavy Splitjoins . . . . . . . . . . . . . . . . . . . . . 107

# List of Tables

3.1 Key Benchmark Characteristics . . . . . . . . . . . . . . . . . . . . . . 32
3.2 Benchmark Suite for Metric Tests . . . . . . . . . . . . . . . . . . . . . 33

4.1 Speedup comparison of Benchmarks . . . . . . . . . . . . . . . . . . . 54

6.1 Projected Benchmark Speedup with Synchronization Metric . . . . . 73

# Chapter 1

# Introduction

## 1.1 Problem Description

As general purpose computer processors become increasingly powerful with respect to both sheer speed and efficiency, novel approaches to utilizing them can be explored. In particular, algorithms that were once implemented solely as specialized hardware are increasingly being implemented in software. This trend is spurred on in part due to the increasing importance of what are called "stream" programs. In the simplest definition, a stream program is one that takes as an input a potentially infinite amount of data over a large amount of time, and likewise outputs a potentially infinite amount of data. From basic Digital Signal Processing (DSP) algorithms such as the Fast Fourier Transform (FFT) to consumer-oriented applications such as MP3 or HDTV, it has even been surmised that such streaming applications are already consuming the majority of cycles on consumer machines [7].

There have been numerous attempts to facilitate the programming of such software implementations. These include specialized programming languages and compilers such as StreamIt [3] [12] and Brook [2] [4]. Among the biggest challenges that software implementations face is that they must match or outperform the realtime performance that traditional specialized hardware achieves. To that end, streaming-application compilers often include multiprocessor support to address the speed issue. However, there have been few studies specifically aimed at measuring the general scalability of

stream programs when split across multiple computation units.

It has been generally theorized that data parallelism will generally produce a speedup for most stream programs, but general purpose models of this have yet to be defined. This thesis aims to model this large scale behavior by introducing numerical models that can accurately predict the performance of stream programs when they are parallelized. We accomplish this by defining two metrics of predicting the speedup of stateless acyclic stream programs, and evaluating the metrics by comparing their small scale performance against two existing networks. We discovered that for this subset of streaming programs, they are indeed scalable across multiple processor units, though this speedup is limited primarily by the amount of internal communication ocurring in the stream.

## 1.2 Prior Work

The field of exploring data parallelization is a well-studied one, with numerous contributions already made. However, their applicability is not specific to stream programs and this unique application domain has yet to be comprehensively studied. There exists some prior work in the field, which was useful for us to establish a starting point in our own modeling work.

Early work by Subhlok, O'Hallaron et al. [8] studies optimal data parallel mappings for stream programs, in particular how the program characteristics affect overall performance for various mappings. It effectively introduces a communication cost model based on the number of items that a stream program consumes and internally routes, which is an approach that this thesis will also use. It also looks at stream program characteristics from a high level such that the study can be applicable towards multiple architectures. However, their performance measurements are based on low level compiler scheduling control, effectively trying to compensate for certain characteristics in stream programs by optimizing the compiler to improve performance. They also do not explore the effects of any characteristics specific to a particular stream program on its overall scalability, which this thesis does. Finally , the work

does not test their models on any real life architectures, making it less usable as a way of modeling aspects of stream programs in general.

Later work by Subhlok and Vondran [9] addresses the concern of applicability by testing their optimal mapping of parallel programs to an existing compiler and testbed, also introducing a more mathematically rigorous justification for their models. They test their mapping models on a wide variety of benchmarks, a test methodology that this thesis borrows from. However, their approach is still based on low level compiler control, which this thesis abstracts away. Instead, this thesis focuses primarily on aspects of the stream programs themselves, which compilers are able to optimize if they wish.

Lipton and Serpanos [6] develop a communication cost model for multiprocessors, with a focus on measuring the scalability of their model as the number of processors increase. This is quite relevant to our work due to its focus on scalability. However, the model they present is quite simplistic, using a uniform communication cost regardless of the number of processors. In addition, they fail to take into acocunt any notion of computation cost for test programs, which can be significant in stream programs. They test their metrics on a very specialized machine (PRAM network with PLAN switches), further limiting its general purpose applicability.

Recent work by Suh, Kim, and Cargo et al [10] perhaps provides the greatest relevance to measuring the performance of stream programs on numerous possible architectures. They introduce numerous relevant stream program benchmarks, and test their potential speedups on three different architecture types, using real systems. Their fundamental approach, however, is different than that of this thesis. The primary goal of their work is to compare the effectiveness of the three differing architectures with respect to their performance, using a common set of benchmarks. While we essentially only validate our models on one of these architecture types, the intent is for the metrics we introduce to be applicable towards others as well, with minimal changes being made to them.

## 1.3 General Overview

The goal of this project is to introduce two models by which one can predict the scalability of stream programs. These models should be apply to a large class of parallel architectures with distributed memory, and be as accurate as possible with respect to what it predicts. In order to do this, we take the following steps: Firstly, we describe our methodology in how we set up and test the numerical metrics that the models are comprised of. Then, we introduce a benchmark suite of stream programs that represent the various characteristics of stream programs. Next, we explain the two metrics that were developed, and compare our results to that of real simulations.

The methodology stage involves explaining our strategy in both creating and validating the two metrics. To this end, we introduce three key tools that are currently being used in the streaming application domain: *Raw*, *StreamIt*, and a *Magic Network*. The Raw machine is a scalable multiprocessor machine that is designed to efficiently parallelize input programs by giving the compiler maximum low level control, reducing many possible sources of overhead. The Magic Network is not a physical network, but instead is an idealized simulator of a tiled multiprocessor machine that is useful in showing best case performance. StreamIt is comprised of both a high level programming language and matching compiler that allows programmers to represent stream programs in a high-level abstraction. The compiler is then able to efficiently convert the high level code into machine language optimized for Raw. We also briefly describe the concept of Steady State, which is essential in measuring the performance characteristics of stream programs.

The benchmark chapter introduces ten sample stream programs that were decided to display a large range of possible characteristics in the streaming application domain. They are categorized by three main qualities, which will be described in a future chapter.

The two metrics in this thesis are introduced by first deriving their numerical form. The metrics are then tested by running the equations on the benchmarks, then comparing these results to when the same benchmarks are run through Raw and the

Magic Network. The results from the three sources are then analyzed with respect to each other, in an effort to validate the metric.

# Chapter 2

# Background and Methodology

Due to the specialized nature of stream programs, using standard programming languages and compilers to compare our metrics with real results would have resulted in numerous problems with respect to obtaining optimal performance. For instance, using a standard Java or C compiler would have severely limited the ability to test data parallelization by splitting a filter amongst multiple processors. Instead, we opted to use tools that were more applicable to stream programs as a whole. First, we introduce the Raw microprocessor, hardware that is well-suited to testing stream program scalability. Then, we introduce the Magic Network, a simulator alternative to the Raw architecture. Finally, we mention the StreamIt programming language and compiler, which enabled us to create high level programming models of stream programs and efficiently compile them to Raw.

## 2.1 The Raw Machine

To minimize any potential overhead incurred by data parallelization across multiple processors, we chose an architecture that allows the compiler to determine and implement an optimal resource allocation for a given stream. The Raw machine [1] [11] is a highly parallel VLSI architecture that accomplishes this. Instead of building one processor on a chip, several processors are connected in a mesh topology. As a result, each tile occupies a fraction of the chip space, so scalability of tiles benefits as well

as latency between them.

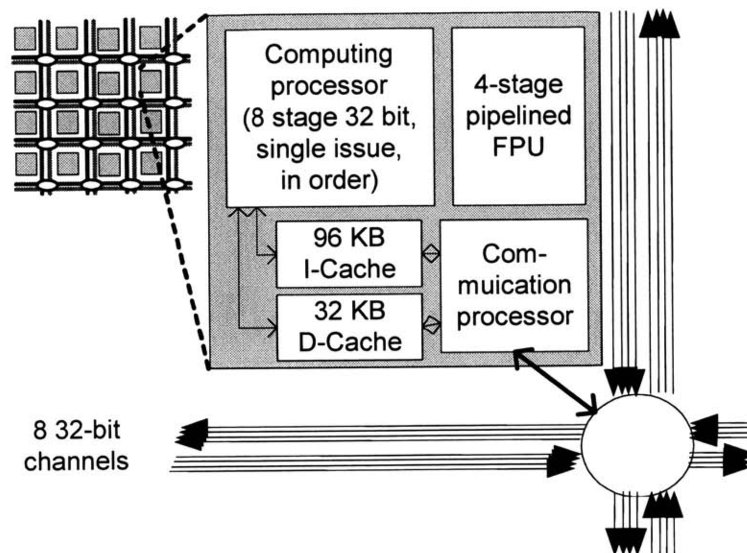



Figure 2-1: Raw Tile Block Diagram

Each tile in a Raw machine, as seen in Figure 2-1, contains a dedicated memory module and cpu. It also has a RISC-type pipeline and is connected to other tiles through a point-to-point network. This is incredibly useful for testing performance characteristics of stream programs, as it allows the compiler to directly assign which filters are placed on certain tiles, to optimize performance. Figure 2-4 is an example of a sample Raw layout when filters are assigned to tiles. The red arrows indicate the communication datapaths between the tiles.

However, since compilers and tools for Raw are still in development, there were numerous limitations in what we could simulate. Our tools currently only allow rectangular tile configurations, limiting the amount of datapoints that could be gathered. Furthermore, successful compilation beyond 16 tiles (4x4 layout) was inconsistent, which forced us to design benchmarks that were sufficiently small to fit into this layout and still have room to be parallelized. In addition, because Raw is a synchronized static network, the StreamIt compiler, as described in a following section, can suffer introduces considerable overhead in assuring synchronicity among all the tiles. There

exists finite size buffers on the communication channels as well, which also introduce additional delay. In order to minimize these extra costs, we opted to also simulate a tiled network that does not have these costs, entitled the Magic network.

## 2.2 Magic Network

The Magic Network is a pure simulation that assumes the characteristics of a Raw-type machine (low level compiler control, data parallelization), but allows exploration of architectural variants of Raw. It allows infinite buffering, removing delays from the previously mentioned finite sized communication buffers on each tile. In addition, it does not make any synchronization requirements between the tiles beyond the requirements of the processes being run, eliminating further overhead costs. The cycle latency per hop is a value that is settable by the user at compile time, allowing for variety in testing various benchmarks. Although this network does not exist as a processing entity, it is useful for comparison purposes, as the metrics we introduce are designed to be applicable for any stream program backend.

## 2.3 The StreamIt Language and Compiler

Since we decided on using the Raw machine, we sought a compiler that could seamlessly interface with Raw and provide an adequate high level programming language with which our benchmarks could be programmed and tested. For these purposes, StreamIt was the natural choice. It has both a programming language and compiler that are tailored towards stream programs and their effective implementation.

The StreamIt compiler contains both a Raw and Java backend, which was extremely useful in our simulation efforts. Using the Java backend, the two metrics we introduce in later chapters were implemented and tested. We could then compare these results with those of the Raw backend, using the same benchmark streams. Since support for Raw is already built into the compiler, no interfacing issues arose in our number gathering.

The StreamIt language provides a simple high level abstraction with which software engineers without lower level DSP knowledge can easily represent stream programs. Perhaps the most useful abstraction that it provides is its categorization of streaming programs. The basic building block is a filter, which on every invocation takes in (`pops`) a certain amount of data, processes it, and outputs (`pushes`) a certain amount at the other end. This is the same way streaming algorithms are viewed from a DSP perspective; consequently, they are highly applicable in this case as well. The language defines three constructs as those that can contain groups of filters to compose them into a network: pipelines, splitjoins, and feedback loops. Our metrics do not currently support feedback loops; we will focus on the former two.

Figure 2-2 shows a sample pipeline. Filters are arranged in series one after the other, and send/receive data in this sequential fashion. Figure 2-3 shows a sample splitjoin element. These contain independent parallel streams that diverge from a common *splitter* module and are combined into a common *joiner* module. The splitter and joiner can vary with respect to how data is taken in and out of them. For instance, there are two main splitter types, *duplicate* and *roundrobin*. A duplicate splitter simply sends identical copies of any data item it receives to its children parallel streams, while a roundrobin($n$,$m$) splitter sends $n$ items one way, then $m$ items the other way, and repeats the process. It is important to note that splitters need not be limited to 2 parallel streams; it can support an arbitrary amount of child streams.

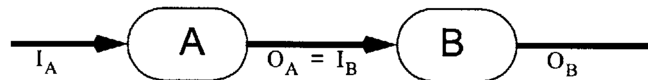



Figure 2-2: A StreamIt Pipeline

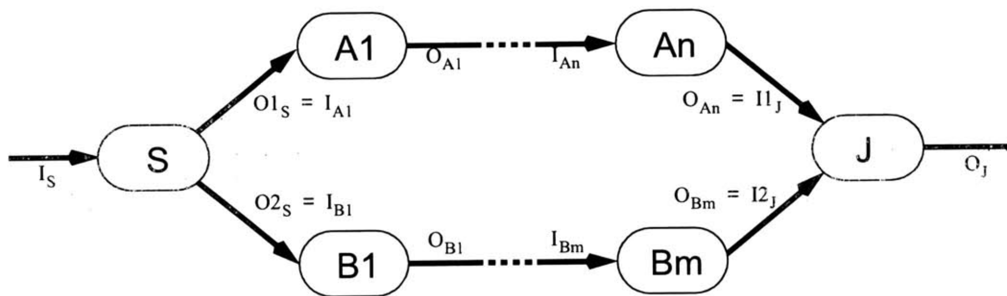



Figure 2-3: A Splitjoin Pipeline

### 2.3.1 The StreamIt to Raw Compiler Path

While details about the StreamIt compilation path are not necessary in either understanding or implementing the models espoused in this thesis, it is useful to show the compiler path as one that is not limited to a tiled architecture like Raw. A large part of the compiler's function is to determine an efficient way to either combine multiple filters on the same processing unit (*fusing*), or splitting a single filter amongst different units (*fissing*), based on the layout it is given.

The StreamIt compiler employs three general techniques that can be applied to compile the StreamIt language to not only the Raw backend, but for other architectures as well: 1) partitioning, which adjusts the granularity of a stream graph to match that of a given target, 2) layout, which maps a partitioned stream graph to a given network topology, and 3) scheduling, which generates a fine-grained static communication pattern for each computational element.

The StreamIt partitioner employs a set of fusion, fission, and reordering transformations to incrementally adjust the stream graph to the desired granularity. To achieve load balancing, the compiler estimates the number of executions by each filter in a given cycle of the entire program; then, computationally intensive filters can be split (fission), and less demanding filters can be combined (fusion). Currently, a simple dynamic programming algorithm is used to automatically select the targets of fusion and fission, based on a simple work estimate in each node.

The layout phase of the StreamIt compiler aims to assign nodes in the stream graph to computation nodes in the target architecture while minimizing the communication and synchronization present in the final layout. This layout assigns exactly one node in the stream graph to one computation node in the target. Because this occurs after the partition phase, a one-to-one mapping is guaranteed.

Figure 2-4 is an example of a sample Raw layout when filters are assigned to tiles. The red arrows indicate the communication datapaths between the tiles.

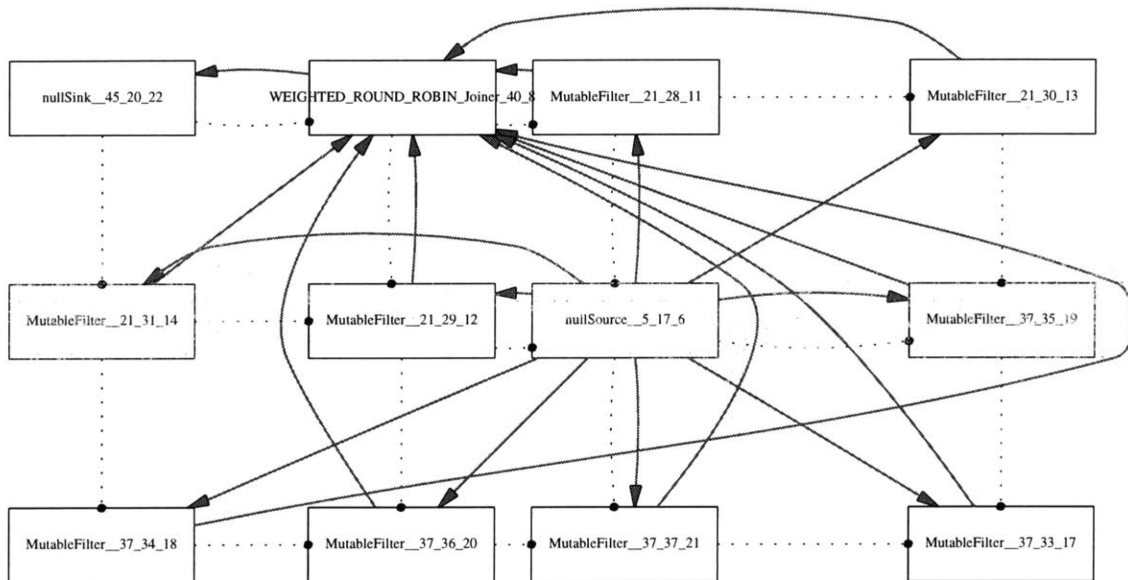



Figure 2-4: A Sample Raw Layout

## 2.4 Description of Steady State

In order to effectively measure our proposed scalability models, it is necessary to first understand key performance characteristics of stream programs. Among the most important of these is a given stream program's Steady State behavior.

A stream's Steady State is best defined as a set of multiplicities for each filter in a stream such that all data that is pushed onto a given channel is completely consumed. Another way of viewing this is that the overall number of data items produced in the stream is also consumed within the stream, for a single Steady State execution.

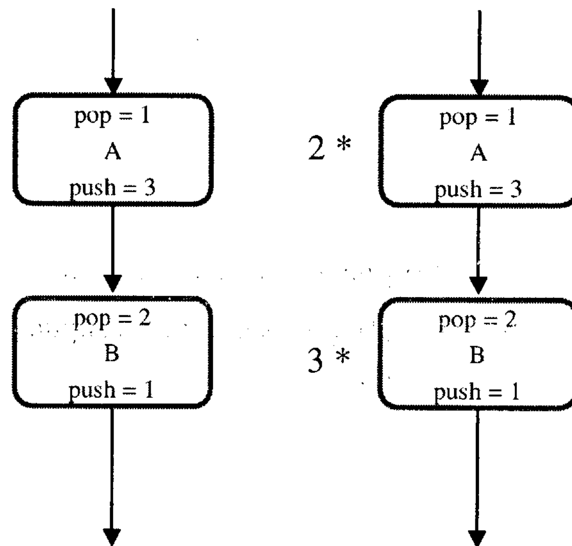



Figure 2-5: Steady State Example

Figure 2-5 shows an example of how a steady state schedule is determined. Because the first filter outputs three data items per execution and the second only consumes two items per execution, if each filter were just run once, then there would be one item left on the channel connecting the two. In order to ensure that this does not occur, in the Steady State, the first filter executes twice, and the second filter executes three times. This is shown in the picture on the right of Figure 2-5. Now, a total of six data items are outputted, and six are consumed by the following filter.

The usefulness of knowing a stream's Steady State is that any schedule that is aware of these multiplicities is able to repeat it endlessly. Because of this, we should

measure the computational characteristics of filters according to their Steady State schedules. In our study, the main computational quantity of each filter that is measured and compared is that of cycles per Steady State output. Due to the properties of the schedule, we can be assured that this quantity remains constant over time. A more detailed explanation of Steady State behavior can be found in [5].

