

MEASUREMENT AND RESOURCE ALLOCATION PROBLEMS IN DATA STREAMING SYSTEMS

A Thesis
Presented to
The Academic Faculty

by

Haiquan (Chuck) Zhao

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August 2010

MEASUREMENT AND RESOURCE ALLOCATION PROBLEMS IN DATA STREAMING SYSTEMS

Approved by:

Prof. Jun (Jim) Xu, Adviser
School of Computer Science
Georgia Institute of Technology

Prof. Mostafa H. Ammar
School of Computer Science
Georgia Institute of Technology

Prof. Ellen W. Zegura
School of Computer Science
Georgia Institute of Technology

Prof. Xiaoli Ma
School of Electrical and Computer
Engineering
Georgia Institute of Technology

Prof. Cathy H. Xia
Department of Integrated Systems
Engineering
Ohio State University

Date Approved: April 20, 2010

To Vrnda, Kana and Nimai.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my adviser Jun (Jim) Xu, who encouraged me to take up Ph.D. study in Georgia Tech, and provided kind guidance and consistent support throughout my entire study. He has been a teacher and a friend. I also must thank Leo Mark and Becky Wilson, who helped to resolve my enrollment status problem.

I would like to thank Cathy Xia, together with Don Towsley and Zhen Liu, who directed and helped my research on fork and join processing networks during my internship at IBM Research, and during the years afterwards to see it through. I want to thank Ashwin Lall who as a co-author practically mentored me on my first paper. I would like to thank all my co-authors, whose participation are invaluable for the many projects I was involved in, and from whom I have learned a lot. I would like to thank my thesis committee, who showed interest in my work and offered valuable suggestions.

My special thanks to my loving wife, Vrnda, who steadfastly supported my study, tolerated reduced income, and shouldered majority of the burden of raising our two young sons, Kana and Nimai – two lovely angels from heaven who can be demanding at times. My deep gratitude to my spiritual master, His Holiness Tamal Krishna Goswami, and to Lord Sri Krishna, for all the blessings in my life, both spiritually and materially.

TABLE OF CONTENTS

DEDICATION	iii
ACKNOWLEDGEMENTS	iv
LIST OF TABLES	vii
LIST OF FIGURES	viii
SUMMARY	ix
I INTRODUCTION	1
1.1 Measurement	1
1.2 Resource Allocation	2
II MEASUREMENT	4
2.1 Tail Bound Problem	4
2.1.1 L_1 norm and stochastic ordering	6
2.1.2 Chernoff method, Majorization and Convex Ordering	7
2.1.3 Selection With or Without Replacement	9
2.2 SRAM/DRAM Counter Array Architecture Revisited	14
2.3 DRAM-based Statistics Counter Array Architecture	16
2.3.1 Introduction	17
2.3.2 Related Work	22
2.3.3 Our Scheme	23
2.3.4 Analysis	26
2.3.5 Evaluation	33
2.4 Detecting Global Icebergs	38
2.4.1 Introduction	38
2.4.2 Background and Related Work	43
2.4.3 Problem Definition	44
2.4.4 Algorithmic Overview	46
2.4.5 Analysis	51

2.4.6	Analysis - Using F_1 Information	56
2.4.7	Properties	61
2.4.8	Empirical Evaluation	63
III	RESOURCE ALLOCATION	68
3.1	Introduction	68
3.1.1	Related Work	70
3.2	Synchronous fork and join only	71
3.2.1	Model and Problem Formulation	72
3.2.2	Distributed Algorithms	76
3.2.3	Primal Algorithm	77
3.2.4	Dual Algorithm	83
3.2.5	Primal vs. Dual	91
3.2.6	Evaluation	92
3.2.7	Concluding Remarks	95
3.3	General fork and join network	96
3.3.1	System Description	96
3.3.2	A Unified Modeling Framework	99
3.3.3	The Resource Allocation Problem	103
3.3.4	Distributed Algorithms	105
3.3.5	Performance Analysis	111
3.3.6	Extensions	116
3.3.7	Evaluation	121
3.3.8	Conclusion	123
APPENDIX A	PROOFS	125
REFERENCES	135

LIST OF TABLES

1	Solution Strategy for \mathcal{A}_L to a^*	7
2	Counter-example for negative correlation	13
3	Counter-example for positive correlation	14
4	Bound Comparison for $N = 10^6, n = 10^{12}, \mu = \frac{1}{10}, l = 4$	16
5	Comparison of different schemes for 16 million 64-bit counters	36
6	Task Graph and Bipartite Graph	101
7	Notations for model and algorithm	104

LIST OF FIGURES

1	Stream processing generating X_a	4
2	Tail Bound problem from \mathcal{A} to \mathcal{A}_L and a^*	5
3	SRAM/DRAM counter array architecture	14
4	Memory architecture for a randomized DRAM-based counter scheme.	23
5	Relationship among q , r , T and τ	29
6	XDR memory chip architecture	35
7	Overflow probability bound as a function of queue size K	36
8	The gap between icebergs and non-icebergs	45
9	Illustration of the sparse gap for real data sets	46
10	The sketches can be aggregated on any connected topology.	62
11	Iceberg Evaluation Results	65
12	Stream Processing	68
13	Example Task Graph	73
14	Several Lossiness Scenarios	75
15	Primal Algorithm Protocol	81
16	Utility Plots	92
17	Output Rate Plots	93
18	Example Task Graph	96
19	From m 's and a 's viewpoints	108
20	(M) \rightarrow (A), simple case	115
21	Simulation Results	124
22	(M) \rightarrow (A), typical case	132

SUMMARY

In a data streaming system, each component consumes one or several streams of data on the fly and produces one or several streams of data for other components. The entire Internet can be viewed as a giant data streaming system. Other examples include real-time exploratory data mining and high performance transaction processing. In this thesis we study several measurement and resource allocation problems of data streaming systems.

Measuring quantities associated with one or several data streams is often challenging because the sheer volume of data makes it impractical to store the streams in memory or ship them across the network. A data streaming algorithm processes a long stream of data in one pass using a small working memory (called a sketch). Estimation queries can then be answered from one or more such sketches. An important task is to analyze the performance guarantee of such algorithms. In this thesis we describe a tail bound problem that often occurs in data streaming algorithms, and we present a technique for solving it using majorization and convex ordering theories. We present two algorithms that utilize our technique. The first is to store a large array of counters in DRAM while achieving the update speed of SRAM. The second is to detect global icebergs across distributed data streams.

Resource allocation decisions are important for the performance of a data streaming system. The processing graph of a data streaming system forms a fork and join network. The underlying data processing tasks consists of a rich set of semantics that include synchronous and asynchronous data fork and data join. The different types of semantics and processing requirements introduce complex interdependence between various data streams within the network. We study the distributed resource allocation

problem in such systems with the goal of achieving the maximum total utility of output streams. For networks with only synchronous fork and join semantics, we present several decentralized iterative algorithms using primal and dual based optimization techniques. For general networks with both synchronous and asynchronous fork and join semantics, we present a novel modeling framework to formulate the resource allocation problem, and present a shadow-queue based decentralized iterative algorithm to solve the resource allocation problem. We show that all the algorithms guarantee optimality and demonstrate through simulation that they can adapt quickly to dynamically changing environments.

CHAPTER I

INTRODUCTION

In the data streaming processing model, data arrives in multiple, continuous, rapid, time-varying data streams. In a data streaming system ¹, each component consumes one or several data streams on the fly and produces one or several streams of data for other components. The entire Internet can be viewed as a giant data streaming system. Other examples include real-time exploratory data mining and high performance transaction processing. For a survey on model and issues in data streaming systems, see [12]. In this thesis we study two areas of challenging problems in data streaming systems: measurement and resource allocation.

1.1 Measurement

Data streaming systems are typically used for data intensive applications. The sheer volume of data makes it impractical to store the entire stream in memory, and the fast arrival speed makes it impractical to update the stream to disk or even DRAM. This makes measurement a challenging task. For example, if we want to find out the number of distinct items in the stream, how can we do so without being able to maintain one counter per item label? A data streaming algorithm is the answer here. A data streaming algorithm processes a long stream of data in one pass using a small working memory (called a sketch). Estimation queries can then be answered from one or more such sketches. The sketches act as randomized summary data structures.

An important aspect of any data streaming algorithm is its performance guarantee, in the form of some probabilistic statement. In Section 2.1, we present a commonly

¹Both the terms “data stream systems” and “data streaming systems” have been used in the literature. In this thesis we use “data streaming systems”

occurring tail bound problem, and a novel technique to solve the problem using majorization and convex ordering theories. We also prove a new convex ordering result that can be used with our technique. In Section 2.2, we briefly show how majorization and Schur-convexity can drastically improve the tail bound in [82]. We then present the following data streaming algorithms where our technique is applied.

In Section 2.3 we present a data structure to store a large array of counters in DRAM while achieving the speed of SRAM. This can be used as a building block for many data streaming algorithms that require a large array of counters.

In Section 2.4 we present an algorithm to detect global icebergs across distributed data streams. An iceberg is an item with high frequency of occurrence. This is challenging problem because a global iceberg across many streams may not be a local iceberg in any of the streams, and we cannot afford to ship all the streams across a network.

1.2 Resource Allocation

In Section 3.1 we introduce the resource allocation problem. A challenge for data streaming systems is the interdependency among the processing tasks. One data processing task may consume multiple streams and produce multiple streams, so the tasks can form a complex fork and join graph. The graph can consist of a rich set of semantics that include synchronous and asynchronous data fork and data join. The different types of semantics and processing requirements introduce complex interdependence between various data streams within the network. There are two important aspects that affect the performance of a data streaming system. The first is how tasks and logical links are placed on underlying servers and physical links. See for example [75]. The second is how to allocate the limited resources to the competing tasks and bandwidth demands. In this thesis we address the second aspect.

We study the distributed resource allocation problem in data streaming systems

with the goal of achieving the maximum total utility of output streams. For networks with only synchronous fork and join semantics, we present several decentralized iterative algorithms using primal and dual based optimization techniques in Section 3.2. We show that all the algorithms guarantee optimality and demonstrate through simulation that they can adapt quickly to dynamically changing environments.

For general networks with both synchronous and asynchronous fork and join semantics, it is hard to formulate the resource allocation problem based on the task graph alone. In Section 3.3, we present a novel modeling framework to transform the task graph into a bipartite graph. We formulate the resource allocation problem as an elegant convex optimization problem on this new graph, and we present a shadow-queue based decentralized iterative algorithm to solve it. We show that the algorithm guarantees optimality and demonstrate through simulation that it can adapt quickly to dynamically changing environments.

CHAPTER II

MEASUREMENT

2.1 *Tail Bound Problem*

In data streaming measurement problems, we are often interested in flow statistics. A data stream consists of a series of packets, and a flow is a group of packets with a common flow header. For simplicity of notation, let us assume that the flow headers are in the set $\{1, \dots, n\}$. Then we have a flow size distribution vector $a = \{a_1, \dots, a_n\}$, where a_i is the number of packets in flow i . One set of statistics of interest is called the frequency moments, defined as $F_p = \sum_{i=1}^n a_i^p, \forall p \geq 0$ [9]. A related definition is the L_p norm, defined as $L_p = (\sum_{i=1}^n a_i^p)^{1/p}, \forall p > 0$. So by definition F_1 is the same as L_1 . If we take p to infinity, we can define $L_\infty = \max_{i=1}^n a_i$. (Here the a_i 's are assumed to be non-negative. For a general definition of F_p and L_p , use $|a_i|$ in place of a_i .)

A common technique is to take an action on each packet based on a (randomly chosen) hash function of the flow label. This ensures that the same action is taken for the packets in the same flow, thus enabling us to extract flow statistics later. Mathematically speaking, each flow i is mapped to a random variable X_i , and the sketch will give us a random variable X_a that is a function of the vector a and the

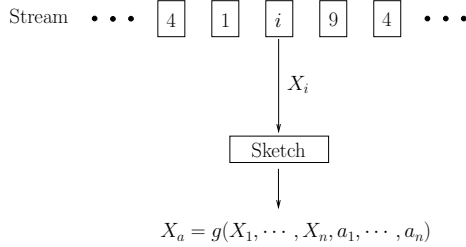


Figure 1: Stream processing generating X_a

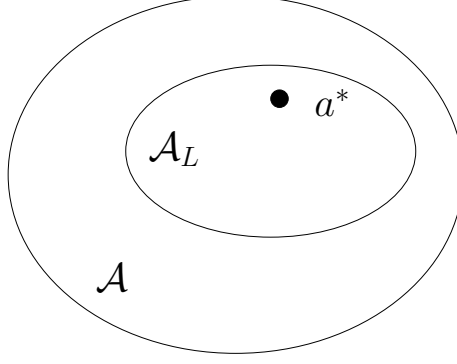


Figure 2: Tail Bound problem from \mathcal{A} to \mathcal{A}_L and a^*

random variables X_i . That is, $X_a = g(X_1, \dots, X_n, a_1, \dots, a_n)$ where $g : \mathbb{R}^{2n} \rightarrow \mathbb{R}$ is some function. See Figure 1 for an illustration. Depending on the algorithm, we may be interested in $\mathbb{E}[X_a]$, $\text{Var}[X_a]$, etc. For example, the F_0 algorithm in [27] gives $X_a = \min_{i=1}^n X_i$ where X_i 's are selected uniformly randomly from $[0, 1]$, and $\mathbb{E}[X_a] = 1/(F_0 + 1)$. The tug-of-war F_2 algorithm in [9] has $X_a = \sum_i a_i X_i$ where X_i 's are selected uniformly randomly from $\{-1, 1\}$, and $\mathbb{E}[X_a^2] = F_2$. The stable distribution L_p norm algorithm in [37] has $X_a = \sum_i a_i X_i$ where X_i is of stable distribution, and median estimator is used to extract the L_p norm with (ϵ, δ) guarantee. From the algorithms to be presented later in this chapter, we will see that an often encountered problem is to bound the tail probability $\Pr[X_a > C]$ where C is some constant. In fact, we want to bound

$$\max_{a \in \mathcal{A}} \Pr[X_a > C] \quad (1)$$

where $\mathcal{A} \in \mathbb{N}^n$ is the space of all possible flow size distributions.

This problem is a *worst-case large deviation* problem in nature [59] because it asks for a bound on the largest (worst case) value among the tail probabilities under all admissible (including adversarial) steaming data patterns. It is large deviation since C is typically much larger than $\mathbb{E}[X_a]$. This is a challenging problem because \mathcal{A} is usually a gigantic set and it is computationally impossible to enumerate over all

elements in \mathcal{A} .

In the next two sections we present a technique to reduce the tail bound problem to only one element in \mathcal{A} . First we reduce from \mathcal{A} to $\mathcal{A}_L = \{a \in \mathcal{A} \mid \sum_i a_i = L\}$ via stochastic ordering, where L is an upper bound of $\sum_i a_i$. Then we reduce from \mathcal{A}_L to $a^* \in \mathcal{A}_L$ via convex ordering. This is illustrated in Figure 2.

2.1.1 L_1 norm and stochastic ordering

It is often the case that we can assume $\sum_{i=1}^n a_i \leq L$ for some constant L , i.e. the L_1 norm of a is bounded. This is because a measurement epoch can be defined in terms of the L_1 norm, or it can be defined in terms of fixed time, where the link speed will limit the L_1 norm, or the L_1 norm can be easily estimated. It is intuitive that we only need to consider $\sum_{i=1}^n a_i = L$ for the worst case. To put this precisely, we use the concept of stochastic ordering.

Definition 1 (Stochastic order [56, 1.2.1, 1.2.8]). *The random variable X is said to be smaller than the random variable Y in stochastic order (written $X \leq_{st} Y$), if $\Pr[X > t] \leq \Pr[Y > t]$ for all real t , or equivalently, if $\mathbb{E}[f(X)] \leq \mathbb{E}[f(Y)]$ holds for all increasing functions f , for which both expectations exist.*

Theorem 1. *Suppose $X_a = g(X; a)$ where $g(x_1, \dots, x_n, y_1, \dots, y_n)$ is increasing in each component y_i , and for every $a \in \mathcal{A}$ there exists $b \in \mathcal{A}_L$ such that $a \leq b$ component-wise, then $\max_{a \in \mathcal{A}} \Pr[X_a \geq C] = \max_{a \in \mathcal{A}_L} \Pr[X_a \geq C]$.*

Proof. For any increasing function f , we have

$$\begin{aligned} \mathbb{E}[f(X_a)] &= \mathbb{E}[f(g(X, a))] = \int f(g(\omega, a)) d\omega \\ &\leq \int f(g(\omega, b)) d\omega = \mathbb{E}[f(g(X, b))] = \mathbb{E}[f(X_b)] \end{aligned}$$

therefore $X_a \leq_{st} X_b$, so $\Pr[X_a > C] \leq \Pr[X_b > C]$, and the theorem follows. \square

Table 1: Solution Strategy for \mathcal{A}_L to a^*		
Majorization $a \leq_m b$		
\downarrow	\leftarrow	Marshall theorems
Convex Order $E[f(X_a)] \leq E[f(X_b)]$		
\downarrow	\leftarrow	$e^{\theta x}$ is convex function
Moment Generating Function (MGF) $E[e^{\theta X}]$		
\downarrow	\leftarrow	Chernoff Bound
Tail Bound $\Pr[X > C]$		

2.1.2 Chernoff method, Majorization and Convex Ordering

The solution strategy to reduce from \mathcal{A}_L to a^* is illustrated in Table 1. We will described the involved steps from bottom up.

First we describe the standard Chernoff method ¹ of obtaining sharp tail bounds from the Moment Generating function (MGF) of a random variable X . We have

$$\Pr[X > C] = \Pr[e^{\theta X} > e^{\theta C}] \leq \frac{E[e^{\theta X}]}{e^{\theta C}},$$

where $\theta > 0$ is any constant, and the last step is due to the Markov inequality.

Since this is true for all θ , we have

$$\Pr[X > C] \leq \min_{\theta > 0} \frac{E[e^{\theta X}]}{e^{\theta C}}. \quad (2)$$

So if we can bound the MGF $E[e^{\theta X_a}]$, we will be able to bound $\Pr[X_a > C]$. Since $\sum_i a_i = L$, and $e^{\theta x}$ is a convex function of x , the concepts of majorization and convex ordering comes into play. We first present a few definitions.

Definition 2 (Majorization [51, 1.A.1]). *For any n -dimensional vectors a and b , let $a_{[1]} \geq \dots \geq a_{[n]}$ denote the components of a in decreasing order, and $b_{[1]} \geq \dots \geq b_{[n]}$ denote the components of b in decreasing order. We say a is majorized by b , denoted $a \leq_M b$, if*

$$\begin{cases} \sum_{i=1}^k a_{[i]} \leq \sum_{i=1}^k b_{[i]}, & \text{for } k = 1, \dots, n-1 \\ \sum_{i=1}^n a_{[i]} = \sum_{i=1}^n b_{[i]} \end{cases} \quad (3)$$

¹This technique was apparently first used by Bernstein according to [33].

Definition 3 (Schur-convex [51, 3.A.1]). *A function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is called Schur-convex (resp. Schur-concave), if $x \leq_M y$ implies $f(x) \leq f(y)$ (resp. $f(x) \geq f(y)$).*

Definition 4 (Exchangeable random variables). *A sequence of random variables X_1, \dots, X_n is called exchangeable, if for any permutation $\sigma : [1, \dots, n] \rightarrow [1, \dots, n]$, the joint probability distribution of the permuted sequence $X_{\sigma(1)}, \dots, X_{\sigma(n)}$ is the same as the joint probability distribution of the original sequence.*

For example, a sequence of independent and identically distributed random variables are exchangeable. Another example is a sequence of random variables resulting from sampling without replacement.

Definition 5 (Convex function). *A real function f is called convex, if $f(\alpha x + (1 - \alpha)y) \leq \alpha f(x) + (1 - \alpha)f(y)$ for all x and y and all $0 < \alpha < 1$.*

Definition 6 (Convex order [56, 1.5.1]). *Let X and Y be random variables with finite means. Then we say that X is less than Y in (increasing) convex order, written $X \leq_{cx} Y$ ($X \leq_{icx} Y$), if $E[f(X)] \leq E[f(Y)]$ holds for all real (increasing) convex functions f , for which both expectations exist.*

Since the MGF ($E[e^{\theta X}]$) is expectation of an increasing convex function ($e^{\theta x}$) of X , establishing convex order or increasing convex order will help to bound the MGF.

The following theorem from Marshall [51] about convex functions and exchangeable random variables has many useful corollaries.

Theorem 2 ([51, 11.B.1]). *Let X_1, \dots, X_n be exchangeable random variables. Let $\Phi : \mathbb{R}^{2n} \rightarrow \mathbb{R}$ be a function of two vector arguments. Suppose that Φ satisfies*

- (i) $\Phi(x; a)$ is convex in a for each fixed x ,
- (ii) $\Phi(x\Pi; a\Pi) = \Phi(x; a)$ for all permutation matrices Π ,
- (iii) $\Phi(x; a)$ is Borel measurable in x for each fixed a .

Then $\Psi(a) = \mathbb{E}[\Phi(X; a)]$ is symmetric and convex, thus Schur-convex [51, 3.C.2]

Corollary 3 ([51, 11.B.2.c]). *If X_1, \dots, X_n are exchangeable random variables, a and b are n -dimensional vectors, then $a \leq_M b$ implies $\sum_{i=1}^n a_i X_i \leq_{cx} \sum_{i=1}^n b_i X_i$.*

The corollary is true because for any convex function f , $f(\sum_{i=1}^n a_i x_i)$ satisfies the conditions in Theorem 2.² Therefore, for $X_a = \sum_{i=1}^n a_i X_i$, if we can find a^* such that $a \leq_M a^*, \forall a \in \mathcal{A}_L$, then $X_a \leq_{cx} X_{a^*}$ according to Corollary 3, thus $\mathbb{E}[e^{\theta X_a}] \leq \mathbb{E}[e^{\theta X_{a^*}}]$ by Definition 6. From the Chernoff bound we get

$$\Pr[X_a > C] \leq \min_{\theta > 0} \frac{\mathbb{E}[e^{\theta X_a}]}{e^{\theta C}} \leq \min_{\theta > 0} \frac{\mathbb{E}[e^{\theta X_{a^*}}]}{e^{\theta C}} \quad (4)$$

Thus we have reduced the tail bound problem to calculating the MGF for X_a for only one $a^* \in \mathcal{A}$.

2.1.3 Selection With or Without Replacement

MGF is easy to compute for sum of independent random variables. In our algorithms we encounter sum of weakly dependent random variables from selection without replacement. Fortunately the follow theorem allows us to use independent random variables from selection with replacement to dominate the dependent random variables from selection without replacement

Theorem 4 ([33, Theorem 4]). *Suppose $n \leq N$. Let the population C consist of N values c_1, \dots, c_N . Let X_1, \dots, X_n denote a random sample without replacement from C and let Y_1, \dots, Y_n denote a random sample with replacement from C . Then $\sum_{i=1}^n X_i$ is dominated by $\sum_{i=1}^n Y_i$ in the convex order.*

In our algorithms we need to use weighted sum of the random variables, therefore we extend Hoeffding's result to the following theorem.

²We will see another corollary of Theorem 2 in Section 2.4.6 as Theorem 13.

Theorem 5. *Same notations as in Theorem 4. Let a_1, \dots, a_n be constants, $a_i > 0, \forall i$, then $\sum_{i=1}^n a_i X_i$ is dominated by $\sum_{i=1}^n a_i Y_i$ in the convex order, i.e. for any convex function f ,*

$$\mathbb{E}f\left(\sum_{i=1}^n a_i X_i\right) \leq \mathbb{E}f\left(\sum_{i=1}^n a_i Y_i\right) \quad (5)$$

Remark: The theorem can be shown to be false if a_i 's can have mixed signs. e.g. $n = N = 2, c_1 = 1, c_2 = 2, a_1 = 1, a_2 = -1$, f is strictly convex, then $\mathbb{E}[f(\sum a_i X_i)] - \mathbb{E}[f(\sum a_i Y_i)] = \frac{1}{2}[f(-1) + f(1)] - \frac{1}{4}[f(-1) + f(1) + 2f(0)] = \frac{1}{2}[\frac{1}{2}f(-1) + \frac{1}{2}f(1) - f(0)] > 0$ by Jensen's inequality. On the other hand, the signs of c_i 's do not matter, since we can do a shift to make all c_i 's positive.

Proof. Following Hoeffding's notation, for an arbitrary function g of n variables we have,

$$\mathbb{E}g(X_1, \dots, X_n) = \frac{1}{N^{(n)}} \sum_{N,n} g(c_{i_1}, \dots, c_{i_n}), \quad (6)$$

$$\mathbb{E}g(Y_1, \dots, Y_n) = \frac{1}{N^n} \sum_{i_1=1}^N \cdots \sum_{i_n=1}^N g(c_{i_1}, \dots, c_{i_n}) \quad (7)$$

Where $N^{(n)} = N(N-1)\dots(N-n+1)$, and $\sum_{N,n}$ is taken over all n -tuples i_1, \dots, i_n of distinct positive integers not exceeding N . The goal is to find function \bar{g} such that the N^n terms of g in (7) can be rewritten into $N^{(n)}$ terms of \bar{g} , and each term of \bar{g} will dominate each term of g in (6) for $g(x_1, \dots, x_n) = f(a_1 x_1 + \dots + a_n x_n)$. So we want

$$\begin{aligned} \mathbb{E}g(Y_1, \dots, Y_n) &= \frac{1}{N^n} \sum_{i_1=1}^N \cdots \sum_{i_n=1}^N g(c_{i_1}, \dots, c_{i_n}) \\ &= \frac{1}{N^{(n)}} \sum_{N,n} \bar{g}(c_{i_1}, \dots, c_{i_n}) = \mathbb{E}\bar{g}(X_1, \dots, X_n) \end{aligned} \quad (8)$$

For $n = 2$, we can use

$$\bar{g}(x_1, x_2) = \frac{N-1}{N} g(x_1, x_2) + \frac{1}{N} \left(\frac{a_1}{a_1 + a_2} g(x_1, x_1) + \frac{a_2}{a_1 + a_2} g(x_2, x_2) \right)$$

In general, let $[n]$ denote the set $\{1, \dots, n\}$. Let $P = \{\rho = \{A_1, \dots, A_k\} | A_i \subset [n], \cup_i A_i = [n]\}$ be the set of all partitions of $[n]$. Let $H = \{h : [n] \rightarrow [n] | h \circ h = h\}$ be the set of all mappings from $[n]$ to itself, with the restriction that the points in image of h , $Im(h)$, are fixed points of h . We can denote h by the vector $(h(1), \dots, h(n))$. Any h naturally induces a partition $\rho_h = \{h^{-1}(j) | j \in Im(h)\}$. For example, the mappings $(1, 1, 3, 3), (2, 2, 3, 3), (1, 1, 4, 4), (2, 2, 4, 4)$ all induce the partition $\{\{1, 2\}, \{3, 4\}\}$. We will let \bar{g} will be linear combinations of $g(x_{h(1)}, \dots, x_{h(n)})$ for all $h \in H$. e.g. in the case of $n = 2$, due to our definition of H , we do not include $g(x_2, x_1)$ in $\bar{g}(x_1, x_2)$. We define \bar{g} as

$$\bar{g}(x_1, \dots, x_n) = \sum_{h \in H} p_h g(x_{h(1)}, \dots, x_{h(n)}), \text{ where } p_h = p_{\rho_h} \prod_{j \in Im(h)} \frac{a_j}{\sum_{i \in h^{-1}(j)} a_i}, \quad (9)$$

where p_ρ are constants. We note that from the definition, p_h and p_ρ satisfy

$$\sum_{\{h | \rho_h = \rho\}} p_h = p_\rho, \quad \forall \rho \in P \quad (10)$$

$$\sum_{h \in H} p_h = \sum_{\rho \in P} p_\rho \quad (11)$$

We need to argue that there exist p_ρ such that (8) is true for any g . Here we will specify p_ρ exactly. Take $n = 4$ for example. For any distinct $c_1, c_2 \in C$, $g(c_1, c_1, c_2, c_2)$ appears once on the LHS of (8). It corresponds to $\rho = \{\{1, 2\}, \{3, 4\}\}$. On the RHS, for any distinct $\alpha, \beta \in C$ that are different from c_1, c_2 , we have

$$\begin{aligned} \bar{g}(c_1, \alpha, c_2, \beta) &= \frac{\mathbf{p}_\rho \mathbf{a}_1 \mathbf{a}_3 \mathbf{g}(\mathbf{c}_1, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_2)}{(\mathbf{a}_1 + \mathbf{a}_2)(\mathbf{a}_3 + \mathbf{a}_4)} + \frac{p_\rho a_2 a_3 g(\alpha, \alpha, c_2, c_2)}{(a_1 + a_2)(a_3 + a_4)} + \frac{p_\rho a_1 a_4 g(c_1, c_1, \beta, \beta)}{(a_1 + a_2)(a_3 + a_4)} + \frac{p_\rho a_2 a_4 g(\alpha, \alpha, \beta, \beta)}{(a_1 + a_2)(a_3 + a_4)} + \dots \\ \bar{g}(\alpha, c_1, c_2, \beta) &= \frac{p_\rho a_1 a_3 g(\alpha, \alpha, c_2, c_2)}{(a_1 + a_2)(a_3 + a_4)} + \frac{\mathbf{p}_\rho \mathbf{a}_2 \mathbf{a}_3 \mathbf{g}(\mathbf{c}_1, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_2)}{(\mathbf{a}_1 + \mathbf{a}_2)(\mathbf{a}_3 + \mathbf{a}_4)} + \frac{p_\rho a_1 a_4 g(\alpha, \alpha, \beta, \beta)}{(a_1 + a_2)(a_3 + a_4)} + \frac{p_\rho a_2 a_4 g(c_1, c_1, \beta, \beta)}{(a_1 + a_2)(a_3 + a_4)} + \dots \\ \bar{g}(c_1, \alpha, \beta, c_2) &= \frac{p_\rho a_1 a_3 g(c_1, c_1, \beta, \beta)}{(a_1 + a_2)(a_3 + a_4)} + \frac{p_\rho a_2 a_3 g(\alpha, \alpha, \beta, \beta)}{(a_1 + a_2)(a_3 + a_4)} + \frac{\mathbf{p}_\rho \mathbf{a}_1 \mathbf{a}_4 \mathbf{g}(\mathbf{c}_1, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_2)}{(\mathbf{a}_1 + \mathbf{a}_2)(\mathbf{a}_3 + \mathbf{a}_4)} + \frac{p_\rho a_2 a_4 g(\alpha, \alpha, c_2, c_2)}{(a_1 + a_2)(a_3 + a_4)} + \dots \\ \bar{g}(\alpha, c_1, \beta, c_2) &= \frac{p_\rho a_1 a_3 g(\alpha, \alpha, \beta, \beta)}{(a_1 + a_2)(a_3 + a_4)} + \frac{p_\rho a_2 a_3 g(c_1, c_1, \beta, \beta)}{(a_1 + a_2)(a_3 + a_4)} + \frac{p_\rho a_1 a_4 g(\alpha, \alpha, c_2, c_2)}{(a_1 + a_2)(a_3 + a_4)} + \frac{\mathbf{p}_\rho \mathbf{a}_2 \mathbf{a}_4 \mathbf{g}(\mathbf{c}_1, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_2)}{(\mathbf{a}_1 + \mathbf{a}_2)(\mathbf{a}_3 + \mathbf{a}_4)} + \dots \end{aligned}$$

Adding up the diagonal terms gives us $p_\rho g(c_1, c_1, c_2, c_2)$. Since there are $(N-2)(N-3)$ choices for α, β , we need to have $\frac{1}{N^4} = \frac{(N-2)(N-3)p_\rho}{N^{(4)}}$, so $p_\rho = \frac{N^{(2)}}{N^4}$. In general, we need to set $p_\rho = \frac{N^{(|\rho|)}}{N^n}$, where $|\rho|$ denotes the number of subsets in partition ρ .

Now we consider the case $g(x_1, \dots, x_n) = f(a_1 x_1 + \dots + a_n x_n)$. If we set $f(x) = 1$,

we see from (8) and (9) that

$$\sum_{h \in H} p_h = 1 \quad (12)$$

This can also be shown directly, since $\sum_{h \in H} p_h = \sum_{\rho \in P} p_\rho = \frac{1}{N^n} \sum_{\rho \in P} N^{(|\rho|)}$. We can view $\sum_{\rho \in P} N^{(|\rho|)}$ as one way to count all ordered samples of size n with replacement, by considering all repetition patterns, thus it equals N^n .

If we set $f(x) = x$, then $\bar{g}(x_1, \dots, x_n)$ is a linear combination of x_1, \dots, x_n . It may be possible to argue that the ratio between the coefficients for x_{i_1} and x_{i_2} will be a_{i_1}/a_{i_2} . Here we calculate it directly. We have

$$\sum_{\{h|\rho_h=\rho\}} p_h(a_1x_{h(1)} + \dots + a_nx_{h(n)}) = p_\rho(a_1x_1 + \dots + a_nx_n), \quad \forall \rho \in P$$

For example, for $\rho = \{\{1, 2\}, \{3, 4\}\}$, we have

$$\begin{aligned} & \sum_{\{h|\rho_h=\rho\}} p_h(a_1x_{h(1)} + \dots + a_nx_{h(n)}) \\ &= \frac{p_\rho a_1 a_3 (a_1x_1 + a_2x_1 + a_3x_3 + a_4x_3)}{(a_1 + a_2)(a_3 + a_4)} + \frac{p_\rho a_2 a_3 (a_1x_2 + a_2x_2 + a_3x_3 + a_4x_3)}{(a_1 + a_2)(a_3 + a_4)} \\ &+ \frac{p_\rho a_1 a_4 (a_1x_1 + a_2x_1 + a_3x_4 + a_4x_4)}{(a_1 + a_2)(a_3 + a_4)} + \frac{p_\rho a_2 a_4 (a_1x_2 + a_2x_2 + a_3x_4 + a_4x_4)}{(a_1 + a_2)(a_3 + a_4)} \end{aligned}$$

For x_1 the first and third terms yield $\frac{p_\rho a_1 a_3 x_1}{x_3 + x_4} + \frac{p_\rho a_1 a_4 x_1}{x_3 + x_4} = p_\rho a_1 x_1$, and similarly for x_2, x_3, x_4 . Therefore

$$\sum_h p_h(a_1x_{h(1)} + \dots + a_nx_{h(n)}) = \sum_\rho p_\rho(a_1x_1 + \dots + a_nx_n) = a_1x_1 + \dots + a_nx_n \quad (13)$$

From (12), (13) and Jensen's inequality we get

$$\begin{aligned} \bar{g}(x_1, \dots, x_n) &= \sum_{h \in H} p_h f(a_1x_{h(1)} + \dots + a_nx_{h(n)}) \\ &\geq f\left(\sum_{h \in H} p_h (a_1x_{h(1)} + \dots + a_nx_{h(n)})\right) \\ &= f(a_1x_1 + \dots + a_nx_n) \end{aligned} \quad (14)$$

Hence $E\bar{g}(X_1, \dots, X_n) \geq Ef(a_1X_1 + \dots + a_nX_n)$, which together with (8) completes the proof. \square

Table 2: Counter-example for negative correlation

		X_1			
		-2	-1	1	2
X_2	-2	1/20	0	3/20	0
	-1	0	3/20	0	3/20
	1	3/20	0	3/20	0
	2	0	3/20	0	1/20

We note that in Theorem 4, X_i 's and Y_i 's have identical distribution, the Y_i 's are independent while the X_i 's are pairwise negatively correlated. However this is not a sufficient condition for the convex order to hold, even if the X_i 's are exchangeable. Consider the following counterexample. Let the joint distribution of X_1, X_2 be as in Table 2. X_1, X_2 are identically distributed as $X : \Pr[X = -2] = \Pr[X = 2] = 1/5$, $\Pr[X = -1] = \Pr[X = 1] = 3/10$, and $E[X] = 0$. They are negatively correlated, since $Cov(X_1, X_2) = E[X_1, X_2] - E[X_1]E[X_2] = -0.5 < 0$. Let Y_1, Y_2 be i.i.d. as X . We have

$$E[f(X_1 + X_2)] = \frac{1}{20}f(4) + 0 \cdot f(3) + \frac{3}{20}f(2) + \dots$$

$$E[f(Y_1 + Y_2)] = \frac{1}{25}f(4) + \frac{3}{50}f(3) + \frac{9}{100}f(2) + \dots$$

If we choose the convex function $f(x) = e^{100x}$, then clearly the $f(4)$ term dominates, so $E[f(X_1 + X_2)] > E[f(Y_1 + Y_2)]$, therefore $X_1 + X_2$ is not dominated by $Y_1 + Y_2$ in the convex order.

