UNIVERSITY OF CALIFORNIA, SAN DIEGO

Designing Network Traffic Managers with Throughput, Fairness, and Worst-case Performance Guarantees

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Electrical Engineering
(Communication Theory and Systems)

by

Hao Wang

Committee in charge:

Professor Bill Lin, Chair
Professor Pamela C. Cosman
Professor Sujit Dey
Professor Rajesh K. Gupta
Professor Andrew B. Kahng
Professor Tajana S. Rosing

2011

The dissertation of Hao Wang is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2011

DEDICATION

To my parents.

EPIGRAPH

If one learns but does not think, one will be bewildered.
If one thinks but does not learn from others, one will be imperiled.
—Confucius, Analects, c.400 B.C.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First and foremost, I owe my deepest gratitude to my advisor, Professor Bill Lin, who has supported me throughout my graduate study with both wisdom and encouragement. He is the one who guided me in my research projects, inspired me with his keen intuitions and logical thinking, and offered me help when there was turmoil in my life.

I give my thanks to my coauthors, Professor Jun Xu and his students Haiquan Zhao and Nan Hua from Georgia Institute of Technology for their help and contributions towards this dissertation. I will always remember the delightful discussions with Professor Xu on work and life. I really appreciate the help from Haiquan on both research projects and my career. My lifetime friendship with Nan will always be with me.

I thank my dissertation committee members, Professor Pamela C. Cosman, Professor Sujit Dey, Professor Rajesh K. Gupta, Professor Andrew B. Kahng, and Professor Tajana S. Rosing, for their insightful comments and advice.

I thank my internship mentors Sailesh Kumar and William Lynch at Huawei Technologies (USA), and Rong Pan and Flavio Bonomi at Cisco Systems, from whom I gained valuable industry experience. I thank the friends I made during my internships, Bin Xiao, Deming Liu, Haoyu Song, Wei Cao, Zhen Chen, Hengwen Tong, Ken Yi, Rongfeng Hong, Joji Philip, Xiangyang Zhang, Hongbo Yang, Jifei Song, Wumao Chen, Patrick Liu, Zhaoming Hu, and Fei Xu at Huawei Technologies (USA), and Michele Caramello, Chiara Piglione, and Mei Wang at Cisco Systems.

I would also like to thank my colleagues and friends at UCSD, especially Rohit Ramanujam, Shan Yan, Jerry Chou, Ting-Lan Lin, Chia-Wei Chang, Junsheng Han, Zheng Wu, Seyhan Karakulak, Rathinakumar Appuswamy, Amir Hadi Djahanshahi, and Abhijeet Bhorkar for providing me with an enjoyable research and learning environment.

Lastly but most importantly, I thank my parents, Liquan Wang and Xiangfen Zhang, for their love and sacrifice that made everything possible. I would have been nothing without them in every conceivable way, and to them I dedicate this dissertation.

Chapter 2, in full, is a reprint of the material as it appears in the following pub-

lications:

- Hao Wang, Haiquan (Chuck) Zhao, Bill Lin, and Jun (Jim) Xu, "DRAM-Based Statistics Counter Array Architecture with Performance Guarantee", *IEEE/ACM Transactions on Networking (ToN)*, 2011.

- Haiquan (Chuck) Zhao, Hao Wang, Bill Lin, and Jun (Jim) Xu, "Design and Performance Analysis of a DRAM-based Statistics Counter Array Architecture", *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, Princeton, NJ, October 19-20, 2009.

- Bill Lin, Jun (Jim) Xu, Nan Hua, Hao Wang, and Haiquan (Chuck) Zhao, "A Randomized Interleaved DRAM Architecture for the Maintenance of Exact Statistics Counters", *ACM Special Interest Group on Performance Evaluation (SIGMETRICS)*, Seattle, WA, June 15-19, 2009.

The dissertation author was the primary investigator and author of the papers.

Chapter 3, in part, has been submitted for publication of material as it may appear in the IEEE Transactions on Parallel and Distributed Systems, Hao Wang, Bill Lin, and Jun (Jim) Xu, "Robust Statistics Counter Arrays with Interleaved Memories". The dissertation author was the primary investigator and author of the paper.