## 2.5 Filter Fission

Among the primary aspects of this thesis involves the accurate modeling of filter fission. When a filter is first fissed into two filters to reduce the steady state computation cost, that filter in the stream is replaced by a splitter, two parallel fissed filters, and a joiner, in that order. Figures 2-6 and 2-7 show an example of this. Here, each the fissed filters now receive some subset of data that the original filter would have received, depending on the splitter module used. If a duplicate splitter is used, then the fissed filters get exactly the same data the original filter would have, and internally decimate the input based on the subset they are assigned to process. If a round robin splitter is used, then each splitter would only receive the data they are to process. However, using a round robin splitter is not without tradeoffs. While the overall communication bandwidth of the stream (and likewise, to the filters individually) decreases, it introduces other causes for overhead. This will be further explored in a following chapter.

It is important to note that this model is only valid for filters without internal state from iteration to iteration. If a filter was not stateless, then the StreamIt compiler is currently unable to fiss it.

## 2.6 Partitioning

As introduced earlier our description of the StreamIt to Raw compiler path, partitioning is a key aspect of exploring scalability. The entire process of filter fission is limited by the granularity with which the compiler can assign filters to tiles. This

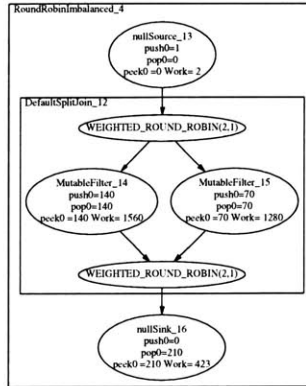



Figure 2-6: Pictoral Representation of Fissed Filters' cost characteristics Pre-fissing

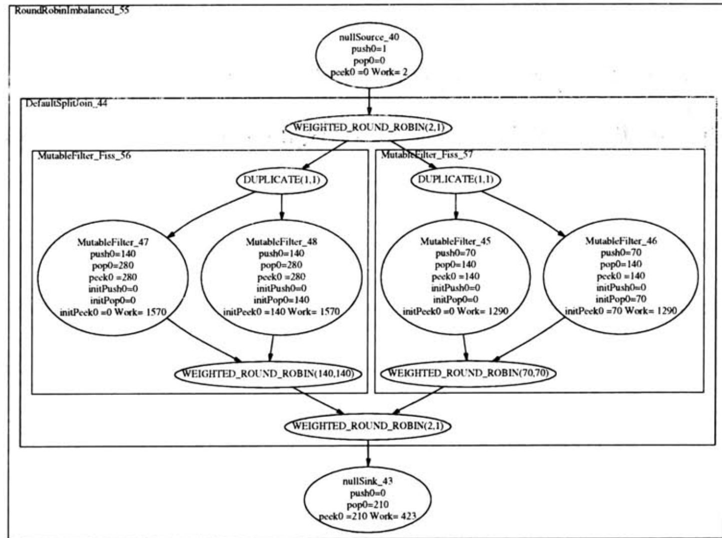



Figure 2-7: Pictoral Representation of Fissed Filters' Cost Characteristics Post-fissing

is a nontrivial thing to accomplish, as the compiler must be aware of which fissing will best benefit overall stream performance. In using the StreamIt compiler, we can be assured that partitioning is being done intelligently. More details about the partitioning algorithms StreamIt uses can be found in [12].

# Chapter 3

# Benchmarks

In order to efficiently and accurately test our metrics, we created a test suite of simple benchmarks whose purpose was to highlight specific characteristics of streaming programs such that the strengths and shortcomings of each metric could be easily visible. Each benchmark stream consists of a dummy source, a dummy sink, and two filters in between them. While it may seem at first glance that few combinations can be made with such a simple model, there are numerous characteristics of streaming programs that can be effectively tested with it.

Although the vast majority of existing programs in the streaming application domain contain far more than two basic filters, when these filters are *fissed*, or split, they are equivalent to more complex stream programs that had larger amounts of filters in their initial state. By starting from a base case of two, we were able to better follow the progression as filters were fissed. It is also important to note that filter *fusion*, or the event of placing more than one filter on the same processing tile, is not considered in this thesis. As a result, our simulations on these benchmarks were started from a number of processors *equal* to the number of filters in the overall stream.

Of the ten metrics that were created from this basis, eight can be categorized fully by the characteristics in Table 3.1.

It is important to note the distinction between pipelines and splitjoins, as described in the previous chapter. While the two constructs are quite different func-

| Stream Type | Dominating Cost | Relative Size |
|---|---|---|
| Pipeline | Computation | Balanced |
| Splitjoin | Communication | Imbalanced |

Table 3.1: Key Benchmark Characteristics

tionally, their differences are less signficant from a perspective of bottleneck cost. If two identical filters are placed in series versus in parallel, this layout difference should not affect which one of the two is the bottleneck in the stream.

The dominating cost characteristic can have either one of two values; either a stream's overall complexity is dominated by the amount of processing done on its input data, or the sheer amount of data that is passed within it. This difference is essential to distinguish in measuring overall scalability, as large amounts of computation are easily parallelizable among different tiles, whereas a high communication bandwidth is less affected by parallelization.

The relative size characteristic is a comparison of the two filters' overall cycle cost per steady state. In a balanced stream, it can be expected that each filter will be fissed evenly, since they have an identical amount of work in them initially. However, for imbalanced filters, the more work-heavy filter will be fissed much more than the work-light one. This is expected to produce a few deviations in terms of the progression of bottleneck work. For instance, it requires a total of two extra fissings for a balanced stream to produce any speedup, since both filters need to be fissed.

The final two benchmarks contain round robin splitters, and will be described separately. Appendix B contains the code for each of the ten benchmarks.

| Stream Type | Dominating Cost | Relative Size | Splitter |
|---|---|---|---|
| Pipeline | Computation | Balanced | n/a |
| Pipeline | Computation | Imbalanced | n/a |
| Pipeline | Communication | Balanced | n/a |
| Pipeline | Communication | Imbalanced | n/a |
| Splitjoin | Computation | Balanced | Duplicate |
| Splitjoin | Computation | Imbalanced | Duplicate |
| Splitjoin | Communication | Balanced | Duplicate |
| Splitjoin | Communication | Imbalanced | Duplicate |
| Splitjoin | Communication | Imbalanced | Roundrobin |
| Splitjoin | Communication | Imbalanced | Roundrobin |

Table 3.2: Benchmark Suite for Metric Tests

## 3.1 Pipeline Benchmarks

The stream graph of the computation heavy pipelines can be seen in Figure ??. The "peek" quantity in the stream graphs are irrelevant for the purposes of our metrics, and thus can be ignored. The "work" quantity is a simple work estimate of each filter, as calculated by the StreamIt backend. Both have almost no communication cost, with a push and pop rate of 2. However, they were given large amounts of computation to cycle through per invocation, which will dominate over the negligible communication cost. The balanced graph has two equally computationally heavy filters, while the imbalanced graph has one heavy filter and one lighter one whose computation cost is lesser than the other by a large factor.

The communication heavy pipelines, whose stream graphs can be seen in Figure 3-1, follow the same approach as the computation heavy benchmarks. However, these have a relatively high communication bandwidth and just enough computation to ensure one of the filters is the bottleneck and not either the sink or source. For the imbalanced pipeline, the datarates of both filters are actually the same, but their computation costs are not. This leads to the event that one filter is fissed far more often than the other one, which is the desired effect.

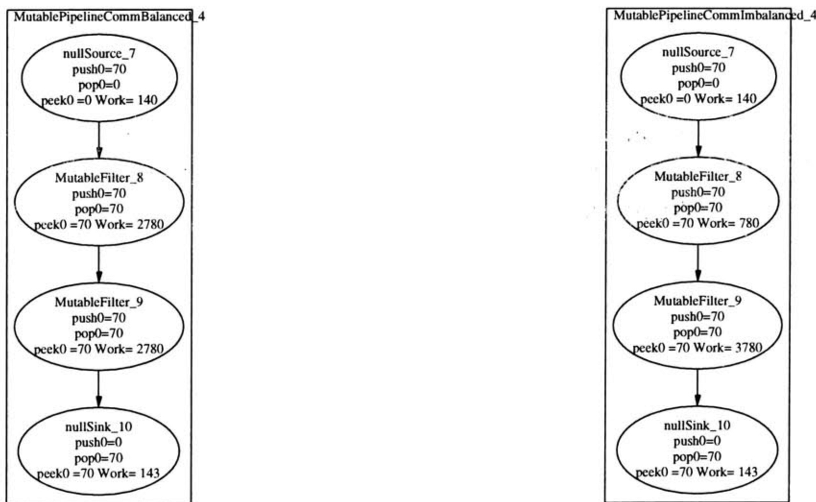
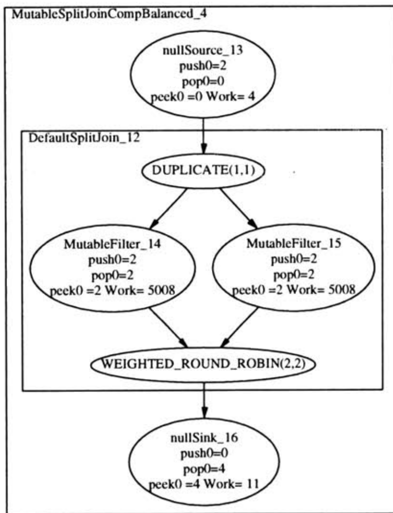



Figure 3-1: Stream Graphs of Communication Heavy Pipelines

## 3.2 Splitjoin Benchmarks

The splitjoin benchmarks are similar to the pipeline benchmarks in terms of individual filters, in that both have a dummy source and sink, and contain two other filters. The primary difference is that in these benchmarks, the two are arranged in parallel instead of in series, in between a duplicate splitter and a roundrobin joiner. Ideally, the communication heavy filters, as can be seen in Figure 3-3, would have a communication datarate much higher than the amount of computation it does. However, in our preliminary tests, we found that this extreme would lead to the bottleneck filter to often be the dummy sink, which is not fissable. This is because the sink is responsible for producing a set number of outputs per steady state, which reasonably incurs a considerable cost. If the filters did not have a computation cost higher than the cost of the sink's outputting, then the bottleneck would quickly converge to the sink. As a result, the amount of computation on even the communication dominant filters had to be sufficiently raised.



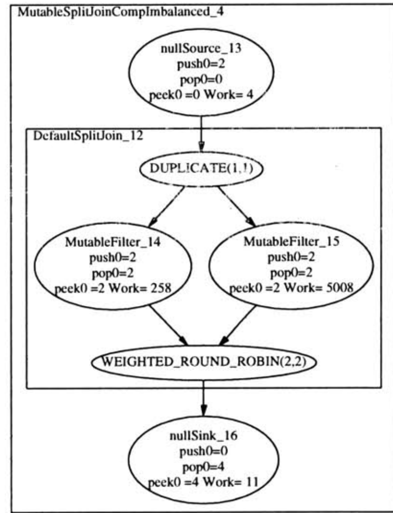




Figure 3-2: Stream Graphs of Computation Heavy Splitjoins

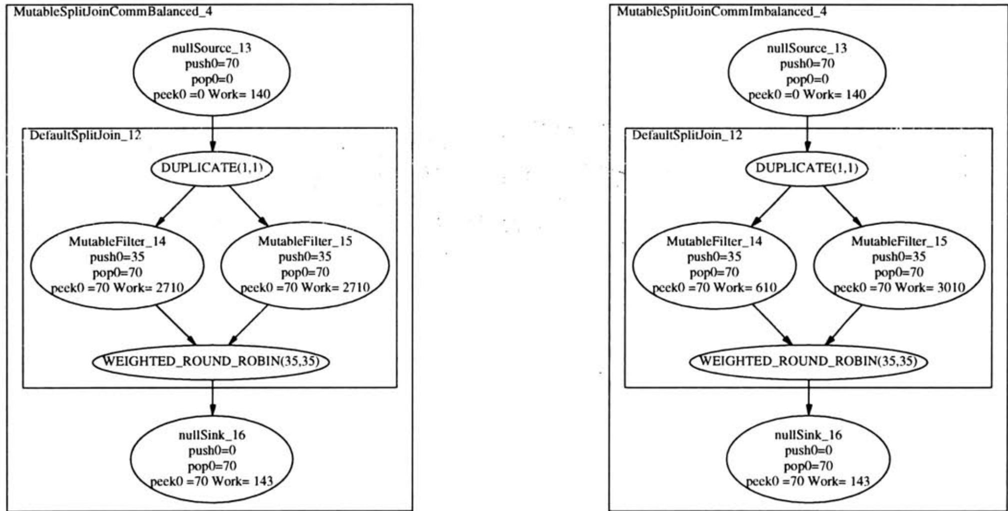



Figure 3-3: Stream Graphs of Communication Heavy Splitjoins

## 3.3 Roundrobin Benchmarks

The final two benchmarks created for the test suite were also splitjoins, but with round robin splitters instead of the duplicate splitters used in Figures 3-2 and 3-3. This was done because round robin splitters tend to incur a greater communication cost than duplicate splitters, as they can only send data to one of its downstream filters at a time. One of the metrics we test in a future chapter uses this assumption.

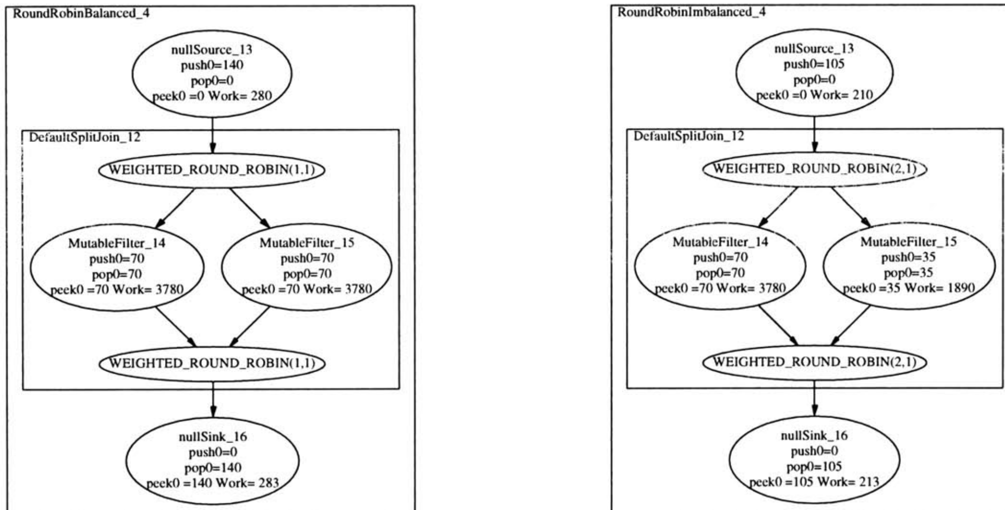



Figure 3-4: Stream Graphs of Round Robin Splitjoins

## 3.4 Testing the Benchmarks

Each of the ten benchmarks were tested in three separate simulations; one with the metric, one with the Magic Network, and one with Raw. Due to limitations in both the Raw and Magic network simulators, we were unable to get consistent datapoints beyond anything in a 4x4 tile configuration. As a result, we were forced to keep

our benchmark streams with minimum complexity, as adding more filters initially would have introduced fusion in the lower tile configurations. However, because our models are essentially numerical equations, we were able to test the metrics to an arbitrarily large number of processors. For the purposes of this thesis, we limited it to 24 processors. This was sufficiently a large amount to display the asymptotic trends for our test benchmarks.

# Chapter 4

# Basic Cost Metric

In our development of models to accurately predict the performance of filter fission across a variety of benchmarks, we followed an iterative process. The first metric tested was one that assumes a communication overhead, solely based on the communication bandwidth of each benchmark. This includes the processing time for any *push, pop,* or *peek*, implying that the data per processor is fed directly into the computation portion of the processor after a fixed amount of cycles. Other effects such as synchronization are not accounted for in this model. While at first glance this metric may seem overly basic to be of use, we found it important to establish an accurate baseline measurement as to the optimal performance of fission, assuming a perfect compiler that would effectively eliminate any costs that would stem from fissing.

## 4.1 The Model

As a baseline measurement for this metric and all following ones, we assume that every filter initially occupies one processor each (for instance, a five filter stream is to occupy a total of five processors as its initial state). This is done in order to remove the potential effects of filter fusion from the calculation. Thus, we focus strictly on data parallelization. Because we make this assumption, the bottleneck work will always stem from a single filter, and not groups of them. For instance, the steady-state cost of a pipeline would be the maximum of its individual component filters, since each

filter is simulated to run in parallel.

The model assumes an iterative process in how new processors are added. It also makes the assumption that all the filters in the stream do not carry with them internal state from iteration to iteration. This is because stateful filters are currently considered to be unfissable without considerable compexity. In the base case, a simple work estimate is taken of all filters in the stream, and this is the initial data that is used. Upon each iteration, the current bottleneck filter in the stream is calculated and fissed. This process is repeated for every new processor that is added. It is important to note that the bottleneck filter is not necessarily the same for each new processor that is added, as parallelizing the current bottleneck filter might make a completely different filter the new bottleneck. Thus, the overall cost of the bottleneck filter scales as follows:

$$p_{N+1} = \frac{p_N * N}{N+1}$$

$$p_0 = \text{Initial filter computation work estimate}$$

where $p_i$ is the total cost of the bottleneck filter when fissed $i$ ways. The decreasing bottleneck cost can be easily explained by the multiplicity of filters that grow with $N$. Within a stream (and hence its original filters), there exists a fixed total cost that needs to be processed in order for outputs to be produced. As filters are fissed, the overall number of filters grow, which reduces the amount of work each individual filter has to do. One can also view this on a filter level, namely that each filter in the original stream is responsible for a certain amount of work, that is then split amongst its fissed filters. Due to data parallelization, each fissed filter only takes upon a fraction of the work of the original, with this fraction decreasing as the number of processors increases.

## 4.2 Results and Analysis

### 4.2.1 Cost Type Comparison

As an example of the difference between communication and computation dominated streams, Figures 4-1 and 4-2 show the distribution of computation vs. communication in the four pipeline benchmarks. In the computation dominant cases of Figure 4-1, the overall cost completely overlaps with the computation cost trendline, which is not surprising given that the communication cost is negligible for all values on the x axis. However, the overall cost trendline is slightly different in Figure 4-2, where the communication cost, albeit still relatively small, does indeed contribute. In particular, it is interesting to note that after a certain number of processors on both graphs in the figure, the communication cost exceeds that of the computation cost. The spikes in communication cost (and respective dips in computation cost) on the bottom graph of Figure 4-2 are due to the imbalanced nature of the benchmark. The graph shows the bottleneck filter a given number of processors, and so at these points, it is the less intensive of the two original filters that is being fissed.