Similarly, if the X_i 's are positively correlated and exchangeable, it is also not a sufficient condition that $\sum_{i=1}^n X_i$ will dominate $\sum_{i=1}^n Y_i$ in the convex order. Let the joint distribution of X_1, X_2 be as in Table 3. X_1, X_2 are uniformly distributed among $\{-2, -1, 1, 2\}$, and they are clearly positively correlated. Let Y_1, Y_2 be i.i.d. as X_1 .

Table 3: Counter-example for positive correlation

		X_1			
		-2	-1	1	2
X_2	-2	0	1/4	0	0
	-1	1/4	0	0	0
	1	0	0	0	1/4
	2	0	0	1/4	0

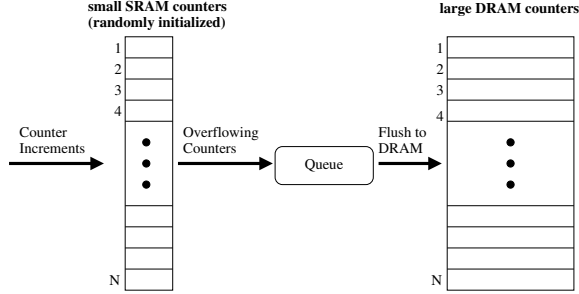


Figure 3: SRAM/DRAM counter array architecture

We have

$$\begin{aligned} \mathbb{E}[f(X_1 + X_2)] &= 0 \cdot f(4) + \frac{1}{2}f(3) + 0 \cdot f(2) + \dots \\ \mathbb{E}[f(Y_1 + Y_2)] &= \frac{1}{16}f(4) + \frac{1}{8}f(3) + \frac{1}{16}f(2) \dots \end{aligned}$$

Again, if we choose the convex function $f(x) = e^{100x}$, the $f(4)$ terms dominates, so $\mathbb{E}[f(X_1 + X_2)] < \mathbb{E}[f(Y_1 + Y_2)]$, therefore $X_1 + X_2$ does not dominate $Y_1 + Y_2$ in the convex order.

2.2 SRAM/DRAM Counter Array Architecture Revisited

Before we present our data streaming algorithms, let us first show how majorization and Schur-convexity could drastically improve the tail bound in the SRAM/DRAM counter array architecture in [82]. The scheme is depicted in Figure 3. Let us first recap some notations: There are n updates to N counters, each SRAM counter has l bits, and the SRAM/DRAM speed difference is μ . The length of the queue is K . Given any interval $[s, t]$, let $\tau = t - s$. Let X_i be the value of counter i at time s , thus

$\{X_i\}$ are i.i.d. random variable uniformly distributed in the set $\{0, 1, \dots, 2^l - 1\}$. Let a_i be the number of updates to counter i during the interval, thus $\sum_{i=1}^N a_i \leq \tau$. The problem is to bound $\Pr[X_a > K + \mu\tau]$ where

$$X_a = \sum_{i=1}^N \lfloor \frac{a_i + X_i}{2^l} \rfloor = \sum_{i=1}^N \lfloor \frac{a_i}{2^l} \rfloor + \lfloor \frac{(a_i \% 2^l) + X_i}{2^l} \rfloor, \quad (15)$$

where $a_i \% 2^l$ stands for a_i modulo 2^l . $\lfloor \frac{(a_i \% 2^l) + X_i}{2^l} \rfloor$ is equivalent to a Bernoulli random variable with expectation $\frac{(a_i \% 2^l)}{2^l}$, and $\lfloor \frac{a_i}{2^l} \rfloor$ can be viewed as $\lfloor \frac{a_i}{2^l} \rfloor$ Bernoulli random variables with expectation 1. So if we ignore all terms of value 0, X_a can be viewed as sum of at most τ independent Bernoulli random variables. From Theorem 1 we can assume that $\sum_{i=1}^N a_i = \tau$, so $E[X_a] = \sum_{i=1}^N \frac{a_i}{2^l} = \frac{\tau}{2^l}$.

We have the following theorem for sum of independent Bernoulli random variables,

Theorem 6 ([28, Theorem 1.1]).³ *Consider n independent Bernoulli trials with success probabilities p_1, \dots, p_n , and $\sum_{i=1}^n p_i = \mu$. Let $p = (p_1, \dots, p_n)$ and S_p be the total number of successes. Then $\Pr[S_p \leq c]$ for $c \leq \lfloor \mu - 2 \rfloor$ is Schur-concave in p . Similarly, $\Pr[S \geq c]$ for $c \geq \lceil \mu + 2 \rceil$ is Schur-concave in p .*

Remark: If we are only interested in comparing p with $(\frac{1}{n}, \dots, \frac{1}{n})$, then it can be extended to $\Pr[S_p \leq c]$ for $c \leq \lfloor \mu - 1 \rfloor$, and $\Pr[S \geq c]$ for $c \geq \lceil \mu + 1 \rceil$, according to [32]. This was also stated in [55, Problem 4.6].

Consider $p = (p_1, \dots, p_\tau)$ where all $p_i = \frac{1}{2^l}$. Clearly p will be majorized by any vector of length τ and sum $\frac{\tau}{2^l}$. Let Y be the sum of τ i.i.d. Bernoulli random variables with expectation $\frac{1}{2^l}$, i.e. Y is a Binomial random variable with parameters τ and $\frac{1}{2^l}$, then from Theorem 6 we get

$$\Pr[X_a > K + \mu\tau] \leq \Pr[Y > K + \mu\tau] = \text{Binotail}(\tau, \frac{1}{2^l}, K + \mu\tau). \quad (16)$$

³In [28], it actually states $\Pr[S \geq c]$ for $c \geq \lfloor \mu + 3 \rfloor$. But $c \geq \lceil \mu + 2 \rceil$ is also correct from the proof.

Table 4: Bound Comparison for $N = 10^6, n = 10^{12}, \mu = \frac{1}{10}, l = 4$

		Hybrid Bound in [82]	Schur Convexity	Geom/D/1
K=150	$\tau = 3000$	1.12×10^{-20}	3.19×10^{-65}	
K=150	Total	1.67×10^{-5}	5.49×10^{-50}	6.47×10^{-53}
K=50	$\tau = 1000$	2.23×10^{-7}	2.89×10^{-23}	
K=50	Total	> 1	2.85×10^{-8}	3.36×10^{-11}

Remark: We can improve upon the above slightly for $\tau > N$, by noting that in (15) there can be at most $N + \lfloor \frac{\tau-N}{2^l} \rfloor$ instead of τ random variables, but this makes little difference numerically.

We will use a concrete example to show how much this bound improves upon the one in [82] using Chernoff bound and second moment bound. Let $N = 10^6, n = 10^{12}, \mu = \frac{1}{10}, l = 4$. The results are listed in Table 4. The first two rows are for $K = 150$. The first row is for the tail bound for $\tau = 3000$, and the second row is for the total overflow probability. The Geom/D/1 column shows the total overflow probability if we treat the arrivals to the queue as independent random variables, which they are not. We believe that this number is close to the actual overflow probability. We can see that the bound using Schur-convexity is drastically better than the bound in [82]. The few orders of magnitude between it and the Geom/D/1 bound is most likely caused by the union bound used in the paper.

In the last two rows we can see that, for $K = 50$ the old bound for overflow probability is larger than 1 (actually 1.24×10^8), however the new bound is in the order of 10^{-8} , so only the new bound can tell us that a queue size of 50 is sufficient for this configuration.

2.3 *DRAM-based Statistics Counter Array Architecture*

In the section we present an algorithm to maintain counters in DRAM while achieving the speed of SRAM. Most of this work has been presented in [78].

2.3.1 Introduction

It is widely accepted that network measurement is essential for the monitoring and control of large networks. For tracking various network statistics (e.g. performing SNMP link counts) and for implementing various network measurement, router management, intrusion detection, traffic engineering, and data streaming applications, there is often the need to maintain very large arrays of statistics counters at wire-speed (e.g. many millions of counters for per-flow measurements [65, 61]). In general, each packet arrival may trigger the updates of multiple per-flow statistics counters, resulting in possibly tens of millions of updates per second. For example, on an 40 Gb/s OC-768 link, a new packet can arrive every 8 ns, and the corresponding counter updates need to be completed within this time. Large counters, such as 64 bits wide, are needed for tracking accurate counts even in short time windows if the measurements take place at high-speed links as smaller counters can quickly overflow. Additionally, a practical counter array solution has to be able to deal with any arbitrary (including adversarial) incoming sequence of counter addresses (i.e., indices) to be incremented because statistics counter arrays may often be used in security applications (e.g., intrusion detection) and/or in settings where an adversary has incentives to compromise their performance guarantees.

While implementing large counter arrays in SRAM can satisfy the performance needs, the amount of SRAM required is often both infeasible and impractical. As reported in [82], real-world Internet traffic traces show that a very large number of flows can occur during a measurement period. For example, an Internet traffic trace from UNC has 13.5 million flows. Assuming 64 bits for each flow counter, 108 MB of SRAM would already be needed for just the counter storage, which is prohibitively expensive. Therefore, researchers have actively sought alternative ways to realize large arrays of statistics counters at wirespeed [65, 61, 64, 82].

Several designs of large counter arrays based on hybrid SRAM/DRAM counter

architectures have been proposed. Their basic idea is to store some lower order bits (e.g. 9 bits) of each counter in SRAM, and all its bits (e.g. 64 bits) in DRAM. The increments are made only to these SRAM counters, and when the values of SRAM counters come close to overflowing, they will be scheduled to be “flushed” back to the corresponding DRAM counter. These schemes all significantly reduce the SRAM cost. In particular, the scheme by Zhao et al. [82] achieves the theoretically minimum SRAM cost of 4 to 5 bits per counters when the SRAM-to-DRAM speed ratio is between 1/10 (4ns/40ns) and 1/20 (3ns/60ns). While this is a substantial reduction over a straightforward SRAM implementation, storing say 4 bits per counter in SRAM for 13.5 million flows would still require nearly 7 MB of SRAM, which is a substantial amount and difficult to implement on-chip. Moreover, since the bounds on SRAM requirements for the hybrid SRAM/DRAM approaches are based on preventing SRAM counter overflows, the SRAM requirements are also dependent on the *size* of the increments. If a wide range of increments is needed, and *large* increments are possible, then the possibility for overflows could occur earlier and more SRAM counter bits would be needed to compensate, resulting in yet larger SRAM requirements. In addition, the existing hybrid SRAM/DRAM approaches do not support arbitrary decrements and are based on an integer number representation, whereas a *floating point number* representation may be needed in some applications [36, 76].

2.3.1.1 DRAM Can Be Plenty Fast

In this section, we challenge the main premise behind previous hybrid SRAM/DRAM architecture proposals. Their main premise is that DRAM access latencies are too slow for wirespeed updates, though DRAMs provide plenty of storage capacity for maintaining exact counts for large arrays of counters. However, our main observation is that modern DRAM architectures have advanced architecture features [34, 44, 74, 73] that can be exploited to make a DRAM solution practical. We then propose a

DRAM-based counter architecture that allows for wirespeed updates to large counter arrays.

Driven by a seemingly insatiable appetite for extremely aggressive memory data rates in graphics, multimedia, video game, and high-definition television applications, the memory semiconductor industry has continually been driving very aggressive roadmaps in terms of ever increasing memory bandwidths that can be provided at commodity pricing (about \$0.01/MB as of this writing). For example, the Cell processor from IBM/Sony/Toshiba [30] uses two 32-bit channels of XDR memories [5] with an aggregated memory bandwidth of 25.6 GB/s. Using an approach called micro-threading [73], the XDR memory architecture provides internally 16 independent banks inside just a *single* DRAM chip, 256 memory banks across 16 DRAM chips that are typically packaged into a single memory module. Next generation memory architectures [6] are expected to achieve a data rate upwards of 16 GB/s on a single 16-bit channel, 64 GB/s on an equivalent dual 32-bit channel interface used by the Cell processor. This enormous amount of memory bandwidth can be shared or time-multiplexed by multiple network functions. The Intel IXP network processor [7] is another example of a state-of-the-art network processor that has multiple high-bandwidth memory channels. Besides XDR, other memory consortia have similar capabilities and advanced architecture features on their roadmaps as well, since they are driven by the same demanding consumer applications. For example, extremely high data efficiency can be achieved using DDR3 memories as well [74].

Although these modern high-speed DRAM offerings provide extraordinary memory bandwidths, the peak access bandwidths are only achievable when memory locations are accessed in a *memory interleaving mode* (to ensure that *internal memory bank conflicts* are avoided). Unlike graphics and video applications with mostly sequential memory access patterns, which are known to be friendly to memory interleaving, the conventional wisdom is that the random (or even adversarial) access

nature of network measurement applications would *render interleaved access modes unusable*. For example, for XDR memories [5], a new memory operation could be initiated every 4 ns when the internal memory banks are interleaved, but a worst-case access latency of 40 ns is required for a read or a write operation if memory bank accesses are unrestricted.

2.3.1.2 Our Approach

Our main idea is to randomly distribute the memory addresses to which consecutive counter indices map across the memory banks so that a near-perfect balancing of memory access loads can be provably achieved, under arbitrary (including adversarial) counter update patterns. In particular, we propose a novel scheme called a *randomized counter architecture* that works as follows. Suppose the SRAM-to-DRAM random access latency ratio is μ (e.g. $\mu = 4\text{ns}/64\text{ns} = 1/16$). The randomized counter scheme works by using $B > 1/\mu$ DRAM banks and randomly distributing the array of counters across these DRAM banks so that when the loads of these memory banks are perfectly balanced, the worst-case load factor of any DRAM bank is $1/B\mu < 1$. In particular, we apply a random permutation function to the counter index to obtain a randomly permuted counter index, which is in turn mapped to a memory bank (according to the traditional memory interleaving/addressing scheme that is in use).

The purpose of this simple randomization scheme is to ensure that the memory load is evenly distributed when different counters are updated over time, *under arbitrary counter update sequences*. Note that an adversary can conceivably overload a memory bank by sending traffic that would trigger the update of the same counter because these counter updates will necessarily be mapped to the same memory bank. However, this case can be easily handled through caching. By caching pending counter update requests, we can ensure that repeated updates to the same counter within a certain time window will not result in any new memory operations. Instead, the

pending counter update request is simply modified to reflect the new counter update request.

While this architecture of randomization plus caching sounds simple and straightforward, it is not trivial to derive the performance guarantee of this scheme. We are able to show that index randomization combined with a reasonably sized cache can handle with overwhelming probability arbitrary (including adversarial) counter update patterns without having overload situations as reflected by long queuing delays (to be made precise in Section 2.3.4). For example, we show that only very small queues (say on the order of $K = 45$ entries per request queue) are required to ensure a negligible overflow probability (say under 10^{-14}). We use the technique in Section 2.1 to establish this result.

Compared to existing hybrid SRAM/DRAM counter architectures [65, 61, 64, 82], our DRAM-based solution offers three clear advantages. First, our solution can achieve the same update speeds to counters, without the need for a non-trivial amount of SRAM for storing partial counts. Second, our solution can easily accommodate increments/decrements of any arbitrary integer (needed for counting bytes) or floating point values (needed in certain data streaming applications [36, 76]), while hybrid SRAM/DRAM counter architectures typically can only accommodate "increment by 1" efficiently. Finally, as we shall show in Section 2.3.4, our DRAM-based solution requires only a small amount of "control" SRAM, the size of which is *independent* of the number of counters being maintained. Therefore, our approach is scalable to future application scenarios in which much larger counter arrays are conceivable (possibly hundreds of millions of counters). This is in contrast to hybrid SRAM/DRAM architectures where the SRAM requirement grows linearly with the number of counters being maintained. Our solution only grows linearly in the DRAM requirement with respect to the number of counters, which is practical given low cost of DRAM⁴.

⁴As of this writing, 2GB of DRAM costs under \$20, over 100MB/\$.

Section 2.3.2 outlines additional related work. Section 2.3.3 describes our proposed randomized counter architecture in details. Section 2.3.4 provides a rigorous analysis on the performance of our architecture in the worst-case. Section 2.3.5 presents an evaluation of our proposed architecture.

2.3.2 Related Work

In this section, we outline prior work related to our problem. As already discussed in Section 2.3.1, the naive approach of storing full counters in SRAM is prohibitively expensive. Although a hybrid SRAM/DRAM architecture [65, 61, 64, 82] significantly reduces the SRAM requirement, the amount of SRAM required for tracking a large number of counters (say in the tens of millions) is still substantial and difficult to implement on-chip.

Besides hybrid SRAM/DRAM architectures, several complementary SRAM-based approaches have also been proposed that aim to make feasible the storage of large counter arrays in SRAM through efficient representations. One category of approaches is approximate counting [53, 21, 69], which are all based on the basic idea invented by Morris [53]. The idea is to probabilistically increment a counter based on the current counter value. This approach is applicable to those network measurement and data streaming applications that can tolerate the inaccuracies.

A second approach, which was recently proposed, is a counter architecture called counter braids [49], which was inspired by the construction of low-density parity-check codes and can keep track of exact counts of all flows without remembering the association between flows and counters⁵. At each packet arrival, counter increments can be performed quickly by hashing the flow label to several counters and incrementing them. The counter values can be viewed as a linear transformation of flow counts,

⁵Counter braids consider a more general problem that also addresses flow association.

where the transformation matrix is the result of hashing all flow labels during a measurement epoch. Flow counts can be decoded through an iterative decoding process at the end of the measurement epoch.

A third approach, which was also recently proposed, is based on an efficient variable-length counter representation called BRICK [35]. It uses a simple operator called rank-indexing to link together counter segments rather than using expensive memory pointers. This counter architecture has the advantage that it can support “active” counter applications in which individual counter values need to be retrieved at wirespeed. For such applications, this approach provides a much more efficient representation than a naive SRAM implementation.

In all three above SRAM-based approaches, significant amounts of SRAM are still necessary for very large counter arrays (say for tens of millions of counters). In contrast, our proposed solution stores all counters only in DRAM. We believe these approaches are complementary as they have different design tradeoffs.

Finally, the idea of using DRAM interleaving to implement large counter arrays was first proposed in [43]. We extend that work considerably in this paper by presenting a new mathematical framework for analyzing the behavior of such architectures under general practical conditions, as detailed in Section 2.3.4. We also introduced a cache module in the architecture to combat adversarial counter update patterns, which adds considerably to the difficulty of our analysis.

2.3.3 Our Scheme

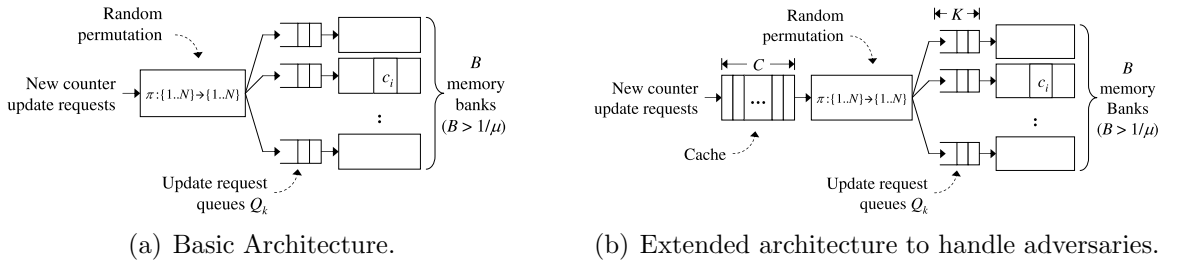


Figure 4: Memory architecture for a randomized DRAM-based counter scheme.

Memory interleaving has been successfully used in the past for improving the performance of computer systems [44, 60, 62], for graphics or video intensive applications [73], and for implementing routing functions like high-performance packet buffers [67]. In this section, we describe how this technique can be employed for statistics counting.

Figure 4(a) depicts a simplified basic version of our randomized counter architecture. Given a SRAM-to-DRAM random access latency ratio of μ (e.g. $\mu = 4\text{ns}/64\text{ns} = 1/16$), we use $B > 1/\mu$ memory banks to store the counters. The basic idea is to randomly distribute the counters evenly across the B memory banks so that with high probability each memory bank will receive about one out of B counter updates to it on average. This is achieved by applying a pseudo-random permutation function $\pi : \{1, \dots, N\} \rightarrow \{1, \dots, N\}$ to a counter index to obtain a permuted index. We then use a simple location scheme where counter c_i will be stored in the k^{th} memory bank, where $k = \pi(i) \bmod B$, at address location $a = \lfloor \pi(i)/B \rfloor$.

At each memory bank k , we maintain a small update request queue Q_k of pending update requests. To update counter c_i that is stored in the k^{th} memory bank, an update request is *inserted* into Q_k . These request queues are then conceptually serviced concurrently. The actual *read* and *write* operations are serviced alternately at the time slot level. In particular, at each memory bank k , suppose a read operation is initiated at time s for the counter request at the head of Q_k . The data will be available $1/\mu$ time slots later. Then a write operation for the incremented counter value can be initiated at time $s + 1/\mu$, which will be finished by time $s + 2/\mu$. Like in [82], we define a *cycle* as two time slots, the equivalent time for reading from SRAM (one time slot) and for writing back to SRAM (another time slot). Although an update would actually take $1/\mu$ cycles ($2/\mu$ time slots) to complete for performing both a DRAM read as well as a DRAM write, our design can effectively keep count of new packet arrivals as long as $B > 1/\mu$, which enables wirespeed throughput. That is, by

randomly load-balancing incoming counter updates to B request queues, the arrival rate of new requests to each request queue is just once every B cycles, but the request queues are serviced at the faster rate of once every $1/\mu$ cycles.

Counter index permutation in our scheme makes it difficult for an adversary to purposely trigger a large number of consecutive counter updates to the same memory bank with updates to distinct counters since the pseudo-random permutation function (or the key it uses) is *not* known to the outside world. An adversary can only try to trigger consecutive counter updates to the same counter, which would result in consecutive accesses to the same memory bank. To safeguard our scheme against this adversarial situation, we add a fully associative cache module (say containing C cache entries) to our architecture to absorb such repetitions, as shown in Figure 4(b). This cache employs a FIFO replacement policy because (1) only FIFO allows us to study the worst-case performance of this system analytically and (2) it has long been proved in the adversarial paging literature that fancy policies such as LRU will not outperform FIFO under adversarial conditions [55, Chapter 13]. With the addition of the cache module, we can catch repeated updates to the same counter within a sliding window of C cycles. That is, if a new counter update request arrives for counter c_i , we can lookup the cache to see if there is already a pending update request to this counter. If there is, then we can just simply modify that request rather than creating a new one (e.g. change the request from “+1” to “+2”). Since these two updates will result in only one (instead of two) eventual DRAM access (read and write), there is no incentive (toward degrading the performance of our system) for an adversary to access the same counter repeatedly within a sliding window of C cycles.

In the next section (Section 2.3.4), we present a detailed theoretical analysis for all counter update sequences for the architecture depicted in Figure 4(b).

2.3.4 Analysis

In this section, we prove the main theoretical result which bounds the probability of having a long queueing delay at any aforementioned update request queue Q_k (associated with the k_{th} DRAM bank) for all (including any adversarial) counter update sequences. As explained earlier, this worst-case large deviation result is proven using the technique in Section 2.1.

We introduced in the previous section the concept of a *cycle*, which consists of an SRAM read time slot and an SRAM write time slot. Throughout the following analysis, we will use cycle as our basic unit of time. As explained in the previous section, we assume the system is continuously 100% loaded – that is, there is one incoming counter update every cycle. We refer to this worst-case workload as an arrival rate of 1. We can use Theorem 1 to show that this assumption indeed represents the worst-case, in the sense that the probability bounds derived for this case will be no better than allowing certain cycles to be idle.

The rest of this section is organized as follows. In Section 2.3.4.1, we describe the overall structure of the tail bound problem, which shows that the overall overflow event \tilde{D} over time period $[0, n]$ is the union of a set of the overflow events $D_{s,t}$, $0 \leq s < t \leq n$, which leads to a union bound. In Section 2.3.4.2, we show how to bound each individual event $D_{s,t}$ by establishing the worst-case number of update requests $X_{s,t}$ during time interval $[s, t]$, in terms of convex order. In Section 2.3.4.3 we bound $X_{s,t}$ with the sum of i.i.d. random variable which can be easily computed.

2.3.4.1 Union bound – the first step

In this section we bound the probability of overflowing a request queue Q . Let $\tilde{D}_{0,n}$ be the event that one or more requests are dropped because Q is full during time interval $[0, n]$ (in units of cycles). This bound will be established as a function of system parameters K , B , μ , and C . Recall that K is the size of each request queue,

B is the number of DRAM banks, μ is the SRAM-to-DRAM random access latency ratio, C is the size of the cache.

In the following, we shall fix n and will therefore shorten $\tilde{D}_{0,n}$ to \tilde{D} . Note that $\Pr[\tilde{D}]$ is the overflow probability for just one out of B such queues. The overall overflow probability can be bounded by $B \times \Pr[\tilde{D}]$ (union bound).

We first show that $\Pr[\tilde{D}]$ is bounded by the summation of probabilities $\Pr[D_{s,t}]$, $0 \leq s \leq t \leq n$, that is,

$$\Pr[\tilde{D}] \leq \sum_{0 \leq s \leq t \leq n} \Pr[D_{s,t}]. \quad (17)$$

Here $D_{s,t}$, $0 \leq s < t \leq n$, represents the event that the number of arrivals during the time interval $[s, t]$ is larger than the maximum possible number of departures in the queue, by more than the queue size K . Formally letting $X_{s,t}$ denote the number of update requests (to the DRAM bank) generated during time interval $[s, t)$, then we have

$$D_{s,t} \equiv \{\omega \in \Omega : X_{s,t} - \mu(t - s) > K\}.$$

Here we will say a few words about the implicit probability space Ω , which is the set of all permutations on $\{1, \dots, N\}$. Since we are considering the worst case bound, we assume that for each cycle there is a request dequeued from the cache, thus creating an arrival for one of the request queues for DRAM banks. We assume that the requests dequeued follow arbitrary pattern, with the only restriction that the same requested address can not repeat within C cycles. This is due to the “smoothing” effect of the cache, i.e. repetitions within C cycles would be absorbed by the cache. Given an arbitrary dequeued pattern satisfying the above restriction, then each instance $\omega \in \Omega$ gives us an arrival sequence to the queues of the DRAM bank.

The inequality (17) is a direct consequence (through the union bound) of the following lemma, which states that if the event \tilde{D} happens, at least one of the events $\{D_{s,t}\}_{0 \leq s < t \leq n}$ must happen, and vice versa.

Lemma 7. $\tilde{D} = \bigcup_{0 \leq s \leq t \leq n} D_{s,t}$

Proof. Given an outcome $\omega \in \tilde{D}$, suppose an overflow happens at time z . The queue is clearly in the middle of a busy period at time z . Now suppose this busy period starts at y . Then the number of departures from y to z is equal to $\lfloor \mu(z - y) \rfloor$. Since an update request happens at time z to find the queue of size K full, $X_{y,z}$, the total number of arrivals during time $[y, z]$ is at least $K + 1 + \lfloor \mu(z - y) \rfloor \geq K + \mu(z - y)$. In other words, $D_{y,z}$ happens and $\omega \in D_{y,z}$. This means for any outcome ω in the probability space, if $\omega \in \tilde{D}$, then $\omega \in D_{s,t}$ for some $0 \leq s < t \leq n$.

On the other hand, given an outcome $\omega \in D_{s,t}$ for some s, t , then obviously the queue will overflow at t or earlier, so $\omega \in \tilde{D}$. \square

Remark: A similar lemma is proved in [82, Lemma 1]. Here we point out the stronger relationship of equivalence.

2.3.4.2 Bounding individual $\Pr[D_{s,t}]$

In this subsection we find the worst-case update request sequence for deriving tail bounds for individual $\Pr[D_{s,t}]$ terms. Recall that $D_{s,t}$ is the event that the number of arrivals during the time interval $[s, t]$, denoted as $X_{s,t}$, is larger than the maximum possible number of departures in the queue, by more than the queue size K . The probability $\Pr[D_{s,t}]$ is clearly a (random) function of the sequence of update requests (viewed as parameters) during the interval $[s, t]$. Fixing any arbitrary sequence, it is not hard to bound $\Pr[D_{s,t}]$ using Chernoff type of techniques, as $X_{s,t}$ can be bound by the sum of independent random variables, in the convex order, using the techniques in Section 2.3.4.3. However, as mentioned before, it is not possible to enumerate over all possible parameter settings (i.e., sequences) to find the worst-case $\Pr[D_{s,t}]$ bound. Fortunately, convex ordering comes to our rescue by allowing us to analytically bound the moment generating function (MGF) of $X_{s,t}$ under all parameter settings by that under a worst-case setting. For simplicity, in this section we will drop the subscripts

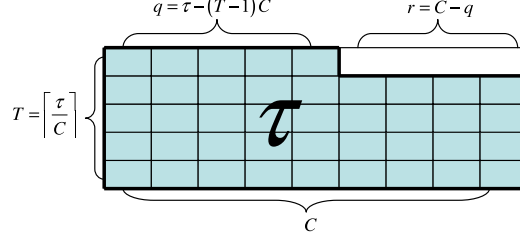


Figure 5: Relationship among q , r , T and τ

of $X_{s,t}$ and use X instead.

We now specify the worst-case update request sequence (in the aforementioned sense of convex ordering) and prove it is indeed the worst-case.

Let $X_i, 1 \leq i \leq N$ be the indicator random variable for whether the i^{th} address is mapped to the DRAM bank under consideration. We have $E[X_i] = \frac{1}{B}$. Thanks to the random counter index permutation scheme, we can view X_i 's as a result of sampling without replacement from N values, $\frac{N}{B}$ of which are 1 and the rest are 0. Therefore the X_i 's are exchangeable random variables, though they are not independent.

Let $a_i, 1 \leq i \leq N$ be the count of the number of appearances of the i^{th} address during time interval $[s, t]$. Then $X = \sum_{i=1}^N a_i X_i$.

Due to caching, the dequeued requests to the same DRAM address should not repeat within any sliding window of C cycles. Therefore none of the counts a_1, \dots, a_N can exceed T , where $T = \lceil \frac{\tau}{C} \rceil$. Moreover, let $q = \tau - (T-1)C$ and $r = C - q$. Then only the first q requests could repeat with count T . Figure 5 will help readers understand the relationship among q , r , T as functions of τ .

We call any vector $a = \{a_1, \dots, a_N\}$ a valid splitting pattern of τ , if $0 \leq a_i \leq T$, $\sum_{i=1}^N a_i = \tau$, and $|\{i : a_i = T\}| \leq q$. Let \mathcal{A} be the set of all valid splitting patterns.⁶ Let X_a denote $\sum_{i=1}^N a_i X_i$.

We are ready to specify the family of worst-case counter update sequences. A

⁶It is possible to prove that for every valid splitting pattern, there is a possible counter update sequence matching the pattern. However this is not essential for our analysis.

worst-case counter update sequence takes the following form: first come $q + r (= C)$ update requests for distinct counter indices i_1, i_2, \dots, i_{q+r} , and they then repeat for $T-1$ times in total, followed finally by q update requests for counter indices i_1, i_2, \dots, i_q . In other words, inside this window of τ cycles, counters i_1, i_2, \dots, i_q (q of them) are accessed T times and counters i_{q+1}, \dots, i_{q+r} (r of them) are accessed $T-1$ times. In the following Theorem 8, we will see that this arrival sequence is indeed the worst-case in the sense of convex ordering.

Let a^* be the aforementioned pattern for one of such counter update sequences, i.e. let $a_1^* = \dots = a_q^* = T, a_{q+1}^* = \dots = a_{q+r}^* = T-1, a_{q+r+1}^* = \dots = a_N^* = 0$.

We now have the main theorem of this section:

Theorem 8. *a^* is the worst case splitting pattern in terms of convex ordering, i.e. $X_a \leq_{cx} X_{a^*}, \forall m \in \mathcal{M}$.*

Proof. Let $a_{[1]}, \dots, a_{[N]}$ denote the components of a in decreasing order. a^* is already in decreasing order. Because a is a valid splitting pattern, we have

$$\begin{aligned} a_{[i]} &\leq T = a_i^* \quad , \quad \text{for } 1 \leq i \leq q \\ a_{[i]} &\leq T-1 = a_i^* \quad , \quad \text{for } q+1 \leq i \leq q+r. \end{aligned}$$

Therefore (3) is true for $1 \leq k \leq q+r$. Since $\sum_{i=1}^{q+r} a_i^* = \tau = \sum_{i=1}^N a_i$, (3) is true for $i > q+r$ as well. Therefore by definition $m \leq_M m^*$.

The theorem follows from Corollary 3 because X_1, \dots, X_n are exchangeable. \square

Remark: Note that stochastic order does not hold here in general, since $E[X_a] = E[X_{a^*}] = \tau/B, \forall a \in \mathcal{A}$. For stochastic order to hold between two random variables of different distributions, their expectations must differ [56, Theorem 1.2.9].

Unfortunately, it is in general not possible to apply the Chernoff bound directly to the MGF of X_{a^*} . For $\tau \leq C$, X_{a^*} is a hypergeometric random variable whose MGF $E[e^{X_{a^*}\theta}]$ is a hypergeometric series [29] that is expensive to compute. For $\tau > C$, X_{a^*}

is a weighted sum of two hypergeometric random variables that are not independent of each other, so its MGF is prohibitively expensive to compute. The next section is devoted to dealing with this problem.

2.3.4.3 Relaxations of X_{a^*} for computational purposes

Our remaining task is to find a way to upper-bound $E[e^{X_{a^*}\theta}]$ by a more computationally friendly formula, to which the aforementioned Chernoff technique can be applied.

We consider the two cases $\tau \leq C$ (i.e., the measurement window τ , in number of cycles, is no larger than the cache size, in number of entries) and $\tau > C$ separately.