Chapter 4, in part, is a reprint of the material as it appears in the following publications:

- Hao Wang, Haiquan (Chuck) Zhao, Bill Lin, and Jun (Jim) Xu, "Robust Pipelined Memory System with Worst Case Performance Guarantee", *IEEE Transactions on Computers (TC)*, 2011.

- Hao Wang, Haiquan (Chuck) Zhao, Bill Lin, and Jun (Jim) Xu, "Design and Analysis of a Robust Pipelined Memory System", *IEEE International Conference on Computer Communications (INFOCOM)*, San Diego, CA, March 15-19, 2010.

The dissertation author was the primary investigator and author of the papers.

Chapter 5, in part, is a reprint of the material as it appears in the following publications:

- Hao Wang and Bill Lin, "Per-flow Queue Scheduling with Pipelined Counting Priority Index", *IEEE Symposium on High-Performance Interconnects (HOTI)*, Santa Clara, CA, August 24-26, 2011.

- Hao Wang and Bill Lin, "Succinct Priority Indexing Structures for the Management of Large Priority Queues", *IEEE International Workshop on Quality of Service (IWQoS)*, Charleston, SC, July 13-15, 2009.

Chapter 5, in full, has been submitted for publication of material as it may appear in IEEE Transactions on Parallel and Distributed Systems, Hao Wang and Bill Lin, "Per-flow Queue Management with Succinct Priority Indexing Structures for High Speed Packet Scheduling". The dissertation author was the primary investigator and author of the papers.

Chapter 6, in part, is a reprint of the material as it appears in the following publications:

- Hao Wang and Bill Lin, "Block-Based Packet Buffer with Deterministic Packet Departures", *IEEE International Conference on High Performance Switching (HPSR)*, Dallas, TX, June 13-16, 2010.

- Hao Wang and Bill Lin, "A Block-Based Reservation Architecture for the Implementation of Large Packet Buffers", *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, Princeton, NJ, October 19-20, 2009.

The dissertation author was the primary investigator and author of the papers.

VITA

2005	Bachelor of Engineering in Electronic Engineering, Tsinghua University, Beijing, P. R. China
2008	Master of Science in Electrical Engineering (Communication Theory and Systems), University of California, San Diego
2008	Intern, Advance Architecture and Research Group, Cisco Systems, San Jose, California
2011	Intern, American Network Division, Huawei Technologies (USA), Santa Clara, California
2011	Doctor of Philosophy in Electrical Engineering (Communication Theory and Systems), University of California, San Diego

PUBLICATIONS

Hao Wang, Haiquan (Chuck) Zhao, Bill Lin, and Jun (Jim) Xu, "DRAM-Based Statistics Counter Array Architecture with Performance Guarantee", *IEEE/ACM Transactions on Networking (ToN)*, 2011.

Hao Wang, Haiquan (Chuck) Zhao, Bill Lin, and Jun (Jim) Xu, "Robust Pipelined Memory System with Worst Case Performance Guarantee", *IEEE Transactions on Computers (TC)*, 2011.

Hao Wang and Bill Lin, "Per-flow Queue Scheduling with Pipelined Counting Priority Index", *IEEE Symposium on High-Performance Interconnects (HOTI)*, Santa Clara, CA, August 24-26, 2011.

Hao Wang and Bill Lin, "Designing Efficient Codes for Synchronization Error Channels", *IEEE International Workshop on Quality of Service (IWQoS)*, San Jose, CA, June 5-7, 2011.

Hao Wang and Bill Lin, "Block-Based Packet Buffer with Deterministic Packet Departures", *IEEE International Conference on High Performance Switching (HPSR)*, Dallas, TX, June 13-16, 2010.

Hao Wang, Haiquan (Chuck) Zhao, Bill Lin, and Jun (Jim) Xu, "Design and Analysis of a Robust Pipelined Memory System", *IEEE International Conference on Computer Communications (INFOCOM)*, San Diego, CA, March 15-19, 2010.

Hao Wang and Bill Lin, "A Block-Based Reservation Architecture for the Implementation of Large Packet Buffers", *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, Princeton, NJ, October 19-20, 2009.