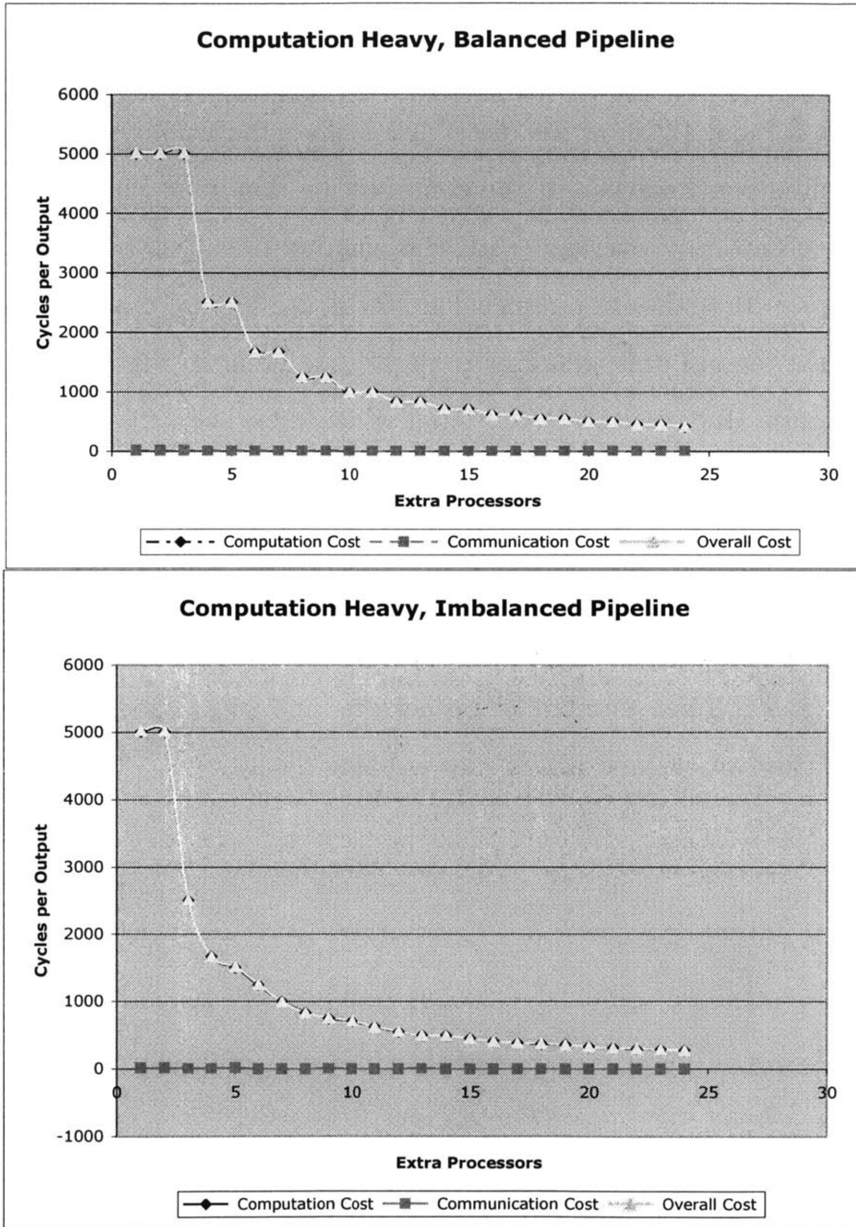



Figure 4-1: Cost Distribution for Computation Dominated Pipelines

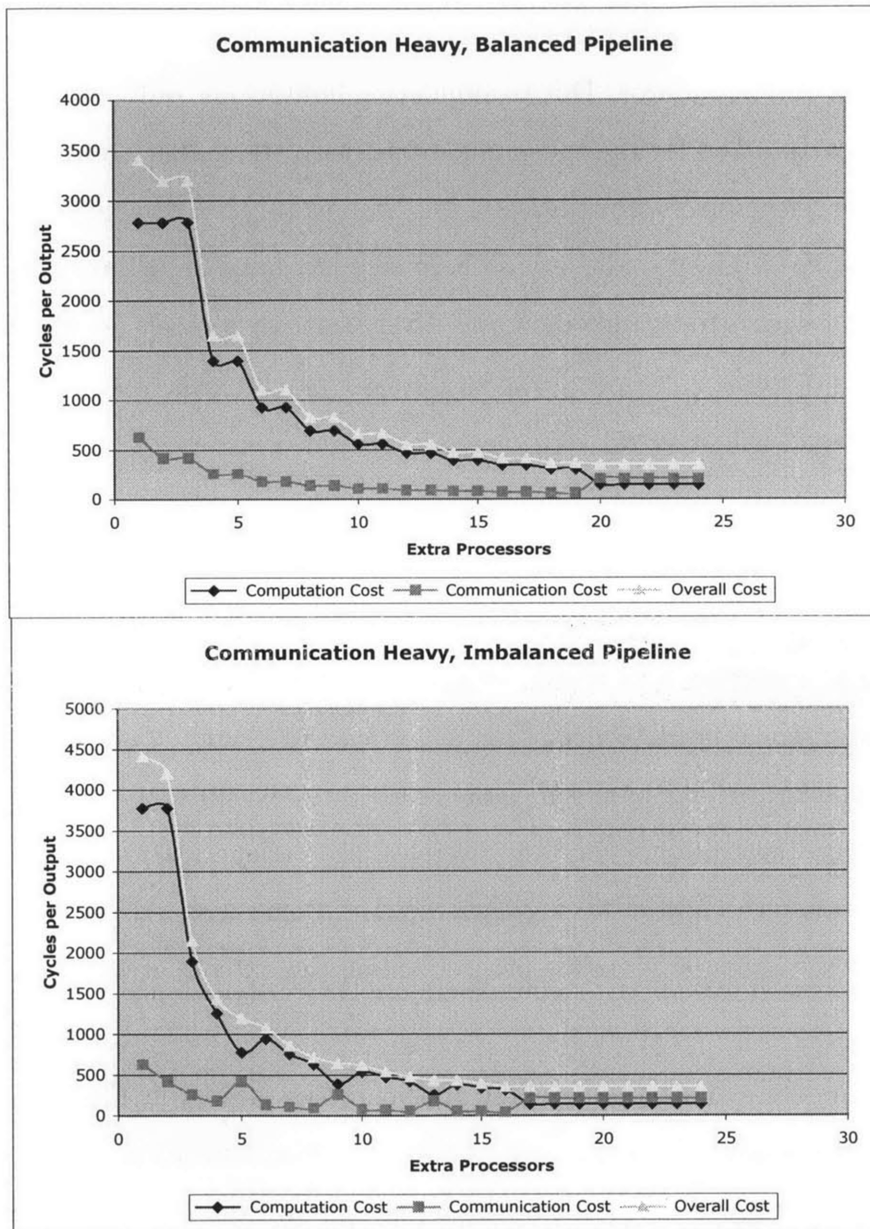



Figure 4-2: Cost Distribution for Communication Dominated Pipelines

### 4.2.2 Metric Comparison

Figures 4-3 and 4-4 show the graphical results of applying the metric to the four pipeline benchmarks, compared to the actual results gathered from both the Magic network and the Raw simulator. Due to simulation limitations, only a limited number of data points were taken for the two simulations, with the metric trendline typically extending to larger numbers of processors. It is important to note that for each of the following comparison graphs, the x axis indicates the number of processors *beyond* the base case for each stream, in which each filter starts by already occupying its own processor. We normalized the results from the base case, which was when each filter in the stream occupied its own tile.

For the computation heavy pipeline benchmarks, the trendline metric matches the Magic Network results fairly well, since communication costs are a negligible in those pipelines by design. In a balanced pipeline or splitjoin, each of the two work filters are split alternately, resulting in a regular stepwise decrease in bottleneck throughput, as exactly two fissings need to occur for a decrease to occur. Conversely, with an imbalanced pipeline or splitjoin, the more work intensive filter is split more often than the non intensive filter, hence the decrease in bottleneck throughput does not follow any set pattern. Both the regular stepwise progression on the top graph of Figure 4-3 and the irregular stepwise progression on the bottom graph are accurately captured.

The Basic metric's shortcomings first appear in the graphs in Figure 4-4, where it severely overestimates the projected speedup for communication heavy pipelines. While the regular stepwise behavior of the top graph and the irregular stepwise progression of the bottom graph are expected, the rate of decrease for the bottleneck cost is far lower than that of either the Magic network or Raw trendlines. This implies that there are other communication costs inherent in data parallelization besides the ones modeled in this metric.

In viewing the Raw trendline for these graphs and all future ones, it is important to note that only datapoints for rectangular Raw configurations are displayed. Because

Raw is limited to data parallelization only across these configurations (e.g. 1x1, 1x3, 2x2, 3x4, etc.), the intermediary data points (e.g. 7, 11) do not give an accurate portrayal of the simulator's performance.

There exists a large disparity between the Raw trendlines and the other two, in particular with respect to the communication heavy pipelines. One potential reason is due to a limited capacity buffer between two filters in a pipeline. When the buffer is full, the preceding filter in the pipeline is blocked, as it is unable to write to the buffer. In Magic network, infinite buffering is assumed, so this is not a factor. A future iteration of this metric might be able include buffer size as a parameter, if we wished to better match the Raw trendline. In addition, we observed increases in bottleneck cost regularly at the 3x3 Raw configuration, which translates to the 5 extra processor datapoint, with this effect most obvious on the top graph of Figure 4-3. This occurrence can most likely be attributed to the peculiarities of Raw's handling this particular layout.

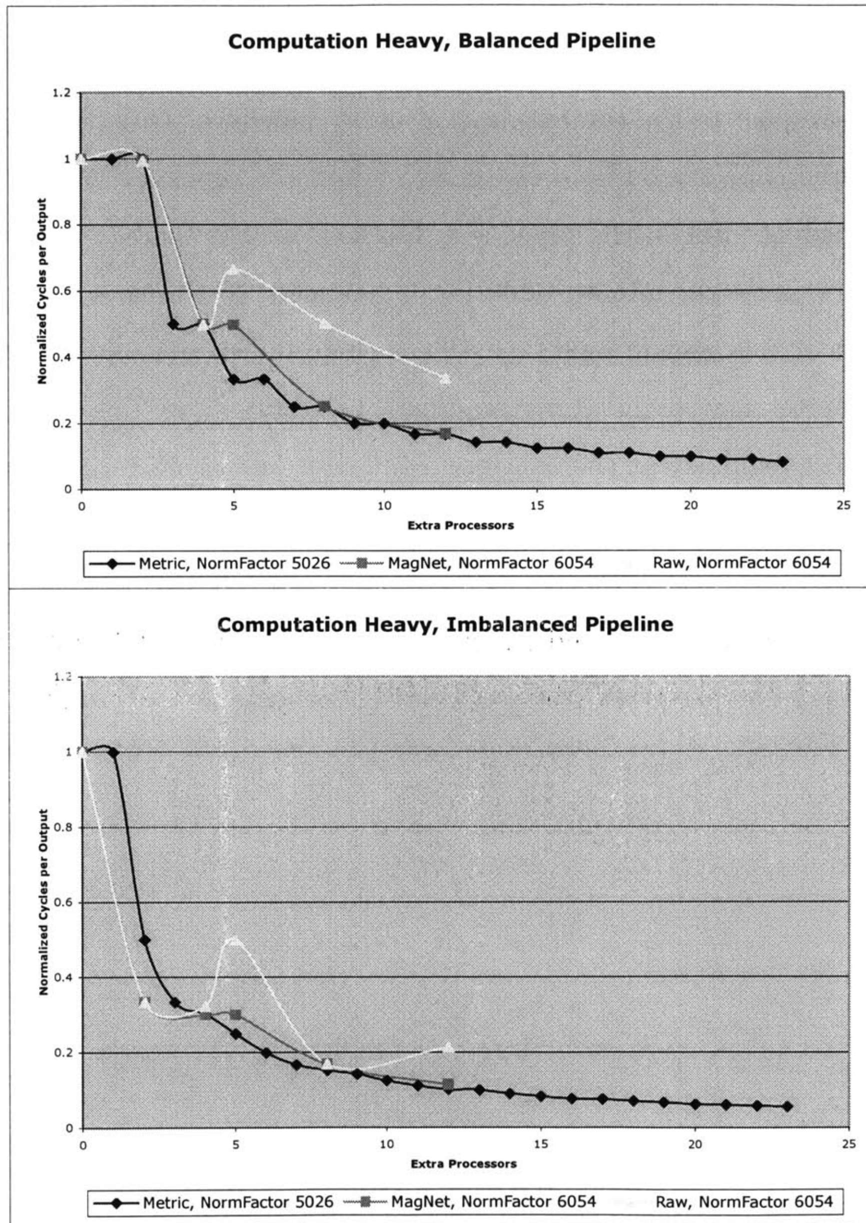



Figure 4-3: Graphical comparison of Basic Metric to Raw simulator for Computation Heavy Pipelines

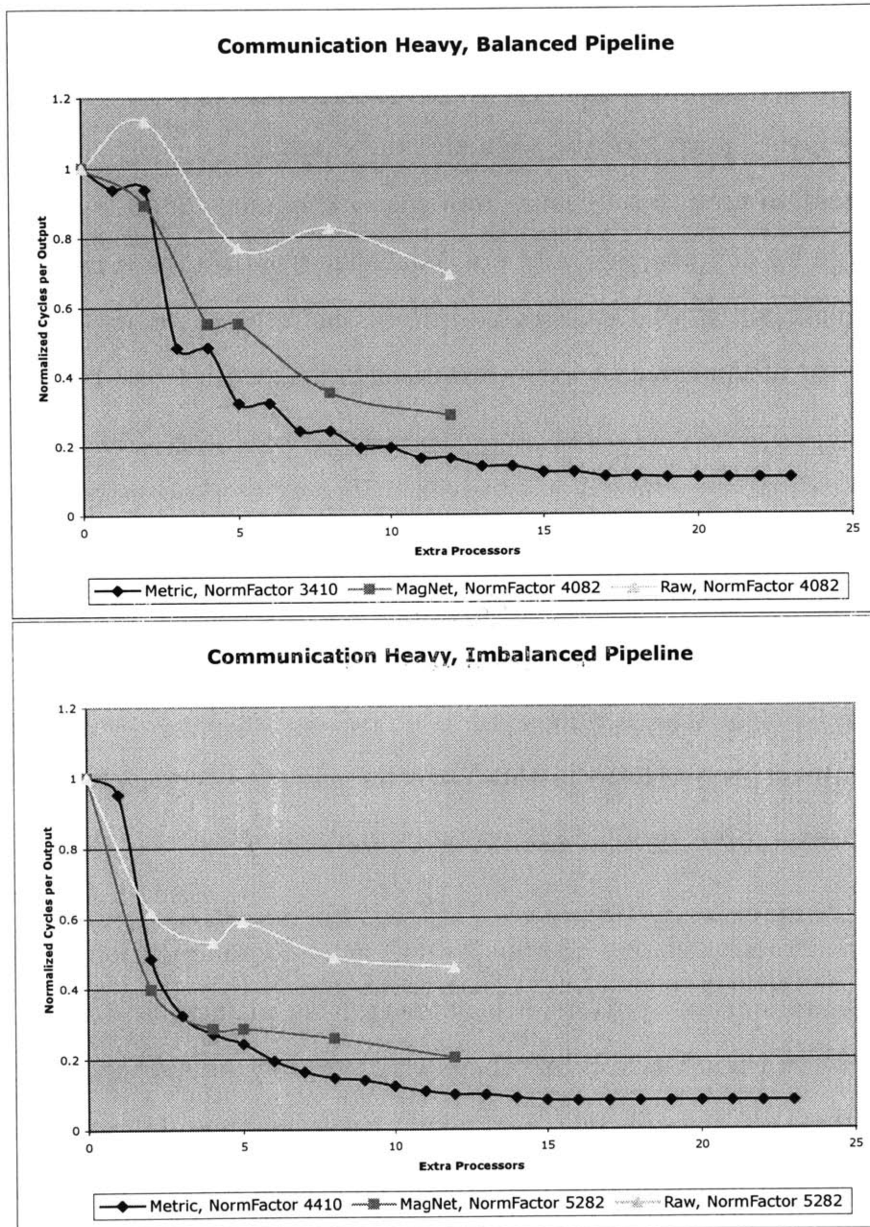



Figure 4-4: Graphical comparison of Basic Metric to Raw simulator for Communication Heavy Pipelines

In viewing the trendlines for the computation heavy splitjoin benchmarks, the basic metric matches both the Raw and Magic network trendlines quite well. In particular, the stepwise behaviors for the bottom graph in Figure 4-5 that are present in Raw and Magic network is accurately reflected in the basic metric trendline. It is possible that it is also the case for the top graph, but it cannot be definitively determined given the spacing of the simulator data points.

It is important to note that because Raw places the joiner module of the splitjoin in a separate tile by default, we could not normalize from the base case of 4 (as we did with the pipelines), as that case would include the fused joiner module. However, we were unable to normalize to 1 extra processor either (5 tiles), as 1x5 results were unavailable for both Raw and Magic network. Because of these limitations, we had to normalize all three trendlines for splitjoins to *two* extra processors, which surely introduced error into our findings.

Similar to the results with the communication dominant pipelines, the metric trendlines in Figure 4-6 vastly overestimates the speedup obtained by fissing. The accuracy of the basic metric to Figure 4-5 is easily explained because of the lack of any real communication overhead in those benchmarks, due to their relatively small push and pop rates. As a result, the constant communication cost in these cases is negligible.

Because round robin splitters inherently have more communication involved with them compared to duplicate splitters, it is not surprising to find that the Basic metric performs poorest on Figure 4-7. While there were very few simulation datapoints for these benchmarks, they are sufficient to highlight the fundamental flaws of the metric.

Another interesting item of note in the splitjoin graphs is that the Raw trendline here matches up with the magic network simulation results much better than with the pipelines. This gives further credence to the theory that there are other unaccounted for costs involved in the placing of two filters in a pipeline as opposed to a splitjoin, that result in vastly different bottleneck costs.