When $\tau \leq C$, $X_{m^*} = X_1 + \dots + X_\tau$. Let the population S consist of c_1, \dots, c_N such that $\frac{N}{B}$ of c_i 's are of value 1 and the rest of them are of value 0. If we let $n = \tau$, then X in Theorem 4 has the same distribution as our X_{a^*} , and Y_i 's there are i.i.d Bernoulli random variables with probability $\frac{1}{B}$. Because $f(x) = e^{x\theta}$ is a convex function of x , from $X_a \leq_{cx} X_{a^*} \leq_{cx} Y$ we get

$$\begin{aligned} E[e^{X_a\theta}] &\leq E[e^{X_{a^*}\theta}] \leq E[e^{Y\theta}] \\ &= E[e^{(Y_1 + \dots + Y_\tau)\theta}] \\ &= E[e^{Y_1\theta}]^\tau \\ &= \left(\frac{1}{B}e^\theta + \left(1 - \frac{1}{B}\right)\right)^\tau. \end{aligned}$$

For $\tau > C$, we have $X_{m^*} = T(X_1 + \dots + X_q) + (T-1)(X_{s+1} + \dots + X_{q+r})$. We can similarly apply our new Theorem 5 and get

$$\begin{aligned} E[e^{X_a\theta}] &\leq E[e^{X_{a^*}\theta}] \leq E[e^{Y\theta}] \\ &= E[e^{(T(Y_1 + \dots + Y_q) + (T-1)(Y_{q+1} + \dots + Y_{q+r}))\theta}] \\ &= E[e^{TY_1\theta}]^q E[e^{(T-1)Y_1\theta}]^r \\ &= \left(\frac{1}{B}e^{T\theta} + \left(1 - \frac{1}{B}\right)\right)^q \left(\frac{1}{B}e^{(T-1)\theta} + \left(1 - \frac{1}{B}\right)\right)^r. \end{aligned}$$

By (2), we now have the following bound:

Theorem 9.

$$\begin{aligned}\Pr[D_{s,t}] &\leq \min_{\theta>0} \frac{(\frac{1}{B}e^\theta + (1 - \frac{1}{B}))^\tau}{e^{(K+\mu\tau)\theta}}, \quad \tau \leq C \\ \Pr[D_{s,t}] &\leq \min_{\theta>0} \frac{(\frac{1}{B}e^{T\theta} + (1 - \frac{1}{B}))^q (\frac{1}{B}e^{(T-1)\theta} + (1 - \frac{1}{B}))^r}{e^{(K+\mu\tau)\theta}}, \quad \tau > C.\end{aligned}$$

Remark: For $\tau \leq C$ we can get away with Theorem 4 alone by taking a different viewpoint. Instead of selecting $C = q + r$ permutation destinations for the addresses and see which ones are among the $\frac{N}{B}$ locations in the DRAM bank, we can treat it as selecting $\frac{N}{B}$ permutation sources, and see which ones are among the addresses that have requested update. So let the population S' be exactly the components of a^* , i.e. let $c_i = a^*_i$. So there are q of c_i 's of value T , r of c_i 's of value $T - 1$, and the rest of them of value 0. Thus $X_{a^*} = X'_1 + \dots + X'_{\frac{N}{B}}$, where X'_i 's are a random sample without replacement from S' . By Theorem 4 we have $X_{a^*} \leq_{cx} Y' = \sum_{i=1}^{N/B} Y'_i$, where Y'_i 's are a random sample with replacement from S' . Thus the Y'_i 's are i.i.d. random variable with

$$\Pr[Y'_i = y] = \begin{cases} \frac{q}{N} & y = T \\ \frac{r}{N} & y = T - 1 \\ \frac{N-q-r}{N} & y = 0 \end{cases}.$$

So similar to the $\tau \leq C$ case, we can derive the following bound:

$$\begin{aligned}\mathbb{E}[e^{X_m\theta}] &\leq \mathbb{E}[e^{X_{m^*}\theta}] \leq \mathbb{E}[e^{Y'\theta}] \\ &= \mathbb{E}[e^{(Y'_1 + \dots + Y'_{\frac{N}{B}})\theta}] \\ &= \mathbb{E}[e^{Y'_1\theta}]^{\frac{N}{B}} \\ &= \left(\frac{q}{N}e^{T\theta} + \frac{r}{N}e^{(T-1)\theta} + \frac{N-q-r}{N} \right)^{\frac{N}{B}} \\ &= \left(1 + \frac{qe^{T\theta} + re^{(T-1)\theta} - q - r}{N} \right)^{\frac{N}{B}} \\ &\leq e^{(qe^{T\theta} + re^{(T-1)\theta} - q - r)/B}.\end{aligned}$$

For the last inequality we used the inequality $(1 + \frac{a}{n})^n < e^a$. By (2), we now have the

following bound (it's not hard to verify that it also applies to $\tau \leq C$) :

$$\Pr[D_{s,t}] \leq \min_{\theta > 0} e^{(qe^{T\theta} + re^{(T-1)\theta} - q - r)/B - (K + \mu\tau)\theta}.$$

However the bound in Theorem 9 are better numerically in our setting.

We also see that the bound is translation invariant, i.e. it only depends on $\tau = t - s$. Therefore, the computation cost of (17) is $O(n)$ instead of $O(n^2)$, where n is the number of cycles during a network measurement interval.

In conclusion, we have established bound for overflow probability for worst-case counter update request sequences. The bound can be computed though $O(n)$ number of numerical minimizations for one-dimensional functions expressed in Theorem 9.

2.3.5 Evaluation

In this section, we present evaluation results for our proposed randomized DRAM-based counter array architecture described in Section 2.3.3. The outline of this section is as follows: Section 2.3.5.1 describes a concrete instantiation of our proposed solution. Section 2.3.5.2 outlines the parameters of two real-world Internet traffic traces that we used for our evaluations. Section 2.3.5.3 presents numerical results derived using the analytical models presented in Section 2.3.4. Finally, Section 2.3.5.4 provides a comparison of our new approach with the state-of-the-art hybrid SRAM/DRAM approach [82].

2.3.5.1 Implementation details

To provide a general formulation, we had assumed in Section 2.3.3 that the request queues are conceptually serviced concurrently. This could be realized by using B separate parallel memory channels to B separate sets of memories. However, this is unnecessarily expensive as modern DRAM architectures already provide a plentiful number of internal memory banks, and a single memory channel can be used to pipeline memory transactions across them at peak rates when performed in an

interleaving manner.

To provide a concrete analysis of our proposed solution, we use the specification of an actual commercial high-bandwidth memory part, namely the XDR memory from Rambus [73, 5]. As depicted in Figure 6, each XDR memory chip contains 16 internal memory banks that can be interleaved to achieve high-bandwidth memory access. Although the XDR memory has a worst-case access latency of 40 ns for a read or a write operation, a new read or write transaction could be initiated every 4 ns if it is initiated to a different memory bank. For an OC-768 link at 40 Gb/s, a new minimum size (40 bytes) packet can arrive every 8 ns. To support a counter update on every packet arrival, about 4 ns is available for a memory read or a memory write. Fortunately, the XDR memory can support this rate of new memory operations.

In particular, one concrete implementation is to use $B = 32$ memory banks and two memory channels, with 16 banks on each memory channel. For each memory channel, we can service its memory banks in round-robin order. Therefore, a new memory operation can be serviced once every 16 cycles. With both memory channels operating in parallel, each of the $B = 32$ memory banks can indeed be serviced deterministically once every 16 cycles. Fortunately, processors with dual memory channels are becoming increasingly common. For example, both the Cell processor from IBM/Sony/Toshiba [30] and the latest mainstream Intel x86 multi-core processor [3] have built-in dual-channel memory controllers. This configuration corresponds to setting $\mu = 1/16$ in our analysis, and it can handle any SRAM-to-DRAM speed ratio that is no smaller than $1/16$. For the remainder of this section (Section 2.3.5), we will use the configuration $\mu = 1/16$ and $B = 32$ in our analysis and comparisons.

2.3.5.2 Traffic traces

For our evaluations, we used parameters derived from two real-world Internet traffic traces. In particular, the traces that we used were collected at different locations

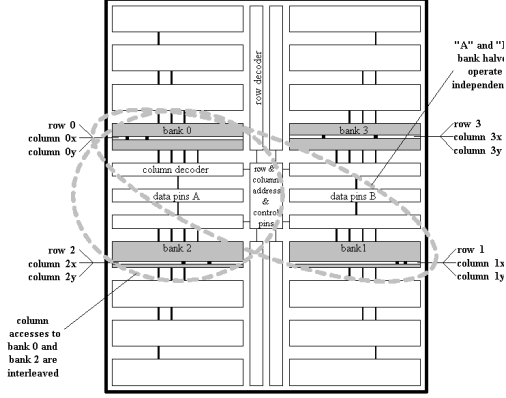


Figure 6: XDR memory chip architecture

in the Internet, namely University of Southern California (USC) and University of North Carolina (UNC), respectively. The trace from USC was collected at their Los Nettos tracing facility on February 2, 2004, and the trace from UNC was collected on a 1 Gbps access link connecting the campus to the rest of the Internet on April 24, 2003. The trace from USC has 120.8 million packets and around 8.6 million flows, and the trace segment from UNC has 198.9 million packets and around 13.5 million flows. To support sufficient counters for both traces, we set the counter array configuration to support $N = 16$ million counters.

2.3.5.3 Numerical examples of the tail bounds

In this section, we present the numerical results computed from the formulae derived in Section 2.3.4 using MATLAB 7.0. We use $n = 10^{10}$ for all the following examples, where n is the total number of cycles for the measurement period.⁷

In Figure 7, the overflow probability bounds with different cache size C as a function of queue length K are presented, where $\mu = 1/16$ and $B = 32$. It is easy to see from this graph that as K increases, the overflow probability bound decreases. However, after K reaches certain thresholds (depending on C), the overflow

⁷In [78] evaluation section, the results apply to $n = 10^{10}$. The value $n = 10^8$ was a typo.

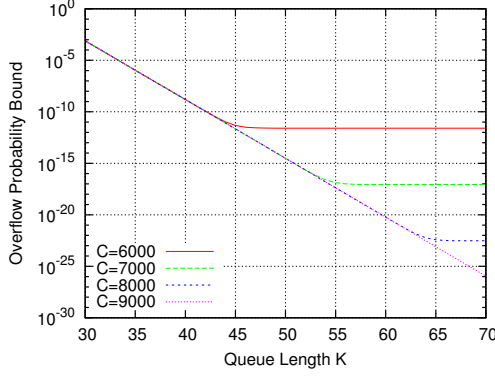


Figure 7: Overflow probability bound as a function of queue size K

Table 5: Comparison of different schemes for 16 million 64-bit counters

	Naive	SRAM/DRAM [82]	Ours
Counter DRAM	None	128 MB DRAM	128 MB DRAM
Counter SRAM	128 MB SRAM	8 MB SRAM	None
Control	None	1.5 KB SRAM	25 KB CAM 5.5 KB SRAM

probability bound stays practically flat. Our bounds at the flat level are better than the ones in [78] thanks to our Theorem 9 which took advantage of Theorem 5.

The flat level depends on C . Actually, as C approaches infinity, the overflow probability as a function of queue length K becomes the result of the overflow probability of a Geom/D/1 incoming traffic with arrival probability $1/B = 1/32$ to a queue of length K . As shown in Figure 7, when $C \geq 8000$, the overflow probability bound has only negligible decreases as cache size C increases. In other words, by increasing the size of the cache, the performance of the system will not be improved accordingly. Therefore, $C = 8000$ is enough for practical purposes in achieving an overflow probability bound of 10^{-14} .

2.3.5.4 Cost-benefit comparison

Table 5 compares our proposed approach with a naive SRAM approach as well as the hybrid SRAM/DRAM counter architecture approaches [65, 61, 64, 82]. The

naive SRAM approach simply implements all counters in SRAM. For the hybrid SRAM/DRAM approach, we specifically compare against the state-of-the scheme proposed in [82] that provably achieves the minimum SRAM requirement for this architecture class. As demonstrated in [82], in one example setting their approach requires almost a factor of six times less SRAM than the first hybrid SRAM/DRAM solution proposed in [65] and more than a factor of two times less SRAM than an improved solution proposed in [61]. For a SRAM-to-DRAM latency ratio of $\mu = 1/12$, the architecture in [82] requires $w = \lceil \log 1/\mu \rceil = \lceil \log 12 \rceil = 4$ bits SRAM bits per counter. In addition, it needs a very small amount of SRAM to maintain a “flush request queue”, on the order of about 500 entries to ensure negligible overflow probabilities. We will compare it with our scheme in Section 2.3.5.1, since that scheme can handle $\mu = 1/12 \geq 1/16$.

For 16 million counters, a naive implementation would require 128 MB of SRAM, which is clearly far too expensive. For the scheme by in [82], it just requires 1.5 KB of control SRAM to implement a flush request queue with 500 entries. The size of each entry is $\lceil \log 16 \text{ million} \rceil = 24$ bits (3 bytes) to encode the counter index. However, even though the scheme requires just 4 bits per counter to store the partial increments, 8 MB of counter SRAM is required for 16 million counters. This is a substantial amount and difficult to implement on-chip. Moreover, this counter SRAM requirement grows linearly with the number of counters, making it difficult to support faster links or longer measurement periods where more counters would be needed.

For our proposed solution, a small update request queue needs to be maintained at each memory bank. As shown in Figure 7, when we use $B = 32$ memory banks and cache size of $C = 7000$, $K = 50$ entries is sufficient for each update request queue to ensure a queue overflow probability bound of 10^{-14} , for a total of $M = B \cdot K = 1600$ entries. Each entry requires 3 bytes to encode the counter indices of 16 million counters and 4 bits for accumulated counts, resulting in a total of about 5.5 KB of

SRAM to implement these update request queues. Since these update request queues are statically sized, they can be simply implemented as an array.

In addition, as shown in Figure 4(b) in Section 2.3.3, our randomized counter architecture also maintains a small cache to keep track of pending update requests. This cache can be implemented as a fully-associative cache using a content-addressable memory (CAM) with a FIFO replacement policy. For $C = 7000$, we need a CAM with 25 KB in size to support 3 bytes encoding of counter indices and 4 bits for accumulated counts⁸. Although our comparisons here are for integer counters and increments of one only, we emphasize that our general scheme supports increments and decrements of arbitrary amounts, and other number representations such as floating point numbers.

2.4 *Detecting Global Icebergs*

In the section we present an algorithm to detect global icebergs. Most of this work has been presented in [77].

2.4.1 Introduction

Today’s Internet applications often generate and collect a massive amount of data at many distributed locations. For example, an ISP (Internet Service Provider) security monitoring application may require that packet traces be collected at hundreds (or even thousands) of ingress and egress routers, and the amount of data collected at each router can be in the order of several terabytes. From time to time, various types of queries need to be performed over the union of these data sets. For example, in this ISP security monitoring application, we may need to query the union of packet trace data sets at all ingress and egress points to look for globally frequent signatures that may correspond to certain Internet worms. Given the gigantic and evolving

⁸An accumulation counter of 4 bits can absorb 16 increments to the same counter into one update request, which can be serviced in the time of 16 increments, thus offering an adversary no advantage in repeatedly hitting the same counter within a sliding window of C cycles.

nature of these physically distributed data sets, it is usually infeasible to ship all the data to a single location for centralized query processing due to the prohibitively high communication cost. Another scenario is that, in a sensor network, constraints on power consumption limit the amount of data that each sensor can transmit to a central server. Therefore, how to execute various types of (approximate) queries over the union of distributed data sets without physically merging them together has received considerable research attention recently.

One such distributed query problem that has been studied extensively is to detect global heavy-hitters or *icebergs*, which are data elements whose aggregate frequency across all these data sets exceed a pre-specified threshold. The hardness of this problem arises from the fact that a global iceberg may be finely distributed across all the measurement points so that it does not appear large at any one location. For example, in security scenarios an adversary may conceal the presence of the iceberg by spreading it thinly across many different nodes. This precludes the possibility of using a naive algorithm that simply reports locally frequent elements. On the other hand, it would be prohibitively expensive for every node to send records for every small fragment to the central server.

We propose a solution with the salient property that it is unaffected by the manner in which the data is distributed across the local nodes. To attain this property we use *summable sketches* that can be computed locally and later summed at a central location to answer queries on the aggregated data. Due to the nature of these sketches, it does not matter how the data was distributed among the nodes, or even in what order the data is aggregated, making the performance of our solution dependent solely on the aggregated data (and independent of how it is split among nodes). Also, the performance guarantee of these sketches is independent of the number of elements inserted into them, making them ideal for this problem.

Now, one longstanding issue with the iceberg detection problem is that it is notoriously difficult to handle elements that are close to the iceberg threshold. If there are many non-icebergs near the iceberg size, then it is virtually impossible for any approximate algorithm to distinguish the iceberg from the non-icebergs. A reasonable requirement for a data set to avoid this issue is that there is a gap between the size of icebergs and non-icebergs with only a few elements that fall within this gap. We call this the *sparsely populated gap* assumption. The analysis of our algorithm makes use of this assumption.

While requiring such a gap between icebergs and non-icebergs sounds restrictive, this assumption is actually quite practical in many real-world applications. This is because many data sources follow a power-law distribution in which the most frequent elements appear many times more often than the average frequency. For example, network data is commonly observed to follow such a heavy-tailed distribution, where extremely large flows are few and far between. It is critical to detect distributed icebergs in such data, e.g., when monitoring for a distributed denial of service attack, no single link may contain sufficient evidence of the attack to raise a flag.

We begin by studying the problem in which there are no elements in the gap. The analysis of our solution takes advantage of this gap. We then show how to reduce the size of the gap by using some additional information about the data. Finally, we discuss the effect of the elements in the sparsely populated gap.

We envision applications of our solution in which a central server is monitoring a large number of distributed nodes for large outliers. Since it is infeasible for all the data to be shipped to the central server, each local node sends a compact summary of its data to the central server in a single round. If the central server detects that there is an iceberg, it may initiate additional rounds of communication to confirm this fact.

Even though we describe our solution using this simple one-tier topology (a server

communicating with many client nodes), we will later show how our solution can be very easily generalized to arbitrary tree topologies with identical communication costs and analytical guarantees. For example, this solution naturally fits the framework of Google MapReduce [22] and the Apache Hadoop architecture [2].

Very often, the data is not found aggregated on the nodes but is presented as a stream of updates (e.g., network packet data). In such cases, it is important to keep the processing requirements of the local algorithm low. Our solution works not only when the data is already locally aggregated (bag case) but also when it appears as a stream.

2.4.1.1 The “sketch” idea of our solution

We approach this problem by making use of *summable sketches* to succinctly encode the data at each local node. A summable sketch is a sketch (i.e., a lossy, succinct representation of a data set or stream) that has the following additional property: the sketch for the union of two data sets $A \cup B$ can be easily computed from the sketches of A and B . In our solution approach, each node computes the sketch of its local data set and ships it to the central server. The server will then in turn “sum up” these sketches to obtain the sketch for the union of the data set, which will be able to detect the global icebergs with high confidence.

The summable sketches we find most useful for our problem are those that compute the *second frequency moment* (i.e., the sum of the squares of the frequencies of all elements) or F_2 of the data aggregated across all the local nodes. The F_2 value is intuitively a good indicator of iceberg existence/nonexistence because of its “squaring effect” that significantly magnifies the skewness of the data (if any). For example, an iceberg item that is 100 times larger than a non-iceberg item contributes $100^2 = 10,000$ times more to the total F_2 value! Conceivably, we could have also used even higher frequency moments (say F_3, F_4, F_5, \dots) to further magnify such skewness.

However, it can be shown that estimating the k^{th} frequency moment ($k > 2$) incurs a minimum communication cost of $\Omega(n^{1-2/k})$ [18], where n is the total number of elements. In sharp contrast, the second frequency moment can be approximated using a sketch with size *independent* of n [9].

While techniques for estimating F_2 have been well-studied, our contribution lies in that (1) we successfully adapt them to the detection of global icebergs in a split-independent manner and (2) we are able to obtain very sharp accuracy bounds using an interesting combination of convex ordering and large deviation techniques.

Because we use summable F_2 sketches (e.g., Alon, Matias, and Szegedy’s tug-of-war sketch [9]), our proposed algorithm has several desirable features that distinguishes it from prior work. First, it has the *split independence* property, i.e., both its performance guarantee and communication overhead are *independent of the way the total frequency of each and every element is split across the nodes*. We will show this is an immediate consequence of F_2 sketches being summable.

Second, since F_2 sketches were designed for streaming updates, our methodology works even when the local nodes have their data streamed to them at very high rates. This makes our algorithm more generally applicable than some previous work (e.g., [81]) that assumes the data is already aggregated without information loss at each local node (so-called “bag case”).

Third, due to the summable nature of the sketch, we can handle arbitrary connected topologies among the nodes and the central server (e.g., flat topology in [81] and hierarchical tree topology in [50]) with the same accuracy and communication overhead guarantees. In other words, our algorithm is “oblivious” to the interconnection topology.

Furthermore, we show that once an iceberg is detected, we can estimate its size approximately with absolutely no additional communication overhead. In the “bag case” we can ascertain the precise size with an additional round of communication.

But we show how to do away with this if we only want an approximate answer, making ours a one-round communication protocol.

The rest of this section is laid out as follows. We describe some related works in Section 2.4.2. In Section 2.4.3 we formally define our problem. We describe our algorithm and the summable sketches upon which it is based in Section 2.4.4. We completely solve a simplified version of our problem, with a gap assumption, in Section 2.4.5 and show how the iceberg size can also be estimated at no additional cost. In Section 2.4.6 we use some additional information about the data to reduce the required magnitude of the gap. We finally discuss how our algorithm can be applied to real data in Section 2.4.7 and highlight some of its useful properties. Our algorithms are evaluated experimentally using Internet flow data in Section 2.4.8.

2.4.2 Background and Related Work

In this section we briefly survey the previous work on the issue of detecting distributed icebergs. The term *iceberg* was introduced by Fang *et al.* [25]. The term “iceberg” for a distributed heavy-hitter comes from the idea that, like icebergs in an ocean, only the tip of an item with gigantic mass can be observed from a single location. Iceberg queries are known to be useful for various applications, including detection of attacks [11], discovery of heavy-hitters in Content Delivery Networks [1], discovery of worms and other anomalies [19], and ensuring SLA compliance [68].

Manjhi *et al.* [50] studied the problem of discovering icebergs in a distributed environment when the nodes are in a multi-level tree topology. Their work differs from ours in that they aim to detect *recently* frequent elements, whereas we consider the problem of detection in a fixed interval. Also, our solution aims solely to detect icebergs, which allows us to discard the identities of the elements when aggregating the streams.

There also has been some work that studies a variation of the problem in which

only the k most frequent items are of interest [13, 58]. Babcock *et al.* [13] studied this “Top- k ” query problem, and their results were extended by Olston *et al.* [58] to support sum and average queries. Their solution has the feature that they assume that an iceberg must appear at some local node with high frequency.

In [81], Zhao *et al.* proposed algorithms for detecting icebergs in distributed data via size-based sampling and summarization of local frequencies using a combination of quantization and Bloom filters. In their analysis, they parameterize their algorithms to give error bounds that are independent of the manner in which the iceberg is split among the local nodes.

Cormode *et al.* [20] recently proposed the problem of functional monitoring, in which local nodes continuously send updates to the central server. The goal is to minimize the amount of information sent by these nodes while still maintaining some global guarantee (e.g., detecting icebergs with high probability). This is a continuous monitoring solution and is hence incomparable with our work.

An important characteristic of our solution is that, no matter how the iceberg is split among the local nodes, the quality of our solution remains unchanged. Whereas [81] designed their scheme to attain the worst-case performance for every distribution of the iceberg across the local nodes, we automatically guarantee the same just by using the summable sketch methodology. In fact, our solution is independent of *any* characteristic of the data other than the aggregate frequency distribution, making our algorithm robust to hidden icebergs.

2.4.3 Problem Definition

Consider a system or network that consists of m distributed nodes (e.g., routers). The data set S_j at node j contains a stream of tuples $\langle element_id, c \rangle$, where $element_id$ is an element identity from a set $\mathcal{U} = \{u_1, u_2, u_3, \dots, u_n\}$ and c is an incremental count. We denote by $c_i = \sum_j \sum_{\langle u_i, c \rangle \in S_j} c$ the frequency of the element u_i when aggregated

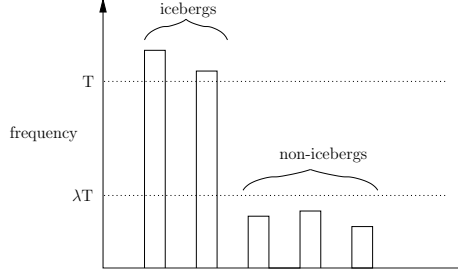


Figure 8: The gap between icebergs and non-icebergs

across all the nodes. We want to detect the presence of elements whose total frequency across all the nodes adds up to exceed a given threshold T . In other words, we would like to find out if there exists an element $u_i \in \mathcal{U}$ such that $c_i \geq T$. We desire our solution to be independent of how the elements are split among the nodes, i.e., our final solution should be dependent on c_1, \dots, c_n , but not on how each c_i is split among the m nodes.

In most iceberg detection scenarios, it is critical to discover the iceberg every time. Hence, we will err on the side of caution by having almost no false negative error even if this means being more permissive to false positive error.

Now, the main issue that we face is that any element that is slightly under the threshold will be nearly indistinguishable from an iceberg. To get around this problem, we will make some simplifying assumptions on the size of non-icebergs and then later demonstrate how these assumptions can be weakened.

The first simplifying assumption that we make is that it is guaranteed that the iceberg is much larger than any non-iceberg. More formally, we say that an element whose aggregate frequency is at least T is an iceberg, and we assume that no element has aggregate frequency in the interval $(\lambda T, T)$, for some $\lambda \in (0, 1)$ (illustrated in Figure 8). This gap parameter λ is independent of n (number of elements) and m (number of nodes).

The gap assumption may be reasonable in certain security scenarios in which

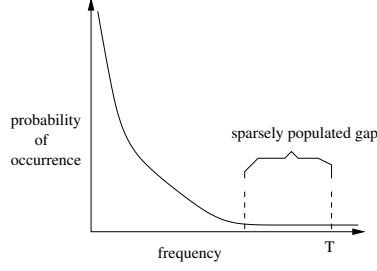


Figure 9: Illustration of the sparse gap for real data sets

massive icebergs are hidden among the many nodes by an adversary. For example, a DDoS attacker may mount an attack that results in the victim receiving hundreds of times more traffic than any other host while spreading this traffic thinly across many different paths to avoid detection. Similarly, a network worm may attempt to avoid detection by spreading very slowly at any single point, even though it has massive aggregate volume.

However, not in all scenarios can we make such a gap assumption. Additionally, even if there is a gap, we may not know how large it is *a priori*. To deal with this issue, we will later weaken this assumption to allow some elements (though not many) to enter this gap. This is reasonable because it is commonly observed in real data that the occurrence of high-frequency items rapidly tails off. We call this a sparsely populated gap (see Figure 9). Our ultimate goal will be to solve the problem of detecting icebergs in real data that exhibits the sparsely populated gap property.

2.4.4 Algorithmic Overview

To solve this problem, we use a *summable sketch*. Summable sketches have the property that we can “sum” the sketches from the individual nodes together to get an aggregate sketch that is identical to the sketch of the aggregate frequencies. This property allows us to guarantee that no matter how the iceberg and non-icebergs are distributed among the nodes, the result of our algorithm will always be the same.

For our solution, we use the sketch for the second frequency moment of the data,

F_2 , which was defined in Section 2.1.

There is typically a gap separating icebergs from non-icebergs in real data. By focusing on the second moment, we magnify this gap to make the difference even easier to detect.

We use F_2 sketches for estimating the second frequency moment for the following reasons:

1. The second moment makes extremal values (i.e., icebergs) stand out distinctly. Intuitively, if we could compute the higher moments (e.g., the tenth frequency moment), then we could further exaggerate this effect. As discussed earlier, however, computing higher frequency moments is much more expensive.
2. We found that some of the existing F_2 sketches for estimating the second moment have the aforementioned summable property. We show in this paper how this property can be exploited for the purpose of iceberg detection.
3. Additionally, the error analysis for these sketches is *independent of the number of elements inserted into it*, which allows us to fix error parameters without a need to account for n , the total number of elements.
4. Finally, the F_2 sketches were designed to be extremely cheap to update. Our solution is viable even if the local nodes process elements as *streams* of updates.

The F_2 sketches we consider enable computation of the second moment with arbitrary precision and confidence. For all $\epsilon, \delta < 1$, these sketches can guarantee an ϵ relative error approximation with probability at least $1 - \delta$ using at most $O(\log(1/\delta)/\epsilon^2)$ counters, which is the asymptotically optimal number [9]. Note that the number of counters necessary is independent of the number of elements that are inserted into the sketch, which is a key property that we need.

PRE-PROCESSING:

Initialize g F_2 sketches S_1, \dots, S_g .
Initialize hash function $h : \mathcal{U} \rightarrow \{1, \dots, g\}$.

ALGORITHM:

for each element/frequency pair $\langle id, count \rangle$ **do**
 Insert id with frequency $count$ into the sketch $S_{h(id)}$.
end for

Algorithm 1: LOCAL SKETCHING ALGORITHM

2.4.4.1 Our Algorithm

Our algorithm works by randomly partitioning all the elements, uniformly at random, into groups and estimating F_2 for each of these groups. Any group with an iceberg in it will stand out from the rest because of its large F_2 . On the other hand, there will be few false positives because non-icebergs are usually much smaller. For example, if the iceberg is ten times larger than most other elements, then one hundred separate non-icebergs would have to fall into a group to make it appear to have an iceberg. We give a more formal description of the algorithm next.

We partition the elements in \mathcal{U} into g groups using a hash function, $h : \mathcal{U} \rightarrow \{1, 2, 3, \dots, g\}$, which is shared by all the nodes. Each node creates a separate F_2 sketch for the elements of each of these groups and updates them over the stream. At the conclusion of the stream (or at regular intervals for infinite streams), each node sends all of its sketches to the central server. See Algorithm 1.

The central server sums the sketches for each of the g groups and obtains an approximation of the second moment for each of these groups. If any group has estimated F_2 over $(1 - \epsilon)T^2$, the algorithm signals that there is an iceberg present. (See Algorithm 2.) For each such group, the central server can poll the nodes for the exact counts for that group. Alternatively, this procedure can be repeated recursively on the suspect group until the iceberg is identified.

Our algorithm has a low false negative rate. The estimate of F_2 for any group with an iceberg in it will be at least $(1 - \epsilon)T^2$ assuming that the F_2 sketch for that

PRE-PROCESSING:

Receive sketches S_1^i, \dots, S_g^i from each local node i .

ALGORITHM:

Sum sketches from each node to create aggregate sketches S_1^*, \dots, S_g^*
for $i := 1$ to g **do**
 Estimate $F_2(S_i^*)$.
 if $F_2(S_i^*) \geq T^2(1 - \epsilon)$ **then**
 Output “There is an iceberg (at least one of $h^{-1}(i)$ is an iceberg).”
 end if
end for

Algorithm 2: CENTRAL AGGREGATION ALGORITHM

group did not err with greater than ϵ relative error—this happens with probability at least $(1 - \delta)$. As a result, we can keep the false negative rate as low as we desire simply by ensuring that the sketches have a suitable small failure rate δ .

In the following section we briefly describe the F_2 sketch of Alon, Matias, and Szegedy [9] and describe how it has all the desirable properties that we require.

2.4.4.2 The Tug-of-War Sketch

As part of our solution, we make use of the Tug-of-War Sketch Algorithm, introduced by Alon, Matias, and Szegedy [9]. The tug-of-war sketch is a means of summarizing frequency data in a stream so that the second moment of the frequencies can be computed efficiently from it. This sketch allows for arbitrary updates (i.e., we may increment the frequency of an element by an arbitrary integer) and is very fast to update.

The tug-of-war sketch enables computation of the second moment with arbitrary precision and confidence, i.e., for all $\epsilon, \delta < 1$, the sketch can guarantee an ϵ relative error approximation with probability at least $1 - \delta$ using at most $O(\log(1/\delta)/\epsilon^2)$ counters. Below, we will briefly describe how it works.

The tug-of-war sketch computes $z = 32 \log(1/\delta)/\epsilon^2$ unbiased estimates for the second moment as follows. Each estimate is the linear projection of the frequencies multiplied by coefficients ± 1 , which are computed from hash functions of the form

$h : \mathcal{U} \rightarrow \{-1, 1\}$. It can be shown that, by choosing h to be 4-wise independent, the square of this sum is an unbiased estimate of the second moment. These estimators are then divided into groups and the median of the averages of the groups can be shown to be an extremely robust estimate of the second moment.

The tug-of-war sketch uses just $O(\log(1/\delta)/\epsilon^2)$ estimators—the asymptotically optimal number [9]—which bounds both the number of counters needed by it and the number of operations needed to update it. This makes it very efficient to update in a stream. Note that the number of counters necessary is independent of the number of elements that have been inserted into the sketch, which allows us to use the same sketch size for each distributed node and group.

Each estimator of the tug-of-war sketch is a linear projection of the form $\vec{a} \cdot \vec{v} = a_1 v_1 + \dots + a_n v_n$, where \vec{a} is the vector of frequencies and \vec{v} is a vector in $\{-1, 1\}^n$. This permits arbitrary updates to the sketch (i.e., updates with both positive and negative integers) since updating the frequency of the i th element by u can be done by simply adding uv_i to the estimator. Additionally, if two sketches use the same hash functions (i.e., the same vector \vec{v}), they can be directly summed to give the sketch that would have resulted from taking the union of the original inputs. *This extremely powerful summable property is what allows us to aggregate the result of the nodes in a split-independent fashion.*

We note that Indyk’s stable distribution sketch [37] also has the same desirable properties as the tug-of-war sketch. Namely, it is summable, is efficient to update, and has the same asymptotic space bound. However, in practice the stable distribution sketch needs considerably more space than the tug-of-war sketch because of its requirement of independent, stably-distributed values [37].

2.4.5 Analysis

One issue that our algorithm, and indeed any approximate algorithm for this problem, must overcome is that it is virtually impossible to distinguish an iceberg from any non-icebergs close to its size. To assist with this issue, we introduce the concept of the gap assumption.

The gap assumption is an assumption that we make about the measured data to assist in correctly detecting icebergs. According to this assumption, there will never be any non-icebergs in the range $(\lambda T, T)$, where T is the threshold for icebergs and $\lambda \in (0, 1)$ is a known gap parameter. In this section we will assume this assumption to be strictly true, and later we will discuss the effect of having a few non-icebergs in the gap.

Our solution for this problem is to simply use Algorithm 1 with $g = 6n\lambda^2$ as the number of groups. In the following sections we prove the communication cost bounds for this algorithm and give analysis showing its accuracy in correctly detecting the presence of icebergs.

2.4.5.1 Communication Cost

Since each sketch has cost $O(\log(1/\delta)/\epsilon^2)$, our algorithm requires each local node to communicate a total of $O(g \log(1/\delta)/\epsilon^2)$ counters to the central server. Taking constant ϵ and δ , we have that the communication cost of our algorithm is $O(g) = O(n\lambda^2)$.

In comparison, while the naive algorithm has each local node send a counter for each element (for a total of $(1+\Omega(1))n$ counters), our algorithm requires $192n\lambda^2 \log(1/\delta)/\epsilon^2$ counters, which gives us large savings when λ is small (e.g., $1/1000$).

The tug-of-war sketch can be modified to use $2/(\delta\epsilon^2)$ counters by just averaging the estimates, rather than taking the median of averages. This is less than $32 \log(1/\delta)/\epsilon^2$ when δ is not too small. For example, if we take $\epsilon = 1/2$, $\delta = 0.05$, then our algorithms

requires $960n\lambda^2$ counters, which gives us considerable savings when λ is as large as $1/100$.

Note that since our algorithm does not need to send the identities of elements along with their counts, we are not burdened with this additional overhead. A naive method, on the other hand, necessarily must transmit element identities to aggregate the counts of all the elements. Hence, our algorithm will especially shine when element identities are large (e.g., IP flow labels).

Numerical Example: Consider a situation each distributed node has $m = 1000000$ (one million) search queries that they need to communicate to the central server. Let us assume that each element has an identity of size 12 bytes and a counter of size 8 bytes. Further, let us assume that we are guaranteed a gap of $\lambda = 1/100$ in the data. Then, a naive solution would require about $20 \times n = 20\text{MB}$ of communication to identify any icebergs in the data. In contrast, our algorithm would need only $8 \times 960n\lambda^2 = 768\text{KB}$ of communication to solve the problem. This gives us over an order of magnitude savings in the communication cost.

2.4.5.2 False Positive Rate

We showed in the previous section that the false negative rate of our algorithm is determined solely by the failure rate of the sketches. By keeping this rate δ small, we will almost never miss a true iceberg. Hence, all that is left for us to show is that it is unlikely for a group without any icebergs in it to be a false positive.

Theorem 10. *For every group with no iceberg in it, the iceberg detection algorithm erroneously signals that it has an iceberg in it with false positive probability at most $\delta + \delta'$, where δ is the failure probability bound of the tug-of-war sketch, $\delta' = (\frac{\epsilon}{4})^{1/(6\lambda^2)}$, and λ is the gap parameter.*

Proof. To simplify our analysis, we consider the worst case input for our algorithm: when all n elements have count λT . It is not hard to see that if the non-icebergs are

smaller than λT this will only decrease the probability of a false positive.

Since the sketches may err with ϵ relative error, a non-iceberg may appear to contribute as much as $T^2\lambda^2(1+\epsilon)$ to the measured F_2 . As the threshold of detection is set to $T^2(1-\epsilon)$, a false positive could only occur when at least $\frac{T^2(1-\epsilon)}{T^2\lambda^2(1+\epsilon)} \geq 1/(3\lambda^2)$ non-icebergs get put in the same group, where we assume that $\epsilon \leq 1/2$. Let us denote by X the random variable indicating how many non-icebergs get put in one particular group and bound the probability of the event that X exceeds $1/(3\lambda^2)$.