Haiquan (Chuck) Zhao, Hao Wang, Bill Lin, and Jun (Jim) Xu, "Design and Performance Analysis of a DRAM-based Statistics Counter Array Architecture", *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, Princeton, NJ, October 19-20, 2009.

Hao Wang and Bill Lin, "Succinct Priority Indexing Structures for the Management of Large Priority Queues", *IEEE International Workshop on Quality of Service (IWQoS)*, Charleston, SC, July 13-15, 2009.

Bill Lin, Jun (Jim) Xu, Nan Hua, Hao Wang, and Haiquan (Chuck) Zhao, "A Randomized Interleaved DRAM Architecture for the Maintenance of Exact Statistics Counters", *ACM Special Interest Group on Performance Evaluation (SIGMETRICS)*, Seattle, WA, June 15-19, 2009.

Hao Wang and Bill Lin, "Pipelined van Emde Boas Tree: Algorithms, Analysis, and Application", *IEEE International Conference on Computer Communications (INFOCOM)*, Anchorage, AK, May 7-11, 2007.

Hao Wang and Bill Lin, "On the Efficient Implementation of Pipelined Heaps for Network Processing", *IEEE Global Communications Conference (GLOBECOM)*, San Francisco, CA, November 27-December 1, 2006.

Hao Wang, Bill Lin, and Jun (Jim) Xu, "Robust Statistics Counter Array with Interleaved Memories", *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, under review.

Hao Wang and Bill Lin, "Efficient Channel Codes for Synchronization Error Correction", *IEEE Transactions on Communications (TCOM)*, under review.

Hao Wang and Bill Lin, "Per-flow Queue Management with Succinct Priority Indexing Structures for High Speed Packet Scheduling", *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, under review.

ABSTRACT OF THE DISSERTATION

Designing Network Traffic Managers with Throughput, Fairness, and Worst-case Performance Guarantees

by

Hao Wang

Doctor of Philosophy in Electrical Engineering
(Communication Theory and Systems)

University of California, San Diego, 2011

Professor Bill Lin, Chair

On the Internet, network routers are typically implemented to provide strategic controls over the growing demands on limited and expensive bandwidth for an increasingly diverse traffic spectrum. A router consists of two major components: a set of switch fabric and multiple linecards. The functionalities of a linecard can be categorized into three parts: packet classification, statistics accounting, and packet scheduling. Packet classification includes functionalities such as routing table lookup, admission control, and deep packet inspection. Statistics accounting is implemented to store essential information such as flow statistics and network counting sketches, for the purpose of traffic monitoring and traffic management. Packet scheduling includes functionalities

such as packet buffering, packet shaping, rate control, and hierarchical queue management. Sophisticated algorithms have been developed to improve the throughput and fairness on the network, however, the costs of implementing the new algorithms constantly outweigh their performance gains. This dissertation focuses on bridging the gap between advanced algorithms and their implementations in real-world network equipments by adopting a throughput- and fairness-driven design of network traffic managers which incorporate most of the functionalities of statistics accounting and packet scheduling, while providing worst-case performance guarantees for the whole router system. First, we explore parallelism to design high throughput statistics counter arrays that are robust against adversarial traffic. Second, we develop robust pipelined memory systems with worst-case performance guarantees for network processing. In our new memory systems, memory operations are finished within a fixed delay, which greatly simplifies the designs of network processors. Third, we present novel succinct priority index data structures that can be implemented for scheduling packets maintained in a large number of priority queues at line rate, which is essential in providing quality-of-service for per-flow queueing. Last, we show several reservation-based packet buffer architectures with interleaved memories that take advantage of the known packet departure times to achieve simplicity and determinism. They are scalable to growing packet storage requirements in routers to provide fine-grained per-flow queue buffering, while matching increasing line rates. All these approaches significantly improve the overall system throughput while providing better fairness, quality-of-service and worst-case performance guarantees over existing solutions.

Chapter 1

Introduction

1.1 Network Traffic Management

Network traffic management has been the focus of research in the networking community in the past decade. The performance of a network is typically measured by its throughput, fairness, quality-of-service (QoS), reliability, response time, latencies, etc. Network traffic management transforms the network into a manageable source. It includes essential functionalities to provide QoS guarantees for millions of flows on the network at the line rate, to buffer packets at the line rate in order to cope with packet retransmissions and network congestions, and to maintain flow statistics to provide support for network accounting, network planning, network forecasting, security monitoring, and many other features.