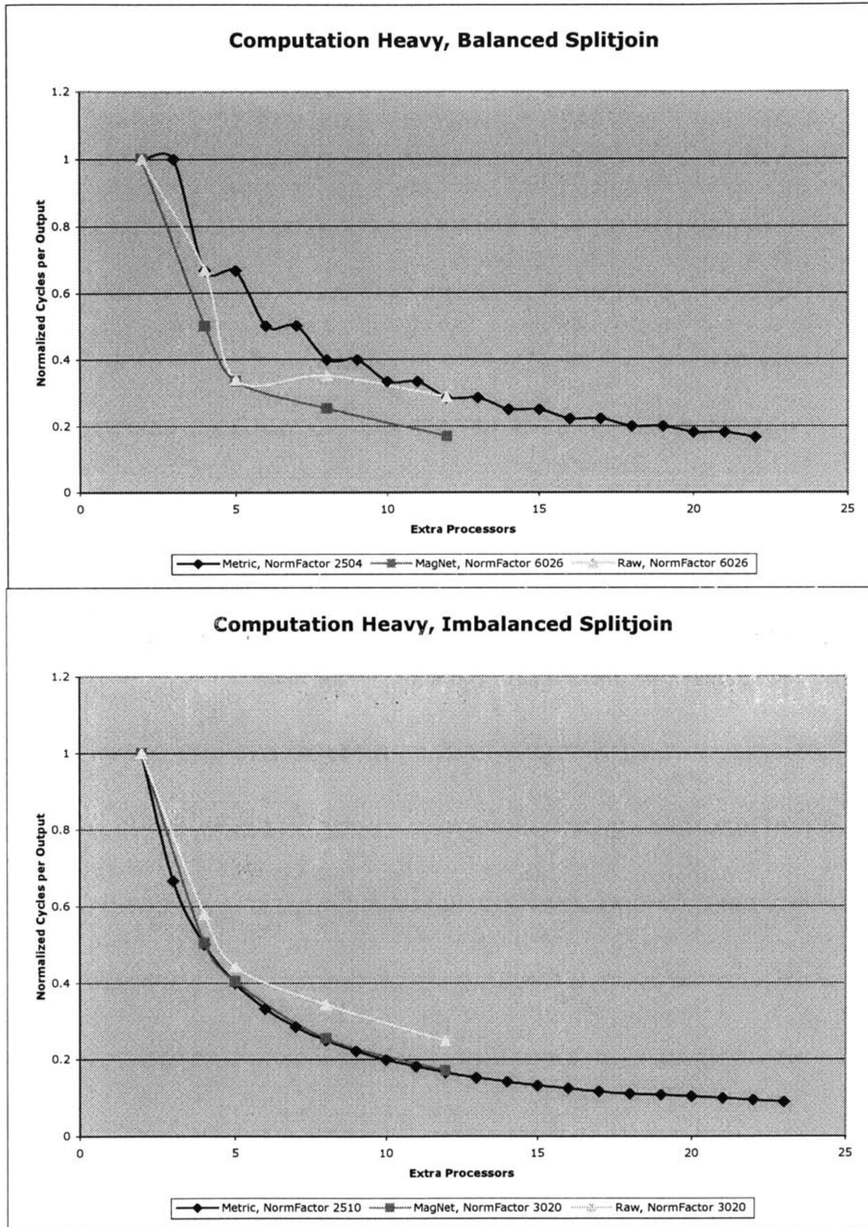



Figure 4-5: Graphical comparison of Basic Metric to Raw simulator for Computation Heavy splitjoins

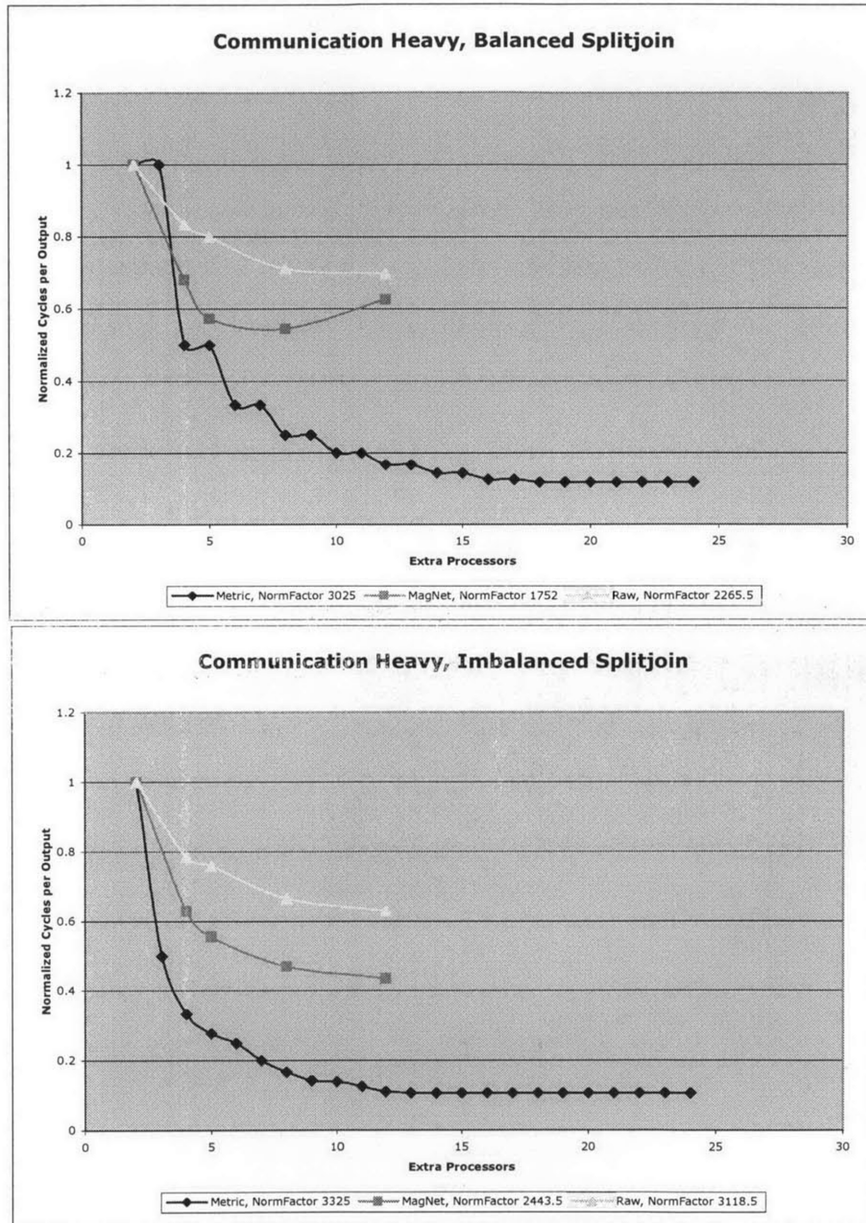



Figure 4-6: Graphical comparison of Basic Metric to Raw simulator for Communication Heavy splitjoins

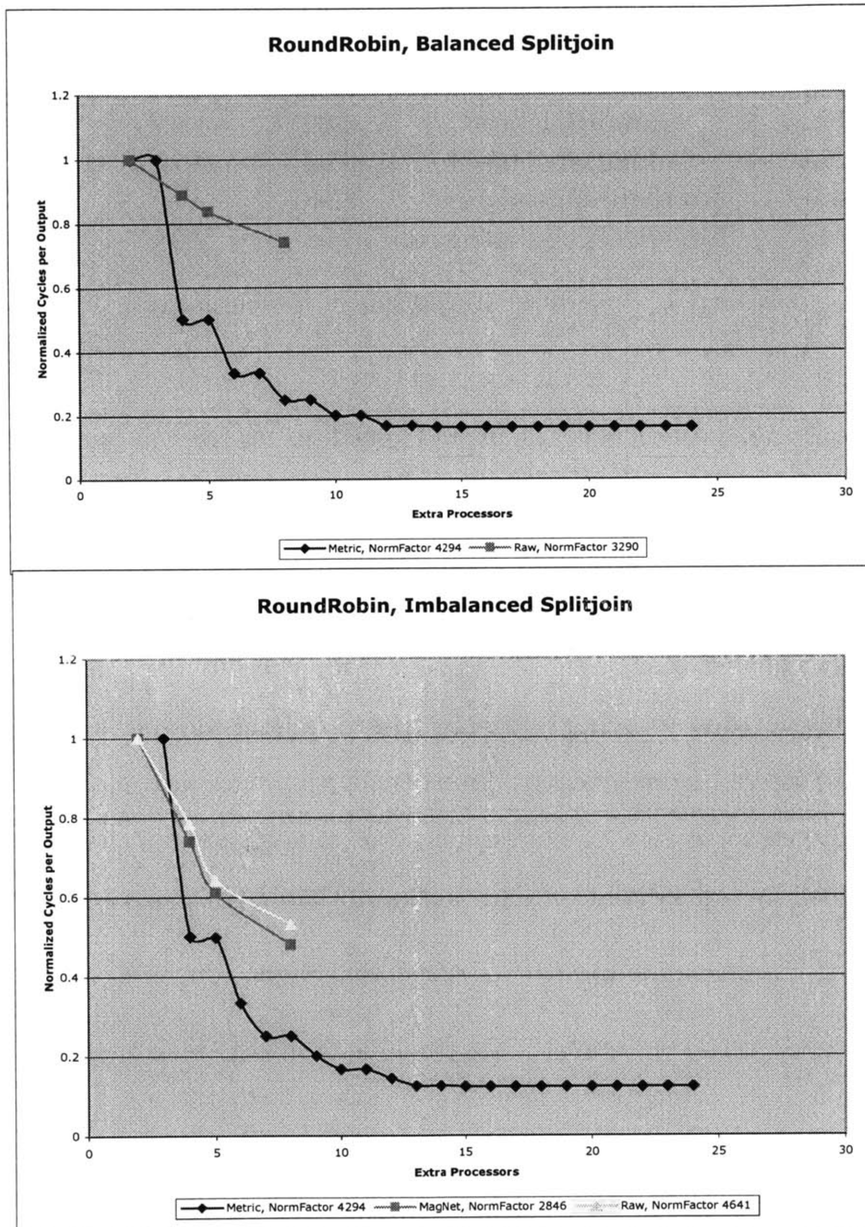



Figure 4-7: Graphical comparison of Basic Metric to Raw simulator for Round Robin splitjoins

| Stream Type | Characteristic | Metric | Magic Net | Raw |
|---|---|---|---|---|
| Pipeline | CompBalanced | 3.51 | 5.93 | 2.96 |
| Pipeline | CompImbalanced | 3.44 | 2.94 | 1.56 |
| Pipeline | CommBalanced | 5.81 | 3.13 | 1.64 |
| Pipeline | CommImbalanced | 4.95 | 1.96 | 1.35 |
| Splitjoin | CompBalanced | 3.51 | 5.92 | 3.47 |
| Splitjoin | CompImbalanced | 6.02 | 5.85 | 4.03 |
| Splitjoin | CommBalanced | 6.02 | 1.59 | 1.43 |
| Splitjoin | CommImbalanced | 9.09 | 2.29 | 1.58 |
| Splitjoin | RoundRobinBalanced | 3.98 | n/a | 1.35 |
| Splitjoin | RoundRobinImbalanced | 4.98 | 2.08 | 1.87 |

Table 4.1: Speedup comparison of Benchmarks

Table 4.1 shows a comparison of the overall speedups for the ten benchmarks after a set amount of extra processors added. The numbers in the table are obtained after exactly twelve fissings total beyond the base case. While numbers for greater fissings were available for some benchmarks, we were limited by the Raw simulator's inability to run a few benchmarks past ten. For instance, for the pipeline benchmarks, one fissing was always devoted to being allocated to the splitter module when one of the base filters were fissed. Consequently, limiting all the comparisons to ten fissings provides the most uniform view of our results. An exception is for the Round Robin benchmarks, which lacked simulation data for more than eight fisses beyond the base case.

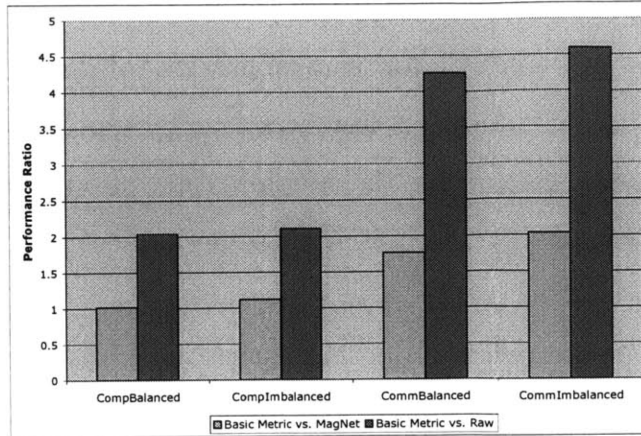



Figure 4-8: Synchronization metric accuracy vs. Magic Network and Raw simulations for pipelines

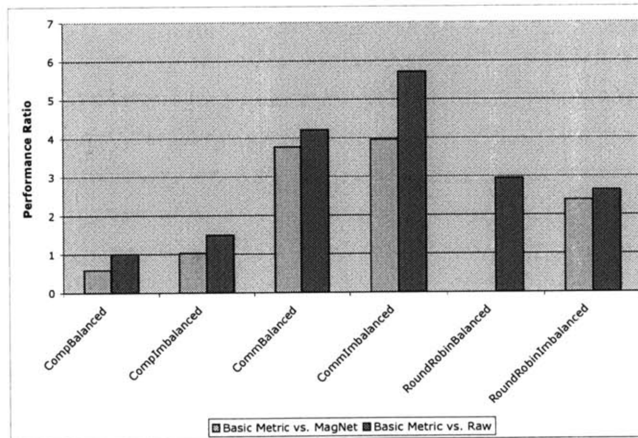



Figure 4-9: Synchronization metric accuracy vs. Magic Network and Raw simulations for splitjoins

The results from figures 4-8 and 4-9 serve to echo the analysis of the graphs previously displayed in the chapter. The basic metric most accurately reflects the predicted speedups for cases with low communication rates , but in general does not do well on the high communication cost benchmarks, especially when compared to Raw. This is not surprising, since the simple nature of our initial model does not capture all of the complexity inherent in building an accurate communication cost model.

Our basic cost metric, while promising, leaves much room to be desired, particularly for the benchmarks containing large amounts of communication overhead. A new metric needs to be introduced, that includes other overhead costs.

# Chapter 5

# Synchronization Cost Metric

The second metric that we developed attempts to implement a communication cost metric that is more accurate than that of the Basic metric, in which a simple constant communication overhead was assumed based on the overall communication rates of the individual filters. We observed that there remained room for improvement on the accurate predictive ability of the basic metric towards communication dominated streams, in particular those containing Splitjoins. To that end, the following metric models a filter's communication cost as it is related to its synchronization cost when the filter is fissed.

## 5.1 The Model

We first define a quantity $k_i$ that represents the cost related to receiving data of a particular filter when it is fissed $i$ ways. This constant is determined as a direct function of a filter's `pop` rate:

$$k_i = b * poprate_i$$

where $b$ is a constant (set to be three for our tests), and $poprate_i$ is simply the fissed filter's respective pop rate. It is interesting to note that in the steady state, this constant is actually a decreasing quantity. This is because in the steady state, the

*pop* rate of any individual fissed filter stays constant even with increasing $i$. Although a filter's *pop* rate increases per-invocation with $i$, the number of outputs of the entire stream graph also increases with $i$, the two effects cancelling each other out.

The crux of the synchronization cost model involves understanding the effects of increasing fissing to a filter's communication overhead. When a filter is first fissed into two filters to reduce the steady state computation cost, that filter in the stream is replaced by a splitter, two parallel fissed filters, and a joiner, in that order. (See Figure 2-6) When these filters are fissed $N$ ways using a roundrobin splitter, each one of them has to wait a total of $k(N-1)$ cycles per invocation to receive its input from the preceding filter (and the following splitter module). This is because the sending filter can only send data to one filter at a time. As $N$ increases, the number of outputs in the entire stream per invocation also increases. Consequently, in the steady state, the relative number of outputs per filter scales as $\frac{1}{N}$. The overall synchronization cost is thus the product of these two quantities, in order to get the units of cycles per output:

$$s_i = k_i * \frac{N-1}{N}$$

The cost $s_i$ is known as the *synchronization cost* that is seen by a given filter, and aims to take communication density into account. This is an additive quantity to the overall cost equation introduced in the previous chapter. The two are added to produce the new overall cost, with $i$ as a parameter:

$$c_i = s_i + p_i$$

where $p_i$ is the overall cost of the bottleneck filter when fissed $i$ ways from the Basic metric. What makes this metric fundamentally different from the Basic metric is that it effectively contributes an additional communication cost $s_i$, to better model the considerable overhead that is present in communication heavy streams.

We emphasize that this model is only valid for fissing that is done using round robin splitters. For duplicate splitters, this delay element does not exist, and is not a

factor in the communication cost of filters. Also, it is important to note that the Raw and Magic Net simulation results used duplicate splitting, not round robin splitting.

The synchronization metric is different from the basic metric in its inclusion of this synchronization cost. It operates on the original graph, and predicts performance for an implementation using either duplicate or roundrobin splitting. The metric is applicable to roundrobin splitjoins as well, and it is this metric's ability to distinguish the effects of round robin splitting that separates it from the basic metric. The results we evaluated the metric against in Raw and the Magic network use duplicate splitting because that is the current implementation of the StreamIt compiler on Raw. However, Appendix C contains graphs using simulated round robin splitting, and they are generally comparable.

## 5.2 Results and Analysis

Figures 5-1 and 5-2 show the graphical results of applying the synchronization cost metric to the four pipeline benchmarks, compared to both the Raw simulator and the Magic network simulator. The trendline for the Basic metric is also included in these graphs.

Not surprisingly, the Sync metric trendline overlaps exactly with that of the Basic metric for both graphs of Figure 5-1. With almost no communication cost existing in these benchmark streams (See Figure 4-1), any metric that has a communication cost proportional to a filter's communication rate would not change these trendlines.

The Sync metric trendlines in Figure 5-2, however, show a significant improvement. The added synchronization cost is responsible for the general upward shift in the metric trendline. This is because the decreasing overall cost as calculated from the basic metric is being partially offset by the synchronization cost, and thus results in an overall slower speedup factor. The overall shape of the trendline is preserved, which is also not surprising considering the synchronization cost is simply added to the cost calculated by the Basic metric. In particular, the sync metric trendline of Figure 5-2 matches that of the Magic network almost exactly for the sample datapoints.

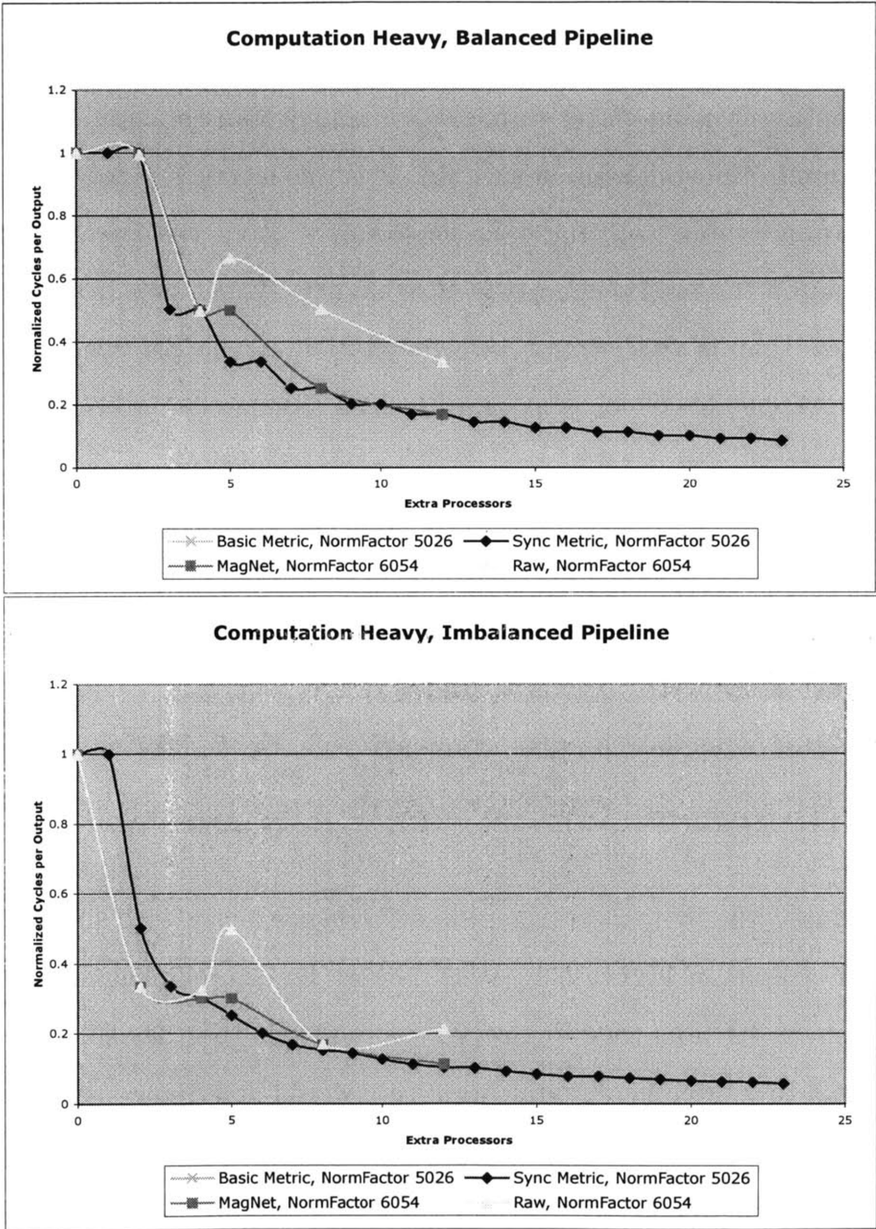



Figure 5-1: Graphical comparison of Synchronization Metric to Raw simulator for Computation Heavy Pipelines