Let us denote by X_i the event that element u_i is in our group, for $i \in \{1, \dots, n\}$, so that $X = \sum_{i=1}^n X_i$. Clearly, X_i 's are i.i.d. Bernoulli random variables with probability $1/g$, since an element may go into any group with equal likelihood. This permits us to use the Chernoff bound:

Theorem 11 (Chernoff Bound). *Let $X_i, 1 \leq i \leq n$ be i.i.d. Bernoulli random variables with probability p , $X = \sum_{i=1}^n X_i$. For $\beta > 1$,*

$$\Pr[X \geq \beta pn] < \left(\frac{e^{\beta-1}}{\beta^\beta} \right)^{pn}.$$

Applying the above Chernoff bound, we get the following

$$\begin{aligned} \Pr[X > 1/(3\lambda^2)] &= \Pr[X > 2(n/g)] \\ &\leq \left(\frac{e^{2-1}}{2^2} \right)^{n/g} = \left(\frac{e}{4} \right)^{1/(6\lambda^2)}. \end{aligned}$$

Since the error in the estimate occurs with probability at most δ , the false positive probability in question is at most $\delta + \delta'$, as desired. \square

Since we expect our algorithm to work only when $\lambda \ll 1$, we expect the δ term to dominate this failure probability. Not only does the above algorithm detect the presence of one or more icebergs, it narrows down the iceberg to a subgroup of the universe. Each group that is above the threshold can be polled to identify the iceberg. Since there are only an expected $1/(6\lambda^2)$ elements in each group, this cost is far lesser than that of sending frequencies of all n elements.

Numerical example: When $\lambda = 0.1$, the false positive probability for a group is at most 0.16%. For $\lambda = 0.05$, this probability drops to less than one in hundred billion (10^{-11}). Clearly, this probability is much smaller than the failure probability of the sketch, δ , which we take to be around 1% in practice. At worst, we expect 1% of the groups (and hence about 1% of the elements) to signal a false positive, which takes very little additional communication to drill down.

2.4.5.3 Estimating Iceberg Size

Besides detecting the presence of an iceberg, it would be useful to get an estimate on its size. Size information is useful in diagnosing the extent of the anomaly and could help in determining what action should be performed next. In this section we show how our solution allows us to obtain an approximate estimate of the actual size of the iceberg in this setting *without any additional communication overhead*. If this estimate indicates a severe problem, a more accurate (but expensive) estimate of the size of an iceberg can be computed using an additional round of communication.

2.4.5.4 Biased Estimator

The first algorithm for estimating the size of the detected iceberg is simple. We take the estimate of F_2 for the group the iceberg was found in and use the square root of this value as an estimate of the iceberg size. There are two sources of error for this estimate: the approximation of the F_2 estimation as well as the collision of non-icebergs in the same group. (We assume that the number of icebergs is small enough that no two icebergs get mapped to the same group with high probability.)

Suppose that we detect an iceberg of size $S \geq T$ in a group that has estimated F_2 above the $T^2(1 - \epsilon)$ threshold. We first estimate by how much we may under-estimate its true frequency: this is bounded by the error of the F_2 estimation. Hence, with probability at least $1 - \delta$, this algorithm returns an estimate \hat{S} such that

$$\hat{S} \geq S\sqrt{1 - \epsilon}.$$

Assuming that $\epsilon \leq 1/2$ (as earlier) we get the guarantee that $\hat{S} \geq S/\sqrt{2}$.

The analysis for the bound on over-estimating the size of the iceberg is slightly more involved since we now have to account for the collisions of non-icebergs in the same group. We start by bounding the probability that the collisions exceed the threshold $T^2/3$. As in the earlier detection analysis, this would require more than $\frac{T^2/3}{T^2\lambda^2} \geq \frac{1}{3\lambda^2}$ non-icebergs to be in the same group as the iceberg. As earlier, we bound this probability:

$$\Pr[X > 1/(3\lambda^2)] < \left(\frac{e}{4}\right)^{1/(6\lambda^2)} = \delta'.$$

Since $S \geq T$, the total overestimation error for the F_2 estimation is at most $(T^2/3 + S^2)(1 + \epsilon) - S^2 \leq S^2$ (again, where we assume $\epsilon \leq 1/2$).

Hence, we finally have the following theorem:

Theorem 12. *With probability at least $1 - \delta - \delta'$ we can estimate the true size of an iceberg of size S by \hat{S} with the guarantee that $S/\sqrt{2} \leq \hat{S} \leq S\sqrt{2}$.*

Obtaining a more accurate estimate of the iceberg size comes at a higher cost. We may obtain much more accurate estimates for the size of the iceberg using smaller ϵ . However, recall that the dependence of the communication cost on ϵ is $1/\epsilon^2$, which grows very rapidly. For example, halving the relative error results in quadrupling the communication cost. Still, there is a natural tradeoff here.

2.4.5.5 Unbiased Estimator

The above estimator has the disadvantage of being biased by the collision of non-icebergs with the iceberg. A more accurate way to approximate the size of the iceberg is to estimate the mass of non-icebergs that collide with it and remove this from our estimate. We can use the average of the F_2 estimates of the other groups for this purpose. Here we show that this estimator is unbiased when there is only one iceberg in the whole dataset and the F_2 sketch used is unbiased (e.g., the tug-of-war sketch).

We will use Y_0 for the F_2 of the non-icebergs in the group with the iceberg, and Y_1, \dots, Y_{g-1} for the F_2 of all the other groups. We know that $E[Y_0] = E[Y_1] = \dots = E[Y_{g-1}]$. Let Π be the (non-real-valued) random variable denoting how the n elements are placed into g groups. Assume the iceberg size is S . So the estimator for iceberg size is $\widehat{S + Y_0} - \frac{1}{g-1} \sum_{j=1}^{g-1} \widehat{Y_j}$, where the hat reflects the F_2 sketch estimator. So we have

$$\begin{aligned}
& E\left[\widehat{S + Y_0} - \frac{1}{g-1} \sum_{j=1}^{g-1} \widehat{Y_j}\right] \\
&= E_{\Pi} \left[E\left[\widehat{S + Y_0} - \frac{1}{g-1} \sum_{j=1}^{g-1} \widehat{Y_j} \mid \Pi\right] \right] \\
&= E_{\Pi} \left[S + Y_0 - \frac{1}{g-1} \sum_{j=1}^{g-1} Y_j \right] \\
&= S + E_{\Pi}[Y_0] - \frac{1}{g-1} \sum_{j=1}^{g-1} E_{\Pi}[Y_j] = S.
\end{aligned}$$

The first equality is due to the law of total expectation. The second equality is because we are using an unbiased F_2 sketch. The third equality is due to the linearity of expectations. Hence, the estimate using this method is an unbiased estimate of the actual iceberg size.

2.4.6 Analysis - Using F_1 Information

In the previous section, we show that in the case of a strong gap assumption (e.g., λ values as small as $1/100$), the tug-of-war sketch allows us to identify icebergs with high probability in one round. We also prove a lower bound communication complexity under these conditions, which show that our detection algorithm is indeed asymptotically optimal, although the result is not practically satisfying.

Fortunately, this is not the end of story. We are able to significantly improve our result, in terms of the required λ , by asking for one additional piece of information

that can be obtained with very little cost. This additional piece of information is the sum of the counts (F_1) of all the elements across all the nodes. It can be obtained by asking every node to send in the sum of the counts of all its items. Adding them up at the central server results in the F_1 of the union of the dataset. The additional communication cost is simply a few more bytes per node. However, strictly speaking, this adds one more round to the protocol as follows. In the first round, the total local counts are sent to the central server and summed to get F_1 . Then the optimal number of groups is computed based on this count, and this is broadcast to all the nodes. Finally, the nodes send grouped tug-of-war sketches to the server.

In reality, however, this additional round can be avoided in continuous monitoring applications when the change from one time window to the next is not gigantic. We can simply use the average F_1 of the past windows as the estimate of the F_1 of the next window, for the purpose of determining the optimal number of groups. Then the total count for the next window can come in together with the tug-of-war sketches in one round. This scheme will work because the accuracy of our detection is not sensitive to the number of groups as computed from F_1 .

Readers may now wonder why this total count alone makes such a huge difference. This is because earlier we only knew that all the items had frequencies between 0 and $B = \lambda T$. Our false positive bound has to assume the worst case from this very large family. However, once we know F_1 , there is a constraint on the item counts, resulting in a much smaller family of counts. The worst case of the smaller family is much better than the worst case of the larger family. However, the worst case of the larger family is mathematically trivial, i.e., every element has count B , which is what we considered in the previous section. Obtaining or even approximating the worst case for the smaller family, however, turns out to be extremely challenging mathematically.

For bounding the false positive rate, we are interested in the second moment F_2 of any group without an iceberg. Each element has an independent and identical

probability to fall into this group. The element counts vector $\{a_i\}$ is only subject to two constraints: $a_i \leq B, \forall i$, and $\sum_i a_i = L$, where L is the aggregate F_1 . Since our scheme has to work with all possible element counts vectors, our bound clearly has to be the worst case (i.e., the maximum) bound over all of them. Fortunately we have the technique developed in Section 2.1 to solve this problem. The twist is that we will use *increasing convex order* here.

Let us denote by X_i the event that element u_i is in the chosen group. X_i 's are i.i.d. Bernoulli random variables with probability $1/g$. Let X_a be the F_2 of the elements in this group, i.e. $X_a = \sum_{i=1}^n a_i^2 X_i$. We denote it X_a to emphasize that its distribution depends on the vector a . We want to bound the probability that $X_a > \frac{T^2(1-\epsilon)}{1+\epsilon} \equiv A$.

Now we can prove the following theorem:

Theorem 13. *Let g be a convex function. Let X_1, \dots, X_n be exchangeable random variables that take only non-negative values. Then $a \leq_M b$ implies $\sum_{i=1}^n g(a_i)X_i \leq_{icx} \sum_{i=1}^n g(b_i)X_i$.*

Proof. Let f be any increasing convex function. Let $\Phi(x; a) = f(\sum_{i=1}^n g(a_i)|x_i|)$. We will verify that $\Phi(x; a)$ satisfies the conditions in Theorem 2. When x is fixed, $g(a_i)|x_i|$ is convex because $|x_i| \geq 0$ and g is convex. So $\sum_{i=1}^n g(a_i)|x_i|$ is a sum of convex functions, thus convex [63, Theorem 5.2]. So $\Phi(x; a)$ is a composition of an increasing convex function with a convex function, thus convex [63, Theorem 5.1], therefore (i) holds. (ii) obviously holds. When a is fixed, $\Phi(x; a)$ is continuous because f is necessarily continuous, so (iii) holds.

Therefore Theorem 2 tells us that $E[\Phi(X; a)]$ is symmetric and convex, thus Schur-convex [51, 3.C.2]. $E[\Phi(X; a)] = E[f(\sum_{i=1}^n g(a_i)|X_i|)] = E[f(\sum_{i=1}^n g(a_i)X_i)]$, due to the assumption $X_i \geq 0$. By definition of Schur-convexity, $a \leq_M b$ implies $E[f(\sum_{i=1}^n g(a_i)X_i)] \leq E[f(\sum_{i=1}^n g(b_i)X_i)]$. Since this is true for any increasing convex function f , by definition of increasing convex order we have $\sum_{i=1}^n g(a_i)X_i \leq_{icx} \sum_{i=1}^n g(b_i)X_i$. \square

Remark: Note that stochastic order does not hold here in general. Suppose $a_1 = a_2 = 1, b_1 = 0, b_2 = 2, g(x) = x^2$, and X_1, X_2 are i.i.d Bernoulli with success probability $0 < p < 1$. Then

$$\begin{aligned}\Pr[X_1 + X_2 \leq 0] &= (1 - p)^2 < 1 - p = \Pr[4X_2 \leq 0] \\ \Pr[X_1 + X_2 \leq 1] &= 1 - p^2 > 1 - p = \Pr[4X_2 \leq 1]\end{aligned}$$

For a stochastic order relation to hold, the two inequalities must be in the same direction.

Now we are ready to specify the worst-case element count vector in terms of increasing convex ordering. The pattern of worst-case item counts is that some item counts take maximum value B while other are 0.⁹ Let a^* be the vector where $a_i = B, 1 \leq i \leq L/B$ and $a_i = 0$ otherwise.

Corollary 14. $X_a \leq_{icx} X_{a^*}$, and consequently

$$\Pr[X_a > A] \leq \min_{\theta > 0} \frac{\mathbb{E}[e^{\theta X_{a^*}/B^2}]}{e^{\theta A/B^2}}.$$

Proof. It is easy to see that $a \leq_M a^*$. Applying Theorem 13 to the i.i.d Bernoulli random variables $\{X_i\}$, the convex function $g(x) = x^2$, we get $X_a \leq_{icx} X_{a^*}$.¹⁰ Since $f(x) = e^{x\theta}$ is an increasing convex function of x , by definition we get $\mathbb{E}[e^{\theta X_a}] \leq \mathbb{E}[e^{\theta X_{a^*}}]$. From our earlier discussion on the Chernoff method we get

$$\begin{aligned}\Pr[X_a > A] &\leq \min_{\theta > 0} \frac{\mathbb{E}[e^{\theta X_a}]}{e^{\theta A}} \leq \min_{\theta > 0} \frac{\mathbb{E}[e^{\theta X_{a^*}}]}{e^{\theta A}} \\ &= \min_{\theta > 0} \frac{\mathbb{E}[e^{\theta X_{a^*}/B^2}]}{e^{\theta A/B^2}}.\end{aligned}$$

For the last step we replaced θ with θ/B^2 . □

⁹For simplicity of computation we round L up to multiples of B . It is simple to prove that the increasing convex ordering still holds.

¹⁰If we were to consider other frequency moments $F_p, p > 1$, the same convex ordering results apply.

Note that X_{a^*}/B^2 is a sum of i.i.d Bernoulli random variables, so the bound in the above corollary is exactly the Chernoff bound for the sum of L/B i.i.d. Bernoulli random variables with probability $1/g$ exceeding A/B^2 . If we pick $g = \beta \frac{B^2}{A} \frac{L}{B} = \lambda^2 \beta \frac{1+\epsilon}{1-\epsilon} \frac{L}{B}$, where $\beta > 1$ is a constant we can choose, then from Theorem 11 the above bound can be relaxed to:

$$\delta' \equiv \left(\frac{e^{(1-1/\beta)}}{\beta} \right)^{\frac{(1-\epsilon)}{(1+\epsilon)\lambda^2}}. \quad (18)$$

We note that δ' can be decreased by either increasing β , or decreasing ϵ , or both. If λ is small we can pick $\epsilon = 1/2$ and $\beta = 2$, and the number of groups simplifies to $g = 6\lambda^2 L/B$. If $L = nB$, then g is the same as in the previous section. However for real data we usually have $L \ll nB$, so we get a much smaller g , thus much less communication cost than if we didn't know L .

If λ is close to 1, we need to pick larger β and smaller ϵ to keep the bound small, which translates to larger communication cost. We want to have $\lambda^2 \frac{1+\epsilon}{1-\epsilon} < 1$, i.e., $\epsilon < \frac{1-\lambda^2}{1+\lambda^2}$, so that a single element of size λT will not become false positive with probability at least $1 - \delta$.

Numerical example: If $\lambda = 0.1$, we can pick $\epsilon = 1/2$ and $\beta = 2$ in Equation 18, so that the false positive rate $\delta' = 0.0016$. If, say, $L/B = 0.001n$ then $g = 6n/100000$, giving a 1000-fold improvement from the analysis in Section 2.4.5. If $\lambda = 1/3$, we can pick $\epsilon = 1/2$ and $\beta = 9$, so that $\delta' = 0.0197$. Again taking $L/B = 0.001n$, we get $g = 3n/1000$. Note that the analysis in Section 2.4.5 breaks down for $\lambda = 1/3$, since it gives $\delta' = 0.56$ and $g = 2n/3$. We need $g \ll n$ to achieve communication cost savings.

2.4.6.1 Estimating Iceberg Size

We can have the same size estimator as in Section 2.4.5.3. We omit the proof for the following theorem as it is similar to the one for Theorem 12.

Theorem 15. *With probability at least $1 - \delta - \delta'$ we can estimate the true size of an iceberg of size S by \hat{S} with the guarantee that $S/\sqrt{2} \leq \hat{S} \leq S\sqrt{2}$.*

We also can use the same unbiased estimator as in Section 2.4.5.3.

2.4.7 Properties

In this section we discuss why we expect our algorithm to perform well on real-world data, as well as several desirable properties that it has.

2.4.7.1 Discussion of the Gap

All our analysis thus far have assumed that there is a gap in the data. However, our algorithm still works even when there is no such large gap, and the gap is necessary only for the purposes of providing guarantees on the performance. In practice, our algorithm performs well even when there is a small gap in the data and when the magnitude of the gap is not known.

We may also loosen the gap assumption by permitting a few elements to appear within the gap. For real problems, this is very often the case—the data usually has a long, thin tail and there are very few elements that come close to the desired threshold. Note that the false negative rate of our scheme is unaffected by such elements. Larger non-icebergs will only increase the F_2 of a group and never allow an iceberg to be missed since we keep the detection threshold fixed. Hence, the only penalty that we pay is a higher false positive rate, which only results in a slightly higher communication cost to drill down a group. This additional cost is, at worst, proportional to the number of elements in the gap. In the case of real data, this is an exceedingly small number (e.g., one or two), and hence barely affects our performance.

2.4.7.2 Streamed Data

When aggregating large volumes of data (e.g., Internet IP packet data), it is necessary to employ streaming algorithms to summarize the data succinctly in a single pass.

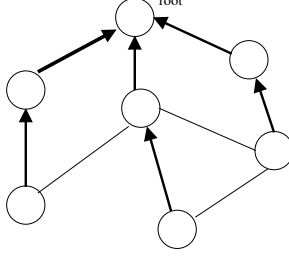


Figure 10: The sketches can be aggregated on any connected topology.

Our algorithm is capable of doing this since it is already based on very light-weight sketches. In particular, each update performed locally at a node can be performed using only $O(\log(1/\delta)/\epsilon^2)$ hash operations and additions since only the sketch of a single group has to be modified for each update in the stream. Since ϵ and δ are small constants, this is essentially a constant-time update, independent of the size of the stream. We find in practice that as few as 5 to 10 estimators may suffice for each sketch.

The memory cost of our approach can be quantified by the product of the number of groups times the number of estimators for each group. In the previous section we gave some tight bounds on how many groups may be required. However, in practice, the number of groups necessary may be much smaller since our bounds assume adversarial (i.e., worst-case) data. Please refer to Section 2.4.8 for more of these details.

2.4.7.3 Application to Arbitrary Topologies

Our solution can be implemented on any arbitrary connected topology due to the summable property of the sketches. Consider any communication graph G . It is possible to choose a spanning tree that is rooted at the node at which we would like to perform the iceberg detection. The protocol is then for every node in the tree to send its sketches to its parents. The parents can then sum these sketches (since all of them use the same hash functions) and pass them along to their parents. Finally

the root of the tree can perform the iceberg detection as described in Section 2.4.4. In this way, only one set of sketches is transmitted on each link. See Figure 10. The edges indicate communication links and the heavy edges are the spanning tree along which the sketches are aggregated.

The communication cost for each non-root node is identical: they all have to send the same number of sketches to their parents. This means that this solution is completely unaffected by the volume of data at each node. Since the sketch sizes are independent of the number of elements inserted into them, every node has the same, succinct set of sketches. Lastly, it should be clear that it does not matter in which order the sketches are summed since the sum operation, which is simply vector addition, is commutative and associative.

2.4.8 Empirical Evaluation

In this section we evaluate our methodology of using F_2 sketches for detecting icebergs in distributed data. We start by fine-tuning the tug-of-war sketch for our purposes. We then evaluate the performance of our algorithm on real network data by varying various parameters. We show that using our proven guarantees we can use as little as 7.5% of the space of the naive algorithm and that using 1% suffices in practice.

2.4.8.1 A Few Words About Sketch Size

For the tug-of-war F_2 sketch, we have an (ϵ, δ) guarantee with $32 \log(1/\delta)/\epsilon^2$ or $2/(\delta\epsilon^2)$ counters. For $\epsilon = 0.5, \delta = 0.02$ this translates to 400 counters. However, this theoretical bound is quite loose. We experimented with the tug-of-war sketch on a variety of data, artificial and real, and found in all cases that a sketch of only 50 counters satisfies the $(0.5, 0.02)$ bound, i.e. it gives estimation with less than 50% relative error for more than 98% of the time.

In the experiments with our iceberg detection algorithm, we need much fewer counters. We found that 10 counter per sketch performed very well. This is due to

the following reasons.

For false negative rate: We are only concerned with groups that happen to contain an iceberg. The group size has been chosen in such a way that with high probability one element (the iceberg) dominates the F_2 of the rest of the elements. We found that the tug-of-war sketch performs extremely well for such datasets, so we only need a small number of counters. (In the case that there are two icebergs in the group, the sketch will need to have at least 75% negative error to cause a false negative, which turned out to be also very unlikely.)

For false positive rate: Two factors could contribute to a false positive—the F_2 of the element counts in the group could be large and the sketch could have a large positive error. For most of the time the F_2 of the element counts is very small compared to the iceberg, and the sketch error with very high probability is not large enough to cause false positive. In other words, the deviation of F_2 plays a bigger role than the deviation of the sketch in causing false positives. Therefore a small sketch with large error still works in practice.

2.4.8.2 Experiments with Network Data

We tested our proposed algorithms on data collected from the Abilene network [4]. We used the destination IP addresses as the element labels. Our trace aggregated packets across several sites over a one day period. In order to simulate a large number of nodes, we distributed the packets to 100 nodes by hashing the source IP addresses uniformly at random to 100 bins. The total raw data size over the 100 nodes is 11.87M (with 4 bytes for label and 4 bytes for flow size). There are in total 140,275 unique destination addresses in this dataset. We set the bound $B = 500,000$ and the iceberg threshold $T = 1,500,000$, therefore $\lambda = 1/3$. There are two element counts between the bound and the threshold, and one element count above the threshold at 1,784,420. Total F_1 is 3.673×10^7 .

Using only the gap assumption we get the number of groups to be $g = 93516$. Adding the F_1 information we get $g = 221$, choosing $\epsilon = 1/2, \beta = 9$ so that (18) is less than 0.02. The communication cost for all the sketches is $0.884M$, counting 4 bytes per counter. The ratio to raw data is 7.5%. We encounter no false negatives or false positives at this setting. Encouraged by this, we pushed the parameters to extreme values to examine the performance of our algorithm on this dataset.

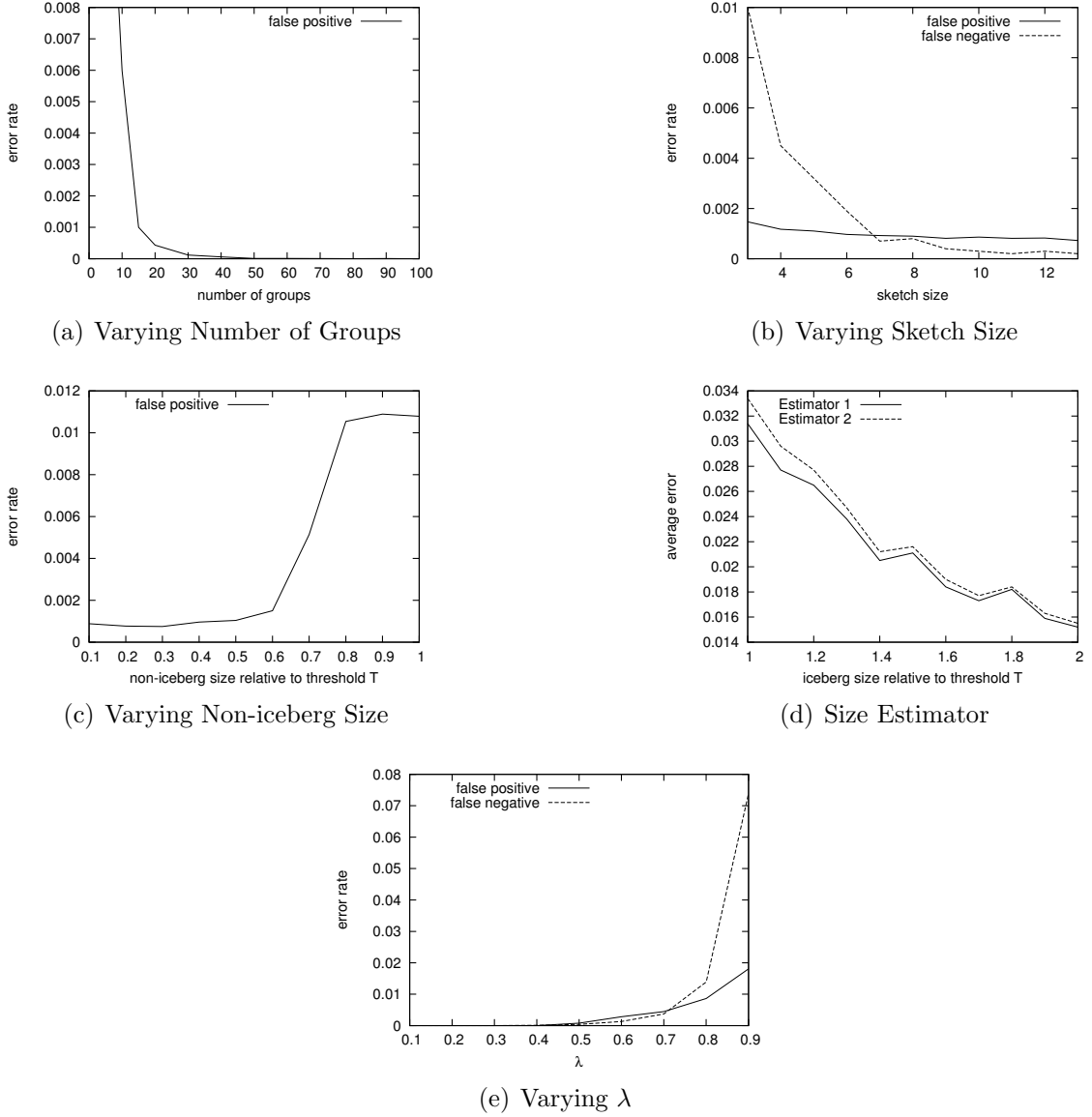


Figure 11: Iceberg Evaluation Results

In Figure 11(a) we reduce the number of groups. The false negative remained at

0. We can see that false positives do not occur when $g = 60$, which corresponds to communication cost of only 2%. Even at $g = 20$ the false positive rate is still very low.

Assume that we underestimated the bound to be $B = 400,000$ instead of $B = 500,000$. So $\lambda = 1/3.75$ and we choose $\beta = 5$ in (18). Further assume that we severely underestimated F_1 to be 2×10^7 instead of 3.673×10^7 . We will get $g = 53$ which still give very good performance. This shows that it is not crucial for us to get accurate estimates of bound B or total F_1 for deriving the number of groups.

In the following we make the problem harder by reducing the iceberg threshold to $T = 1,000,000$, i.e. $\lambda = 0.5$. We also replace the large iceberg by one right at the threshold, i.e. with size 1,000,000. We fix $g = 100$ and study how other parameters affect performance.

In Figure 11(b) we vary the sketch size, i.e. number of counters per sketch. We can see that false negative rate increases as sketch size decreases, which is expected. We see that false positive rate is not very sensitive to sketch size, verifying our remark about sketch size and false positive rate in the previous section.

Next we study how the size of a non-iceberg element affects the false positive rate. We insert non-iceberg of various sizes into the data. In Figure 11(c), the x -axis ratio is relative to the threshold T . We see that after the element reaches a certain size it starts to increase false positive rate, then it reaches a plateau where the group containing this element is very likely to report positive. We have remarked before that when a non-iceberg is close to the threshold it is hard to distinguish it from an iceberg.

Figure 11(d) plots the average relative error for the two size estimators when the iceberg size changes. Estimator 1 is the simple estimator, and Estimator 2 removes the bias. The x -axis ratio is relative to the threshold T . The peculiar result is that although estimator 2 is unbiased, estimator 1 has slightly less average relative error

in this case.

Next we will change λ and see how it affects performance. We still use $g = 100$ and sketch size 10. We remove the 3 elements above the bound, and for each λ we set the threshold and insert an iceberg at $1/\lambda$ times the bound. Figure 11(e) shows the result. We see that even for higher λ s (e.g., $\lambda = 0.7$, where the iceberg is less than 1.5 times the bound) the algorithm still performs well. For λ closer to 1 we will need larger g to control false positive and larger sketch size to control false negative.

Hence, we see that for real data our algorithms greatly out-perform the provided theoretical guarantees. The reason for this is that all the guarantees we give are worst-case, whereas real network data follows a highly skewed power-law distribution.

CHAPTER III

RESOURCE ALLOCATION

3.1 *Introduction*

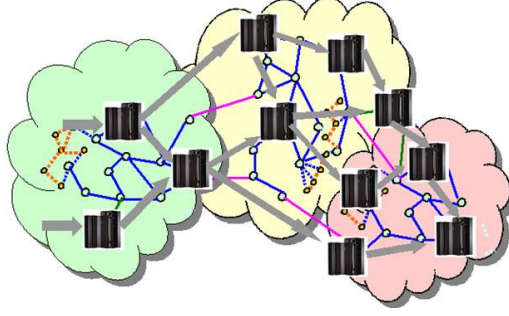


Figure 12: Stream Processing

Fork and Join Processing Networks arise in many networked or distributed computing areas, including cloud computing, parallel processing, distributed stream processing, and various applications in distributed data intensive computing. Due to the complex processing requirements in these applications, the underlying data processing software typically consists of a rich set of semantics that define how data from various input links are processed (which will be referred to as join), and how output data are disseminated to each output link (which we will refer to as fork).

We abstract the various fork and join semantics into four types: synchronous join (denoted \otimes -join), asynchronous join (\oplus), synchronous fork (\otimes), and asynchronous fork (\oplus). With a *synchronous join*, data (or jobs) from all input buffers are assembled simultaneously; whereas an *asynchronous join* processes data (jobs) from one input buffer at a time. Synchronous join operations such as correlation and aggregation, are at the core of most stream processing systems [8, 10]. Similarly, a *synchronous fork* dis-assemble or duplicate data over all output links simultaneously; while an

asynchronous fork sends data over one output link at a time. Synchronous fork operations used for data separation, duplication, and task decomposition, are common in stream processing, in grid computing [17], in parallel processing [14], and in multi-cast communication [40]. Asynchronous fork or join operations represent multi-path options for data routing or processing which are common in multipath routing and load balancing.

Typically, a processing task (think in terms of a piece of software program) is associated with a specific join semantic and a specific fork semantic, which defines how input data are processed and how output data are disseminated. We refer to an information processing network interconnected with many such generalized fork and join tasks as a *general fork and join processing network*.¹

Such general fork and join (information) processing networks are often deployed over a network of geographically dispersed servers with limited resources. The servers could have diverse characteristics. Edge devices such as sensors typically are low power with low computing capability and low reliability, while high-end server clusters are at the opposite end of the spectrum. Data sources often produce large volumes of data at high rates, while workload spikes cannot be predicted in advance. Providing high performance execution of such distributed applications can place considerable strain on both communication and processing resources. Resource management is complicated by the diversity in server capabilities.

The task-to-server mapping itself is an important research problem (see, e.g. [75]), but it is out of the scope of this thesis. Instead, we assume that the task to server mapping is given, and we address the problem of how to allocate the limited resources to the various tasks so as to maximize the total utility of system output. Different tasks on the same server must compete for limited resources. The upstream tasks

¹We use the modifier “general” because the terms “fork and join processing network” has been used to refer to networks with only synchronous forks and joins.

must satisfy the demand of downstream tasks. A task may need inputs from multiple streams simultaneously, and may output multiple streams simultaneously. The streams may expand or shrink after processing. The external streams into the system may need to be curtailed in order not to overload the system. Different final output streams may have different values. We need to design an algorithm to perform admission control, routing and resource allocation under all these constraints to achieve optimal system utility, and it needs to do so in a distributed and adaptive way due to the distributed and time varying nature of the system. Our goal is to address the problem of distributed resource allocation in such general fork and join processing networks.

In Section 3.2 we use “pipelined” algorithms to solve a special case of the problem where all the forks and joins are synchronous. In Section 3.3 we use back-pressure based algorithms to solve the general case of the problem.

3.1.1 Related Work

The problems of load shedding and resource management have recently gained increasing attention in stream processing systems. Most studies address the problem from a system perspective, where solutions are provided using heuristic or statistical methods [54, 71, 10] while making no optimality guarantees.

In this thesis, we take a quantitative approach and address the problem using the rigorous framework of network utility maximization. The advantage of this approach is that one can derive distributed algorithms systematically with guaranteed convergence to the global optimum. Starting with the work reported in [70], the approach has been successfully applied to solve many rate allocation problems in communication networks in the unicast setting [41, 48, 42, 45, 24]. See [46] for a comprehensive survey. Several recent studies have advanced the state-of-the-art for unicast networks to the multirate multicast setting [39, 40, 23, 16]. Such multicast networks are special

cases of fork and join networks since there are only \otimes -forks but no \otimes -joins. In the context of data intensive computing, this approach has been used in [26] for sequential processing graphs (with only \oplus -forks and \oplus -joins). To the best of our knowledge, the distributed resource allocation problem for general fork and join processing networks has not been addressed fully.

Our “pipelined” algorithms in Section 3.2 can be viewed as an extension of the work in [39, 40] to the synchronous fork and join setting. [72, 26] assume simple sequential processing task graphs [10] deals with fork networks (with no joins) and uses feedback control to derive local resource control policies but provides no guarantee on the global optimality. [47] has both forks and joins, but it only handles the limited case that all processing units are on the same server. Both [10] and [47] try to maximize weighted throughput instead of general system utility.

Our back-pressure based algorithm in Section 3.3 is inspired by the shadow queue algorithm in [16] for multirate multicast in multihop wireless networks. [16] does not deal with \otimes -join, and its \oplus -join and \oplus -forks nodes are separate from the \otimes -fork nodes. In our model all four types can interconnect with each other. We also present a different approach for the optimality of the real outputs and the stability of the real queues.

We initially came up with the bipartite model purely as a helper to formulate the optimization problem, but it turned out to be a processing network after we reinterpreted the multiplier nodes as queue nodes. Our research is conducted independently from the work in [38]. [38] deals with a lossless network while our paper deals with a lossy network.

3.2 Synchronous fork and join only

In this section we study a special case where the task graph has only synchronous fork and join. An abbreviated version of this work appeared in [79].

3.2.1 Model and Problem Formulation

3.2.1.1 System Model

We denote the task graph as *directed acyclic graph* (DAG) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \mathcal{S} \cup \mathcal{P} \cup \mathcal{D}$ consists of three types of nodes: sources \mathcal{S} , tasks \mathcal{P} and sinks \mathcal{D} . Note that the term “node” here is applied to vertices in the task graph. It does not refer to physical servers. In addition, we do not require \mathcal{G} to be a connected graph.

The sources only have outgoing edges, the sinks only have incoming edges and the tasks have both types of edges. For each $v \in \mathcal{V}$, let \mathcal{I}_v denote the set of parent nodes (immediate predecessors), and \mathcal{O}_v the set of children nodes (immediate successors).

The tasks are deployed on heterogeneous servers. We denote the set of all servers as \mathcal{R} . We denote the server on which a task $v \in \mathcal{P}$ is placed as r_v , and the set of tasks located on the same server $r \in \mathcal{R}$ as V_r . Each server can have one or more tasks deployed on it. The server computation resources serve the computation demands of the tasks, and the bandwidth between the servers serve the communication demands between the tasks. Each server has R_r units of total resource. With a slight abuse of notation, we use R_v to represent the maximum flow input rate for each source $v \in \mathcal{S}$. We also use R_v to represent the maximum desired flow output rate for each $v \in \mathcal{D}$.