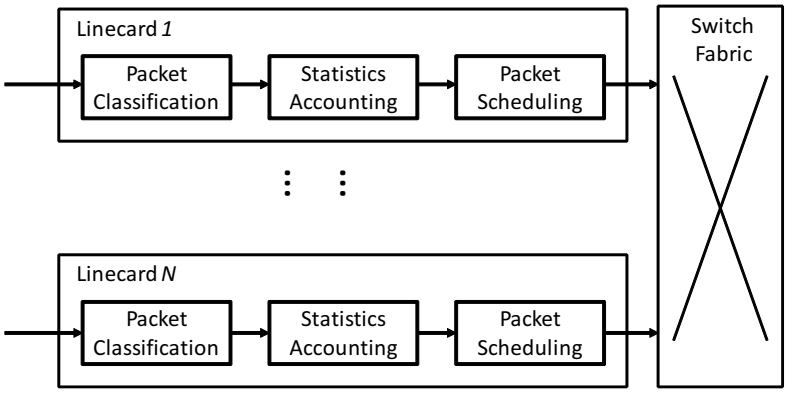


Figure 1.1: Router architecture.

The growing demands on limited and expensive bandwidth for an increasingly diverse traffic spectrum require strategic controls, which are typically implemented in network routers. The architecture of a typical router is shown in Figure 1.1. A router consists of two major components: a set of switch fabrics and multiple linecards. There have been extensive research efforts on the designs and implementations of switch fabrics [4, 15–17, 47, 56, 62, 69, 93] in order to provide high throughput in sending packets from input linecards to their destination output linecards. In general, current switch fabric designs can be categorized into two groups. One group includes the designs with a centralized arbiter configuring the interconnection of linecards at any instant of time to optimize the number of packets that can be sent across the switch fabric, which is essentially solving a matching problem. Various algorithms have been proposed to solve the matching problem on a switch fabric, such as output scheduling [69], wavefront arbiter (WFA) [93], parallel iterative matching (PIM) [4], and iSLIP [62]. The other group includes the designs without a centralized arbiter, where load-balancing techniques are typically deployed to better utilize the links on a switch fabric. Load-balanced routers are first proposed in [15, 16] which do not require a centralized arbiter unit to make matching decisions. In general, they suffer from the problem that packets sent through the switch fabric may be miss-sequenced, which would dramatically degrade the performance for TCP traffic by causing excessive retransmissions [2]. Many load-balance router extensions have been proposed to provide for in-order packet departures, such as the full-frame-first switch [47], the mailbox switch [17], and the concurrent matching switch [56]. State-of-the-art switch fabric technologies are mature enough so that various merchant chips have been built to implement these algorithms.

Inside a linecard, there are three major components provided: packet classification, statistics accounting, and packet scheduling. Packet classification includes functionalities such as access control list [55, 75], routing table lookup [61, 92], admission control [25, 103], and deep packet inspection [23, 53, 104], which have all been well studied and widely deployed in network routers. This dissertation focuses on the other two components, statistics accounting and packet scheduling, which are essential for network traffic management.