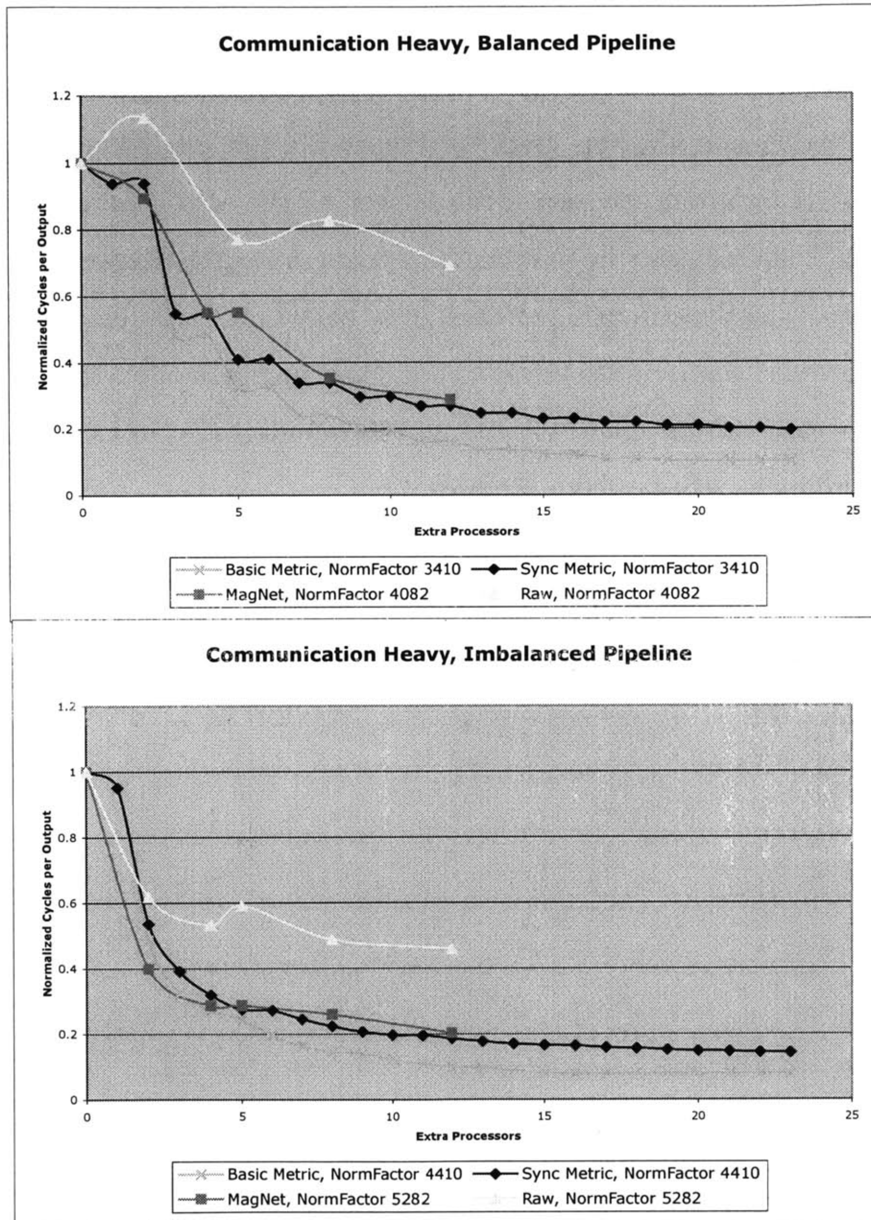



Figure 5-2: Graphical comparison of Synchronization Metric to Raw simulator for Communication Heavy Pipelines

The Sync metric trendlines for splitjoins, as seen in Figures 5-3 , 5-4 and 5-5 show improvement similar to that which was observed in the pipeline benchmarks. In Figure 5-3, the sync metric trendline is again identical to the Basic metric trendline.

The primary shortcoming of the Basic metric was in the communication heavy splitjoins, and in this respect, the Sync metric performs marginally better. While the metric's trendline on the top graph of Figure 5-4 for small numbers of extra processors (1-5) is far from accurate with respect to the Magic network and Raw trendlines, it is slightly better in that its asymptotic speedup is lower than that of the Basic metric. The Sync metric performs even better on the imbalanced splitjoin benchmark, providing an accurate trendline for small numbers of extra processors as well. A similar case can be made to analyze the roundrobin graphs in Figure 5-5, though the trendline is still far from accurate.

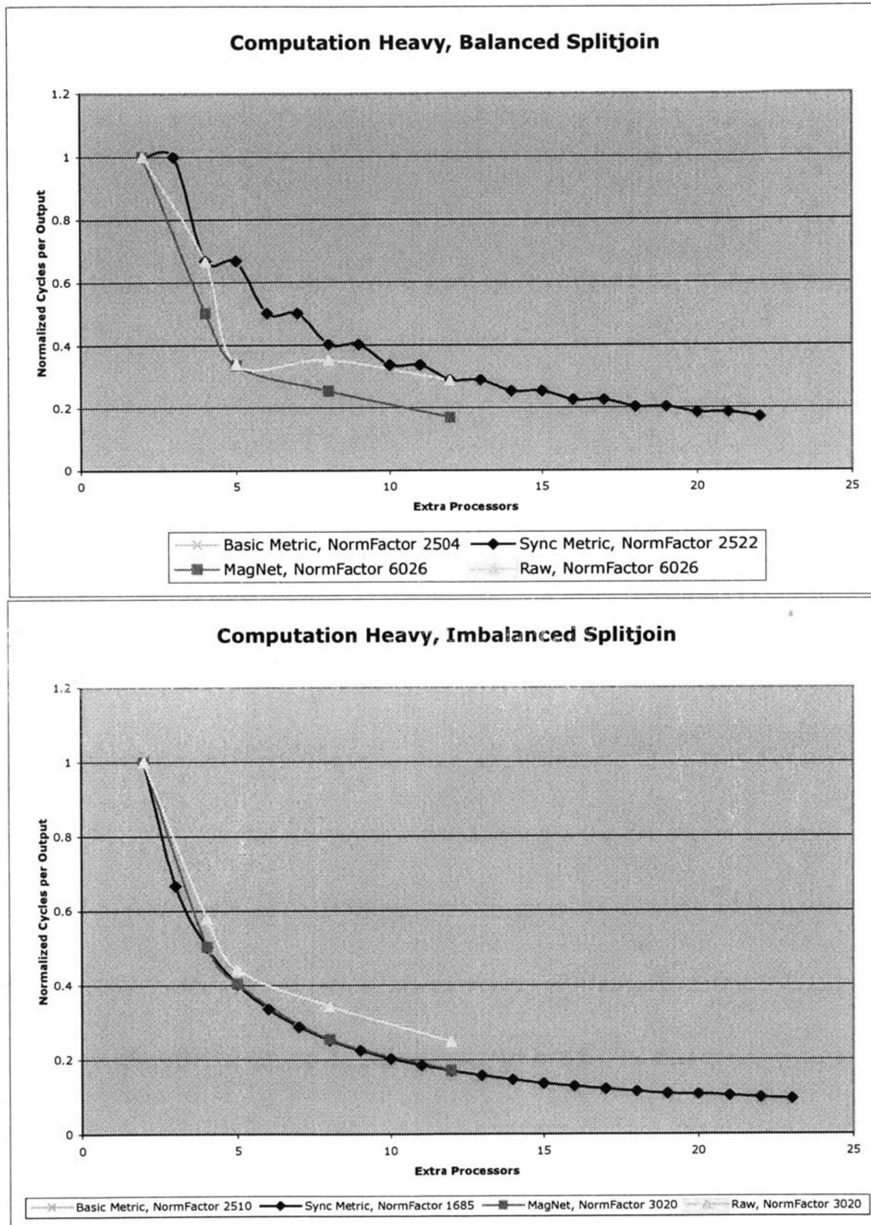



Figure 5-3: Graphical comparison of Synchronization Metric to Raw simulator for Computation Heavy splitjoins

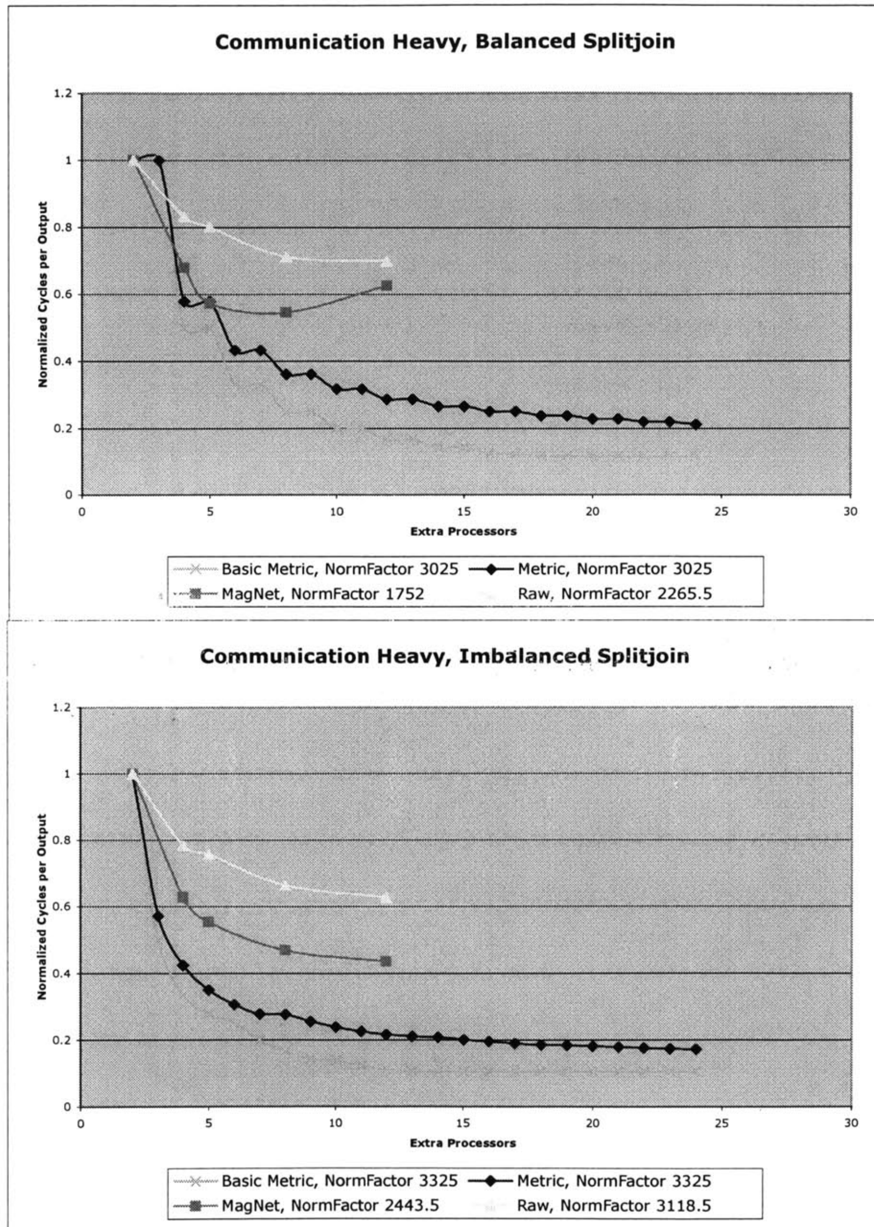



Figure 5-4: Graphical comparison of Synchronization Metric to Raw simulator for Communication Heavy splitjoins

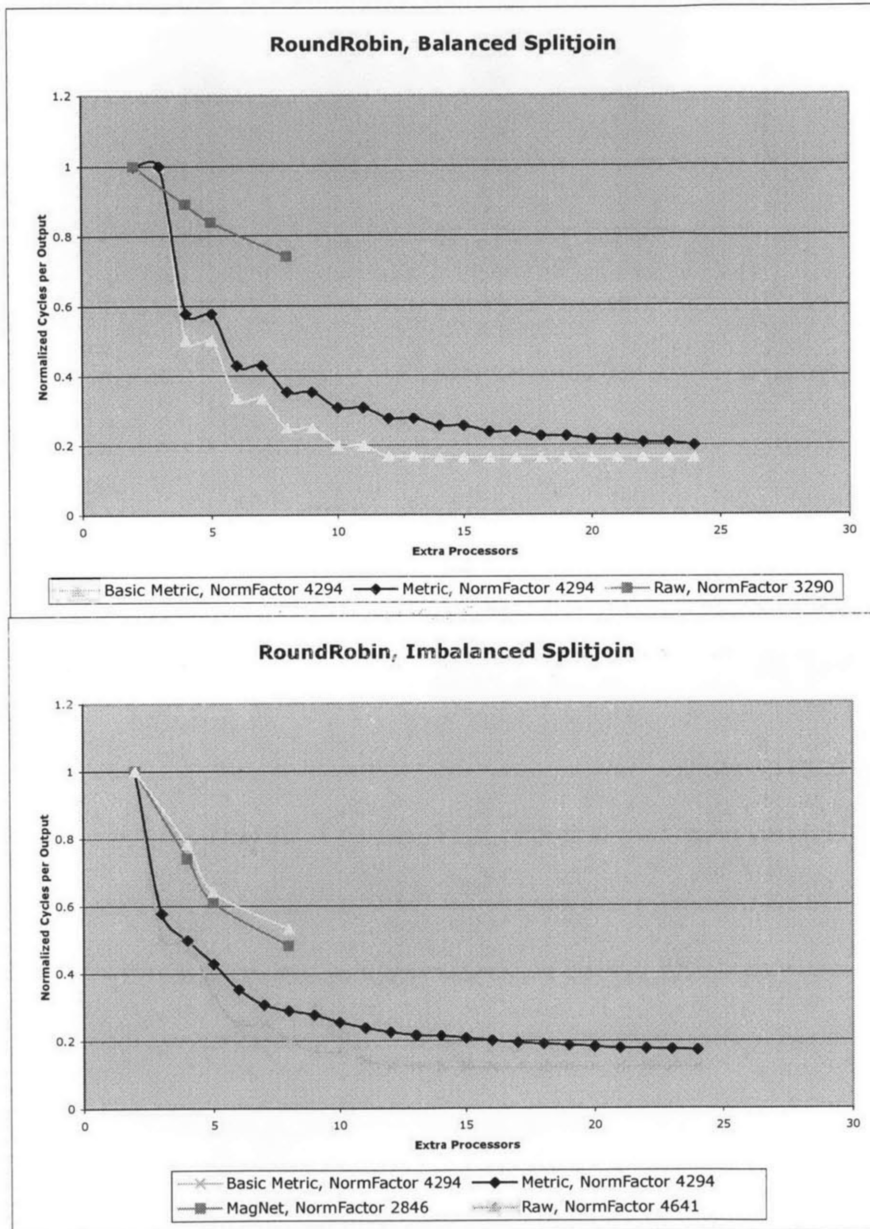



Figure 5-5: Graphical comparison of Synchronization Metric to Raw simulator for Round Robin splitjoins

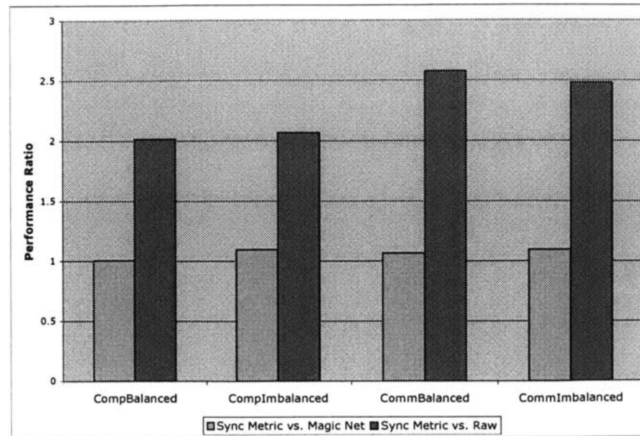



Figure 5-6: Synchronization metric accuracy vs. Magic Network and Raw simulations for pipelines

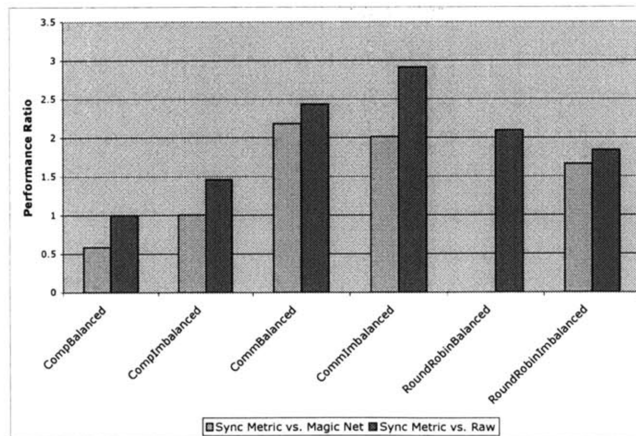



Figure 5-7: Synchronization metric accuracy vs. Magic Network and Raw simulations for splitjoins

As can be seen in figure 5-6, the Sync metric is more accurate than the Basic metric in nearly pipeline benchmark, sometimes by a factor of two or three. The percentage accuracy with respect to the Magic network in the computation heavy benchmarks is

preserved, but there is a drastic difference in the communication heavy benchmarks. Where the Basic metric was off by about 100%, the Sync metric now matches Magic Network almost exactly, with little room for improvement. There remains a high percentage difference from the Raw simulator, due to overhead issues described in the first chapter.

Figure 5-7 highlights the improved accuracy of the Synchronization metric with respect to splitjoins. The percentage correlation for the communication dominated splitjoins remain low, but are an improvement over the Basic metric. It is also noteworthy that the metric matches the Round Robin balanced benchmark on Magic Net almost exactly, a vast improvement from the Basic metric for this particular stream.

The improved overall performance of the Synchronization metric indicates that it is be a better overall metric for predicting real performance, in particular for pipelines, in which it predicts Magic network performance almost exactly. It still does not seem to go far enough for splitjoins, but the metric has shown itself to be a fairly accurate predictor.

# Chapter 6

# Conclusion

This thesis aims to introduce two models by which stream programs can be measured for their overall scalability. It does this in two phases: First, the introduction of a metric, and a brief derivation as to its numerical formulation. The second phase is analyzing the performance of the metric on a test suite, comparing it to two other networks to demonstrate its accuracy.

## 6.1 Results Summary

### 6.1.1 Metric Validation

In this thesis, we analyze two models of scalability, the Basic metric and the Sync metric. Figure 6-1 shows the accuracy with which the two metrics predict pipeline speedup, with respect to the Magic network. It is not surprising to note that for the computation heavy cases, both metrics match the Magic network almost exactly, since communication cost is negligible in these cases. Since communication cost is known to be responsible for the majority of the cost overhead in fissing, the speedup due to computation cost is relatively simple to calculate.

However, the advantage of including synchronization overhead is visible in Figure 6-1. Here, the sync metric is far more accurate in predicting the projected speedup using the Magic network, giving credence to the presence of considerable synchroniza-

tion overhead in communication dominant stream programs.

Figure 6-2 exhibits a similar trend, though both metrics are generally less accurate with respect to Raw. This is due to a number of factors, among them the presence of finite size buffers in the communication channel for each tile.