Associated with a given task-to-server mapping, there is a unique server dependency graph whose vertices are the servers, and there is a directed edge from server r_1 to server r_2 if there is a directed edge in the task graph from a task in r_1 to a task in r_2 . We do not require the server dependency graph to be a DAG.

Figure 13 offers an example task graph. It consists of 3 sources, 3 sinks and 7 tasks. There are 3 servers r_1 , r_2 and r_3 , represented by the rectangle shapes. Clearly, $V_{r_1} = \{p_2, p_3, p_5\}$, $V_{r_2} = \{p_1, p_4\}$, $V_{r_3} = \{p_6, p_7\}$. These are the groups of tasks that need to compete for the shared resources.

The amount of resource allocated to each task determines the amount of input data it will consume from each upstream link and the amount of output it will produce for

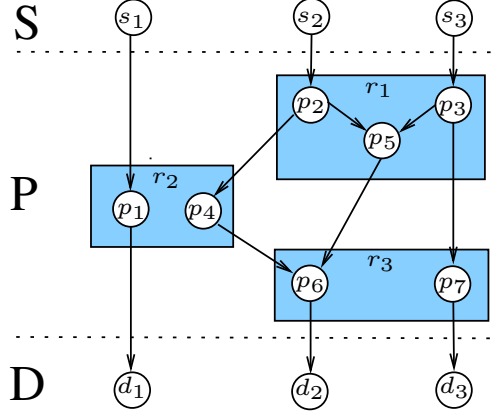


Figure 13: Example Task Graph

each downstream link. Denote by x_v the amount of resource allocated to task $v \in \mathcal{P}$. We extend the notation x_v to source and sink nodes as well. For source $v \in \mathcal{S}$, x_v represents the flow admission rate. For sink $v \in \mathcal{D}$, x_v represents the flow output rate.

Each task processes data flows from its parents *simultaneously* at given proportions and generates output flows to its children *simultaneously* at possibly different proportions. The ratio between input and output rates also may not be one. Similar to [47], we introduce parameters α and β to capture these proportions and ratios as well as the resource requirements for this task, as follows. Each task v in \mathcal{P} , with a unit of resource², consumes α_{iv} units of flow from parent i for any $i \in \mathcal{I}_v$, and produces β_{vj} units of flow for child j for any $j \in \mathcal{O}_v$. As convention, we will always use the two subscripts in the direction of the edge. The β parameters for source nodes and α parameters for sink nodes will be set to one.

These parameters $\{\alpha_{iv}\}, \{\beta_{vj}\}$ can be thought of as slowly time varying running averages. Each task measures its own parameters and there is no need to communicate them to neighbors, as we will show later.

²The unit for resource is specific to each server. If the unit is changed, we only need to change the α and β parameters of all the tasks on it accordingly.

For simplicity of notation for expressing resource constraints, we define $\tilde{\mathcal{R}} = \mathcal{R} \cup \mathcal{S}$. We extend the notations V_r to \mathcal{S} : $V_v = \{v\}, \forall v \in \mathcal{S}$.

Although this is a node-capacitated model, we can represent link capacities as well using similar techniques as in [26]. For a physical link, we introduce a “server” r where R_r is set to the link capacity. For each edge in the task graph that goes through this physical link, we introduce a “task” placed in r , and we add appropriate edges. The parameters α_{iv} and β_{vj} for the new task are set to one.

The multirate multicast problem in [40] can be seen as a special case of our model. Branches of multicast sessions are the tasks while the physical links are the servers. However, for multirate multicast problem there are only forks and no joins, so the graph consists of only rooted trees; there are no α or β parameters; there are no connections between the tasks located on the same server. Our model, algorithms and proofs must handle the general scenario required of stream processing. Due to having both forks and joins, our model, algorithms and proofs have a nice symmetry to them.

3.2.1.2 Lossiness and flow constraints

One important characteristic of a fork and join processing network is that it can be “lossy”. This is This is due to the presence of synchronous forks as described next. Consider the fork in Figure 14(a). For simplicity of exposition we omit the parameters α and β . Node i_1 sends output simultaneously to node j_1 and j_2 . Suppose j_1 is on a server with substantial resources, but j_2 is on a server with limited resources. It makes sense for i_1 to generate as much output as possible for j_1 , but j_2 will not be able to handle all of it. In this case, j_2 is allowed to drop part of the output from i_1 that it cannot consume.

Another scenario involves synchronous joins. Consider the fork in Figure 14(b). If j_2 receives a limited input from i_2 , this may cause j_2 not to consume all outputs

from i_1 and thus drop a fraction of them.

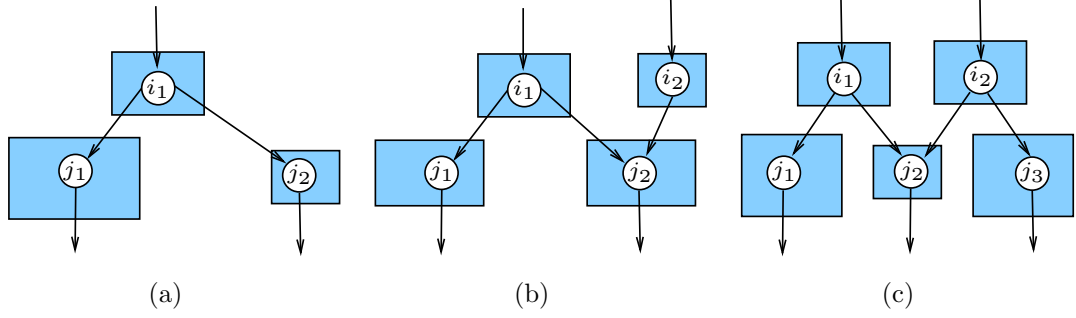


Figure 14: Several Lossiness Scenarios

The “loss” can happen at either upstream or downstream ends of an edge. In some applications it may make sense for i_1 to proactively discard some packets destined for j_2 , knowing the limited consumption rate of j_2 . While in other applications it may make sense for i_1 to send all packets to j_2 and let j_2 decide which ones to discard.

We need to distinguish “loss” from flow shrinkage. Flow expansion or shrinkage is caused by the nature of the task and the data it processes. “Loss” is caused by excessive demand or insufficient production of other tasks, as discussed above. So in the example of multirate multicast, there is “loss” but no flow shrinkage.

In such a “lossy” network, the flow constraint is that a child node cannot consume more than what the parent node can output, i.e., we must have

$$x_i \beta_{ij} \geq x_j \alpha_{ij}, \quad \forall (i, j) \in \mathcal{E} \quad (19)$$

Of course, in Figure 14(a) it will be a waste of resources if i_1 produces more data than what both j_1 and j_2 can handle. Each node only needs to produce the maximum of what its downstream nodes demand. Hence, the flow constraint is

$$x_i = \max_{j \in \mathcal{O}_i} x_j \alpha_{ij} / \beta_{ij}, \quad \forall i \in \mathcal{S} \cup P \quad (20)$$

However, the “dual” of (20), i.e. $x_j = \min_{i \in \mathcal{I}_j} x_i \beta_{ij} / \alpha_{ij}$, need not hold. For example in Figure 14(c), both i_1 and i_2 may output more than what j_2 can handle, because they need to satisfy the higher demands of j_1 and j_3 respectively.

3.2.1.3 The resource allocation problem

From the above examples, we see that the resource allocation problem is complicated by the presence of the interconnected forks and joins. Lossiness and the parameters $\{\alpha_{iv}\}$ and $\{\beta_{vj}\}$ make the problem even more convoluted. One consequence of the lossiness is that the sink output rates are not determined by the source input rates, but can depend on the resource allocation. So we have to judge the system utility based on output rates, not on input rates.

We assume there is a utility function $U_v(x)$ associated with every sink node $v \in \mathcal{D}$ that is increasing, strictly concave and differentiable. We want to maximize the sum of the utilities for all the flow output rates, i.e. $\sum_{v \in \mathcal{D}} U_v(x_v)$.

We now have abstracted the admission control and resource allocation problem for a fork and join processing network as the following convex optimization problem:

$$\begin{aligned}
 (P) \quad & \max \sum_{v \in \mathcal{D}} U_v(x_v) \\
 \text{subject to} \quad & x_i = \max_{j \in \mathcal{O}_v} x_j \alpha_{vj} / \beta_{vj}, \quad \forall i \in \mathcal{S} \cup \mathcal{P}, \\
 & x_v \geq 0, \quad \forall v \in \mathcal{V}, \\
 & \sum_{v \in V_r} x_v \leq R_r, \quad \forall r \in \tilde{\mathcal{R}}, \\
 & x_v \leq R_v, \quad \forall v \in \mathcal{D}.
 \end{aligned}$$

Due to the strict concavity of the utility functions, the solution to (P) is unique. We denote this unique value as the vector $x_{\mathcal{D}}^*$.

3.2.2 Distributed Algorithms

Resource allocation problem (P) is a convex optimization problem with a unique solution. In this paper, we present two different ways to solve the problem in a distributed manner, namely using primal algorithms and dual algorithms.

The primal algorithm focuses directly on the utility function, introduces a penalty for each violated constraint, and adjusts the flow rate adaptively. A dual algorithm

assigns Lagrangian multipliers (shadow prices) to flow constraints, and uses the prices to guide local resource allocation decisions. We present several variants of the duality approach by assigning shadow prices intelligently to different sets of constraints, which would impose different computation requirements on local servers. One can choose the primal algorithm or a variant of the dual algorithms depending on the needs of underlying applications and the computing capabilities of local servers.

We call the algorithms presented in this paper “pipelined” algorithms, in the sense that the algorithms try to determine proper resource allocation for all of the nodes so as to form “capacity pipelines” along the paths of the task graph. Properly admitted data can then flow from source to sink throughout the pipelines with sufficient resources secured along the way. The pipelines may have “leaks” due to the “lossiness” of the network.

3.2.3 Primal Algorithm

For the primal approach, we capture each resource constraint by adding a penalty term to the objective function based on resource usage; this yields a new optimization problem. In general, the modified problem has a different optimal solution than the original one, but with proper choice of the penalty terms, the two solutions can be made arbitrarily close. We will also show that for (P) there are choices for the penalty terms such that the modified problem has exactly the same solution as the original one.

We define a penalty function $P_r(y)$ for each $r \in \tilde{\mathcal{R}}$. $P_r(y)$ is assumed to be convex and increasing. Let $p_r(y)$ be a subdifferential of $P_r(y)$. $p_r(y)$ is necessarily non-negative and non-decreasing. We also choose a constant $K > 0$. We design a mechanism for the penalty information to be passed along the network and aggregated at the sinks, and the sinks will adjust their requests accordingly.

The modified primal problem is:

$$\begin{aligned}
(\tilde{P}) \quad & \max \sum_{v \in \mathcal{D}} U_v(x_v) - K \sum_{r \in \tilde{\mathcal{R}}} P_r \left(\sum_{v \in V_r} x_v \right) \\
\text{subject to} \quad & x_i = \max_{j \in \mathcal{O}_v} x_j \alpha_{vj} / \beta_{vj}, \quad \forall i \in \mathcal{S} \cup \mathcal{P}, \\
& x_v \geq 0, \quad \forall v \in \mathcal{V}, \\
& x_v \leq R_v, \quad \forall v \in \mathcal{D}.
\end{aligned}$$

We will solve it using a two-phase iterative algorithm. Each iteration consists of a bottom-up phase and a top-down phase.

Top-down phase: At the beginning of each iteration, each source $v \in \mathcal{S}$ is assigned a penalty $p_v(x_v)$, and each server $r \in \mathcal{R}$ assigns a penalty $p_r(\sum_{v \in V_r} x_v)$ to all the nodes $v \in V_r$. These penalties are propagated down the task graph from the sources. They are added together at joins and split at forks as described below.

The crucial thing is that a node i should only send penalties to the children that are “responsible” for the value of x_i . These are the nodes that satisfy the max in (20). So a node j is only responsible for the value of x_i if $x_i = x_j \alpha_{ij} / \beta_{ij}$. Among these responsible children, the aggregated penalty can be split among them arbitrarily. To be more precise, for each iteration, any choice of splitting factors z_{ij} for all edges (i, j) are allowed, so long as they satisfy:

$$z_{ij} \geq 0, \quad \forall (i, j) \in \mathcal{E} \quad (21)$$

$$\sum_{j \in \mathcal{O}_i} z_{ij} = 1, \quad \forall i \in \mathcal{S} \cup \mathcal{P} \quad (22)$$

$$z_{ij} = 0 \text{ if } x_i > x_j \alpha_{ij} / \beta_{ij}, \quad \forall (i, j) \in \mathcal{E} \quad (23)$$

The algorithm at each node v is: it receives a penalty value from each parent $i \in \mathcal{I}_v$ and multiplies it by α_{iv} . The node then adds them together with its own penalty. Denote the sum by q_v . The node then sends $z_{vj} q_v / \beta_{vj}$ to each child $j \in \mathcal{O}_v$.

Consider the case in Figure 14(b) for example, if node j_2 receives penalty 3 from parent node i_1 and penalty 5 from parent node i_2 , and it has a penalty of 1 by itself,

then $q_{j_2} = 3\alpha_{i_1j_2} + 5\alpha_{i_2j_2} + 1$, and it sends $z_{j_2k}q_{j_2}/\beta_{j_2k}$ to its child node $k \in \mathcal{O}_{j_2}$.

Note that in the algorithm, neighboring nodes only need to exchange requested rates and aggregated penalties. They do not need to exchange the α and β parameters.

At the end of the top-down phase, each sink $v \in \mathcal{D}$ computes its aggregated penalty q_v . It then updates its requested rate x_v , with step size $\eta_k > 0$, as follows,³

$$x_v^{(k+1)} = [x_v^{(k)} + \eta_k(U'_v(x_v^{(k)}) - Kq_v^{(k)})]_0^{R_v} \quad (24)$$

Bottom-up phase to determine resource allocation:

Notice the importance of (20). Once the output rates x_v for all sinks $v \in \mathcal{D}$ are chosen, we can determine the desired resource allocation at all nodes bottom up using (20) by induction.

- 1). Each sink sends $x_v\alpha_{iv}$ to its parent i , for all $i \in \mathcal{I}_v$.
- 2). A non-sink node $v \notin \mathcal{D}$ receives a value from each child $j \in \mathcal{O}_v$ and divides it by β_{vj} . It then chooses the maximum value to be x_v . Node v then sends $x_v\alpha_{iv}$ to each parent $i \in \mathcal{I}_v$.

3.2.3.1 Convergence

Theorem 16. *If we choose η_k such that $\lim_{k \rightarrow \infty} \eta_k = 0$, $\sum_{k=0}^{\infty} \eta_k = \infty$, then $\{x_v^{(k)}\}$ converges to the optimal solution of the primal problem (\tilde{P}) . On the other hand, we can choose a small constant η_k for $\{x_v^{(k)}\}$ to converge to any small neighborhood of the optimal solution of (\tilde{P}) .*

Proof. See the appendix. □

The next theorem states that certain choices of penalty terms yield the same solution to (\tilde{P}) as (P) .

³The notation $[x]_a^b$ stands for $\min(\max(x, a), b)$.

Theorem 17. *Assume that the utility functions have bounded first derivatives, i.e. $U'_v(x_v) < A$ for $x_v \in [0, R_v]$, $\forall v \in \mathcal{D}$, where A is a constant. If we choose the penalty functions $P_r(y) = \max(0, y - R_r)$, and $p_r(y) = 1_{y \geq R_r}$, then for sufficiently large K , the primal problems (P) and (\tilde{P}) have the same solutions.*

Proof. See the appendix. □

The penalty rate function $p_r(y) = 1_{y \geq R_r}$ in Theorem 17 has a simple interpretation. A source $v \in \mathcal{S}$ receives penalty 1 if requested rate exceeds maximum input rate, and all tasks in a server $r \in \mathcal{R}$ receive penalty 1 whenever the sum of the resource allocation exceeds R_r .

3.2.3.2 Implementation

Now we describe how the primal algorithm can be implemented in a network. Here we use constant step size η . We use the penalty term in Theorem 17. For the splitting factors z_{ij} , we use only 0-1 splits. In this case, if there are multiple children satisfying the max in (20), only one of them (selected randomly by the algorithm) is penalized by receiving all the penalties from the parent.

Each server $r \in \mathcal{R}$ stores penalty q_r . Each node $v \in \mathcal{V}$ stores rate x_v and cumulative penalty q_v . Each node communicates only with its parents and children. Each parent sends downward control packets (DCPs) to its children, and each child sends upward control packets (UCPs) to its parents. A DCP contains penalty information in a \bar{q} field, and a UCP contains rate information in a \bar{x} field. As we will observe shortly, these values already account for the α and β parameters, so they differ slightly from the x and q defined in the algorithm. This removes the need to communicate the α and β parameters explicitly. In the protocol description we use \bar{x}_j to indicate the \bar{x} field in a UCP received from a child j , and \bar{q}_i to indicate the \bar{q} field in a DCP received from a parent i .

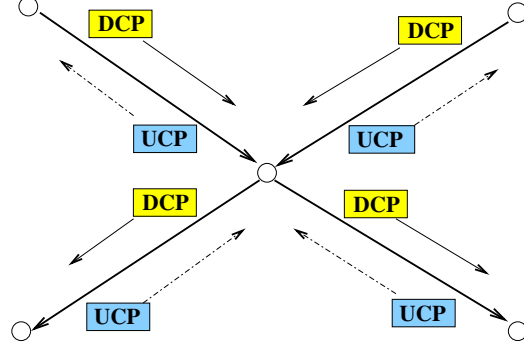


Figure 15: Primal Algorithm Protocol

These UCP and DCP control packets do not have to be separate packets. Their information can be piggy-backed on actual data packets and acknowledgment packets.

Here we present a synchronized protocol where the tasks and servers maintain iteration counters, but the control packets do not have iteration number fields. We assume that each UCP and DCP is transmitted reliably and exactly once, and the iteration numbers of all nodes and servers are kept in sync, so there is no need to carry iteration numbers in control packets. For more flexibility we can add iteration number fields to DCPs and UCPs to allow each control packet to be sent multiple times to ensure delivery, and the protocols can be easily adjusted accordingly.

Algorithm for source $v \in \mathcal{S}$

On receiving UCPs from all children:

1. Read the \bar{x} field to get the rate requested by children. Update $x_v \leftarrow \max_{j \in \mathcal{O}_v} \bar{x}_j / \beta_{vj}$.

2. Update the penalty as:

$$q_v \leftarrow \begin{cases} 1 & \text{if } x_v > R_v \\ 0 & \text{if } x_v \leq R_v \end{cases}$$

.

3. Send a DCP to each child $j \in \mathcal{O}_v$ with the field \bar{q} set to q_v / β_{vj} .

Algorithm for sink $v \in \mathcal{D}$

On receiving DCPs from all parents:

1. Read the \bar{q} field to get the current penalty from parents, update $q_v \leftarrow \sum_{i \in \mathcal{I}_v} \bar{q}_i \alpha_{iv}$.
2. Update the rate as:

$$x_v = [x_v + \eta(U'_v(x_v) - Kq_v)]_0^{R_v}$$

3. Send a UCP to each parent $i \in \mathcal{I}_v$, setting the field \bar{x} to $x_v \alpha_{iv}$.

Algorithm for task $v \in \mathcal{S}$

On receiving UCPs from all children:

1. Read the \bar{x} field to get the rate requested by each child
 - (a) Update the rate as: $x_v \leftarrow \max_{j \in \mathcal{O}_v} \bar{x}_j / \beta_{vj}$.
 - (b) Update max children as:

$$\mathcal{O}_v^{max} \leftarrow \{j : j \in \mathcal{O}_v, x_v = \bar{x}_j / \beta_{vj}\}$$

2. Increment iteration number $n_v \leftarrow n_v + 1$
3. Send an UCP to each parent $i \in \mathcal{I}_v$ with the field \bar{x} set to $x_v \alpha_{iv}$.

On receiving DCPs from all parents:

1. Wait until the server iteration counter has the same value as the node iteration counter, i.e. $n_{r_v} = n_v$.
2. Read the \bar{q} fields to get the penalties from the parents.
 - (a) Update the penalty as: $q_v \leftarrow \sum_{i \in \mathcal{I}_v} \bar{q}_i \alpha_{iv} + q_{r_v}$.
2. Randomly choose a child $j \in \mathcal{O}_v^{max}$ and send it a DCP with \bar{q} field set to q_v / β_{vj} . For all other children, send a DCP with \bar{q} field set to 0.

Algorithm for server $r \in \mathcal{R}$

When task iteration counters for all tasks on the server have higher values than the server iteration counter, i.e. $n_v = n_r + 1, \forall v \in V_r$

1. Update the penalty as:

$$q_r \leftarrow \begin{cases} 1 & \text{if } \sum_{v \in V_r} x_v > R_r \\ 0 & \text{if } \sum_{v \in V_r} x_v \leq R_r \end{cases}$$

.

2. increment iteration number: $n_r \leftarrow n_r + 1$.

The implementation protocol described above is a synchronous protocol, in the sense that the x 's and q 's are updated alternatively as described in the algorithm. There is no need to use large counters for iteration numbers. Small cyclic counters are sufficient to distinguish one iteration from the next.

A more robust approach is to use an asynchronous protocol. There is no iteration counter. The sources penalties, servers penalties and sink rates are all updated periodically and independently. A task does not need to wait for the server to update penalty. Typically, the asynchronous algorithms converge when the corresponding synchronous algorithms do, under reasonable assumptions. However, proof of the convergence of the asynchronous algorithm is outside of the scope of this paper.

3.2.4 Dual Algorithm

The key idea of the dual algorithm is to assign shadow prices to the flow constraints so that local resource allocation decisions can be guided by price differences. In order to introduce Lagrangian multipliers for the flow constraints in (P), we need to consider a slightly different primal problem so that we can apply dualization. We replace the flow constraint (20) by (19). We also replace x by y to make this distinction. The

primal problem (P) then becomes the following:

$$(P') \quad \max \sum_{v \in \mathcal{D}} U_v(y_v)$$

$$\text{subject to } y_i \beta_{ij} \geq y_j \alpha_{ij}, \quad \forall (i, j) \in \mathcal{E}, \quad (25)$$

$$\sum_{v \in V_r} y_v \leq R_r, \quad \forall r \in \tilde{\mathcal{R}}, \quad (26)$$

$$y_v \geq 0, \quad \forall v \in \mathcal{V},$$

$$y_v \leq R_v, \quad \forall v \in \mathcal{D}.$$

The solutions for (P') are not unique, but they take unique values for $y_v, v \in \mathcal{D}$. We denote these unique values as a vector $y_{\mathcal{D}}^*$. It is straightforward to prove that the objective functions for (P) and (P') must achieve the same maximum value, and thus the solutions agree for $v \in \mathcal{D}$. We state the theorem without proof.

Theorem 18. *The solutions for (P) and (P') agree for $v \in \mathcal{D}$, i.e. $y_{\mathcal{D}}^* = x_{\mathcal{D}}^*$.*

We now add Lagrangian multipliers $p_{ij} \geq 0$ for all $(i, j) \in \mathcal{E}$ for the flow constraints (25). The resource constraints (26) remain in the dual problem.

$$\begin{aligned} L_1(y, p) &= \sum_{v \in \mathcal{D}} U_v(y_v) + \sum_{(i, j) \in \mathcal{E}} p_{ij} (y_i \beta_{ij} - y_j \alpha_{ij}) \\ &= \sum_{v \in \mathcal{D}} (U_v(y_v) - \sum_{i \in \mathcal{I}_v} p_{iv} \alpha_{iv}) \\ &\quad + \sum_{r \in \tilde{\mathcal{R}}} \sum_{v \in \mathcal{V}_r} (y_v (\sum_{j \in \mathcal{O}_v} p_{vj} \beta_{vj} - \sum_{i \in \mathcal{I}_v} p_{iv} \alpha_{iv})) \end{aligned}$$

So the dual problem is

$$(D1) \quad \min_{p \geq 0} D(p)$$

where the dual objective function $D(p)$ is given as

$$D(p) = \max L_1(y, p)$$

subject to

$$\begin{aligned} y_v &\geq 0, \quad \forall v \in \mathcal{V}, \\ \sum_{v \in V_r} y_v &\leq R_r, \quad \forall r \in \tilde{\mathcal{R}}, \\ y_v &\leq R_v, \quad \forall v \in \mathcal{D}. \end{aligned}$$

$D(p)$ can be decomposed as

$$D(p) = \sum_{v \in \mathcal{D}} \tilde{D}_v(p) + \sum_{r \in \tilde{\mathcal{R}}} \tilde{D}_r(p)$$

where $\tilde{D}(p)$ is given as

$$\tilde{D}_v(p) = \max_{0 \leq y_v \leq R_v} U_v(y_v) - y_v \sum_{i \in \mathcal{I}_v} p_{iv} \alpha_{iv}, \quad \forall v \in \mathcal{D}, \quad (27)$$

$$\begin{aligned} \tilde{D}_r(p) &= \max_{\substack{0 \leq y_v \\ \sum_{v \in V_r} y_v \leq R_r}} \sum_{v \in V_r} y_v \left(\sum_{j \in \mathcal{O}_v} p_{vj} \beta_{vj} - \sum_{i \in \mathcal{I}_v} p_{iv} \alpha_{iv} \right), \\ &\quad \forall r \in \tilde{\mathcal{R}}. \end{aligned} \quad (28)$$

The solution for $\tilde{D}(p)$ is

- For sink $v \in \mathcal{D}$, we allocate resource according to the price.

$$y_v = [(U'_v)^{-1}(\sum_{i \in \mathcal{I}_v} p_{iv} \alpha_{iv})]_0^{R_v}. \quad (29)$$

- For source $v \in \mathcal{S}$, we always set $y_v = R_v$ since $\sum_{j \in \mathcal{O}_v} p_{vj} \beta_{vj}$ is always non-negative.
- For each server $r \in R$, we evaluate the price differences d_v for all tasks $v \in V_r$, and allocates all resources to the task with maximum positive price difference. Here d_v is defined as

$$d_v := \sum_{j \in \mathcal{O}_v} p_{vj} \beta_{vj} - \sum_{i \in \mathcal{I}_v} p_{iv} \alpha_{iv}. \quad (30)$$

So the server picks a $v' \in V_r$ such that $d_{v'} = \max_{v \in V_r} d_v$, and set

$$y_v = \begin{cases} R_r & v = v' \text{ and } d_{v'} \geq 0 \\ 0 & o.w. \end{cases}$$

The shadow prices p are updated using the standard subgradient method with step size $\eta_k > 0$:⁴

$$p_{ij}^{(k+1)} = [p_{ij}^{(k)} - \eta_k(y_i^{(k)}\beta_{ij} - y_j^{(k)}\alpha_{ij})]^+$$

3.2.4.1 Convergence

Theorem 19. *If we choose η_k such that $\lim_{k \rightarrow \infty} \eta_k = 0$, $\sum_{k=0}^{\infty} \eta_k = \infty$, then $\{y_{\mathcal{D}}^{(k)}\}$ converges to the optimal solution of the primal problem $y_{\mathcal{D}}^*$. On the other hand, we can choose a small constant η_k for $\{y_{\mathcal{D}}^{(k)}\}$ to converge to any small neighborhood of $y_{\mathcal{D}}^*$.*

Proof. See the appendix. □

3.2.4.2 Deriving real rates

Although $\{y_v\}_{v \in \mathcal{D}}$ converges to the optimal output rates, $\{y_v\}_{v \notin \mathcal{D}}$ only takes extreme values of 0 or R_{r_v} . Therefore a parent node can get a full resource allocation while a child node gets no resources, or vice versa. We refer to this situation as a "broken pipeline", as $\{y_v\}$ does not form a capacity pipeline for the data to flow from source to sink.

Therefore $\{y_v\}$ cannot be used as resource allocations in a pipelined algorithm, as defined in the beginning of Section 3.2.2. We need to calculate the resource allocations $\{x_v\}$ separate from $\{y_v\}$. For sinks $v \in \mathcal{D}$, we can set $x_v = y_v$, thanks to Theorem 18. For non-sink nodes, we use the bottom-up phase similar to that of the primal algorithm to derive $\{x_v\}_{v \notin \mathcal{D}}$. Therefore the algorithm consists of two parts.

⁴The notation $[x]^+$ stands for $\max(x, 0)$.

One part is the mechanism for $\{y_v\}$ and $\{p_{ij}\}$ to be updated alternatively through local information exchange. The other part is the bottom-up mechanism to calculate $\{x_v\}$ according to $\{y_v\}$. These two parts can run on separate schedules.

3.2.4.3 Variations

The dual algorithm presented above is based on (D1), which dualizes just the flow constraints (25). A server needs to compare the pricing differences for all the tasks it hosts and allocates its resource to the one with the maximum pricing difference. Depending on the computing capabilities of local servers, we can dualize more (resp. less) constraints so that less (resp. more) computation is carried out locally. This will result in several variants of the dual algorithm as follows.

Variant 1: If we want an algorithm where the servers do not need to coordinate between the tasks, we could do as follows. We add Lagrangian multipliers q_r for all $r \in \tilde{\mathcal{R}}$ for the resource constraints (26), in addition to p_{ij} for all $(i, j) \in \mathcal{E}$ for the flow constraints (25). The Lagrangian is then

$$\begin{aligned}
L_2(y, p, q) &= \sum_{v \in \mathcal{D}} U_v(y_v) + \sum_{(i, j) \in \mathcal{E}} p_{ij}(y_i \beta_{ij} - y_j \alpha_{ji}) + \sum_{r \in \tilde{\mathcal{R}}} q_r (R_r - \sum_{v \in V_r} y_v) \\
&= \sum_{v \in \mathcal{D}} U_v(y_v) + \sum_{(i, j) \in \mathcal{E}} p_{ij}(y_i \beta_{ij} - y_j \alpha_{ij}) + \sum_{r \in \tilde{\mathcal{R}}} q_r R_r - \sum_{v \in S \cup \mathcal{P}} q_{r_v} y_v \\
&= \sum_{v \in \mathcal{D}} (U_v(y_v) - \sum_{i \in \mathcal{I}_v} p_{iv} \alpha_{iv}) + \sum_{v \in S \cup \mathcal{P}} y_v (\sum_{j \in \mathcal{O}_v} p_{vj} \beta_{vj} - \sum_{i \in \mathcal{I}_v} p_{iv} \alpha_{vi} - q_{r_v}) \\
&\quad + \sum_{r \in \tilde{\mathcal{R}}} q_r R_r,
\end{aligned}$$

and the dual problem becomes

$$(D2) \quad \min_{p, q \geq 0} \left(\max_y L_2(y, p, q) \right)$$

where the un-dualized constraints remain in $\max_y L_2(y, p, q)$.

Note that the new Lagrangian is again separable and the last term is a constant when q is fixed. The solution to (D2) is similar to that of (D1) except that no coordination among the tasks on each server is needed. Each task $v \in \mathcal{P}$ only needs

to know its own price pressure and the price on the resource node to determine its own allocation.

Variation 2: We can also dualize fewer constraints so that servers perform more local computations. We now only assign Lagrangian multipliers p_{ij} to flow constraints that cross servers. Let $\mathcal{E}_0 = \{(i, j) | r_i = r_j, i \in \mathcal{P}, j \in \mathcal{P}\}$, and $\mathcal{E}_1 = \mathcal{E} \setminus \mathcal{E}_0$. Assign Lagrangian multipliers p_{ij} only for $(i, j) \in \mathcal{E}_1$. The Lagrangian is

$$\begin{aligned} L_3(y, p) &= \sum_{v \in \mathcal{D}} U_v(y_v) + \sum_{(i, j) \in \mathcal{E}_1} p_{ij} (y_i \beta_{ij} - y_j \alpha_{ij}) \\ &= \sum_{v \in \mathcal{D}} (U_v(y_v) - \sum_{i \in \mathcal{I}_v} y_v p_{iv} \alpha_{iv}) + \sum_{r \in \tilde{\mathcal{R}}} \left(\sum_{\substack{v \in V_r \\ (v, j) \in \mathcal{E}_1}} y_v p_{vj} \beta_{vj} - \sum_{\substack{v \in V_r \\ (i, v) \in \mathcal{E}_1}} y_v p_{iv} \alpha_{iv} \right) \end{aligned}$$

and the dual problem becomes

$$(D3) \quad \min_{p \geq 0} \left(\max_y L_3(y, p) \right),$$

where all un-dualized flow constraints for \mathcal{E}_0 and the resource constraints remain in $\max_y L_3(y, p)$.

Again the Lagrangian is separable. The solution to (D3) is similar to that of (D1) except that the decomposed dual problem for a server becomes a linear programming problem subject to internal flow constraints, external pricing pressure and its own resource constraint. We expect this approach to lead to faster convergence for situations where the sub-graphs inside servers are deep. Clearly such an approach requires servers to carry out more computation locally, which may be feasible for high power servers such as server clusters in a cloud, but not for networks consisting of low-power low capability servers such as sensors.

Hybrid approach: The three approaches (D1), (D2) and (D3) are not exclusive. A hybrid approach is possible by designing the dualization intelligently according to the computing capabilities of local servers. This is especially attractive for hybrid networks which contain both low-power low capability resources and high-end computing servers.

3.2.4.4 Implementation

Now we describe how the dual algorithm (D1) can be implemented in a network. (D2) and (D3) can be implemented similarly. Here we use constant step size η .

Although the algorithm does not require the $\{y_v\}$ and $\{p_{ij}\}$ to be updated on the same schedule as the $\{x_v\}$, here we present a protocol where they are updated on the same schedule. Each iteration consists of a top-down phase and a bottom-up phase.

Each node $v \in \mathcal{V}$ stores rates y_v , x_v and pricing pressure d_v . The link price p_{ij} is calculated and stored by the parent node i . Each node communicates only with its parents and children. Each parent sends downward control packets (DCPs) to its children, and each child sends upward control packets (UCPs) to its parents. A DCP contain a link price in a \bar{p} field, and a UCP contain rate information in \bar{x} and \bar{y} fields. The \bar{x} and \bar{y} fields already account for the α and β parameters, so they differ slightly from the x_v and y_v defined in the algorithm. This removes the need to communicate the α and β parameters explicitly. However the \bar{p} field does have the same value as the p_{ij} in the algorithm. In the protocol description we use \bar{x}_j and \bar{y}_j to indicate the \bar{x} and \bar{y} fields in a UCP received from a child j , and \bar{p}_i to indicate the \bar{p} field in a DCP received from a parent i .

The flow of the control packets is the same as depicted in Figure 15. The only difference is that the UCP and DCP contain different fields.

These UCP and DCP control packets do not have to be separate packets. Their information can be piggy-backed on the actual data packets and acknowledgment packets.

Here we present a synchronous protocol where the tasks and servers maintain iteration counters, but the control packets do not have iteration number fields. As with the primal algorithm, we could also add iteration number fields to control packets.

Now we describe the algorithm at each node.

Algorithm for source $v \in \mathcal{S}$

On receiving UCPs from all children:

1. Read the \bar{x} and \bar{y} fields to get the rates from each child

(a) Update the link price for each $j \in \mathcal{O}_v$ as:

$$p_{vj} \leftarrow [p_{vj} - \eta(y_v \beta_{vj} - \bar{y}_j)]^+$$

(b) Update actual rate as: $x_v \leftarrow \max_{j \in \mathcal{O}_v} \bar{x}_j / \beta_{vj}$

2. Send a DCP to each child $j \in \mathcal{O}_v$ with the field \bar{p} set to p_{vj} .

Algorithm for sink $v \in \mathcal{D}$

On receiving DCPs from all parents:

1. Read the \bar{p} field to get the incoming link price from each parent, update the new rate as

$$\begin{aligned} y_v &\leftarrow [(U'_v)^{-1}(\sum_{i \in \mathcal{I}_v} \bar{p}_i \alpha_{iv})]_0^{R_v} \\ x_v &\leftarrow y_v \end{aligned}$$

2. Send a UCP to each parent $i \in \mathcal{I}_v$, with the field \bar{x} set to $x_v \alpha_{iv}$ and \bar{y} set to $y_v \alpha_{iv}$.