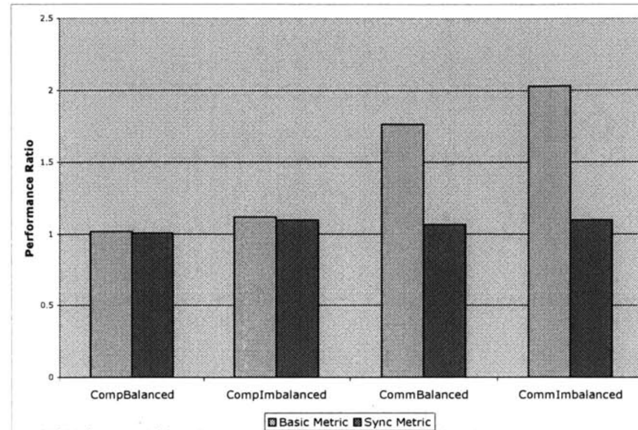



Figure 6-1: Metric Comparison of Pipelines with Respect to Magic Network

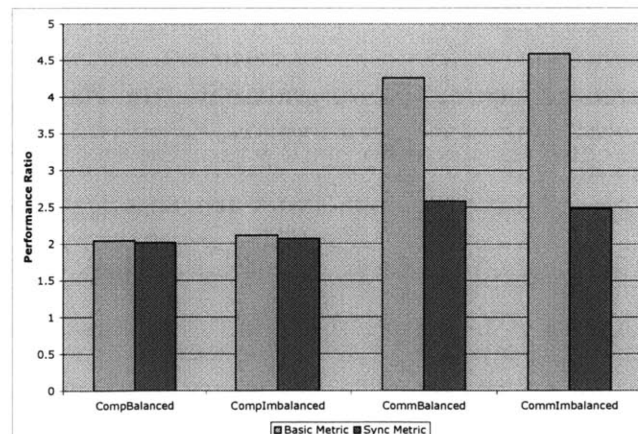



Figure 6-2: Metric Comparison of Pipelines with Respect to Raw

Figures 6-3 and 6-4 show similar accuracy results to those of the pipeline results, though both metrics are generally less accurate at predicting performance for these structures. We were unable to obtain Magic network comparison for the balanced round robin benchmark due to simulator errors. Part of the potential error can be attributed to issues related to normalization, as we were unable to normalize the speedups to the base case for splitjoins. This was because the number of tiles in the base case (five) for splitjoins could not be generated with the current state of the Raw and Magic network simulators. The general trend of the sync metric outperforming the basic metric is still clearly visible in Figures 6-3 and 6-4, indicating that the synchronization overhead is present regardless of the stream layout.

In general, our metrics are less predictively accurate for streams with imbalanced filters versus streams with balanced ones. This indicates that there is perhaps either something inherent in the stream graph itself, or within the compiler that introduces additional overhead when fissing one filter much more than the other. As we were unable to determine the exact cause, exploring this would be a potential improvement for future work.

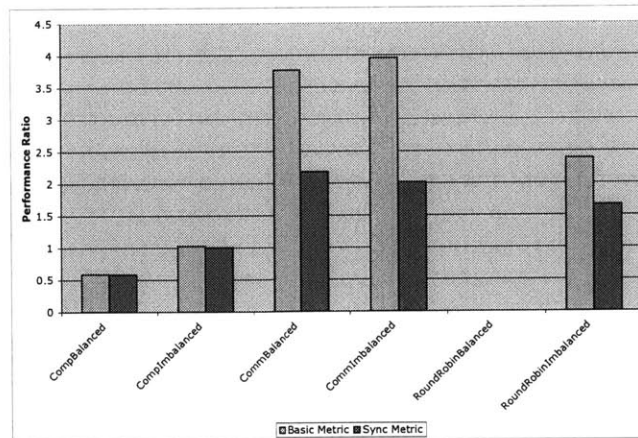



Figure 6-3: Metric Comparison of Splitjoins with Respect to Magic Network

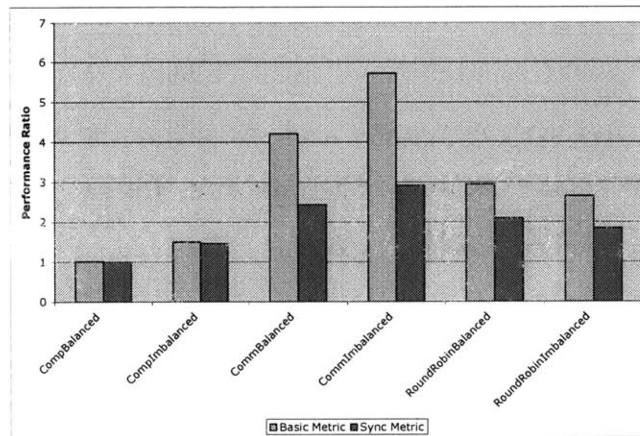



Figure 6-4: Metric Comparison of Splitjoins with Respect to Raw

### 6.1.2 Scalability Analysis

Since we have shown that the sync metric is accurate to a high degree in predicting the scalability of stream programs for small scale processors, we now extrapolate the potential speedups of our benchmarks for larger amounts of processors. Table 6.1 shows the results when the sync metric simulates filter fissing to 24 extra processors.

| Stream Type | Characteristic | Speedup |
|---|---|---|
| Pipeline | CompBalanced | |
| Pipeline | CompImbalanced | 11.70 |
| Pipeline | CommBalanced | 17.39 |
| Pipeline | CommImbalanced | 5.11 |
| Splitjoin | CompBalanced | 7.01 |
| Splitjoin | CompImbalanced | 5.87 |
| Splitjoin | CommBalanced | 4.70 |
| Splitjoin | CommImbalanced | 5.83 |
| Splitjoin | RoundRobinBalanced | 4.98 |
| Splitjoin | RoundRobinImbalanced | 5.79 |

Table 6.1: Projected Benchmark Speedup with Synchronization Metric

As can be seen in Table 6.1, while all ten benchmarks benefit considerably from scaling to a large number of processors, we can draw key distinctions from the disparities in speedups. It is not surprising to see that the computation dominated benchmarks in general experience a much larger speedup than that of the communication ones. As described in the Basic metric, the computation cost of a filter decrease geometrically as the number of processors fissing it increases. We also showed in our derivation of the Synchronization metric that the communication cost does not experience this decrease; rather, the cost actually increases for sufficiently large numbers of processors due to the synchronization overhead, if roundrobin splitters are used for the fission process.

It is also interesting to note that splitjoins in general experience a lesser speedup than pipelines, buy a relatively large margin. While our metric did not specifically draw a distinction between filters being placed in a pipeline versus being placed in a splitjoin, it is quite conceivable that added complexity in a splitjoin structure contributes to an increased cost overhead for all filters. For instance, the metrics

do not take into account the extra splitter and joiner modules present in splitjoins versus pipelines. Another source of error was that due to simulator limitations, we were unable to normalize splitjoin speedups to the base case of one filter per tile. Instead, we had to normalize and compare based on a base case of $n+1$ tiles to $n$ computation units. In the case of splitjoins, these computation units consist of the filters and the joiner module. However, despite these sources of error, the metric still predicts on average a five-fold speedup on all splitjoin streams.

## 6.2 Current State

This thesis is novel in that it proposes metrics for modeling the scalability of stream programs in specific, and compares the metrics to existing real-life networks. Other studies have explored the field of data parallelization in general, but none have applied directly to the streaming application domain, nor tested their findings on stream programs. Because of this, there was very little material for the thesis to build on; consequently, the state of these models are effectively still in their infancy.

The Sync metric in particular showed considerable promise in accurately modeling the bottleneck cost of pipeline streams, though it only takes into account a single possible source of overhead, the synchronization cost. The datapoints used to test it from Raw and Magic network were also somewhat lacking, due to the fact that the StreamIt compiler and Raw tools are still in a developmental phase. As a result, more tests are sorely needed in order to validate the metric.

There are a plethora of other potential costs that would strengthen this metric even more, in particular costs that are more specific to splitjoin constructs. However, we have demonstrated that this model is highly applicable to stream programs that are acyclic and stateless.

## 6.3 Future Work

As the streaming application domain has yet to be explored to a large degree, there are numerous things that can be done to further the contributions introduced in this thesis.

From a testing perspective, a benchmark suite with more complex stream programs would be more helpful in validating the metrics espoused in this thesis. We were unable to do this due to shortcomings in our current compiler and simulation infrastructure, but it is highly conceivable that these will be remedied in the future, and further testing can be done.

The system that the metrics were compared against, the Raw machine, is a grid-based VLSI architecture. The models should ideally be architecture-independent, as they were conceived and derived without regard to any particular architecture. It would be extremely helpful to run tests on other systems, such as a cluster-based architecture.

A limitation of the metrics introduced in this thesis was that they made certain assumptions about compiler behavior on how data parallelization was implemented at a low level. We did this due to our knowledge of the StreamIt to Raw compiler, and have demonstrated with some accuracy that we can indeed model acyclic stateless programs in this manner. However, there exist many other compiler implementations that have vastly differing characteristics, from their cost models to their partitioning and scheduling algorithms. It would be useful to both test our metrics on other compilers, and extend the metrics we introduced in order to be applicable to these, as well.

Lastly, extending the metrics to incorporate other characteristics of the streaming application domain would vastly increase their applicability to real world programs. For instance, adding support for cyclic programs with feedback loops would greatly increase its usefulness, as most complex DSP algorithms have these structures.

# Appendix A

# Metric Implementation File

## A.1 ParallelizationGathering.java

```
package at.dms.kjc.sir.stats;

import at.dms.kjc.*;
import at.dms.kjc.flatgraph.*;
import at.dms.kjc.iterator.*;
import at.dms.kjc.sir.*;
import at.dms.kjc.sir.lowering.partition.*;
import at.dms.kjc.sir.lowering.fission.*;
import at.dms.kjc.sir.lowering.fusion.*;
import at.dms.kjc.raw.*;

import java.util.*;
import java.io.*;

/**
 * This class gathers statistics about the parallelization potential
 * of stream programs.  It overrides the VisitFilter, PostVisitPipeline,
 * and PostVisitSplitJoin methods (extends EmptyStreamVisitor)
 * 11/12/03, v1.00 jnwong
 * v.1.10:
 *  Added communication cost parallelization, refined nocost
 *  option such that it only halves filters along the critical path.
*/

public class ParallelizationGathering {
```

```
    public ParallelizationGathering() {

    }

    //procedure that's called by Flattener.java
    public static void doit(SIRStream str, int processors) {
new ParallelWork().doit(str, processors);
    }

    static class MutableInt {
static int mutint;
    }


    static class ParallelWork extends SLIREmptyVisitor{

//Toggles for type of synchronization I want to do
        static boolean commcost;
static boolean syncremoval;
/** Mapping from streams to work estimates. **/
static HashMap filterWorkEstimates;
static HashMap filterCommunicationCosts;
static HashMap filterTimesFizzed;
static Vector statelessFilters;
static MutableInt numFilters;
static SIRFilter lastFizzedFilter;
static int[] computationCosts;
static int[] communicationCosts;
static Vector bottleneckFilters;
//additions for syncremoval support:
static HashMap fizzedFiltersToOriginal;
static SIRStream originalStream;
static HashMap origToNewGraph;
static HashMap newToOrigGraph;
static boolean printgraphs;
static boolean synccost;

public ParallelWork() {
    filterWorkEstimates = new HashMap();
    filterTimesFizzed = new HashMap();
    filterCommunicationCosts = new HashMap();
    statelessFilters = new Vector();
    numFilters = new MutableInt();
    lastFizzedFilter = null;
```

```
bottleneckFilters = new Vector();
    fizzedFiltersToOriginal = new HashMap();
    originalStream = null;
    origToNewGraph = null;
    newToOrigGraph = null;
    printgraphs = false;
    commcost = true;
    synccost = true;
    syncremoval = false;
}


public static void doit(SIRStream str, int numprocessors)
{
    originalStream = str;
    computationCosts = new int[numprocessors];
    communicationCosts = new int[numprocessors];

    if(syncremoval)
    {
//StatelessDuplicate.USE_ROUNDROBIN_SPLITTER = true;
    }
    else
    {
//StatelessDuplicate.USE_ROUNDROBIN_SPLITTER = false;
    }
    if(commcost) {
SIRStream copiedStr = (SIRStream) ObjectDeepCloner.deepCopy(str);
GraphFlattener initFlatGraph = new GraphFlattener(copiedStr);
HashMap[] initExecutionCounts = RawBackend.returnExecutionCounts(copiedStr, initFlatGraph);

origToNewGraph = new HashMap();
newToOrigGraph = new HashMap();
WorkEstimate initWork = WorkEstimate.getWorkEstimate(copiedStr);

createFilterMappings(origToNewGraph, newToOrigGraph, str, copiedStr);  //sync thing!
IterFactory.createFactory().createIter(str).accept(new BasicVisitor(filterTimesFizzed));
IterFactory.createFactory().createIter(copiedStr).accept
    (new ParallelVisitor(initFlatGraph, initWork, newToOrigGraph, fizzedFiltersToOriginal,
 filterWorkEstimates, filterCommunicationCosts, filterTimesFizzed,
 initExecutionCounts, numFilters));

SIRFilter initHighestFilter = getHighestWorkFilterSync(1);
for(int i = 2; i < numprocessors + 1; i++)
{
```

```
    //sync implementation
    filterWorkEstimates = new HashMap();
    filterCommunicationCosts = new HashMap();
    statelessFilters = new Vector();
    GraphFlattener flatGraph = new GraphFlattener(copiedStr);
    if(syncremoval){
Lifter.lift(copiedStr);
    }
    //do sync removal if syncremoval is true
    //get number of prints, so I can divide everything by it to get
    //executions per steady state
    String outputDot = new String("at-fizz-" + i + ".dot");
    StreamItDot.printGraph(copiedStr, outputDot);
    HashMap[] filterExecutionCosts = RawBackend.returnExecutionCounts(copiedStr, flatGraph);
    WorkEstimate work = WorkEstimate.getWorkEstimate(copiedStr);
    IterFactory.createFactory().createIter(copiedStr).accept
(new ParallelSyncVisitor(flatGraph, work, filterExecutionCosts,
 filterWorkEstimates, filterCommunicationCosts,
 fizzedFiltersToOriginal, filterTimesFizzed,
 newToOrigGraph, synccost));
    copiedStr = (SIRStream) ObjectDeepCloner.deepCopy(originalStream);
    parallelizesync(copiedStr, i);
}
generateOutputFiles();

    }//if(commcost)
    else
    {
WorkEstimate initWork = WorkEstimate.getWorkEstimate(str);
GraphFlattener initFlatGraph = new GraphFlattener(str);
HashMap[] initExecutionCounts = RawBackend.returnExecutionCounts(str, initFlatGraph);

IterFactory.createFactory().createIter(str).accept
    (new NoSyncVisitor(initWork, filterWorkEstimates,
       filterCommunicationCosts, filterTimesFizzed,
       initExecutionCounts, initFlatGraph, numFilters));
report(1);
for(int i = 2; i < numprocessors + 1; i++)
{
    parallelize();
    report(i);
}
generateOutputFiles();
    }//endif(commcost)
```

```
}


/**   parallelizesync(copiedStr):
 * Similar to parallelize, but it does not use greedy approach, calculates everything straight
 * from originalStream.  Also increments part of HashMap filterTimesFizzed, depending on what it fizzed.
 * returns the new parallelized stream in copiedStr.
 */

public static void parallelizesync(SIRStream copiedStr, int numprocessors)
{
    SIRFilter nextFizzedFilter = getHighestWorkFilterSync(numprocessors);  //returns real original filter, reports
    int oldfizzfactor = ((Integer)filterTimesFizzed.get(nextFizzedFilter)).intValue();
    filterTimesFizzed.put(nextFizzedFilter, new Integer(oldfizzfactor + 1));
    Iterator filtersToFizz = filterTimesFizzed.keySet().iterator();
    //do the mappings!
    origToNewGraph.clear();
    newToOrigGraph.clear();
    createFilterMappings(origToNewGraph, newToOrigGraph, originalStream, copiedStr);

    while(filtersToFizz.hasNext())
    {
SIRFilter currentFilter = (SIRFilter)filtersToFizz.next();
int timestofizz = ((Integer)filterTimesFizzed.get(currentFilter)).intValue();
if(timestofizz > 1)
{
    SIRFilter fizzingFilter = (SIRFilter)origToNewGraph.get(currentFilter);
    if(StatelessDuplicate.isFissable(fizzingFilter))
    {
SIRSplitJoin newFilters = StatelessDuplicate.doit(fizzingFilter, timestofizz);
//populate fizzedFiltersToOriginal HashMap
for(int i = 0; i < newFilters.getParallelStreams().size(); i++)
{
    SIRFilter splitFilter = (SIRFilter) newFilters.get(i);
    fizzedFiltersToOriginal.put(splitFilter, currentFilter);
}
    }
}

    }

}//void parallelizesync


/**   getHighestWorkFilterSync:
```

```
 *  Returns the *ORIGINAL* filter in the input stream with the highest overall cost (comp plus comm.),
 *  also populates computationCosts, communicationCosts[] for reporting
 */

public static SIRFilter getHighestWorkFilterSync(int numprocessors)
{
    SIRFilter maxFilter = null;
    int maxwork = Integer.MIN_VALUE;
    Iterator sirFilters = filterWorkEstimates.keySet().iterator();
    int maxcomm = 0;
    int maxcomp = 0;

    while(sirFilters.hasNext())
    {
SIRFilter currentFilter = (SIRFilter)sirFilters.next();
int currentcompcost = ((Integer)filterWorkEstimates.get(currentFilter)).intValue();
int currentcomcost = ((Integer)filterCommunicationCosts.get(currentFilter)).intValue();
int currentwork = currentcompcost + currentcomcost;
if(currentwork >= maxwork)
{
    maxFilter = currentFilter;
    maxwork = currentwork;
    maxcomm = currentcomcost;
    maxcomp = currentcompcost;
}
    }
    computationCosts[numprocessors - 1] = maxcomp;
    communicationCosts[numprocessors - 1] = maxcomm;
    bottleneckFilters.add(maxFilter.toString());

    SIRFilter realMaxFilter = null;
    if(fizzedFiltersToOriginal.containsKey(maxFilter))
realMaxFilter = (SIRFilter)fizzedFiltersToOriginal.get(maxFilter);
    else
realMaxFilter = (SIRFilter)newToOrigGraph.get(maxFilter);
    return realMaxFilter;

}//SIRFilter getHighestWorkFilterSync


/**   createFilterMappings:
 *  produces a mapping of filters from the original stream to the new stream and vice versa,
 *  and stores these in two HashMaps.
 */
```

```
public static void createFilterMappings(HashMap origToNew, HashMap newToOrig, SIRStream origStream, SIRStream newStream)
{
    if(origStream instanceof SIRFilter)
    {
newToOrig.put(newStream, origStream);
origToNew.put(origStream, newStream);
    }
    if(origStream instanceof SIRContainer)
    {
SIRContainer origContainer = (SIRContainer)origStream;
for(int i=0; i < origContainer.size(); i++)
{
    createFilterMappings(origToNew, newToOrig, origContainer.get(i), ((SIRContainer)newStream).get(i));
}
    }

}//createFilterMappings

/**    generateOutputFiles():
 * prints to various output files the results of the parallel modelling experiments.
 *
 */

public static void generateOutputFiles()
{
    try
    {
File compSyncFile = new File("compsync.txt");
File commSyncFile = new File("commsync.txt");
File totalSyncFile = new File("totalsync.txt");
File filterFile = new File("bfilters.txt");
File costsFile = new File("costs.txt");

FileWriter compSync = new FileWriter(compSyncFile);
FileWriter commSync = new FileWriter(commSyncFile);
FileWriter totalSync = new FileWriter(totalSyncFile);
FileWriter filter = new FileWriter(filterFile);
FileWriter costsSync = new FileWriter(costsFile);

costsSync.write("Processors; Computation Cost; Communication Cost; Overall Cost; \n");
for(int i = 0; i < computationCosts.length; i++)
{
    Integer compCost = new Integer(computationCosts[i]);
    Integer commCost = new Integer(communicationCosts[i]);
    int totalcost = compCost.intValue() + commCost.intValue();
```

```
        Integer totalCost = new Integer(totalcost);
        compSync.write(compCost.toString());
        compSync.write(";");
        commSync.write(commCost.toString());
        commSync.write(";");
        totalSync.write(totalCost.toString());
        totalSync.write(";");
        filter.write((String)bottleneckFilters.get(i));
        filter.write(";");

        //costsFile writing
        Integer currentProcessor = new Integer(i+1);
        costsSync.write(currentProcessor.toString());
        String costLine = new String(";" + compCost.toString() + ";" +
    commCost.toString() + ";" +
    totalCost.toString() + "; \n");
        costsSync.write(costLine);
}

costsSync.close();
totalSync.close();
filter.close();
compSync.close();
commSync.close();
    }

    catch(IOException e)
    {
System.err.println("File Output Error with message " + e.getMessage());
    }


}//generateOutputFiles()




/*  report():
 *  Print to int[]computationCosts and int[]communicationCosts
 *  both the communication and the computation costs of the bottleneck filter, takes current number of processors as
 */
public static void report(int currentprocessors)
{
    SIRFilter maxFilter = getHighestWorkFilter();
    int maxwork = ((Integer)filterWorkEstimates.get(maxFilter)).intValue();
    int commcost = ((Integer)filterCommunicationCosts.get(maxFilter)).intValue();
```

```
    computationCosts[currentprocessors - 1] = maxwork;
    communicationCosts[currentprocessors - 1] = commcost;
    bottleneckFilters.add(maxFilter.toString());
}


/*  parallelize();
 *  takes the highest work stateless filter on the critical path, cuts it in half, putting
 *  new work into filterWorkEstimates (replacing the old mapping)
 */
public static void parallelize()
{
    SIRFilter maxFilter = getOptimalFizzFilter();
    if(StatelessDuplicate.isFissable(maxFilter))
    {
int oldfizzfactor = ((Integer)filterTimesFizzed.get(maxFilter)).intValue();
Integer maxWork = (Integer)filterWorkEstimates.get(maxFilter);
int mwork = maxWork.intValue();
int totalwork = mwork * oldfizzfactor;

int newwork = totalwork / (oldfizzfactor + 1);  //simulate the fizzing
filterTimesFizzed.put(maxFilter, new Integer(oldfizzfactor + 1));
filterWorkEstimates.put(maxFilter, new Integer(newwork));
    }
    //else
}//void parallelize()


/**   getHighestWorkFilter
 *  Returns the filter with the highest overall work
 *  (computation + communication cost)
 */

public static SIRFilter getHighestWorkFilter()
{
    SIRFilter maxFilter = null;
    int maxwork = Integer.MIN_VALUE;
    Iterator sirFilters = filterWorkEstimates.keySet().iterator();

    while(sirFilters.hasNext())
    {
SIRFilter currentFilter = (SIRFilter)sirFilters.next();
int currentcompcost = ((Integer)filterWorkEstimates.get(currentFilter)).intValue();
int currentcomcost = ((Integer)filterCommunicationCosts.get(currentFilter)).intValue();
int currentwork = currentcompcost + currentcomcost;
```

```
if(currentwork >= maxwork)
{
    maxFilter = currentFilter;
    maxwork = currentwork;
}

    }

    return maxFilter;


}//getHighestWorkFilter

/**   getOptimalFizzFilter:  Returns the filter that would most
 *  benefit from being fizzed, using filterWorkEstimates()
 *  and filterTimesFizzed().
 */

public static SIRFilter getOptimalFizzFilter()
{
    int worksavings = Integer.MIN_VALUE;
    SIRFilter optFilter = null;
    Iterator sirFilters = filterWorkEstimates.keySet().iterator();
    while(sirFilters.hasNext())
    {
SIRFilter currentFilter = (SIRFilter)sirFilters.next();
Integer currentWork = (Integer)filterWorkEstimates.get(currentFilter);
int work = currentWork.intValue();
int fizzfactor = ((Integer)filterTimesFizzed.get(currentFilter)).intValue();
int totalwork = work * fizzfactor;
int nwork = totalwork / (fizzfactor + 1);
int currentsavings = work - nwork;

if(currentsavings >= worksavings)
{
    worksavings = currentsavings;
    optFilter = currentFilter;
}
    }

    return optFilter;
} //getOptimalFizzFilter()
```

```
/** getOptimalCommFizzFilter:  Similar to getOptimalFilter,
 *  but incorporates fan-in/fan-out cost k-(k/N), where
 *  k is constant.
 */

public static SIRFilter getOptimalCommFizzFilter()
{
    int fancost = 10;

    int worksavings = Integer.MIN_VALUE;
    SIRFilter optFilter = null;
    Iterator sirFilters = filterWorkEstimates.keySet().iterator();
    while(sirFilters.hasNext())
    {
SIRFilter currentFilter = (SIRFilter)sirFilters.next();
Integer currentWork = (Integer)filterWorkEstimates.get(currentFilter);
int work = currentWork.intValue();
int fizzfactor = ((Integer)filterTimesFizzed.get(currentFilter)).intValue();
int totalwork = work * fizzfactor;
int cfanwork = fancost - (fancost/fizzfactor);

int nwork = totalwork / (fizzfactor + 1);
int nfanwork = fancost - (fancost/(fizzfactor + 1));
int currentsavings = cfanwork + work - (nwork + nfanwork);

if(currentsavings >= worksavings)
{
    worksavings = currentsavings;
    optFilter = currentFilter;
    //System.err.println("blah with worksavings " + currentsavings);
}
    }
    //System.err.println("Saving " + worksavings);
    return optFilter;


}//getOptimalCommFizzFilter

    }


    //Static class ParallelWork

    static class ParallelVisitor extends EmptyStreamVisitor {
```

```
/** Mapping from streams to work estimates. **/
static HashMap newToOrigGraph;
static HashMap fizzedFiltersToOriginal;
static HashMap filterWorkEstimates;
static HashMap filterTimesFizzed;
static HashMap filterCommunicationCosts;
static MutableInt numFilters;
static int currentfilters;
static WorkEstimate work;
static HashMap[] executionCounts;
static GraphFlattener flatGraph;
//static int printsperexec;

public ParallelVisitor
    (GraphFlattener graph, WorkEstimate wrk, HashMap newormap, HashMap filtermapping,
     HashMap filters, HashMap comcosts, HashMap timesfizzed, HashMap[] execcounts, MutableInt num)
{
    flatGraph = graph;
    work = wrk;
    newToOrigGraph = newormap;
    fizzedFiltersToOriginal = filtermapping;
    filterTimesFizzed = timesfizzed;
    filterWorkEstimates = filters;
    filterCommunicationCosts = comcosts;
    numFilters = num;
    currentfilters = 0;
    executionCounts = execcounts;
    //printsperexec = 0;
}

    //Visitor Methods:

    /** visitFilter:
     *  Gets the work estimate of the filter, and stores it in
     *  filterWorkEstimates.  If the filter is stateless, put in statelessFilters vector
     *  also populates the communication cost hashmap, based on pushing, popping, and peeking
     */
    public void visitFilter(SIRFilter self, SIRFilterIter iter) {
int costperitem = 3;  //can be varied to see how it affects things!
NumberGathering execNumbers = new NumberGathering();
execNumbers.getPrintsPerSteady(flatGraph.getFlatNode(self), executionCounts);
int printsperexec = execNumbers.printsPerSteady;
int workestimate = work.getWork(self);
int numreps = work.getReps(self);
```

```
boolean nostate = StatelessDuplicate.isStateless(self);

if(printsperexec > 0)
    filterWorkEstimates.put(self, new Integer(workestimate /printsperexec));
else
    filterWorkEstimates.put(self, new Integer(workestimate));
fizzedFiltersToOriginal.put(self, (SIRFilter)newToOrigGraph.get(self));
s
int numpop = self.getPopInt();
int numpush = self.getPushInt();
int numpeek = self.getPeekInt();

int initcost = costperitem * (numpop + numpush + numpeek) * numreps;
if (printsperexec > 0)
    initcost = initcost / printsperexec;
filterCommunicationCosts.put(self, new Integer(initcost));

//end addition for constant communication cost
numFilters.mutint = numFilters.mutint + 1;

return;
}


    /** postVisitPipeline
     *  Requires that all the children have already been visited, and their
     *  estimated work calculations already be in streamsToWorkEstimates or filterWorkEstimates
     *  (Hence, this being postVisit, and not preVisit).  Since it is just a pipeline,
     *  function just sums up all the work of its children.
     *  Also puts the work of the pipeline in streamsToWorkEstimates.
     */

    public void postVisitPipeline(SIRPipeline self, SIRPipelineIter iter)
    {

return;

    }


    /** postVisitSplitJoin
     *  Requires that all the children have already been visited, and their
     *  estimated work calculations already be in either streamsToWorkEstimates
     *  or filterWorkEstimates.
     *  Since it is a splitjoin, it will take the maximum of the work
```

```
     * of its children (bottleneck path)
     */

    public void postVisitSplitJoin(SIRSplitJoin self, SIRSplitJoinIter iter)
    {

return;
    }



    }//class ParallelVisitor



    /** ParallelSyncVisitor
     * Similar to ParallelVisitor, except it populates FilterCommunicationCosts with the synchronization
     * costs dynamically.
     */

    static class ParallelSyncVisitor extends EmptyStreamVisitor {

static HashMap filterWorkEstimates;
static HashMap filterCommunicationCosts;
static HashMap[] executionCounts;
static GraphFlattener flatGraph;
//static int printsperexec;
static WorkEstimate work;
//static Vector splittersVisited;
static HashMap filterTimesFizzed;
static HashMap fizzedFiltersToOriginal;
static HashMap newToOrigGraph;
boolean synccost;

public ParallelSyncVisitor(GraphFlattener flat, WorkEstimate wrk,
   HashMap[] execcounts, HashMap workcosts, HashMap comcosts,
   HashMap fizzedtooriginal, HashMap timesfizzed,
   HashMap tooriggraph, boolean scost)
{
    executionCounts = execcounts;
    flatGraph = flat;
    filterWorkEstimates = workcosts;
    filterCommunicationCosts = comcosts;
    filterTimesFizzed = timesfizzed;
    fizzedFiltersToOriginal = fizzedtooriginal;
    newToOrigGraph = tooriggraph;
    work = wrk;
```

```
    synccost = scost;
}



//Visitor Methods:

/**  visitFilter:
 * Gets the workEstimate of the filter, and stores it in filterWorkEstimates.  Also
 * populates initial communication costs
 */

public void visitFilter(SIRFilter self, SIRFilterIter iter)
{
    NumberGathering execNumbers = new NumberGathering();
    execNumbers.getPrintsPerSteady(flatGraph.getFlatNode(self), executionCounts);
    int printsperexec = execNumbers.printsPerSteady;
    int costperitem = 3; //can be varied
    int workestimate = work.getWork(self);
    int numreps = work.getReps(self);
    if(printsperexec > 0)
filterWorkEstimates.put(self, new Integer(workestimate /printsperexec));
    else
filterWorkEstimates.put(self, new Integer(workestimate));

    //add initial comm cost for all filters, based on their push/pop/peek rates
    int numpop = self.getPopInt();
    int numpush = self.getPushInt();
    int numpeek = self.getPeekInt();

    //sync one
    int initcost = costperitem * numreps * (numpop + numpush);
    if(printsperexec > 0)
 initcost = initcost / printsperexec;
    //System.err.println("syncfactor bool is " + synccost);

    if(synccost)
    {
 SIRFilter origFilter = null;
 if(fizzedFiltersToOriginal.containsKey(self))
     origFilter = (SIRFilter) fizzedFiltersToOriginal.get(self);
 else
     origFilter = (SIRFilter)newToOrigGraph.get(self);
 int fizzfactor = ((Integer)filterTimesFizzed.get(origFilter)).intValue();
 //System.err.println("Synchro fizzfactor is " + fizzfactor);
```

```
int synchrocost = initcost * (fizzfactor - 1) / fizzfactor;
initcost += synchrocost;

    }


    filterCommunicationCosts.put(self, new Integer(initcost));
}//void visitFilter




/**   postVisitSplitJoin:
 * Ah, annoying procedure #1.  Use getSplitFactor to return what N value should be
 * used in synchronization calculation, based on potential parent splitjoins.
 * Then use flatnode to find the (filtery) children and add the synchro. cost to
 * its current synchro cost.
 */

public void postVisitSplitJoin(SIRSplitJoin self, SIRSplitJoinIter iter)
{

    int fconstant = 2;
    MutableInt syncfactor = new MutableInt();
    SIRSplitter splitter = self.getSplitter();
    FlatNode flatSplitter = flatGraph.getFlatNode(splitter);

    SIRSplitType splitterType = splitter.getType();
    if((splitterType.equals(SIRSplitType.ROUND_ROBIN)) || (splitterType.equals(SIRSplitType.DUPLICATE)) ||
       splitterType.equals(SIRSplitType.WEIGHTED_RR))
    {
if(((splitterType == SIRSplitType.ROUND_ROBIN) ||
   (splitterType == SIRSplitType.WEIGHTED_RR)) &&
   (synccost))
{
    syncfactor.mutint = self.getParallelStreams().size();

    int blah = self.size();

}
else
{
    syncfactor.mutint = 1; //if duplicate, don't include the size in the cost
}
getSplitFactor(syncfactor, flatSplitter);   //recursion happens in this procedure
for(int i = 0; i < flatSplitter.edges.length; i++)
```

```
{
    if(flatSplitter.edges[i].contents instanceof SIRFilter)
    {
SIRFilter currentFilter = (SIRFilter)flatSplitter.edges[i].contents;
NumberGathering execNumbers = new NumberGathering();
int numreps = work.getReps(currentFilter);
execNumbers.getPrintsPerSteady(flatGraph.getFlatNode(currentFilter), executionCounts);
int printsperexec = execNumbers.printsPerSteady;
int oldCommCost = ((Integer)filterCommunicationCosts.get(currentFilter)).intValue();
int fancost = 0;
if(printsperexec > 0)
    fancost = (fconstant * currentFilter.getPopInt() * numreps) / printsperexec;
else
    fancost = fconstant * currentFilter.getPopInt() * numreps;
int scost = fancost - (fancost/(syncfactor.mutint));
int newCommCost = oldCommCost;
if(syncfactor.mutint != 0)
    newCommCost = oldCommCost + fancost - (fancost/(syncfactor.mutint));

filterCommunicationCosts.put(currentFilter, new Integer(newCommCost));
    }
    //if not a filter, do nothing!
}
    }

}



/**  getSplitFactor(MutableInt syncfactor, FlatNode flatsplitter)
 * recurses up to find if it has any top level splitters.  If so, keep adding to syncfactor
 */

public void getSplitFactor(MutableInt syncfactor, FlatNode flatsplitter)
{
    FlatNode incomingNode = flatsplitter.incoming[0];
    if(incomingNode.contents instanceof SIRSplitter)
    {
SIRSplitter topSplitter = (SIRSplitter)incomingNode.contents;
if ((topSplitter.getType().equals(SIRSplitType.ROUND_ROBIN)) || (topSplitter.getType().equals(SIRSplitType.WEIGHTED_RR)))
{
    syncfactor.mutint += topSplitter.getWays();
    getSplitFactor(syncfactor, incomingNode);
}
if (topSplitter.getType().equals(SIRSplitType.DUPLICATE))
{
```

```
    getSplitFactor(syncfactor, incomingNode);
}
    }
    else
    {
SIRSplitter splitter = (SIRSplitter)flatsplitter.contents;
if ((splitter.getType().equals(SIRSplitType.ROUND_ROBIN)) || (splitter.getType().equals(SIRSplitType.WEIGHTED_RR)))
{
    syncfactor.mutint += splitter.getWays();
}

    }

}//getSplitFactor

    }//class ParallelSyncVisitor



    /**   BasicVisitor:
     * Class that takes the original input stream, and populates filterTimesFizzed with (1) values
     * as an initial state
     */

    static class BasicVisitor extends EmptyStreamVisitor
    {
static HashMap filterTimesFizzed;

public BasicVisitor(HashMap fizzmapping)
{
    filterTimesFizzed = fizzmapping;
}

public void visitFilter(SIRFilter self, SIRFilterIter iter)
{
    filterTimesFizzed.put(self, new Integer(1));
}

    }//class BasicVisitor



    /**   NoSyncVisitor:
     * Used in no communication model.  All it does is increment numfilters,
     * and populate filterWorkEstimates and filterCommunicationCosts
     */
```

```
    static class NoSyncVisitor extends EmptyStreamVisitor {

static MutableInt numFilters;
static HashMap filterWorkEstimates;
static HashMap filterCommunicationCosts;
static HashMap filterTimesFizzed;
static WorkEstimate work;
static GraphFlattener flatGraph;
static HashMap[] executionCounts;


public NoSyncVisitor
    (WorkEstimate wrk, HashMap filterwork, HashMap filtercomm,
     HashMap fizzmapping, HashMap[] execcounts,
     GraphFlattener graph, MutableInt nfilts)
{
    numFilters = nfilts;
    filterWorkEstimates = filterwork;
    filterCommunicationCosts = filtercomm;
    work = wrk;
    flatGraph = graph;
    executionCounts = execcounts;
    filterTimesFizzed = fizzmapping;
}

public void visitFilter(SIRFilter self, SIRFilterIter iter)
{
    filterTimesFizzed.put(self, new Integer(1));

    NumberGathering execNumbers = new NumberGathering();
    execNumbers.getPrintsPerSteady(flatGraph.getFlatNode(self), executionCounts);
    int printsperexec = execNumbers.printsPerSteady;
    int costperitem = 3; //can be varied
    int workestimate = work.getWork(self);
    int numreps = work.getReps(self);
    if(printsperexec > 0)
filterWorkEstimates.put(self, new Integer(workestimate /printsperexec));
    else
filterWorkEstimates.put(self, new Integer(workestimate));

    //add initial comm cost for all filters, based on their push/pop/peek rates
    int numpop = self.getPopInt();
    int numpush = self.getPushInt();
    int numpeek = self.getPeekInt();
```

```
    //sync one
    int initcost = costperitem * numreps * (numpop + numpush);
    if(printsperexec > 0)
initcost = initcost / printsperexec;

    filterCommunicationCosts.put(self, new Integer(initcost));

    numFilters.mutint = numFilters.mutint + 1;
}



    }//class NoSyncVisitor


}//class ParallelizationGathering
```