Algorithm for task $v \in \mathcal{P}$

On receiving DCPs from all parents:

1. Read the \bar{p} field to get link price for each incoming link

(a) Update the price difference as:

$$d_v \leftarrow \sum_{j \in \mathcal{O}_v} p_{vj} \beta_{vj} - \sum_{i \in \mathcal{I}_v} \bar{p}_i \alpha_{iv}$$

2. Increment iteration number $n_v \leftarrow n_v + 1$

3. send a DCP to each child $j \in \mathcal{O}_v$ with the field \bar{p} set to p_{vj} .

On receiving UCPs from all children:

1. Wait until the server iteration counter has the same value as the node iteration counter, i.e. $n_{r_v} = n_v$.

2. Read the \bar{x} and \bar{y} fields to get the rates from each child.

(a) Update the link price for each $j \in \mathcal{O}_v$ as:

$$p_{vj} \leftarrow [p_{vj} - \eta(y_v \beta_{vj} - \bar{y}_j)]^+$$

(b) Update actual rate as: $x_v \leftarrow \max_{j \in \mathcal{O}_v} \bar{x}_j / \beta_{vj}$

3. Send a UCP to each parent $i \in \mathcal{I}_v$ with the field \bar{x} set to $x_v \alpha_{iv}$ and \bar{y} set to $y_v \alpha_{iv}$.

Algorithm for server $r \in \mathcal{R}$

When task iteration counters for all tasks on the server have higher values than the server iteration counter, i.e. $n_v = n_r + 1, \forall v \in V_r$

1. Update $d^{max} = \max_{v \in V_r} d_v$.

2. If $d^{max} < 0$, update $y_v \leftarrow 0$ for all $v \in V_r$.

3. If $d^{max} \geq 0$, choose one $v' \in V_r$ that achieves the maximum pricing pressure, i.e. $d_{v'} = \max_{v \in V_r} d_v$.

(a) update $y_{v'} \leftarrow R_r$.

(b) update $y_v \leftarrow 0$ for all $v \in V_r, v \neq v'$.

2. increment iteration number: $n_r \leftarrow n_r + 1$.

The discussion in Section 3.2.3.2 regarding synchronous v.s. asynchronous protocols also apply here.

3.2.5 Primal vs. Dual

Each type of algorithm has its advantages and disadvantages. As pointed out in [39], the primal algorithms require fewer messages and less memory compared with the dual algorithms, because there is no need to communicate or store pseudo-rates (the

y_v 's). Also, they provide the freedom to select the penalty terms. For example we can use piecewise linear increasing functions to model resource constraints such as "target at 80 with hard limit at 100".

On the other hand, we have pointed out that for dual algorithms, we can choose which constraints to dualize to match the computing capabilities of the servers. If the subgraph inside a server is complex, and the server has sufficient computational resources, we can take advantage of it by only dualizing the flow constraints between servers. This will most likely lead to faster convergence. (In the extreme scenario that all tasks are on the same server, it requires only one iteration.)

3.2.6 Evaluation

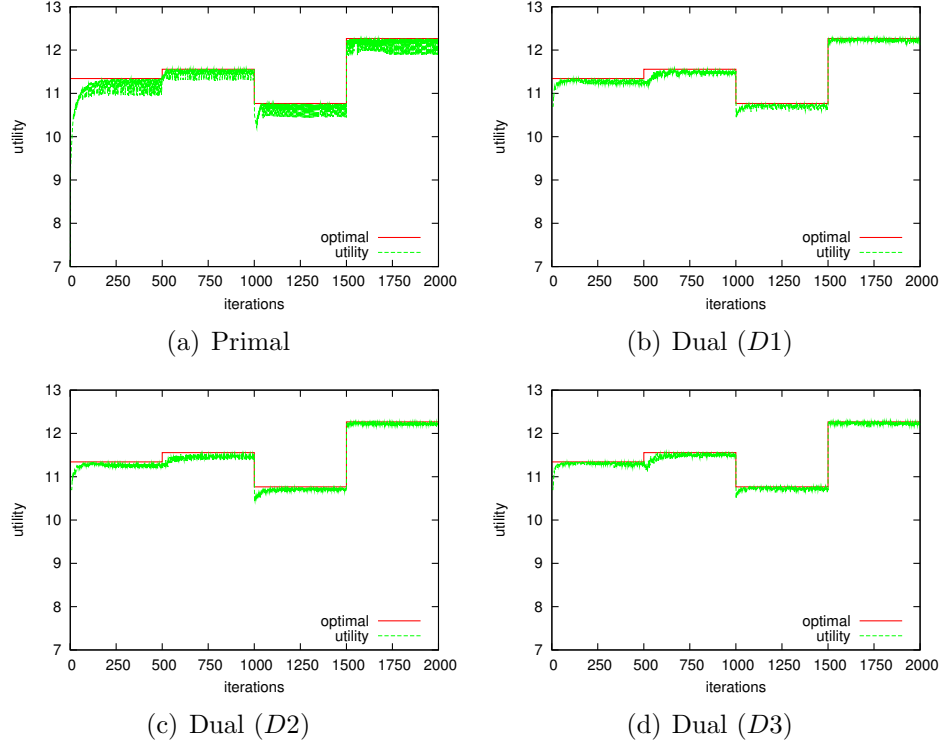


Figure 16: Utility Plots

Since we have already proved the convergence theorems for the primal and dual algorithms, our goal in this section is to evaluate whether they can adapt well in

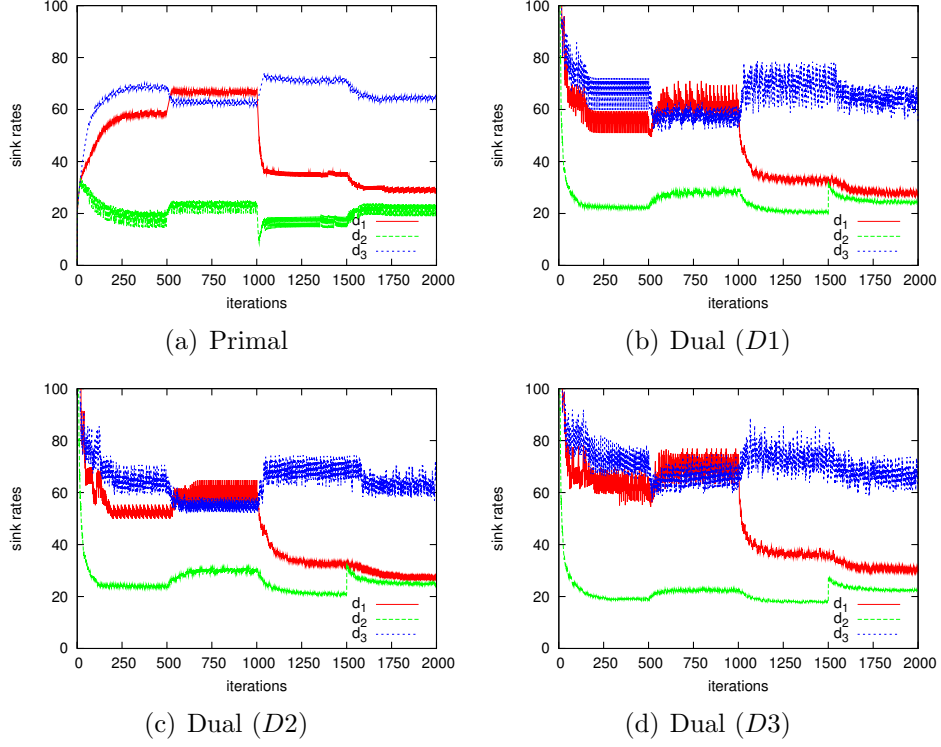


Figure 17: Output Rate Plots

slowly time varying environments.

We ran some simulations for synchronized versions of both the primal and dual algorithms. We used the task graph in Figure 13. This graph contains many contention points. Input streams s_2 and s_3 contend on both servers r_1 and r_3 . Stream s_2 also contend with s_1 on server r_2 . There are multiple forks and joins. All resource constraints are set to 100. We set all the α and β parameters to 1 except $\beta_{p_2p_4} = \beta_{p_5p_6} = \alpha_{p_2p_5} = \alpha_{p_4p_6} = 2$. All utility functions are identical and equal to $\ln(x)$. Such a utility function has been known to represent a proportional fairness [52] on the rate distribution among multiple streams.

We introduce different types of changes to the parameters after some number of iterations to see how the algorithms adapt to slowly time varying environments.

- After 500 iterations, $\alpha_{p_2p_4}$ and $\beta_{p_4p_6}$ are increased from 1 to 1.5.

- After 1000 iterations, the total resource for server r_2 is reduced from 100 to 60.
- After 1500 iterations, the importance of the d_2 output stream is increased by changing its utility function from $\ln(x)$ to $1.5\ln(x)$.

For the primal algorithm we choose a constant step size $\eta = 20$ and constant $K = 0.1$. For the dual algorithms we choose a constant step size $\eta = 0.00002$. Figure 16 plots the utility achieved at each iteration for all the algorithms. The straight lines show the theoretical optimal values. We observe that for all the algorithms the utility converges to a small neighborhood of the optimal value in a few hundred iterations. The algorithms also quickly adapt to changes in slowly time varying environments. After each change, the utility converges to a small neighborhood of the new optimal value in a few hundred or just tens of iterations.

Figure 17 plots sink output rates at each iteration for all the algorithms. We observe that sink rates converge in a few hundreds of iterations after each change for all the algorithms. We also observe that the output rates for d_2 at the 2000th iteration do not agree completely between the algorithms. This is because there can be small tradeoffs between the output rates that barely affect the total utility.

(D3) algorithm appears to converge slightly faster than (D1) and (D2), for example after the change introduced at the 1000th iteration. The effect is barely present because the task graphs inside servers are not deep.

An interesting observation is that for output rates the primal algorithm oscillates more for lower values while the dual algorithms oscillate more for higher values. For utility the primal algorithm oscillates more than the dual algorithms. This phenomenon is due to the increasing and concave objective functions and the update formulae of the algorithms.

For the primal algorithm the sink rates are updated according to (24). Since U_v is a concave function, U'_v is a decreasing function. So the one step change for x_v is larger for smaller x_v . Therefore the primal algorithm has more oscillations for smaller

sink rates. Since the utility function has larger derivative for smaller sink rates, this causes the utility to oscillate more.

On the other hand, for the dual algorithms the sink rates are updated according to (29). Since U_v is a concave function, $(U'_v)^{-1}$ is an increasing functions. So after the same amount of pricing change, the one step change for y_v is larger for larger y_v . Therefore the dual algorithms have more oscillations for larger sink rates.

In conclusion, our simulation has shown that all the algorithms can perform well in slowly time varying environments. We have also tried to explain the different oscillation behaviors between the primal and the dual algorithms.

3.2.7 Concluding Remarks

In this section we studied the resource allocation problem in a processing network where the processing nodes can involve simultaneous fork and join semantics. We formulated the problem as a convex optimization problem and proposed several variants of distributed primal and dual algorithms that achieve the optimal global system utility. We've proven the convergence results and demonstrated the performance via simulation.

We've mentioned that the choice among the variants of the dual algorithms depends on the actual application and computing capabilities of local servers. Similarly, choice between the primal and dual algorithms also depends on the application and system capabilities. This study provides some engineering insights on how the resource allocation algorithm can be tailored according the computing and communication capabilities of the underlying network. For future work, we plan to evaluate and compare of performance of these different choices in testbeds and real networks.

Our pipelined algorithms assign resource allocations to tasks to form “capacity-pipelines” for the data to flow through. This approach may not be suitable for situation where not all links can be active at the same time, for example in a multihop

wireless network. The back-pressure based algorithms presented in the next section are suitable for such situations.

3.3 General fork and join network

In this section we study the resource allocation problem on a general fork and join processing network. Most of this work has been presented in [80].

3.3.1 System Description

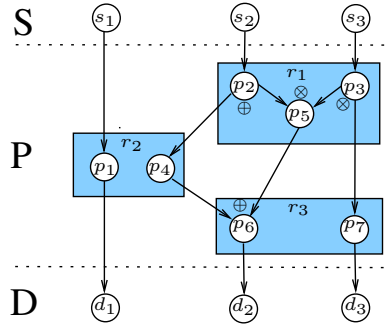


Figure 18: Example Task Graph

We represent the various (and possibly interconnected) data intensive computing applications by a logical task graph, $\mathcal{G}_0 = (\mathcal{V}_0, \mathcal{E}_0)$, where $\mathcal{V}_0 = \mathcal{S}_0 \cup \mathcal{P}_0 \cup \mathcal{D}_0$ consists of three types of nodes: (data) sources \mathcal{S}_0 , (processing) tasks \mathcal{P}_0 and (data) sinks \mathcal{D}_0 . Data streams enter the system at the sources and exit at the sinks. \mathcal{E}_0 is a set of directed edges indicating the direction of data flow in between the various nodes. We assume \mathcal{G}_0 is a *directed acyclic graph* (DAG), but do not require \mathcal{G}_0 to be a connected graph. We assume that each (non-source) node is associated with a unique join semantic that defines how data from various input links are processed. Similarly, each (non-sink) node is associated with a unique fork semantic that defines how output data are generated on the output links. We next introduce the different semantics associated with the various forks and joins.

3.3.1.1 Fork and Join Semantics

A join occurs when a node v has more than one incoming edges, and a fork occurs when a node v has more than one outgoing edges. We define two types of fork semantics and two types of join semantics.

Synchronous join: also called *and-join*, or \otimes -join. An \otimes -join requires data from all input streams simultaneously, at some fixed ratio. For example, an \otimes -join task may merge an audio stream and a video stream into a multimedia stream.

Asynchronous join: also called *or-join*, or \oplus -join. An \oplus -join processes data independently from each input stream. For example, an \oplus -join with two input links processes data from either the left link or from the right link at one time but not both simultaneously. Such asynchronous join is common in multi-path routing where packets from any input link can be immediately processed by the router.

Synchronous fork: also called *and-fork*, or \otimes -fork. An \otimes -fork produces multiple output streams simultaneously, at some fixed ratio. For example, an \otimes -fork task may split a multimedia stream into an audio stream and a video stream for further processing.

Asynchronous fork: called *or-fork*, or \oplus -fork. An \oplus -fork chooses one of the output links to send the data it produces. For example, an \oplus -fork with two output links will send output data over either the left output link or the right output link at one time but not both simultaneously.

We assume that each task is associated with a unique join and a unique fork type, denoted by $v(\cdot, \cdot)$, where the first argument specifies its join type and second argument its fork type. For example, $v(\oplus, \otimes)$ denotes a task with an \oplus -join and an \otimes -fork. When the task has a single input link, it can be viewed as the degenerative case of either the \otimes -join or the \oplus -join, and one does not need to specify the join type. Similarly for the case of a single output link.

In practice, there can be more complicated *and* and *or* logic combinations associated with each task. For example, a task may do an and-join \otimes of two input streams and then do an or-join \oplus with a third input stream. Such a situation can be converted to our representation above by introducing dummy tasks so that each task is associated with only one type of join and one type of fork. In the rest paper, we simply assume that each task is associated with a unique join logic and a unique fork logic.

3.3.1.2 Resource Constraints

Assume there is a finite set of resources \mathcal{R} . We view each resource as a “server” $r \in \mathcal{R}$, which has limited capacity R_r . That is, a server can be a computer with max processing rate R_r or a communication link with limited bandwidth R_r . Using similar techniques as in [26], one can convert link capacity constraints into node capacity constraints by introducing virtual nodes. Thus we can handle simultaneously resource constraints on communication bandwidth and computing resources.

Assume the tasks have been assigned to the various servers, where a server could be shared by multiple tasks. Figure 18 offers an example where there are 3 servers r_1, r_2 and r_3 , represented by rectangles. There are 3 sources, 3 sinks and 7 tasks. The fork and join types have been specified. Based on the task to server deployment, p_2, p_3 , and p_5 need to compete for resource r_1 , p_1 and p_4 compete for resource r_2 , whereas p_6 and p_7 compete for resource r_3 .

We assume that, for each sink $v \in \mathcal{D}_0$, there is a utility function $U_v(x_v)$ associated with delivering output to sink v at rate x_v , where $U_v(\cdot)$ is non-negative, increasing, strictly concave and differentiable. We focus on the resource allocation problem with the objective of maximizing the sum of the utilities for all the flow output rates, i.e. $\sum_{v \in \mathcal{D}_0} U_v(x_v)$.

For simplicity of exposition, for now we assume that each task consumes one unit

of input and produces one unit of output with one unit of resource. Thus the amount of resource allocated to each task directly maps to the amount of input data it will consume from each upstream link and the amount of output it will produce for each downstream link. In Section 3.3.6.2 we will deal with the general case that the flows may expand or shrink after processing, that is, data may join or fork at certain ratios, and different task requires different amount of resources.

The different types of semantics and processing requirements introduce complex interdependencies among the various data flows. In order to fully describe the flow constraints of the resource allocation problem, one has to enumerate the fork and join semantics of all upstream and downstream tasks. It is thus difficult to provide a general mathematical formulation of the problem. We next present a unified modeling framework that will eliminate the need for such enumeration.

3.3.2 A Unified Modeling Framework

In this section, we present a unified modeling framework that can represent all combinations of the various fork and join semantics, and formulate the resource allocation problem into an elegant optimization problem without the need for enumeration.

3.3.2.1 Flow Constraints and Lossiness

Before we introduce the framework, we first need to understand the basic flow characteristics of fork and join processing networks.

Let us first consider the case where there are only \otimes -forks and \otimes -joins. In this case, it suffices to introduce one decision variable for each task. Let x_v denote the amount of resource allocated to task $v \in \mathcal{P}_0$.

One important characteristic of a fork and join processing network is that it can be “lossy”. This has been described in Section 3.2.1. In such a “lossy” network, the flow constraint is that a child node cannot consume more than what the parent node

can output, i.e., we must have

$$x_i \geq x_j, \quad \forall (i, j) \in \mathcal{E} \quad (31)$$

Now consider the presence of \oplus -forks and \oplus -joins. When there is an \oplus -fork, we need to know the amount produced for each outgoing link. We thus introduce *one resource allocation variable per outgoing link* (see cases 3 and 4 in Table 6). When there is an \oplus -join, since the task can consumes data from any of the parents, the flow constraint becomes that the sum of the parents' output is no less than the child's consumption rate (see cases 2 and 4 in Table 6). The two left columns of Table 6 lists all four possible combinations of fork and join type, and the corresponding resource constraints.

We extend the notation x_v to source and sink nodes as well. For source $v \in \mathcal{S}_0$, x_v represents the flow admission rate. For sink $v \in \mathcal{D}_0$, x_v represents the flow output rate.

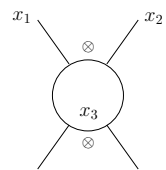
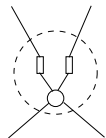
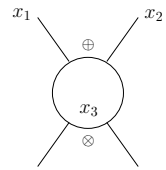
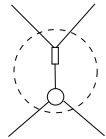
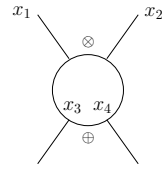
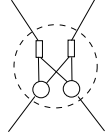
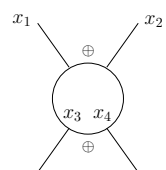
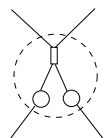
It is clear that, simply based on the graph model \mathcal{G}_0 , one has to enumerate the fork and join semantics of all processing task in order to fully describe the flow constraints of the processing network. We next present a bipartite model that expresses these rich semantics in a unified framework.

3.3.2.2 Bipartite Model

We transform the process graph \mathcal{G}_0 into a bipartite DAG $\mathcal{G} = (\mathcal{A} \cup \mathcal{M}, \mathcal{E})$. We call the two partitions the allocation layer \mathcal{A} and the multiplier layer \mathcal{M} . All paths in \mathcal{G} start and terminate in \mathcal{A} .

\mathcal{A} can be decomposed as $\mathcal{A} = \mathcal{S} \cup \mathcal{P} \cup \mathcal{D}$, where \mathcal{S} denote all nodes with no incoming links, \mathcal{D} all nodes with no outgoing links, and \mathcal{P} the rest of the nodes in \mathcal{A} . The nodes in \mathcal{A} correspond to the sources, sinks, and resource allocation decision variables for tasks, so we call it the allocation layer. The nodes in \mathcal{M} correspond to the flow constraints. Later we will see that each node in \mathcal{M} corresponds to a Lagrangian

Table 6: Task Graph and Bipartite Graph

\mathcal{G}_0	Flow constraints	\mathcal{G}
	$x_1 \geq x_3$ $x_2 \geq x_3$	
	$x_1 + x_2 \geq x_3$	
	$x_1 \geq x_3 + x_4$ $x_2 \geq x_3 + x_4$	
	$x_1 + x_2 \geq x_3 + x_4$	

multiplier that appears in the dual approach of the optimization problem we will formulate (Section 3.3.4.1), and has a shadow queue and a real queue associated with it by the distributed algorithm. Hence we call \mathcal{M} the multiplier layer or queue layer. Each node $v \in \mathcal{V}_0$ maps to multiple nodes in \mathcal{A} and \mathcal{M} , as shown in the right column of Table 6. The exact rules of transformation are as follows:

Allocation Layer \mathcal{A} ,

- task with \otimes -fork: assign one allocation node to the task (which will be origin

for all the outgoing links)

- task with \oplus -fork: assign one allocation node to each outgoing link
- assign one allocation node to each source or sink.

Multiplier/Queue Layer \mathcal{M} ,

- task or sink with \oplus -join: assign one multiplier node to each task (this will be the destination for all incoming links)
- task or sink with \otimes -join: assign one multiplier node to each incoming link

Links \mathcal{E} ,

- all the links in \mathcal{E}_0 are copied into \mathcal{E}
- inside each task node, create links connecting each multiplier node to each allocation node.

Table 6 illustrates the local transformation from the original graph \mathcal{G}_0 to the bipartite graph \mathcal{G} in all four possible combinations. In the new graph \mathcal{G} , the small circles represent the allocation nodes, and the small rectangles represent the multiplier nodes. All combinations of various fork and join semantics in the original task graph are now fully represented in the bipartite graph, eliminating the need to enumerate the semantics associated with each task.

Note that, in the bipartite graph, *all forks and joins at the allocation nodes are synchronous, while all forks and joins at the multiplier/queue nodes are asynchronous*. This is useful for designing the distributed algorithms in later sections.

We assign one resource allocation variable x_a to each allocation node $a \in \mathcal{A}$. The flow constraints are,

$$\sum_{a \in P_m} x_a - \sum_{a \in C_m} x_a \geq 0, \quad \forall m \in \mathcal{M}. \quad (32)$$

Here, we use P_m and C_m to denote respectively the set of parent nodes and the set of children nodes of any $m \in \mathcal{M}$. We define P_a and C_a for $a \in \mathcal{A}$ similarly. Since \mathcal{G} is bipartite, for $m \in \mathcal{M}$, $P_m, C_m \in \mathcal{A}$, and for $a \in \mathcal{A}$, $P_a, C_a \in \mathcal{M}$.

Not all bipartite graphs \mathcal{G} can result from the above transformation on a graph \mathcal{G}_0 . For example, in a graph \mathcal{G} transformed from some \mathcal{G}_0 , if a set of multiplier nodes share one common child node, they must share the same set of children nodes. This is evident from Table 6. However, we will formulate the resource allocation problem on any bipartite DAG \mathcal{G} , with the only restriction that all paths start and terminate in \mathcal{A} .

3.3.3 The Resource Allocation Problem

Denote A_r the set of allocation nodes on server r . Recall that each server r has R_r units of resource. With slight abuse of notation, we use R_a to represent the maximum flow input rate for each source $a \in \mathcal{S}$, and the maximum desired flow output rate for each sink $a \in \mathcal{D}$. See Table 7 for a complete list of notations.

We now can abstract the resource allocation problem for a general fork and join processing network as the following convex optimization problem for the bipartite model:

$$(P) \quad \max \sum_{a \in \mathcal{D}} U_a(x_a) \quad (33)$$

subject to

$$\sum_{a \in P_m} x_a - \sum_{a \in C_m} x_a \geq 0, \forall m \in \mathcal{M}, \quad (34)$$

$$\sum_{a \in A_r} x_a \leq R_r, \forall r \in \mathcal{R}, \quad (35)$$

$$x_a \leq R_a, \forall a \in \mathcal{S} \cup \mathcal{D}, \quad (36)$$

$$x_a \geq 0, \forall a \in \mathcal{A} \quad (37)$$

Note that the decision variable $\mathbf{x} := (x_a, a \in \mathcal{A})$ carries a rich set of decisions at

different nodes. For source nodes, x_a corresponds to the admission rate at each source $a \in \mathcal{S}$. For sink nodes, x_a corresponds to the achieved flow rate at each sink $a \in \mathcal{D}$. For allocation nodes, x_a represents the amount of resource allocated to processing node $a \in \mathcal{P}$.

Due to the strict concavity of the utility functions, the optimal solution for the primal problem is unique for $a \in \mathcal{D}$. Let us call it $x_{\mathcal{D}}^*$.

In the coming section, we will see that the multiplier layer \mathcal{M} is not only useful in providing a unified mathematical formulation of the resource allocation problem, but also helpful in deriving distributed shadow queue based solutions to problem (P).

Table 7: Notations for model and algorithm

Notation	Meaning
\mathcal{G}_0	the task graph
\mathcal{S}_0, D_0, P_0	sources, sinks, tasks
\mathcal{G}	the bipartite graph
\mathcal{A}	allocation layer
\mathcal{M}	multiplier/queue layer
\mathcal{S}, D, P	sources, sinks, allocation nodes
P_m, P_a	parent nodes of $a \in \mathcal{A}$ or $m \in \mathcal{M}$
C_m, C_a	children nodes of $a \in \mathcal{A}$ or $m \in \mathcal{M}$
x_a	resource allocation for $a \in \mathcal{A}$
U_a	utility function at sink $a \in \mathcal{D}$
\mathcal{R}	resources(servers)
R_r, R_a	resource limit for server $r \in \mathcal{R}$ or $a \in \mathcal{S}, \mathcal{D}$
A_r	The set of all allocation nodes on server r
p_m	Lagrangian multiplier for constraint at m
q_m, \tilde{q}_m	real and shadow queue length at m
η_{ma}	real queue departure from m to child a
φ_{am}	real queue arrival to m from parent a
$\tilde{\eta}_{am}$	shadow queue departure from m to parent a
$\tilde{\varphi}_{ma}$	shadow queue arrival to m from child a
η_a	real flow through a
$\tilde{\varphi}_a$	shadow flow through a
V	a system parameter to control convergence

3.3.4 Distributed Algorithms

In Section 3.3.4.1 we present the standard dual approach to solve the dual problem. In Section 3.3.4.2 we present a distributed back-pressure based algorithm that essentially implements the solution to that dual problem.

3.3.4.1 The Dual Decomposition

Consider the dual problem of the primal problem (P) by assigning Lagrangian multipliers p_m (≥ 0) to the flow constraints (34) for every $m \in \mathcal{M}$. We get the Lagrange function

$$\begin{aligned}
L(x, p) &= \sum_{a \in \mathcal{D}} U_a(x_a) + \sum_{m \in \mathcal{M}} p_m \left(\sum_{a \in P_m} x_a - \sum_{a \in C_m} x_a \right) \\
&= \sum_{a \in \mathcal{D}} (U_a(x_a) - x_a \sum_{m \in P_a} p_m) + \sum_{a \in \mathcal{S}} x_a \sum_{m \in C_a} p_m \\
&\quad + \sum_{r \in \mathcal{R}} \sum_{a \in A_r} x_a \left(\sum_{m \in C_a} p_m - \sum_{m \in P_a} p_m \right)
\end{aligned} \tag{38}$$

The equality is derived through interchanging the summations, then grouping under resources $r \in \mathcal{R}$. So the dual problem is

$$(D) \quad \min_{\mathbf{p} \geq 0} D(p)$$

where the dual objective function $D(p)$ is given as

$$D(p) = \max_{\mathbf{x}} L(x, p) \quad \text{subject to (35)-(37)}$$

From (38), we see that $D(p)$ can be decomposed into two problems:

$$D(p) = \sum_{a \in \mathcal{D} \cup \mathcal{S}} \tilde{D}_a(p) + \sum_{r \in \mathcal{R}} \tilde{D}_r(p),$$

where $\tilde{D}_a(p)$ corresponds to a *Congestion Control Problem* at the source and sink nodes:

$$\tilde{D}_a(p) = \begin{cases} \max_{0 \leq x_a \leq R_a} U_a(x_a) - x_a \sum_{m \in P_a} p_m & \forall a \in \mathcal{D} \\ \max_{0 \leq x_a \leq R_a} x_a \sum_{m \in C_a} p_m & \forall a \in \mathcal{S} \end{cases}. \tag{39}$$

And $\tilde{D}_r(p)$ corresponds to a local *Resource Allocation Problem* at each server $r \in \mathcal{R}$:

$$\tilde{D}_r(p) = \max_{\sum_{a \in A_r} x_a \leq R_r} \sum_{a \in A_r} x_a \left(\sum_{m \in C_a} p_m - \sum_{m \in P_a} p_m \right) \quad \forall r \in \mathcal{R}$$

When the multipliers $(p_m, m \in \mathcal{M})$ are given, the solution to the above congestion control and resource allocation problems are straight-forward:

- For sink node $a \in \mathcal{D}$, we allocate resource according to the price as follows,

$$x_a = [(U'_a)^{-1} \left(\sum_{m \in P_a} p_m \right)]_0^{R_a}, \quad (40)$$

where the notation $[x]_a^b$ stands for $\min(\max(x, a), b)$.

- For source node $a \in \mathcal{S}$, we always set $x_a = R_a$ since $\sum_{m \in C_a} p_m$ is always nonnegative.
- For each server $r \in \mathcal{R}$, we evaluate the price difference d_a for all nodes $a \in A_r$, and allocate all resources to the node with maximum positive price difference.

Here d_a is defined as

$$d_a := \sum_{m \in C_a} p_m - \sum_{m \in P_a} p_m \quad (41)$$

Let $\hat{a} = \operatorname{argmax}_{a \in A_r} d_a$, and set

$$x_a = \begin{cases} R_r & \text{if } a = \hat{a} \text{ \& } d_a > 0 \\ 0 & \text{otherwise} \end{cases}, \quad (42)$$

If more than one node have the maximum positive d_a , the resources can be divided among them arbitrarily.

The standard subgradient method is an iterative algorithm to solve the dual problem (D). For each iteration, the multipliers can be updated as follows:

$$p_m(t+1) = \left[p_m(t) - h(t) \left(\sum_{a \in P_m} x_a(t) - \sum_{a \in C_m} x_a(t) \right) \right]^+ \quad (43)$$

where the notation $[x]^+$ stands for $\max(x, 0)$. Here $h(t)$ is a step size that affects the convergence behavior. We then have the following theorem from the subgradient method. The proof can be found in the appendix.

Theorem 20. *If we choose the step size $h(t)$ such that $\lim_{t \rightarrow \infty} h(t) = 0$ and $\sum_{t=0}^{\infty} h(t) = \infty$, and update $\mathbf{p}(t)$ and $\mathbf{x}(t)$ iteratively according to (43), (40) and (42), then the solution $\{x_{\mathcal{D}}(t)\}$ converges to the optimal solution of the primal problem $x_{\mathcal{D}}^*$. On the other hand, we can choose a small constant $h(t)$ to converge to any small neighborhood of $x_{\mathcal{D}}^*$.*

3.3.4.2 Shadow Queue Based Algorithm

Due to the lossy characteristics of the fork and join processing networks, it is no longer feasible to relate the multipliers to the queue size as done in many past back-pressure algorithms [70, 45, 57, 24]. Since utility is charged at the sinks, update equation (43) suggests that the shadow prices (multipliers) can be related to credits (or tokens) created at the sinks and that flow bottom up towards the sources. That is, a back-pressure algorithm can be applied to the shadow queues on the reversed graph and credits in the shadow queues can be used to guide the processing of the real queues. Such shadow queue based algorithms have been recently developed in [26] for stream processing networks and in [16] for multicast wireless networks. In this section, we present a shadow queue based algorithm for general fork and join networks with arbitrary fork and join semantics, which is built on top of the bipartite graph. We associate a shadow queue \tilde{q}_m and a real queue q_m with each multiplier node $m \in \mathcal{M}$ in graph \mathcal{G} . Credits in the shadow queues move in the reverse direction of the directed graph \mathcal{G} (following a time-slotted back-pressure algorithm on the reversed graph). The shadow queues determine resource allocations to various tasks at each server, thus guiding real data streams flow in the forward direction.

We now describe the shadow queue based algorithm for the resource allocation problem. It is important to remember that under the bipartite representation, all forks and joins at the allocation nodes are synchronous, while all forks and joins at the multiplier/queue nodes are asynchronous. At all times, we also keep in mind

simultaneously the original graph for the real flow and the reversed graph for the shadow flow as shown in Figures 19(a) and 19(b).

The algorithm is time slotted. For any $m \in \mathcal{M}$, let $q_m(t)$ and $\tilde{q}_m(t)$ be the sizes of the real and shadow queues at the beginning of time slot t . For simplicity, we assume that the arrivals at any queue for a time slot will only be available for departure in the next time slot.

Suppose a is a parent node of m and a' is a child node of m in the original graph, as shown in Figure 19(a). Denote $\varphi_{am}(t)$ the amount arriving at the real queue q_m over the link (a, m) , and $\eta_{ma'}(t)$ the amount departing from q_m over the link (m, a') , both during time slot t . Similarly, denote $\tilde{\varphi}_{ma'}(t)$ the amount arriving at the shadow queue \tilde{q}_m over the link (m, a') , and $\tilde{\eta}_{am}(t)$ the amount departing from \tilde{q}_m over the link (a, m) during time slot t . Note that *the subscripts are always in the direction of the edges of the original bipartite graph*. Figure 19(b) illustrates these notations from the viewpoint of an allocation node a .

Our algorithm determines these arrival and departure quantities during each time slot. Over time, the real queues q_m and shadow queues \tilde{q}_m will evolve as follow:

$$\tilde{q}_m(t+1) = \tilde{q}_m(t) - \sum_{a \in P_m} \tilde{\eta}_{am}(t) + \sum_{a \in C_m} \tilde{\varphi}_{ma}(t), \quad \forall m \in \mathcal{M}, \quad (44)$$

$$q_m(t+1) = q_m(t) - \sum_{a \in C_m} \eta_{ma}(t) + \sum_{a \in P_m} \varphi_{am}(t), \quad \forall m \in \mathcal{M}. \quad (45)$$

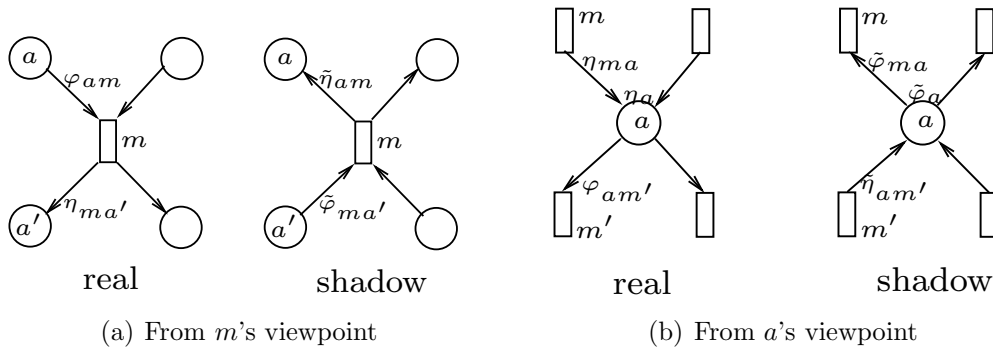


Figure 19: From m 's and a 's viewpoints

The back-pressure algorithm on shadow queues mimics the subgradient method with a constant step size $h \equiv 1/V$, where V is a system parameter. At each iteration, parent allocation nodes (in the original bipartite graph) request credits from children shadow queues \tilde{q}_m (i.e. $a \in P_m$) and move them bottom up (in the reverse direction of the edges in the original graph \mathcal{G}). The algorithm is detailed below:

Back-pressure algorithm on shadow queues:

Step 1. Set $p_m(t) = \tilde{q}_m(t)/V$ for all $m \in \mathcal{M}$ and calculate $x_a(t)$ according to equations (40) and (42) for all $a \in \mathcal{A}$.