# Appendix B

# Benchmark Source Files

```
int->int filter MutableFilter(int pushrate, int poprate, int workiter) {

    init {
    }

    work push pushrate pop poprate {
int a;
int i;
int j;
int k;
int x;
for (j = 0; j < poprate; j++)
    {

a = pop();
    }

int b = 2;
for(i=0; i<workiter; i++)
    {
b++;
x+= b * i;

    }

for (k = 0; k < pushrate; k++)
    {
push(x);

    }
```

```
    }

}


int->void filter nullSink(int datarate){

    init{}

    work push 0 pop datarate{
int i;
int x;
print(1);
for(i = 0; i < datarate; i++)
    {
x = pop();

    }

    }


}



void->int filter nullSource(int datarate){

    init{}

    work push datarate pop 0 {

for(int i = 0; i < datarate; i++)
{
    push(1);

}


    }
}


void->void pipeline MutablePipelineCompBalanced() {
```

```
    add nullSource(2);
    add MutableFilter(2,2,1000);
    add MutableFilter(2,2,1000);
    add nullSink(2);

}

void->void pipeline MutablePipelineCompImbalanced() {

    add nullSource(2);
    add MutableFilter(2,2,300);
    add MutableFilter(2,2,1000);
    add nullSink(2);

}

void->void pipeline MutablePipelineCommBalanced() {

    add nullSource(70);
    add MutableFilter(70,70,500);
    add MutableFilter(70,70,500);
    add nullSink(70);

}

void->void pipeline MutablePipelineCommImbalanced() {

    add nullSource(70);
    add MutableFilter(70,70,100);
    add MutableFilter(70,70,700);
    add nullSink(70);

}

int->int splitjoin DefaultSplitJoin1()
{
    split duplicate;
    add MutableFilter(2,2,1000);
    add MutableFilter(2,2,1000);
    join roundrobin(2,2);

}
```

```
void->void pipeline MutableSplitJoinCompBalanced() {

    add nullSource(2);
    add DefaultSplitJoin1();
    add nullSink(4);

}


int->int splitjoin DefaultSplitJoin2()
{
    split duplicate;
    add MutableFilter(2,2,50);
    add MutableFilter(2,2,1000);
    join roundrobin(2,2);

}

void->void pipeline MutableSplitJoinCompImbalanced() {

    add nullSource(2);
    add DefaultSplitJoin2();
    add nullSink(4);

}

int->int splitjoin DefaultSplitJoin3()
{
    split duplicate;
    add MutableFilter(35,70,500);
    add MutableFilter(35,70,500);
    join roundrobin(35,35);

}

void->void pipeline MutableSplitJoinCommBalanced() {

    add nullSource(70);
    add DefaultSplitJoin3();
    add nullSink(70);

}


int->int splitjoin DefaultSplitJoin4()
```

```
{
    split duplicate;
    add MutableFilter(35,70,100);
    add MutableFilter(35,70,700);
    join roundrobin(35,35);

}

void->void pipeline MutableSplitJoinCommImbalanced() {

    add nullSource(70);
    add DefaultSplitJoin4();
    add nullSink(70);

}

int->int splitjoin DefaultSplitJoin5()
{
    split roundrobin(1,1);
    add MutableFilter(70,70,700);
    add MutableFilter(70,70,700);
    join roundrobin(1,1);

}

void->void pipeline RoundRobinBalanced() {

    add nullSource(140);
    add DefaultSplitJoin5();
    add nullSink(140);

}

int->int splitjoin DefaultSplitJoin6()
{
    split roundrobin(2,1);
    add MutableFilter(70,70,700);
    add MutableFilter(35,35,350);
    join roundrobin(2,1);

}

void->void pipeline RoundRobinImbalanced() {

    add nullSource(105);
```

```
    add DefaultSplitJoin();
    add nullSink(105);

}
```

# Appendix C

# Simulation Graphs for Round Robin Splitting

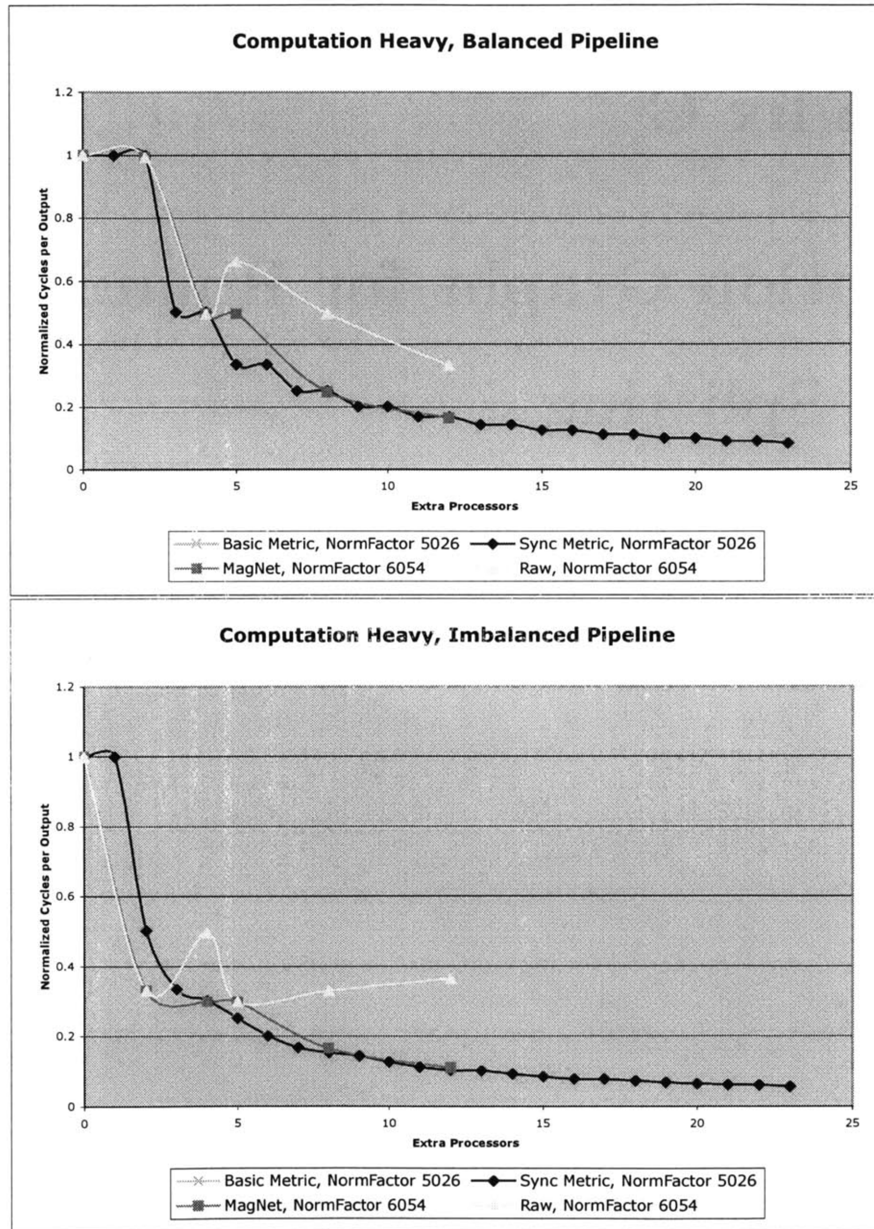



Figure C-1: Graphical comparison of Synchronization Metric to Raw simulator for Computation Heavy Pipelines

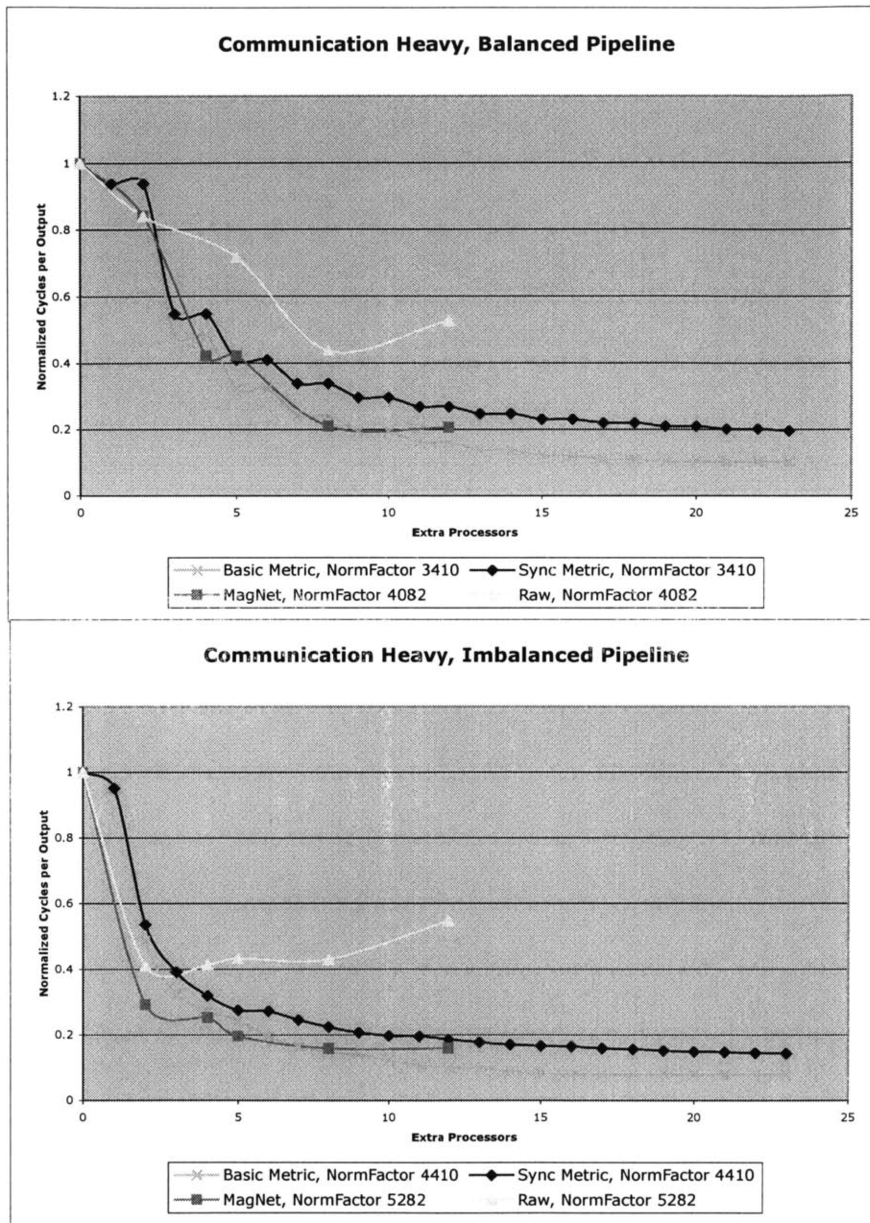



Figure C-2: Graphical comparison of Synchronization Metric to Raw simulator for Communication Heavy Pipelines

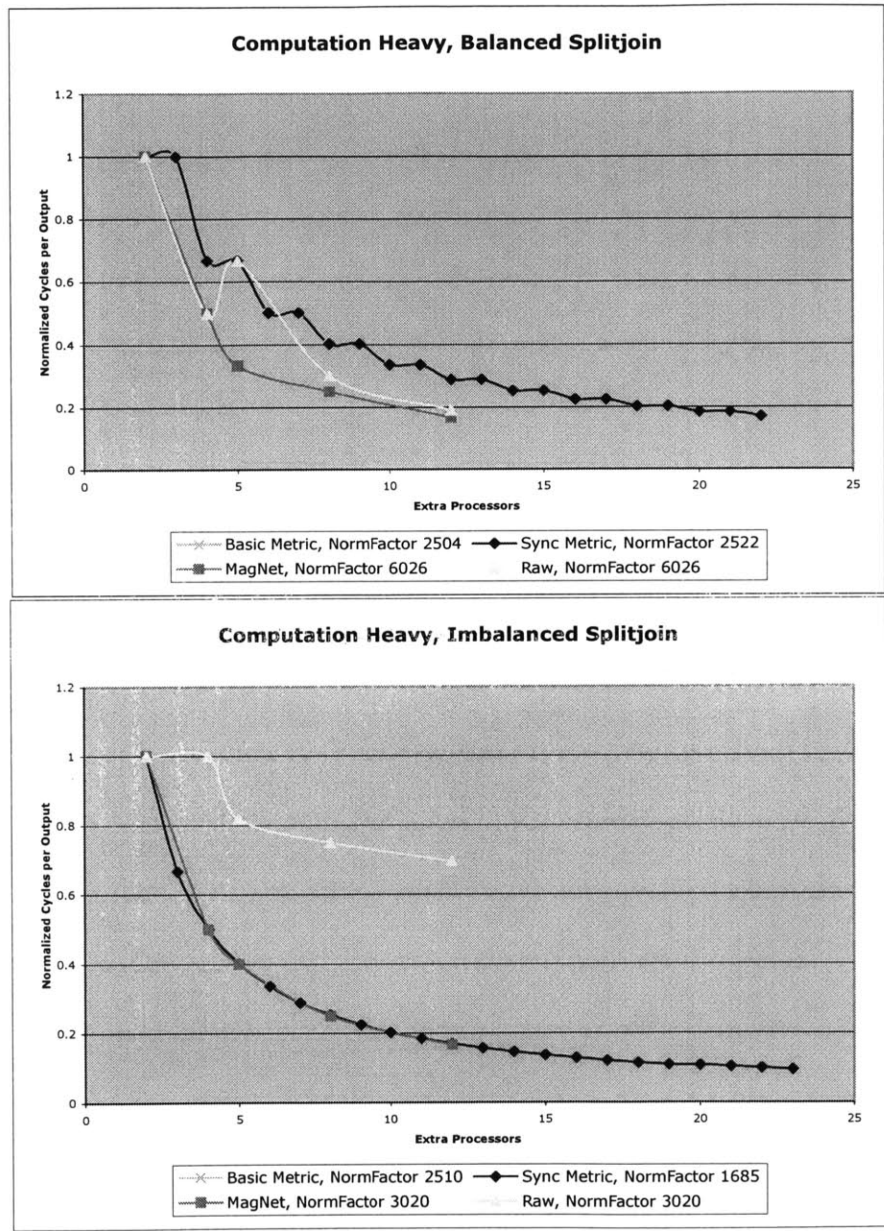



Figure C-3: Graphical comparison of Synchronization Metric to Raw simulator for Computation Heavy Splitjoins

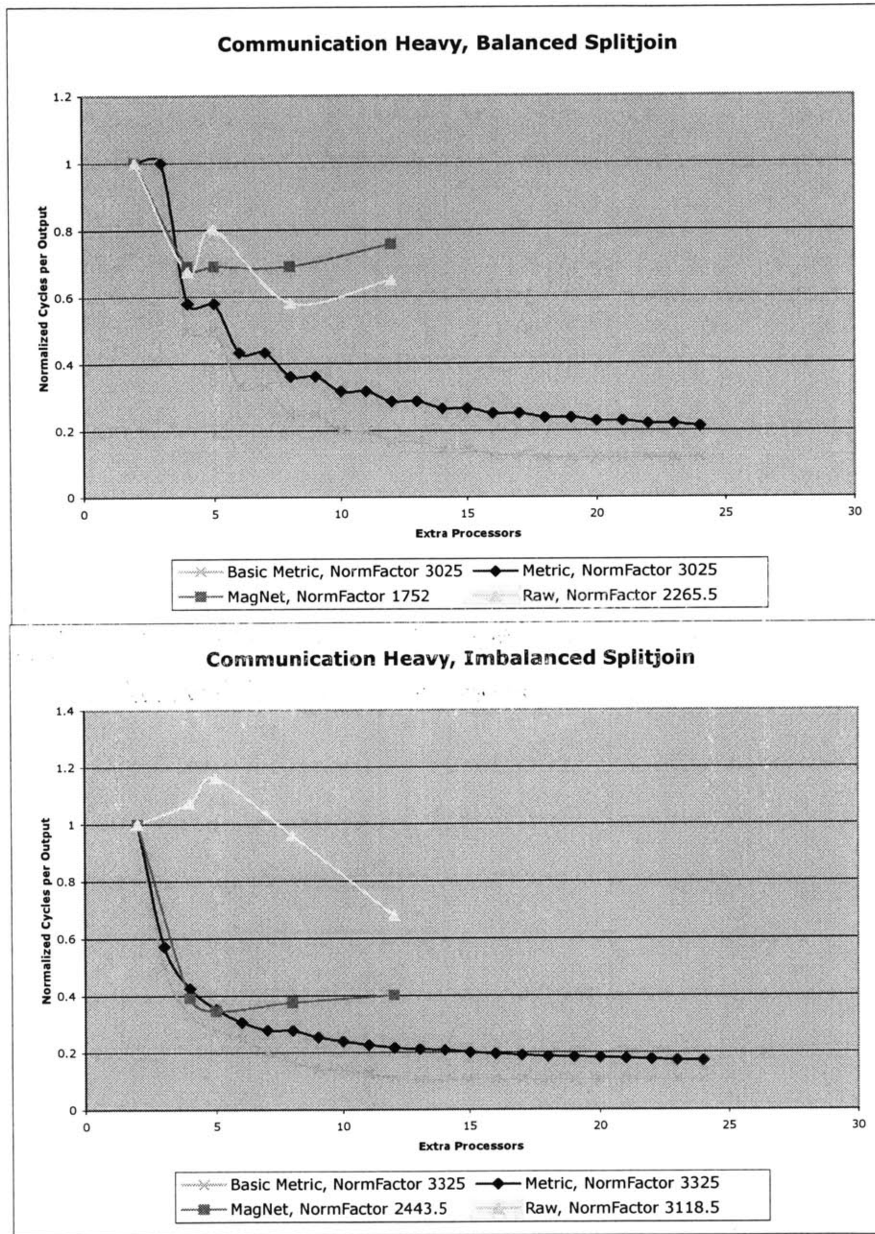



Figure C-4: Graphical comparison of Synchronization Metric to Raw simulator for Communication Heavy Splitjoins

# Bibliography

[1] Rajeev Barua, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Compiler support for scalable and efficient memory systems. *IEEE Transactions on Computers*, 50(11), November 2001.

[2] Ian Buck. Brook specification, 2003. http://merrimac.stanford.edu/brook/brookspec-v0.2.

[3] Michael Gordon, William Thies, Michal Karczmarek, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. *Proceedings of the Tenth Conference on Architectural Support for Programing Languages and Operating Systems*, 2002.

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

[5] Michal Karczmarek, William Thies, and Saman Amarasinghe. Phased scheduling of stream programs. In *Languages, Compilers, and Tools for Embedded Systems*, San Diego, CA, June 2003.

[6] R.J. Lipton and D.N. Serpanos. Uniform-cost communication in scalable multiprocessors. *International Conference on Parallel Processing*, 1990.

[7] Scott Rixner and et al. A bandwidth-efficient architecture for media processing. *HPCA*, 1998.

[8] Jaspal Subhlok, David O'Hallaron, Thomas Gross, Peter A. Dinda, and Jon Webb. Communication and memory requirements as the basis for mapping task and data parallel programs. *IEEE Proceedings of Supercomputing*, 1994.

[9] Jaspal Subhlok and Gary Vondran. Optimal use of mixed task and data parallelism for pipelined communications. *Journal of Parallel and Distributed Computing*, 2000.

[10] Jinwoo Suh, Eun-Gyu Kim, Stephen P. Crago, Lakshmi Srinivasan, and Matthew C. French. A performance analysis of pim, stream processing, and tiled processing on memory-intensive signal processing kernels. *IEEE Symposium on Computer Architecture*, 2003.

[11] Michael Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Benjamin Greenwald, Henry Hoffman, Jae-Wook Lee, Paul Johnson, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matthew Frank, Saman Amarasinghe, and Anant Agarwal. The raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, May 2002.

[12] William Thies, Michal Karczmarek, Michael Gordon, David Maze, Jeremy Wong, Henry Hoffman, Matthew Brown, and Saman Amarasinghe. Streamit: A compiler for streaming applications. Technical report, MIT/LCS, 2002.



6075-92