Step 2. $\forall a \in \mathcal{A}$: Request $x_a(t)$ credits from each child shadow queue $\tilde{q}_{m'}$. If a child queue does not have enough credits for all parents, the credits can be divided among the parents arbitrarily. (The queue is not required to be emptied in this case.) Take the maximum of the credits received from each children, i.e. $\tilde{\varphi}_a(t) := \max_{m' \in C_a} \tilde{\eta}_{am'}(t)$. Forward the amount to each parent queue \tilde{q}_m of a unless it is a source node.

Step 3. Sink a injects $x_a(t)$ credits and sends to parents, i.e. $\tilde{\varphi}_a(t) = x_a(t)$.

Step 4. Each shadow queue stores all credits received from its children.

Remark: In the above algorithm, the amount of credits issued to a parent node a indicates the amount of output desired from that parent node. Since each allocation node forks its output in a synchronous manner to all its children, taking the maximum (in Step 2) of the credits received from all children ensures that node a will produce enough to satisfy the output requirements desired by its succeeding nodes.

At each iteration, the processing of the real flow is guided by the shadow flow. The amount of real flow to be processed in each iteration is derived based on the following algorithm.

Algorithm for Real Flow Processing:

Step 1. $\forall a \in \mathcal{A}$: Process up to $\tilde{\varphi}_a(t)$ amount of data from each parent (real) queue q_m .

If a parent queue q_m cannot satisfy the processing demand of all its children, its data can be divided among the children arbitrarily. (It need not be optimal in the sense of draining the parents as much as possible.) Since the join at each allocation node is synchronous, a will drain equal amount from each parent. We denote this amount as $\eta_a(t)$. Then the departure amount from each parent queue $\eta_{ma}(t) = \eta_a(t)$. The processed amount is sent to each child queue, unless it is a sink node.

Step 2. Each source a injects $\tilde{\varphi}_a(t)$ amount of real flow, i.e. $\eta_a(t) = \tilde{\varphi}_a(t)$.

Step 3. $\forall m' \in \mathcal{M}$: Receive processed real flow from each parent node $a \in P_{m'}$, up to the amount of credits issued by corresponding shadow queue to a . i.e. $\varphi_{am'}(t) = \min(\eta_a(t), \tilde{\eta}_{am'}(t))$. In other words, if $\eta_a(t) > \tilde{\eta}_{am'}(t)$, discard $\eta_a(t) - \tilde{\eta}_{am'}(t)$. (The amount could also be discarded by parent a before sending to m' .)

Remark: The need to discard a portion of processed data over some links in Step 3 is related to the lossy characteristics discussed earlier in subsection 3.3.2.1. Since each allocation node forks its outputs in a synchronous manner to all children nodes, some children nodes may need to drop the amount exceeding its quota (i.e. the allocated shadow credits). Otherwise the real queue may grow indefinitely. We will show in the next section that the real flow rate delivered to the sink nodes will converge to the optimal rates.

The algorithm implies that real and shadow flow are bounded by the corresponding $x_a(t)$, and the real flow is bounded by the shadow flow,

$$\phi_{am}(t) \leq \tilde{\eta}_{am}(t) \leq x_a(t), \quad \forall m \in \mathcal{M}, \forall a \in P_m \quad (46)$$

$$\eta_{ma}(t) \leq \tilde{\varphi}_{ma}(t) \leq x_a(t), \quad \forall m \in \mathcal{M}, \forall a \in C_m \quad (47)$$

The real departure and shadow arrival are the same for all m linked to the same

child a . The shadow arrival and real arrival follow the max and discard rules.

$$\eta_{ma}(t) = \eta_a(t), \quad \forall a \in \mathcal{A}, \forall m \in P_a \quad (48)$$

$$\tilde{\varphi}_{ma}(t) = \tilde{\varphi}_a(t) = \max_{m' \in C_a} \tilde{\eta}_{am'}(t), \quad \forall a \in \mathcal{A}, \forall m \in P_a \quad (49)$$

$$\varphi_{am}(t) = \min(\eta_a(t), \tilde{\eta}_{am}(t)), \quad \forall a \in \mathcal{A}, \forall m \in C_a. \quad (50)$$

3.3.5 Performance Analysis

In this section we analyze the performance of our algorithm for fluid flows. We first show that the utility based on the credit injection rate is close to the utility evaluated on the optimal output rates, and that the shadow queues are bounded. We then show that the real flow closely tracks the shadow flow, therefore the real flow also achieves optimality and stability.

We first define a few constants. Let M denote the number of \mathcal{M} -layer nodes, A the number of \mathcal{A} -layer nodes, N the number of sink nodes, D the maximum in-degree or out-degree of any node ($D = \max_{n \in \mathcal{G}} \max(|P_n|, |C_n|)$), C the maximum of any resource capacity and input and output rates ($C = \max(\max_{r \in \mathcal{R}} R_r, \max_{a \in \mathcal{S} \cup \mathcal{D}} R_a)$). Therefore all $x_a(t)$ are bounded by C . Let U_{max} denote the maximum value of all the utility function over $[0, C]$ ($U_{max} = \max_{a \in \mathcal{D}} U_a(C)$ since $U_a(x)$ is an increasing function).

We adopt the following notation for time average: For a discrete time series $y(t)$, $\bar{y} \equiv \lim_{T \rightarrow \infty} \frac{1}{T} \sum_{t=0}^{T-1} y(t)$. We also use \bar{y}^{sup} and \bar{y}^{inf} to replace \lim with \limsup and \liminf respectively.

3.3.5.1 Optimality and Stability

Since the shadow queue update formula does not exactly follow the update equation (43) for the multipliers of the dual algorithm, we cannot directly use Theorem 20 to establish optimality. We need to do a separate proof. We use a similar Lyapunov argument as in [57]. We define the Lyapunov function $F(\tilde{q}(t)) \equiv \sum_{m \in \mathcal{M}} \tilde{q}_m^2(t)$. So

the Lyapunov drift is

$$\Delta F(\tilde{q}(t)) \equiv \sum_{m \in \mathcal{M}} \tilde{q}_m^2(t+1) - \tilde{q}_m^2(t) \quad (51)$$

The optimality and stability results on the back-pressure algorithm for the shadow queues are stated below. Remember that for sink nodes the credit injection rate $\tilde{\varphi}_a(t)$ is always the same as its allocation $x_a(t)$.

Theorem 21. *Under the back-pressure algorithm on shadow queues described in section 3.3.4.2, the credit injection rates at all sinks satisfy the following,*

$$\sum_{a \in \mathcal{D}} U_a(\bar{x}_a^{inf}) \geq \sum_{a \in \mathcal{D}} U_a(x_a^*) - \frac{3MD^2C^2}{2V}. \quad (52)$$

Furthermore, the shadow queue lengths are bounded at all times, i.e.

$$\exists B, \quad \sum_{m \in \mathcal{M}} \tilde{q}_m(t) \leq B, \quad \forall t \geq 0 \quad (53)$$

In order to prove the above result, we need the following lemma on discrete time queues.

Lemma 22. *Consider a discrete time queue whose length at time t is $q(t)$. Suppose the arrival rate is $\varphi(t)$ and service rate is $\mu(t)$ at time t , where $\varphi(t) \leq \lambda(t)$, and the actual departure rate $\eta(t)$ satisfies*

$$\eta(t) = \begin{cases} \mu(t) & \text{if } q(t) \geq \mu(t) \\ \leq q(t) & \text{if } q(t) < \mu(t) \end{cases}.$$

Then we have

$$q^2(t+1) - q^2(t) \leq 2\mu_{max}^2 + \lambda_{max}^2 - 2q(t)(\mu(t) - \lambda(t)),$$

where μ_{max} and λ_{max} are upper bounds on $\mu(t)$ and $\lambda(t)$ respectively, i.e., $\mu_{max} \geq \mu(t), \forall t$ and $\lambda_{max} \geq \lambda(t), \forall t$.

The proof of the lemma can be found in the appendix. This lemma can be applied to shadow queue $\tilde{q}_m(t)$, with $\lambda(t) = \sum_{a \in C_m} x_a(t)$ and $\mu(t) = \sum_{a \in P_m} x_a(t)$, and it shows that the Lyapunov function has negative drift. Please see the appendix for the detailed proof for Theorem 21.

3.3.5.2 Real Queue Output Rate

In this section we study the relation between the real flow and the shadow flow. By design the real flow never exceeds the corresponding shadow flow in the opposite direction. We will show that the difference between the two is bounded, as stated in the following theorem.

Theorem 23.

(A): $\sum_t (\tilde{\varphi}_a(t) - \eta_a(t))$ is bounded, $\forall a \in \mathcal{A}$.

(M): $\sum_t (\tilde{\eta}_{am}(t) - \varphi_{am}(t))$ is bounded, $\forall m \in \mathcal{M}, \forall a \in P_m$.

By design, $\eta_a(t) \leq \tilde{\varphi}_a(t)$, and equality always holds at source nodes. We want to show that the total difference between the two is bounded, i.e. a node processes almost the same amount of data as the credits issued to it.

By design, $\varphi_{am}(t) \leq \tilde{\eta}_{am}(t)$. We want to show that the total difference between the two is bounded, i.e. a real queue receives almost the same amount of flow as what leaves the corresponding shadow queue.

We defer the proof for Theorem 23, and first show that given the results of Theorem 23, the optimality and stability of the real queue immediately follows.

Corollary 24. *The real output rates are optimal, and the real queue lengths are bounded. That is,*

$$\sum_{a \in \mathcal{D}} U_a(\overline{\eta}_a^{inf}) \geq \sum_{a \in \mathcal{D}} U_a(x_a^*) - \frac{3MD^2C^2}{2V} \quad (54)$$

$$\exists B, \quad \sum_{m \in \mathcal{M}} q_m(t) \leq B, \quad \forall t \geq 0 \quad (55)$$

Proof. Applying Theorem 23 (A) to $a \in \mathcal{D}$, we get $\overline{\tilde{\phi}_a} = \overline{\eta_a}$, so the optimality follows from Theorem 21 since $\tilde{\phi}_a(t) = x_a(t)$ always for sink nodes.

For any $m \in \mathcal{M}$, we have

$$\begin{aligned}
& [q_m(T) - q_m(0)] + [\tilde{q}_m(T) - \tilde{q}_m(0)] \\
&= \sum_{t=0}^{T-1} \left[\sum_{a \in P_m} \varphi_{am}(t) - \sum_{a \in C_m} \eta_{ma}(t) \right] - \sum_{t=0}^{T-1} \sum_{a \in C_m} [\tilde{\varphi}_{ma}(t) - \sum_{a \in P_m} \tilde{\eta}_{am}(t)] \\
&= \sum_{t=0}^{T-1} \sum_{a \in C_m} (\tilde{\varphi}_{ma} - \eta_{ma}(t)) - \sum_{t=0}^{T-1} \sum_{a \in P_m} (\tilde{\eta}_{am}(t) - \phi_{am}(t))
\end{aligned}$$

Using the fact that $\tilde{\eta}_{am}(t) - \phi_{am}(t) \geq 0$ and $\tilde{q}_m(T) > 0$, we get

$$q_m(T) \leq q_m(0) + \tilde{q}_m(0) + \sum_{t=0}^{T-1} \sum_{a \in C_m} (\tilde{\varphi}_{ma} - \eta_{ma}(t))$$

and the stability also follows from Theorem 23 (A). \square

Proof of Theorem 23. The proof is by induction on the maximum distance that a node is from any source. Let $l(a)$ and $l(m)$ denote this distance, henceforth called *level*, for nodes $a \in \mathcal{A}$ and $m \in \mathcal{M}$. $l(a) = 0$ for source nodes. $l(m) = \max_{a \in P_m} l(a) + 1$, and $l(a) = \max_{m \in P_a} l(m) + 1$.

We already mentioned that $\eta_a(t) = \tilde{\varphi}_a(t), \forall a \in \mathcal{S}$ by design, so (A) holds for level 0. If the theorem is true up to level $2k$, we can go to level $2k + 1$ of queueing nodes. If the theorem is true up to level $2k + 1$, we can go to level $2k + 2$ of allocation nodes.

(A) \rightarrow (M) Let us prove that given $m \in \mathcal{M}$, if (A) holds for all $a \in P_m$, then (M) holds for m .

From (50) and (49) we get

$$\begin{aligned}
0 &\leq \tilde{\eta}_{am}(t) - \varphi_{am}(t) = \tilde{\eta}_{am}(t) - \min(\eta_a(t), \tilde{\eta}_{am}(t)) \\
&= [\tilde{\eta}_{am}(t) - \eta_a(t)]^+ \leq [\tilde{\varphi}_a(t) - \eta_a(t)]^+ = \tilde{\varphi}_a(t) - \eta_a(t)
\end{aligned}$$

The induction assumption implies that $\sum_t(\tilde{\varphi}_a(t) - \eta_a(t))$ is bounded, therefore $\sum_t(\tilde{\eta}_{am}(t) - \varphi_{am}(t))$ is bounded by the same bound. Done.

(M)→(A) We need to prove that given $a \in \mathcal{A}$, if (M) holds for all $m \in P_a$, then (A) holds for a . This step is more difficult due to the fact that a can only process $\tilde{\varphi}_a(t)$ amount when all parent real queue lengths are over that amount.

Let us first consider the simple case that a has only one parent queue m , and m has only one parent and one child. Since $\eta_{ma} = \eta_a$, we denote as η . Similarly we drop all the subscripts for $\varphi, \tilde{\eta}, \tilde{\varphi}$ in the following discussion.

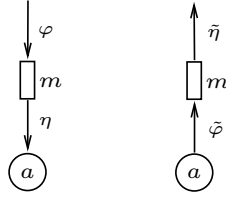


Figure 20: (M)→(A), simple case

The idea behind the proof is that when the real queue length stays above C , $\eta(t)$ and $\tilde{\varphi}(t)$ stay the same. If the real queue length periodically drops below C , then $\sum_t \eta(t)$ is close to $\sum_t \varphi(t)$, which is close to $\sum_t \tilde{\eta}(t)$ due to the induction assumption, which is close to $\sum_t \tilde{\varphi}(t)$ due to the shadow queue being bounded.

For any time T , let T_1 be the last time before T that the real queue length lies below C , i.e. $T_1 = T_1(T) = \max\{0 \leq t \leq T : q_m(t) < C\}$. T_1 is a non-decreasing function of T . So we have $\eta(t) = \tilde{\varphi}_m(t)$ for $T_1 < t \leq T$, and $2C \geq q(T_1 + 1) = q(0) + \sum_{t=0}^{T_1}(\varphi(t) - \eta(t))$, so $\sum_{t=0}^{T_1} \eta(t) \geq (\sum_{t=0}^{T_1} \varphi(t)) - 2C$.

$$\begin{aligned}
0 &\leq \sum_{t=0}^T (\tilde{\varphi}(t) - \eta(t)) = \sum_{t=0}^{T_1} (\tilde{\varphi}(t) - \eta(t)) \\
&\leq 2C + \sum_{t=0}^{T_1} (\tilde{\varphi}(t) - \varphi(t)) \\
&= 2C + \sum_{t=0}^{T_1} (\tilde{\varphi}(t) - \tilde{\eta}(t)) + \sum_{t=0}^{T_1} (\tilde{\eta}(t) - \varphi(t)) \\
&= 2C + \tilde{q}_m(T_1 + 1) - \tilde{q}_m(0) + \sum_{t=0}^{T_1} (\tilde{\eta}(t) - \varphi(t))
\end{aligned}$$

$\tilde{q}_m(T_1 + 1)$ is bounded due to Theorem 21, and the last term is bounded by induction assumption. So (A) holds for a .

For the general case the same proof idea applies. For details please see the appendix. \square

3.3.6 Extensions

In this Section we describe a few extensions to our model and algorithm.

3.3.6.1 Discrete Flow

In the discrete flow model we assume all real flow to be in integer units. The unit doesn't have to represent fixed size data. For example the unit could be number of service data objects (SDO), while the sizes of SDO's may differ. Since the real flow is guided by the shadow flow, we will also require shadow flow to be in integer units. This means that both the real queue and shadow queue lengths are (non-negative) integers. We also require the maximum resources to be integers.

The algorithm needs to be adjusted at sink nodes. When we solve $x_a(t)$ at a sink node $a \in \mathcal{D}$, the solution in (40) may not be an integer. Let $\hat{x}_a(t)$ be the solution in (40). We will let $x_a(t)$ be a random variable with expected value $\hat{x}_a(t)$, i.e.

$$\mathbb{E}[x_a(t) | \tilde{q}(t)] = \hat{x}_a(t) = [(U'_a)^{-1} \left(\sum_{m \in P_a} \tilde{q}_m(t) / V \right)]_0^{R_a} \quad (56)$$

We have complete freedom in choosing the random variable $x_a(t)$. To make analysis easier we require it to be bounded by C . One simple way to do this is to let $x_a(t)$ be $\lceil \hat{x}_a(t) \rceil$ with probability $\hat{x}_a(t) - \lfloor \hat{x}_a(t) \rfloor$, and $\lfloor \hat{x}_a(t) \rfloor$ with probability $\lceil \hat{x}_a(t) \rceil - \hat{x}_a(t)$.

Now we have a countable state space discrete time aperiodic Markov chain. We just need to assume that when the algorithm makes some arbitrary decision, for example when a shadow queue does not have enough credits to satisfy all parents, the decision follows some random distribution that only depends on current state.

The optimality and stability results are stated below. The proof is in the appendix.

Theorem 25. *In the discrete flow model, the Markov chain is positive recurrent. The credit injection rates are optimal, and the shadow queue lengths are stable. That is,*

$$\sum_{a \in \mathcal{D}} U_a(E[x_a(\infty)]) \geq \sum_{a \in \mathcal{D}} U_a(x_a^*) - \frac{3MD^2C^2}{2V} \quad (57)$$

$$\sum_{a \in \mathcal{D}} U_a(E[\bar{x}_a]) \geq \sum_{a \in \mathcal{D}} U_a(x_a^*) - \frac{3MD^2C^2}{2V} \quad a.s. \quad (58)$$

$$E[\tilde{q}_m(\infty)] < \infty \quad (59)$$

where the time ∞ indicates the stationary regime.

What about the real queues? One possible approach is to follow the “thinning” arguments presented in [16], where the nodes probabilistically drop packets from the real queues, thus real queue stability can be established.

Another approach is to examine the arguments in Section 3.3.5.2 more closely. Basically, from the assumption that the shadow queue length $\tilde{q}_m(T) \in O(1)$, we were able to prove that $\sum_{t=0}^{T-1} (\tilde{\phi}_a(t) - \eta_a(t)) \in O(1)$, and $\sum_{t=0}^{T-1} (\tilde{\eta}_{am}(t) - \varphi_{am}(t)) \in O(1)$. The whole argument still holds if we replace $O(1)$ with $o(T)$. From the above theorem we know $\overline{\tilde{q}_m(T)} < \infty$ almost surely. From this and the fact that $q_m(t+1) - q_m(t) \leq DC$ we can prove $\tilde{q}_m(T) \in o(T)$. Therefore $\sum_{t=0}^{T-1} (\tilde{\phi}_a(t) - \eta_a(t)) \in o(T)$, and $\sum_{t=0}^{T-1} (\tilde{\eta}_{am}(t) - \varphi_{am}(t)) \in o(T)$, i.e., the time average of the different between real flow

and shadow flow is 0. So the real flows achieve optimality. However for real queues we will only be able to ascertain that $q_m(T) \in o(T)$.

Here we present a third approach. We make an additional assumption that $U'_a(0) < \infty, \forall a \in \mathcal{D}$. Let $U'_{max} = \max_{a \in \mathcal{D}} U'_a(0)$. This has the property that credits are not injected into the system when the shadow queue lengths get too large, i.e.

$$\sum_{m \in P_a} \tilde{q}_m(t) > VU'_{max} \Rightarrow x_a(t) = \hat{x}_a(t) = 0. \quad (60)$$

Since credit queue lengths can increase by at most DC in each iteration, this implies that all credit queues with only sink nodes as children are bounded by $VU'_{max} + DC$. We also know that for $a \in \mathcal{P}$, x_a will be positive only when there is a positive price difference (40), i.e., credits will only flow when the children shadow queues have more credits than parent shadow queues. So we can prove that all shadow queues are bounded, by induction on the maximum distance that a queue is from any sink.

Therefore we have a finite state Markov chain, which implies the queue is positive recurrent. This gives another proof for the positive recurrent property.

The analysis in Section 3.3.5.2 now applies to any sample path, therefore the real queues are also bounded, and the total real output only differs from the total credit injection by a bounded amount. As a consequence of Theorem 25 the real output rate is optimal. We state these in the following theorem.

Theorem 26. *In the discrete flow model, assuming $U'_a(0) \leq U'_{max}, \forall a \in \mathcal{D}$, then all the credit queues are bounded. As a consequence, the real flow rates are optimal, and the real queue lengths are bounded. That is,*

$$\sum_{a \in \mathcal{D}} U_a(\bar{\eta}_a^{inf}) \geq \sum_{a \in \mathcal{D}} U_a(x_a^*) - \frac{3MD^2C^2}{2V} \quad a.s. \quad (61)$$

$$\exists B, \quad \sum_{m \in \mathcal{M}} q_m(t) \leq B, \quad \forall t \geq 0 \quad (62)$$

3.3.6.2 Resource Usage Parameters

In a real fork and join processing network, the join or fork ratios may not be one, as the data rates may expand or contract after processing, and different tasks on the same server may have different usage requirements. [47] introduced α and β parameters to capture these ratios and the resource requirement for \otimes forks and \otimes joins. Here we expand the notations to all types.

For the original task graph \mathcal{G}_0 , each task v in \mathcal{P} , with one unit of resource ⁵, will consume α_{iv} units of flow from each parent $i \in I_v$ (\otimes -join); or consume α_v units of combined flow from all parents (\oplus -join). Similarly, with one unit of resource, the task will produce β_{vj} units of flow for each child $j \in \mathcal{O}_v$ (\otimes -fork); or β_v units of flow to be divided among all children (\oplus -fork). The β parameters for source nodes and α parameters for sink nodes are set to one.

When mapped to the bipartite graph \mathcal{G} , the α parameters are associated with the links between parent queue nodes and children allocation nodes, and the β parameters are associated with the links between parent allocation nodes and children queue nodes.

We need to distinguish “loss” from flow shrinkage. Flow expansion or shrinkage is caused by the nature of the task and the data it is processing. “Loss” is caused by excessive demand or insufficient production of other tasks, as discussed before. For example, if we view multirate multicast as a fork and join processing network, there is “loss” but no flow shrinkage.

It is a straightforward but tedious exercise to show that previous discussions still hold with the α and β parameters inserted at proper places. We omit the details

⁵The unit for resource is specific to each server. If the unit is changed, we only need to change the α and β parameters of all the tasks on it accordingly.

here. We only mention a few examples. The flow constraints become

$$\sum_{a \in P_m} x_a \beta_{am} - \sum_{a \in C_m} x_a \alpha_{ma} \geq 0, \quad \forall m \in \mathcal{M} \quad (63)$$

In the shadow queue algorithm Step 2, a requests $x_a(t) \beta_{am'}$ credits from child queue m' . a takes the maximum $\tilde{\varphi}_a(t) := \max_{m' \in C_a} \tilde{\eta}_{am'}(t) / \beta_{am'}$. Then a forward $\tilde{\varphi}_a(t) \alpha_{ma}$ to each parent queue \tilde{q}_m of a unless it is a source node.

There is one subtlety when using α and β parameters with discrete flow. Suppose that with one unit of resource, node a consumes one packet from parent m ($\alpha_{ma} = 1$) and produces two packets for its child m' ($\beta_{am'} = 2$). Suppose $x_a(t) = 3$ for some iteration. If $\tilde{q}_{m'}$ has more than 6 credits, then it should issue 6 credits to a . But if m' has only 5 credits, it should only issue 4 credits to a , so that a won't be asked to process half packets. m' is also allowed to issue only 2 or 0 credits in this case. Therefore m' will not be emptied, as allowed in the shadow queue algorithm Step 2 description. This is also the reason that in Lemma 22 we do not require $\eta(t) = q(t)$ when $q(t) < \mu(t)$.

3.3.6.3 General Resource Constraints

In the most general setting, we denote Γ to be the set of all allowed allocation vectors. Let $\hat{\Gamma} = \mathcal{CH}\{\Gamma\}$ be the convex hull of Γ . It is well-known that by time-sharing between different rate vectors in Γ , any point in $\hat{\Gamma}$ is achievable. (In the fluid flow case we discussed before, Γ and $\hat{\Gamma}$ are identical.) So the resource constraint for the primal problem (P) is simply $x \in \hat{\Gamma}$.

In the dual approach, $\sum_{r \in \mathcal{R}} \tilde{D}_r(p)$ is replaced by

$$\begin{aligned} \tilde{D}(p) &= \max_{x \in \hat{\Gamma}} \sum_{a \in \mathcal{A}} x_a \left(\sum_{m \in C_a} p_m - \sum_{m \in P_a} p_m \right) \\ &= \max_{x \in \Gamma} \sum_{a \in \mathcal{A}} x_a \left(\sum_{m \in C_a} p_m - \sum_{m \in P_a} p_m \right) \end{aligned} \quad (64)$$

The second equality is due to the fact that it is a linear optimization problem, and $\hat{\Gamma}$ is convex hull of Γ .

In the cases we have discussed so far, $\tilde{D}(p)$ can be decomposed into local problems on each server. However if such a decomposition is not possible, for example in a wireless transmission setting, then we need a scheduling algorithm to find a maximal solution $x \in \Gamma$ for $\tilde{D}(p)$. This is outside of the scope of this paper and is a future research topic for us.

However, if we have a way to find maximal solution $x \in \Gamma$ for $\tilde{D}(p)$, then our algorithm still applies, and the optimality and stability results still hold, provided that Γ is bounded.

3.3.7 Evaluation

In this section we demonstrate through a small example that our algorithm does converge to the optimal utility, and that it adapts well in a slowly time varying environment.

We use the task graph in Figure 18. This graph contains many contention points. Input streams s_2 and s_3 contend on both servers r_1 and r_3 . Stream s_2 also contend with s_1 on server r_2 . There are different types of fork and join. All resource constraints are set to 100 except $R_{r_1} = 90$. We also set the minimum output rate to 1. The utility functions are set to the natural log function ($U_a(x) = \ln(x)$). Such an utility function has been known to represent a weighted proportional fairness [52] on the rate distribution among multiple streams.

We introduce different types of changes to the parameters after some iterations to see how the algorithms adapt to slowly time varying environments.

- After 1000 iterations, the resource for server r_2 is reduced from 100 to 70.
- After another 1000 iterations, the importance of the d_2 output stream is increased by changing its utility function from $\ln(x)$ to $2\ln(x)$.

We simulated the discrete model of our algorithm with V set to 20000. When there are not enough credits to serve from a shadow queue, we adopt the simple

policy of not serving any credits at all. Similarly, when there is insufficient data in a real queue, we choose not to process any.

Figure 21(a) plots sink output rates at each iteration for all three sinks. To present smoother curves the values are rolling averages over 50 iterations. We observe that sink rates converge in a couple hundreds of iterations after each change.

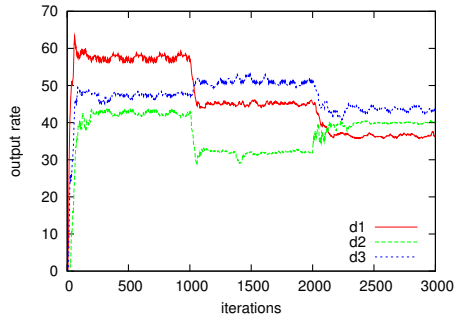
Figure 21(b) plots the total utility based on average output rate over 50 iterations. The straight lines show the theoretical optimal values. We observe that after each change, the moving average of utility converges to a small neighborhood around the new optimal value in less than a hundred iterations. So the algorithm quickly adapts to changes in slowly time varying environments.

Figure 21(c) shows the difference between credits injected and real output at sink d_2 . In Section 3.3.6.1 we have stated that the total real output only differs from the total credit injection by a bounded amount. Figure 21(c) illustrates this. Mostly the credits and output differ during the convergence stage, then it happens less frequently and eventually stops. This trend is more obvious when we run the simulation for a longer period while keeping the parameters unchanged.

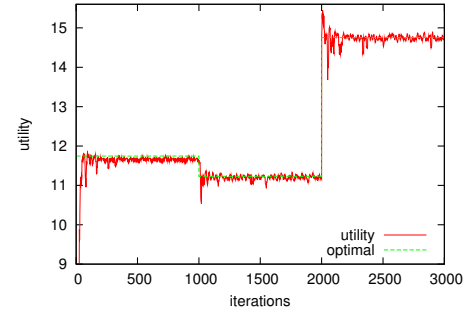
Figure 21(d) shows the shadow queue and real queue lengths at sink d_2 . The values shown are rolling averages. We can see that although the shadow queue length increased significantly due to the way the parameters have changed, the real queue lengths remained low. This is desirable. Credit queues are just counters so the size of their values does not matter. We observed similar effect in most of the queues except for the queue between p_5 and p_3 , which is depicted in Figure 21(e). This seems to be related to the fact that p_5 is a simultaneous join and is involved in a lossy link from p_3 . Even in this case the real queue size is stable after the algorithm converges.

3.3.8 Conclusion

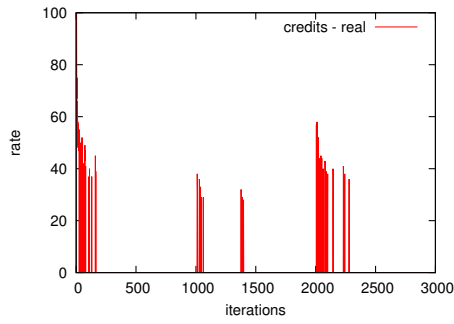
In this section we studied the resource allocation problem in a processing network where the processing nodes can involve arbitrary combination of fork and join semantics. We presented a novel modeling framework that allows us to formulate the problem as an elegant convex optimization problem, and proposed a shadow queue based algorithm for distributed admission control, routing and resource allocation. We proved that the algorithm converges to the optimal solution and the queues remain bounded. We further demonstrated the robustness and convergence of our distributed algorithm through simulation. Although our framework can handle general resource usage parameters where these parameters can be estimated online, it would be interesting to explore algorithms that do not rely on the specific knowledge of these parameters or allow them to be stochastic. This is an interesting topic for future research.



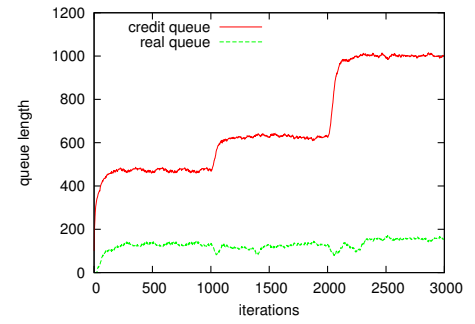
(a) Sink Output Rates



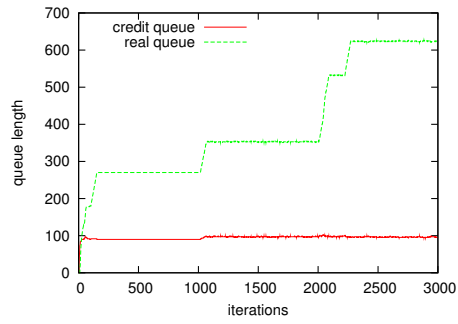
(b) Total Utility



(c) Difference between injected credits and real output at sink d_2



(d) Queue lengths at sink d_2



(e) Queue lengths between p_5 and p_3

Figure 21: Simulation Results

APPENDIX A

PROOFS

A.1 Proofs of Theorem 16

A few properties of subdifferential that we will need is listed after the proof.

Proof. The solution for the primal problem (\tilde{P}) is a $|\mathcal{D}|$ dimensional vector $x_{\mathcal{D}} = \{x_v | v \in \mathcal{D}\}$. All the other $x_u, u \notin \mathcal{D}$ are viewed as functions of $x_{\mathcal{D}}$. Let $e_v, v \in \mathcal{D}$ be the $|\mathcal{D}|$ dimensional unit vector that is all 0 except 1 for sink node v .

Since x_v updates according to (24), we need to show that $\sum_{v \in \mathcal{D}} e_v \left(U'_v(x_v^{(k)}) - K q_v^{(k)} \right)$ is a subgradient of the primal objective function in (\tilde{P}) . Since U'_v is already a derivative of U_v , we only need to show that the penalty rate vector $q = \sum_{v \in \mathcal{D}} q_v$ is a subgradient of the penalty term $\sum_{r \in \tilde{\mathcal{R}}} P_r(h_r)$, i.e. $q \in \partial \sum_{r \in \tilde{\mathcal{R}}} P_r(h_r)$. Here we use the notation $h_r \equiv \sum_{u \in V_r} x_u$, the total allocation of all nodes in r .

We need to introduce a few more definitions. Let $\gamma_{ij} \equiv \alpha_{ij}/\beta_{ij}$. So (20) becomes:

$$x_i = \max_{j \in \mathcal{O}_n} x_j \gamma_{ij}, \quad \forall i \in \mathcal{S} \cup P \quad (65)$$

We will use \mathcal{W}_{uv} to denote all directed paths (walks) from u to v in graph \mathcal{G} . We will use $w \in \mathcal{W}_{uv}$ to denote a particular directed path from u to v . We will use $(i, j) \in w$ to denote that a directed link (i, j) is in the directed path w . Let $z_w \equiv \prod_{(i,j) \in w} z_{ij}$, $z_{uv} \equiv \sum_{w \in \mathcal{W}_{uv}} z_w$, $\gamma_w \equiv \prod_{(i,j) \in w} \gamma_{ij}$, $\gamma_{uv} \equiv \max_{w \in \mathcal{W}_{uv}} \gamma_w$.

If there is no path from u to v , then \mathcal{W}_{uv} is understood to be the empty set, and doing max or \sum over the set is understood to produce 0, therefore $z_{uv} = 0$, $\gamma_{uv} = 0$ in that case.

It is straightforward to derive the following:

$$\sum_{v \in \mathcal{D}} z_{uv} = 1, \quad \forall u \in \mathcal{S} \cup \mathcal{P}, \quad (66)$$

$$x_u = \max_{v \in \mathcal{D}} x_v \gamma_{uv}, \quad \forall u \in \mathcal{S} \cup \mathcal{P}, \quad (67)$$

$$\gamma_w = \gamma_{uv}, x_u = x_v \gamma_{uv} \text{ if } z_w > 0, \quad \forall u \in \mathcal{S} \cup \mathcal{P}, v \in \mathcal{D}, w \in \mathcal{W}_{uv}, \quad (68)$$

$$x_u = x_v \gamma_{uv} \text{ if } z_{uv} > 0, \quad \forall u \in \mathcal{S} \cup \mathcal{P}, v \in \mathcal{D}. \quad (69)$$

(66) follows from (22) which shows that all the splitting factors z_{ij} add to 1. (67) follows from (65). (68) follows from (23). This is because we have $x_u = x_v \gamma_w$ from (23), and $x_u \geq x_v \gamma_{uv}$ from (67), and $\gamma_{uv} \geq \gamma_w$ by definition. (69) follows from (68).

A sink $v \in \mathcal{D}$ receives penalty rate from all the non-sink nodes over all possible paths, and the penalty rate is modified by all the γ_{ij} and z_{ij} factors. So the penalty vector $q = \{q_v | v \in \mathcal{D}\}$ can be expressed as:

$$\begin{aligned} q &= \sum_{v \in \mathcal{D}} e_v q_v \\ &= \sum_{v \in \mathcal{D}} e_v \sum_{r \in \tilde{\mathcal{R}}} \sum_{u \in V_r} p_r(h_r) \sum_{w \in \mathcal{W}_{uv}} \prod_{(i,j) \in w} z_{ij} \gamma_{ij} \end{aligned} \quad (70)$$

$$= \sum_{v \in \mathcal{D}} e_v \sum_{r \in \tilde{\mathcal{R}}} \sum_{u \in V_r} p_r(h_r) \sum_{w \in \mathcal{W}_{uv}} z_w \gamma_w \quad (71)$$

$$= \sum_{v \in \mathcal{D}} e_v \sum_{r \in \tilde{\mathcal{R}}} \sum_{u \in V_r} p_r(h_r) \sum_{w \in \mathcal{W}_{uv}} z_w \gamma_{uv} \quad (72)$$

$$= \sum_{v \in \mathcal{D}} e_v \sum_{r \in \tilde{\mathcal{R}}} \sum_{u \in V_r} p_r(h_r) z_{uv} \gamma_{uv} \quad (73)$$

$$= \sum_{r \in \tilde{\mathcal{R}}} p_r(h_r) \sum_{u \in V_r} \sum_{v \in \mathcal{D}} z_{uv} \gamma_{uv} e_v \quad (74)$$

In (71) we used definition of z_w and γ_w . In (72) we replaced γ_w by γ_{uv} . We can do this because if $z_w = 0$, then the replacement is fine; if $z_w > 0$, then (68) gives us $\gamma_w = \gamma_{uv}$. In (73) we used definition of z_{uv} . In (74) we switched the three summations.

Now we will show that the penalty rate vector q is a subgradient of the penalty term $\sum_{r \in \tilde{\mathcal{R}}} P_r(h_r)$.

For $v \in \mathcal{D}$, $\gamma_{uv} e_v$ is the gradient of $x_v \gamma_{uv}$ (as a function on $x_{\mathcal{D}}$). We want to show that $\sum_{v \in \mathcal{D}} z_{uv} \gamma_{uv} e_v$ is a subgradient of $x_u = \max_{v \in \mathcal{D}} x_v \gamma_{uv}$ from (67).

By (69), z_{uv} is only non-zero when the max is achieved by $x_v \gamma_{uv}$. By (66), this is a convex combination. So Lemma 27 applies. Thus $\sum_{v \in \mathcal{D}} z_{uv} \gamma_{uv} e_v \in \partial x_u$. Summing over $u \in V_r$, we get $\sum_{u \in V_r} \sum_{v \in \mathcal{D}} z_{uv} \gamma_{uv} e_v \in \partial \sum_{u \in V_r} x_u = \partial h_r$.

Now apply the chain rule in Lemma 27. By definition $p_r(y) \in \partial P_r(y)$, so we have $p_r(h_r) \sum_{u \in V_r} \sum_{v \in \mathcal{D}} z_{uv} \gamma_{uv} e_v \in \partial P_r(h_r)$. Summing over all r , we get that the aggregated penalty vector q is a subgradient of the penalty term $\sum_{r \in \tilde{\mathcal{R}}} P_r(h_r)$.

So far we have proved that the algorithm executes a subgradient algorithm for primal problem (\tilde{P}) . The convergence statements follow from standard results for a subgradient algorithm [66]. \square

Lemma 27. ([31, Section D, Theorem 4.3.1 and Corollary 4.3.2])

(a) *Subdifferential of maximum of convex functions is the convex hull of the subdifferential of the functions achieving the maximum. Let f_1, \dots, f_m be convex functions from \mathcal{R}^n to \mathcal{R} , and $f := \max\{f_1, \dots, f_m\}$, then*

$$\partial f(x) = \text{co}\{\cup \partial f_i(x) : f_i(x) = f(x)\}$$

(b) *Chain Rule: Let $f : \mathcal{R}^n \rightarrow \mathcal{R}$ be a convex function, $g : \mathcal{R} \rightarrow \mathcal{R}$ be a convex and increasing function, then*

$$\partial(g \circ f)(x) = \{\rho s : \rho \in \partial g(f(x)), s \in \partial f(x)\}$$

A.2 Proof of Theorem 17

Proof. Let B be the maximum number of downstream sinks that any node can have, and Γ be the minimum value of all γ_{uv} . Let x^* be the optimal solution for \tilde{P} . If it is not the optimal solution for P , then some node must have non-zero penalty, i.e. $h_r > R_r$ for some $r \in \tilde{\mathcal{R}}$. Pick any node u in V_r that has resource allocation $x_u > 0$. Assume u has b downstream sinks with non-zero request. If we reduce the request of these b sinks by a small δ , we reduce the utility on these sinks, but we also reduce the penalty on r . We reduce the total utility by at most $BA\delta$, but we reduce

the penalty on r by at least $K\Gamma\delta$. So if we pick $K > AB/\Gamma$, then we have increased the total objective, contradicting the assumption that x^* is the optimal solution. \square

A.3 Proof of Theorem 19

Proof. The optimal rates and optimal prices are both non-unique. However, as we mentioned earlier, the optimal rates on $v \in \mathcal{D}$ are unique and is denoted y_v^* . For any set of optimal prices, because there is no duality gap, $y_{\mathcal{D}}^*$ must satisfy the maximizer in (27), therefore it satisfies (29). So the optimal prices must satisfy

$$\forall v \in \mathcal{D}, \sum_{i \in \mathcal{I}_v} p_{iv} \alpha_{iv} \begin{cases} = U'_v(y_v^*) & 0 < y_v^* < R_v \\ \geq U'_v(0) & y_v^* = 0 \\ \leq U'_v(R_v) & y_v^* = R_v \end{cases}$$

By standard subgradient arguments [66], if $\lim_{k=0}^{\infty} \eta_k = 0$ and $\sum_{k=0}^{\infty} \eta_k = \infty$, the prices $p^{(k)}$ will converge to the set of optimal prices. So $\sum_{i \in \mathcal{I}_v} p_{iv}^{(k)} \alpha_{iv}$ will also converge to the above regions. Again, due to (29), we can see that the rates $y_v^{(k)}$ will converge to $y_v^*, v \in \mathcal{D}$.

We can use “converge to a small neighborhood of” in the above argument for the constant step size case. \square

A.4 Proof of Theorem 20

Proof. The optimal rates and optimal prices are both non-unique. However, as we mentioned earlier, the optimal rates on $a \in \mathcal{D}$ are unique and is denoted x_a^* . For any set of optimal prices, because there is no duality gap, $x_{\mathcal{D}}^*$ must satisfy the maximizer in (39), therefore it satisfies (40). So the optimal prices must satisfy

$$\sum_{m \in P_a} p_m \begin{cases} = U'_a(x_a^*) & 0 < x_a^* < R_a \\ \geq U'_a(0) & x_a^* = 0 \\ \leq U'_a(R_a) & x_a^* = R_a \end{cases} \quad \forall a \in \mathcal{D}.$$

By standard subgradient arguments [66], if $\lim_{t \rightarrow \infty} h(t) = 0$ and $\sum_{t=0}^{\infty} h(t) = \infty$, the prices $p(t)$ will converge to the set of optimal prices. So $\sum_{m \in P_a} p_m(t)$ will also converge to the above regions. Again, due to (40), we can see that the rates $x_a(t)$ will converge to x_a^* , $a \in \mathcal{D}$.

We can use “converge to a small neighborhood of” in the above argument for the constant step size case. \square

A.5 Proof of Lemma 22

Proof. When $q(t) \geq \mu(t)$,

$$\begin{aligned} q^2(t+1) - q^2(t) &\leq (q(t) - \mu(t) + \lambda(t))^2 - q^2(t) \\ &= \mu^2(t) + \lambda^2(t) - 2q(t)(\mu(t) - \lambda(t)) - 2\mu(t)\lambda(t) \\ &\leq 2\mu_{max}^2 + \lambda_{max}^2 - 2q(t)(\mu(t) - \lambda(t)) \end{aligned}$$

When $q(t) < \mu(t)$,

$$\begin{aligned} q^2(t+1) - q^2(t) &\leq (q(t) + \lambda(t))^2 - q^2(t) \\ &= \lambda^2(t) + 2q(t)\lambda(t) \\ &= 2q(t)\mu(t) + \lambda^2(t) - 2q(t)(\mu(t) - \lambda(t)) \\ &\leq 2\mu^2(t) + \lambda^2(t) - 2q(t)(\mu(t) - \lambda(t)) \\ &\leq 2\mu_{max}^2 + \lambda_{max}^2 - 2q(t)(\mu(t) - \lambda(t)) \end{aligned}$$

\square

Comment: If we change the service rate behavior to $\eta(t) = q(t)$ when $q(t) < \mu(t)$, then we can get a tighter bound $q^2(t+1) - q^2(t) \leq \mu_{max}^2 + \lambda_{max}^2 - 2q(t)(\mu(t) - \lambda(t))$, similar to [57].

A.6 Proof of Theorem 21

Proof. For shadow queue $\tilde{q}_m(t)$, apply Lemma 22 with $\mu(t) = \sum_{a \in P_m} x_a(t)$, $\lambda(t) = \sum_{a \in C_m} x_a(t)$, $\mu_{\max} = \lambda_{\max} = DC$, we have

$$\begin{aligned}
& \sum_{m \in \mathcal{M}} (\tilde{q}_m^2(t+1) - \tilde{q}_m^2(t)) \\
& \leq 3MD^2C^2 - 2 \sum_{m \in \mathcal{M}} \tilde{q}_m(t) \left(\sum_{a \in P_m} x_a(t) - \sum_{a \in C_m} x_a(t) \right) \\
& = 3MD^2C^2 + 2V \sum_{a \in \mathcal{D}} U_a(x_a(t)) - 2VL(x(t), p(t)) \tag{75}
\end{aligned}$$

where $L(x, p)$ is defined in (38) and $p_m(t) = \tilde{q}_m(t)/V$.

Consider a Primal Problem (P_ϵ) , $\epsilon > 0$, which is the same as the original Primal Problem (P) except that the flow constraints $\sum_{a \in P_m} x_a - \sum_{a \in C_m} x_a \geq 0$ are changed to $\sum_{a \in P_m} x_a - \sum_{a \in C_m} x_a \geq \epsilon$. (P_ϵ) is obviously feasible for small ϵ since we can construct a feasible x from the sink nodes bottom up. We define the maximum ϵ such that (P_ϵ) is feasible to be ϵ_{\max} . When $\epsilon = 0$, we are just talking about the original (P) . Let x^ϵ be an optimal solution for (P_ϵ) .

Our algorithm picks $x(t)$ to maximize $L(x, p(t))$ under the resource constraints. Since x^ϵ satisfies the resource constraints, we get $L(x(t), p(t)) \geq L(x^\epsilon(t), p(t))$. Therefore

$$\begin{aligned}
& \sum_{m \in \mathcal{M}} \tilde{q}_m^2(t+1) - \tilde{q}_m^2(t) \\
& \leq 3MD^2C^2 + 2V \sum_{a \in \mathcal{D}} U_a(x_a(t)) - 2VL(x^\epsilon(t), p(t)) \\
& \leq 3MD^2C^2 + 2V \sum_{a \in \mathcal{D}} U_a(x_a(t)) \\
& \quad - 2V \sum_{a \in \mathcal{D}} U_a(x^\epsilon(t)) - 2\epsilon \sum_{m \in \mathcal{M}} \tilde{q}_m(t) \tag{76}
\end{aligned}$$

By setting $\epsilon = 0$, picking an optimal solution for (P) as x^ϵ , averaging over $0 \leq t \leq T-1$, rearranging the terms and taking limits, using non-negativity of the Lyapunov

function, we can derive

$$\overline{\sum_{a \in \mathcal{D}} U_a(x_a)}^{inf} \geq \sum_{a \in \mathcal{D}} U_a(x_a^*) - \frac{3MD^2C^2}{2V} \quad (77)$$

(52) immediately follows due to $\sum_{a \in \mathcal{D}} U_a(\overline{x_a}^{inf}) \geq \overline{\sum_{a \in \mathcal{D}} U_a(x_a)}^{inf}$, which is due to Jensen's inequality.

If we do the same thing by setting $\epsilon = \epsilon_{max}$ and picking any x^ϵ , we get

$$\overline{\sum_{m \in \mathcal{M}} \tilde{q}_m}^{sup} \leq \frac{3MD^2C^2 + 2VNU_{max}}{2\epsilon_{max}} \equiv B_1 \quad (78)$$

This shows that the time average of the shadow queue length is bounded. However we can get a stronger result. In (76), by setting $\epsilon = \epsilon_{max}$, we get

$$\sum_{m \in \mathcal{M}} (\tilde{q}_m^2(t+1) - \tilde{q}_m^2(t)) \leq 2\epsilon_{max}(B_1 - \sum_{m \in \mathcal{M}} \tilde{q}_m(t))$$

Let $B_2 = B_1 + MDC + \sum_m \tilde{q}_m(t)$, and we will argue that $\sum_m \tilde{q}_m^2(t)$ is bounded by $B_2^2, \forall t$. This is obviously true for $t = 0$. If this is not true for all t , then there exists time T such that $\sum_m \tilde{q}_m^2(T) \leq B_2^2$ and $\sum_m \tilde{q}_m^2(T+1) > B_2^2$. This implies $\sum_m (\tilde{q}_m^2(T+1) - \tilde{q}_m^2(T)) > 0$, so from the above equation we get $\sum_m \tilde{q}_m(T) < B_1$, which implies $\sum_m \tilde{q}_m(T+1) < B_1 + MDC \leq B_2$, which implies $\sum_m \tilde{q}_m^2(T+1) \leq (\sum_m \tilde{q}_m(T+1))^2 < B_2^2$, a contradiction.

$\sum_m \tilde{q}_m^2(t) \leq B_2^2$ implies that $\tilde{q}_m^2(t) \leq B_2$, so $\sum_m \tilde{q}_m(t) \leq MB_2$ and (53) is proven. We can improve the bound a little by using the Cauchy-Schwarz inequality $(\sum_i x_i y_i)^2 \leq (\sum_i x_i^2)(\sum_i y_i^2)$, which implies $(\sum_m \tilde{q}_m(t))^2 \leq M \sum_m \tilde{q}_m^2(t) \leq MB_2^2$, so $\sum_m \tilde{q}_m(t) \leq \sqrt{M}B_2$. \square

A.7 Proof of Theorem 23 Continued

Proof. Here we prove the $(M) \rightarrow (A)$ part for general case. Given $a \in \mathcal{A}$, we know from the induction assumption that for all $m \in P_a$, $\sum_t (\tilde{\eta}_{a'm}(t) - \varphi_{a'm}(t))$ is bounded for all $a' \in P_m$.

Without loss of generality and to simplify notations, we'll just consider a typical case as illustrated in Figure 22. Proof for general case is the same but messy to write.

a_0 is our node of interest. Suppose a_0 has two parents m_1 and m_2 , m_1 has an additional child a_1 , and m_2 has an additional child a_2 . a_1 or a_2 may have other parents but this does not affect our discussion. m_1 and m_2 only have single parents. (For multiple parents we just need to view $\varphi_1, \varphi_2, \tilde{\eta}_1, \tilde{\eta}_2$ as sums of multiple variables.)

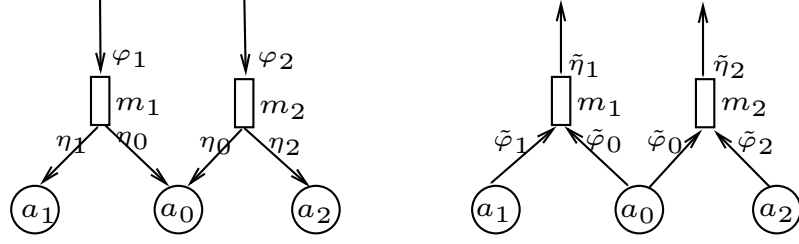


Figure 22: (M) \rightarrow (A), typical case

For any time T , let T_1 be the last time before T that $q_{m_1}(t)$ is below DC , i.e. $T_1 = T_1(T) = \max\{0 \leq t \leq T : q_{m_1}(t) < DC\}$, and similar definition for T_2 . We can cover all the time slots Z^+ by union of 2 subsets $Z_1 \cup Z_2$ where $Z_1 = \{T : T_1(T) \geq T_2(T)\}$, and the other way around for Z_2 .

Let us consider $T \in Z_1$. For $T_2 \leq T_1 < t \leq T$, both queues have sufficient length, so we have $\eta_0(t) = \tilde{\varphi}_0(t)$. And $2DC \geq q_1(T_1 + 1) = q_1(0) + \sum_{t=0}^{T_1} (\varphi_1(t) - \eta_1(t) - \eta_0(t))$, so $\sum_{t=0}^{T_1} \eta_0(t) + \eta_1(t) \geq -2DC + \sum_{t=0}^{T_1} \varphi_1(t)$.

$$\begin{aligned}
0 &\leq \sum_{t=0}^T (\tilde{\varphi}_0(t) - \eta_0(t)) = \sum_{t=0}^{T_1} (\tilde{\varphi}_0(t) - \eta_0(t)) \\
&\leq \sum_{t=0}^{T_1} (\tilde{\varphi}_0(t) - \eta_0(t) + \tilde{\varphi}_1(t) - \eta_1(t)) \\
&\leq 2DC + \sum_{t=0}^{T_1} (\tilde{\varphi}_0(t) + \tilde{\varphi}_1(t) - \varphi_1(t)) \\
&= 2DC + \sum_{t=0}^{T_1} (\tilde{\varphi}_0(t) + \tilde{\varphi}_1(t) - \tilde{\eta}_1(t)) + \sum_{t=0}^{T_1} (\tilde{\eta}_1(t) - \varphi_1(t)) \\
&= 2DC + \tilde{q}_{m_1}(T_1 + 1) - \tilde{q}_{m_1}(0) + \sum_{t=0}^{T_1} (\tilde{\eta}_1(t) - \varphi_1(t))
\end{aligned}$$

The second inequality is due to $\eta_1(t) \leq \tilde{\phi}_1(t)$ by design. In the last line, $\tilde{q}_{m_1}(T_1 + 1)$ is bounded due to Theorem 21, and the last term is bounded due to the induction assumption. So $\sum_{t=0}^T (\tilde{\varphi}_0(t) - \eta_0(t))$ is bounded.

Since Z^+ is covered by finite number of subsets, and we can prove a bound for each subset, there is a bound for all time slots. So (A) holds for a_0 . \square

Comment: For (M) \rightarrow (A) simple case, there is an alternative argument in [15, Proposition 3] that can directly establish bound on real queues from bound on shadow queues, which easily implies (M) \rightarrow (A). However the argument cannot be extended to the general case.

A.8 Proof of Theorem 25

Proof. Since we are dealing with a Markov chain, the Lyapunov drift becomes $E[\sum_{m \in \mathcal{M}} (\tilde{q}^2(t+1) - \tilde{q}^2(t)) | \tilde{q}(t)]$. We add and subtract $U_a(\hat{x}_a(t)) = U_a(E[x_a(t) | \tilde{q}(t)])$ instead of $U_a(x_a(t))$ in our arguments. Similar to (76) we get

$$\begin{aligned}
&E[\sum_{m \in \mathcal{M}} (\tilde{q}_m^2(t+1) - \tilde{q}_m^2(t)) | \tilde{q}_m(t)] \\
&\leq 3MD^2C^2 + 2V \sum_{a \in \mathcal{D}} U_a(\hat{x}_a(t)) - 2V \sum_{a \in \mathcal{D}} U_a(x^\epsilon(t)) - 2\epsilon \sum_{m \in \mathcal{M}} \tilde{q}_m(t) \quad (79)
\end{aligned}$$

By setting $\epsilon = \epsilon_{max}$, we get

$$\mathbb{E} \left[\sum_{m \in \mathcal{M}} (\tilde{q}^2(t+1) - \tilde{q}^2(t)) | \tilde{q}(t) \right] \leq 2\epsilon_{max} (B_1 - \sum_{m \in \mathcal{M}} \tilde{q}_m(t)),$$

where B_1 was defined in (78). Let C be the set of states that satisfy $\sum_{m \in \mathcal{M}} \tilde{q}_m(t) \leq B_1 + 1$. So the Lyapunov drift is bounded by $2\epsilon_{max} B_1$ always, and the drift is $\leq -2\epsilon_{max}$ outside of the finite subset C . According to Foster-Lyapunov criteria, the Markov chain is positive recurrent. It is easy to see that the chain is aperiodic, so we can apply ergodic theory and get:

$$\overline{\mathbb{E}[x_a]} = \mathbb{E}[x_a(\infty)], \quad \forall a \in \mathcal{A} \quad (80)$$

$$\overline{x_a} = \mathbb{E}[x_a(\infty)] \quad a.s., \quad \forall a \in \mathcal{A} \quad (81)$$

$$\overline{\mathbb{E}[\tilde{q}_m]} = \mathbb{E}[\tilde{q}_m(\infty)], \quad \forall m \in \mathcal{M} \quad (82)$$

Going back to (79), we can take its expectation and use Jensen's inequality to get

$$\begin{aligned} & \mathbb{E} \left[\sum_{m \in \mathcal{M}} (\tilde{q}_m^2(t+1) - \tilde{q}_m^2(t)) \right] \\ & \leq 3MD^2C^2 + 2V \sum_{a \in \mathcal{D}} U_a(\mathbb{E}[x_a(t)]) - 2V \sum_{a \in \mathcal{D}} U_a(x^\epsilon(t)) - 2\epsilon \sum_{m \in \mathcal{M}} \mathbb{E}[\tilde{q}_m(t)] \end{aligned}$$

Now by the same technique of taking time average, and setting $\epsilon = 0$, we get

$$\sum_{a \in \mathcal{D}} \overline{U_a(\mathbb{E}[x_a])} \geq \sum_{a \in \mathcal{D}} U_a(x_a^*) - \frac{3MD^2C^2}{2V} \quad (83)$$

(57) immediately follows due to $U_a(\overline{\mathbb{E}[x_a]}) \geq \overline{U_a(\mathbb{E}[x_a])}$, which is due to Jensen's inequality, and (80). Then (58) follows from (81).

Similarly by taking ϵ to ϵ_{max} , we get

$$\sum_{m \in \mathcal{M}} \overline{\mathbb{E}[\tilde{q}_m]} \leq B_1 \quad (84)$$

Then (59) follows due to (82). \square

REFERENCES

- [1] “Akamai Technologies Inc.” <http://www.akamai.com> (Apr/2010).
- [2] “Apache hadoop.” <http://hadoop.apache.org> (Apr/2010).
- [3] “Intel Lynnfield processor.” *Intel Corporation*.
- [4] “Internet2 Abilene Network.” <http://abilene.internet2.edu> (Apr/2010).
- [5] “XDR datasheet,” *Rambus, Inc.*, 2002-2003.
- [6] “XDR-2 datasheet,” *Rambus, Inc.*, 2004-2005.
- [7] “Intel IXP 2855 network processor product brief,” *Intel Corporation*, 2005.
- [8] ABADI, D. J. and OTHERS, “The design of the borealis stream processing engine,” in *CIDR*, pp. 277–289, 2005.
- [9] ALON, N., MATIAS, Y., and SZEGEDY, M., “The space complexity of approximating the frequency moments,” in *Proceedings of ACM Symposium on Theory of Computing (STOC)*, 1996.
- [10] AMINI, L., JAIN, N., SEHGAL, A., SILBER, J., and VERSCHURE, O., “Adaptive control of extreme-scale stream processing systems,” in *Proc. of ICDCS '06*, IEEE Computer Society, 2006.
- [11] AYRES, P., SUN, H., CHAO, H., and LAU, W., “ALPi: A DDoS defense system for high-speed networks,” *IEEE Journal on Selected Areas in Communications*, vol. 24(10), pp. 1864–1876, 2006.
- [12] BABCOCK, B., BABU, S., DATAR, M., MOTWANI, R., and WIDOM, J., “Models and issues in data stream systems,” in *PODS*, pp. 1–16, 2002.
- [13] BABCOCK, B. and OLSTON, C., “Distributed top-k monitoring,” in *Proceedings of ACM SIGMOD*, 2003.
- [14] BACCELLI, F. and LIU, Z., “On the execution of parallel programs on multiprocessor systems-a queuing theory approach,” *JACM*, vol. 37, pp. 373–417, April 1990.
- [15] BUI, L., SRIKANT, R., and STOLYAR, A., “Optimal resource allocation for multicast flows in multihop wireless networks,” in *IEEE CDC*, 2007.
- [16] BUI, L., SRIKANT, R., and STOLYAR, A., “Optimal resource allocation for multicast sessions in multihop wireless networks,” *Philosophical Transactions of The Royal Society*, vol. 366, no. 1872, pp. 2059–2074, 2008.

- [17] CAO, J., JARVIS, S. A., SAINI, S., and NUDD, G. R., “Gridflow: Workflow management for grid computing,” in *Intl. Symposium on Cluster Computing and the Grid (CCGrid’03)*, (Tokyo, Japan), May 2003.
- [18] CHAKRABARTI, A., KHOT, S., and SUN, X., “Near-optimal lower bounds on the multi-party communication complexity of set disjointness,” in *Proceedings of IEEE Conference on Computational Complexity (CCC)*, 2003.
- [19] CHEETANCHERI, S. G., AGOSTA, J. M., DASH, D. H., LEVITT, K. N., ROWE, J., and SCHOOLER, E. M., “A distributed host-based worm detection system,” in *Proceedings of the ACM SIGCOMM Workshop on Large-scale Attack Defense (LSAD)*, 2006.
- [20] CORMODE, G., MUTHUKRISHNAN, S., and YI, K., “Algorithms for distributed functional monitoring,” in *Proceedings of the 19th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2008.
- [21] CVETKOVSKI, A., “An algorithm for approximate counting using limited memory resources,” in *ACM Sigmetrics*, 2007.
- [22] DEAN, J. and GHEMAWAT, S., “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, 2008.
- [23] DEB, S. and SRIKANT, R., “Congestion control for fair resource allocation in networks with multicast flows,” in *IEEE CDC*, 2001.
- [24] ERYILMAZ, A. and SRIKANT, R., “Joint congestion control, routing and mac for stability and fairness in wireless networks,” *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 8, pp. 1514–1524, 2006.
- [25] FANG, M., SHIVAKUMAR, N., GARCIA-MOLINA, H., MOTWANI, R., and ULLMAN, J. D., “Computing iceberg queries efficiently,” in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1998.
- [26] FENG, H., LIU, Z., XIA, C. H., and ZHANG, L., “Load shedding and distributed resource control of stream processing networks,” *Performance Evaluation*, vol. 64, no. 9–12, pp. 1102–1120, 2007.
- [27] FLAJOLET, P. and MARTIN, G., “Probabilistic counting,” in *FOCS*, 1983.
- [28] GLEESER, L. J., “On the distribution of the number of successes in independent trials,” *The Annals of Probability*, vol. 3, pp. 182–188, Feb. 1975.
- [29] GRAHAM, R., KNUTH, D., and PATASHNIK, O., *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 2nd ed., 1994.
- [30] GSCHWIND, M., HOFSTEE, H. P., FLACHS, B., HOPKINS, M., WATANABE, Y., and YAMAZAKI, T., “Synergistic processing in cell’s multicore architecture,” *IEEE Micro*, vol. 26, no. 2, pp. 10–24, 2006.

- [31] HIRIART-URRUTY, J.-B. and LEMARÉCHAL, C., *Fundamentals of Convex Analysis*. Springer, 2001.
- [32] HOEFFDING, W., “On the distribution of the number of successes in independent trials,” *The Annals of Mathematical Statistics*, vol. 27, no. 3, pp. 713–721, 1956.
- [33] HOEFFDING, W., “Probability inequalities for sums of bounded random variables,” *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 13–30, 1963.
- [34] HONG, S. I., MCKEE, S. A., SALINAS, M. H., KLENKE, R. H., AYLLOR, J. H., and WULF, W. A., “Access order and effective bandwidth for streams on a direct rambus memory,” in *IEEE HPCA*, p. 80, 1999.
- [35] HUA, N., LIN, B., XU, J., and ZHAO, H., “BRICK: A novel exact active statistics counter architecture,” in *ACM/IEEE ANCS*, 2008.
- [36] INDYK, P., “Stable distributions, pseudorandom generators, embeddings, and data stream computation,” in *IEEE FOCS*, 2000.
- [37] INDYK, P., “Stable distributions, pseudorandom generators, embeddings, and data stream computation,” *Journal of the ACM*, vol. 53, no. 3, pp. 307–323, 2006.
- [38] JIANG, L. and WALRAND, J., “Stable and utility-maximizing scheduling for stochastic processing networks,” in *Allerton Conference*, September 2009.
- [39] KAR, K., SARKAR, S., and TASSIULAS, L., “A low-overhead rate control algorithm for maximizing aggregate receiver utility for multirate multicast sessions,” in *SPIE ITCOM*, 2001.
- [40] KAR, K., SARKAR, S., and TASSIULAS, L., “Optimization based rate control for multirate multicast sessions,” in *IEEE INFOCOM*, pp. 123–132, 2001.
- [41] KELLY, F., MAULLOO, A., and TAN, D., “Rate control in communication networks: shadow prices proportional fairness and stability,” *Journal of the Operational Research Society*, vol. 49, no. 3, pp. 237–252, 1998.
- [42] KUNNIYUR, S. and SRIKANT, R., “End-to-end congestion control schemes: Utility functions, random losses and ECN marks,” in *INFOCOM*, pp. 1323–1332, 2000.
- [43] LIN, B. and XU, J., “DRAM is plenty fast for wirespeed statistics counting,” in *ACM HotMetrics*, June 2008.
- [44] LIN, W., REINHARDT, S. K., and BURGER, D., “Reducing DRAM latencies with an integrated memory hierarchy design,” in *Proc. of IEEE HPCA*, (Washington, DC, USA), p. 301, 2001.

- [45] LIN, X. and SHROFF, N. B., “The impact of imperfect scheduling on crosslayer rate control in multihop wireless networks,” in *Proc. of IEEE INFOCOM*, 2005.
- [46] LIU, S., BASAR, T., and SRIKANT, R., “Controlling the internet: A survey and some new results,” in *IEEE Conference on Decision and Control*, 2003.
- [47] LIU, Z., TANG, A., XIA, C., and ZHANG, L., “A decentralized control mechanism for stream processing networks,” *Annals of Operations Research*, vol. 170, pp. 161–182, 2008.
- [48] LOW, S. H. and LAPSLEY, D. E., “Optimization flow control–i: basic algorithm and convergence,” *IEEE/ACM Trans. Netw.*, vol. 7, no. 6, pp. 861–874, 1999.
- [49] LU, Y., MONTANARI, A., PRABHAKAR, B., DHARMAPURIKAR, S., and KARBANI, A., “Counter braids: A novel counter architecture for per-flow measurement,” in *ACM SIGMETRICS*, 2008.
- [50] MANJHI, A., SHKAPENYUK, V., DHAMDHERE, K., and OLSTON, C., “Finding (recently) frequent items in distributed data streams,” in *Proceedings of International Conference on Data Engineering (ICDE)*, 2005.
- [51] MARSHALL, A. W. and OLKIN, I., *Inequalities: Theory of Majorization and Its Applications*. Academic Press, 1979.
- [52] MO, J. and WALRAND, J., “Fair end-to-end window-based congestion control,” *IEEE/ACM Trans. on Networking*, vol. 8, no. 5, pp. 556–567, 2000.
- [53] MORRIS, R., “Counting large numbers of events in small registers,” *Commun. ACM*, vol. 21, no. 10, 1978.
- [54] MOTWANI, R. and OTHERS, “Query processing, approximation, and resource management in a data stream management system,” in *CIDR*, 2003.
- [55] MOTWANI, R. and RAGHAVAN, P., *Randomized Algorithms*. Cambridge, 1995.
- [56] MULLER, A. and STOYAN, D., *Comparison Methods for Stochastic Models and Risks*. Wiley, 2002.
- [57] NEELY, M., MODIANO, E., and LI, C., “Fairness and optimal stochastic control for heterogeneous networks,” in *IEEE INFOCOM*, March 2005.
- [58] OLSTON, C., JIANG, J., and WIDOM, J., “Adaptive filters for continuous queries over distributed data streams,” in *Proceedings of ACM SIGMOD*, 2003.
- [59] PANDIT, C. and MEYN, S., “Worst-case large-deviation asymptotics with application to queueing and information theory,” *Stochastic Processes and their Applications*, vol. 116, no. 5, pp. 724–756, 2006.
- [60] PATTERSON, D. and HENNESSY, J., *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2nd ed., 1996.

- [61] RAMABHADRAN, S. and VARGHESE, G., “Efficient implementation of a statistics counter architecture,” *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 261–271, 2003.
- [62] RAU, B. R., “Pseudo-randomly interleaved memory,” in *Proc. 18th Annual International Symposium on Computer Architecture*, 1991.
- [63] ROCKAFELLAR, R. T., *Convex Analysis*. Princeton University Press, 1970.
- [64] ROEDER, M. and LIN, B., “Maintaining exact statistics counters with a multi-level counter memory,” in *IEEE GLOBECOM*, vol. 2, pp. 576–581, Nov - Dec 2004.
- [65] SHAH, D., IYER, S., PRAHHAKAR, B., and MCKEOWN, N., “Maintaining statistics counters in router line cards,” *Micro, IEEE*, vol. 22, pp. 76–81, Jan/Feb 2002.
- [66] SHOR, N. Z., *Minimization Methods for Non-differentiable Functions*. Springer-Verlag, 1985.
- [67] SHRIMALI, G. and MCKEOWN, N., “Building packet buffers using interleaved memories,” in *Workshop on High Performance Switching and Routing (HPSR)*, May 2005.
- [68] SOMMERS, J., BARFORD, P., DUFFIELD, N., and RON, A., “Accurate and efficient sla compliance monitoring,” in *Proceedings of ACM SIGCOMM*, 2007.
- [69] STANOJEVIC, R., “Small active counters,” in *IEEE Infocom*, 2007.
- [70] TASSIULAS, L. and EPHREMIDES, A., “Stability properties of constrained queueing systems and scheduling for maximum throughput in multihop radio networks,” *IEEE Transactions on Automatic Control*, vol. 37, no. 12, pp. 1936–1949, 1992.
- [71] TATBUL, N., ÇETINTEMEL, U., ZDONIK, S., CHERNIACK, M., and STONEBRAKER, M., “Load shedding in a data stream manager,” in *VLDB*, 2003.
- [72] TURAGA, D., VERSCHEURE, O., and V. U. CHAUDHARI, L. A., “Resource management for networked classifiers in distributed stream mining systems,” in *Proc. of the Sixth International Conference on Data Mining (ICDM’06)*, Dec. 2006.
- [73] WARE, F. A. and HAMPEL, C., “Micro-threaded row and column operations in a dram core,” *Rambus White Paper*, Mar 2005.
- [74] WARE, F. and HAMPEL, C., “Improving power and data efficiency with threaded memory modules,” in *International Conference on Computer Design (ICCD)*, pp. 417–424, Oct. 2006.

- [75] YING, L., LIU, Z., TOWSLEY, D., and XIA, C. H., “Distributed operator placement and data caching in large-scale sensor networks,” in *IEEE INFOCOM*, 2008.
- [76] ZHAO, H., LALL, A., OGIHARA, M., SPATSCHECK, O., WANG, J., and XU, J., “A data streaming algorithm for estimating entropies of OD flows,” in *ACM IMC*, 2007.
- [77] ZHAO, H., LALL, A., OGIHARA, M., and XU, J., “Global iceberg detection over distributed data streams,” in *IEEE ICDE*, 2010.
- [78] ZHAO, H., WANG, H., LIN, B., and XU, J., “Design and performance analysis of a DRAM-based statistics counter array architecture,” in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2009.
- [79] ZHAO, H., XIA, C. H., LIU, Z., and TOWSLEY, D., “Distributed resource allocation for synchronous fork and join processing networks,” in *IEEE Infocom Mini-conference*, 2010.
- [80] ZHAO, H., XIA, C. H., LIU, Z., and TOWSLEY, D., “A unified modeling framework for distributed resource allocation of general fork and join processing networks,” in *ACM Sigmetrics*, 2010.
- [81] ZHAO, Q., OGIHARA, M., WANG, H., and XU, J., “Finding global icebergs over distributed data sets,” in *Proceedings of the Symposium on Principles of Database Systems (PODS)*, 2006.
- [82] ZHAO, Q., XU, J., and LIU, Z., “Design of a novel statistics counter architecture with optimal space and time efficiency,” *SIGMETRICS Perform. Eval. Rev.*, vol. 34, no. 1, pp. 323–334, 2006.