1.1.1 Statistics Accounting

Statistics accounting is essential in network performance measurement, router management, intrusion detection, traffic engineering, and data streaming management. After a packet is processed by the packet classification module, its related information such as flow record and packet size will be recorded by the statistics accounting module. Statistics counter arrays are typically implemented to store the information for statistics analysis. For example, by counting the total size of the packets with the same flow record, it is possible to study the network traffic pattern, distinguish heavy users, and quickly identify network abnormalities such as a virus exploitation. The QoS provided by the traffic manager also relies heavily on the information recorded in the statistics counters. For example, counters are essential in providing committed access rate (CAR) support, which includes the committed information rate (CIR) and the peak information rate (PIR) statistics, so that a traffic manager can offer differentiated services to different flows based on the service level agreements (SLA) of their fair-share bandwidth and the amount of bandwidth they are actually utilizing. Counters are implemented to keep track of the leftover bandwidth for each specific type of traffic, which can be implemented at flow, user group, virtual queue, port, or other service levels. For the packets belonging to the same traffic type, the corresponding counter values will be decremented according to arriving packet sizes to reflect the leftover bandwidth. On the other hand, the counter values need to be incremented periodically to reflect the fair-share bandwidth allocated to that traffic type for the next service period. It is easy to see that the bandwidth can be more accurately managed if the service period is made arbitrarily small and the amount of increments to the counters in each service period are reduced accordingly. Therefore, it is highly desirable to build counters that support both increment and decrement operations at high speed. Moreover, a practical counter array solution needs to be able to cope with any arbitrary incoming sequence of counter update requests. This is true especially for security applications such as intrusion detection where an adversary has incentives to compromise any performance guarantees provided by the statistics counter array system. The problem of efficiently maintaining a large number (say millions) of statistics counters that need to be updated at very high speeds (e.g. 40 Gb/s) has received considerable research attention in recent years [20, 36, 37, 59, 65, 76, 81, 87, 91, 108]. It proves

too costly to store such large counter arrays entirely in static random access memory (SRAM) while dynamic random access memory (DRAM) is viewed as too slow for providing wirespeed updates at high line rates.

1.1.2 Packet Scheduling

The packet scheduling module includes functionalities such as managing the packets stored in a packet buffer, implementing a traffic shaper based on the SLA, and also scheduling packets for departure.

Network routers need to temporarily store a large number of packets in response to network congestion and retransmission. Packet buffers are constantly built into linecards on the commercial routers deployed at the center or edge of the network. When a packet arrives at a router, it will be stored in a packet buffer for a period of time which typically corresponds to the round-trip-time of the network. The common buffer sizing rule is that $B = RTT \times C$, where B is the size of the buffer, RTT is the average round-trip-time of a flow passing through the link, and C is the line rate [96]. At an average round-trip-time of 250 ms on the Internet, and a line rate of 100 Gb/s (100GE), it translates to a buffer size of 3.125 GB at each linecard. A packet buffer implemented with SRAM is clearly infeasible, since the state-of-the-art SRAM has a capacity of only about 32 Mb. On the other hand, there are typically millions of flows arriving at the router. Packets are necessarily aggregated in the packet buffer physically or logically according to the flows they belong to in order to provide for differentiated services. Later, these packets need to be retrieved from the corresponding flows when they are due for departure. On a 100 Gb/s link, a minimum sized packet of 64 bytes may arrive every 5 ns. In order to match the line rate, the corresponding packet insertion to the packet buffer has to be completed within this time frame. Also, a packet previously inserted into the packet buffer may need to be removed for departure within the same time frame. Current DRAM has a random access latency of about 32 ns, which makes a naïve DRAM implementation of the packet buffer too slow for the current and future line rates.

Based on the SLA, a traffic shaper makes use of the information collected by the statistics accounting module to manage the packets stored in a packet buffer. For

example, if a flow is sending at a rate lower than its committed rate, the packets in the flow will not incur any further delay in the traffic shaper. However, if the flow is sending at a rate higher than its committed rate, the packets in the flow will experience a certain delay in order to effectively "smooth" the flow. If the flow is sending at a rate higher than the peak allowable rate, its packets will be labeled accordingly so that they can be dropped or processed at a much lower priority by the packet scheduling module. The information collected by the statistics accounting module and the updated information from the traffic shaper need to be stored in a memory system that is robust against adversarial attacks, while providing high throughput to meet the ever-increasing line rate. Many other network applications also require wirespeed accesses to large data structures or a large amount of packet and flow-level data, such as routing table lookups, admission control, and deep packet inspection, which all require memory systems specialized for network processing. It is essential for the memory systems of a router to be able to support both read and write accesses to such data at link speeds. As link speeds continue to increase, router designers are constantly grappling with the necessary tradeoffs between the speed and cost of SRAM and DRAM. The capacity of SRAMs is woefully inadequate in many cases and it generally proves too costly to store large data structures entirely in SRAM, while DRAM is too slow for providing wirespeed updates at high line rates.

In the advanced scheduling of per-flow queues with QoS requirements, priority queues have received the most attention. Different flows may have different QoS requirements that correspond to different data rates, end-to-end latencies, and packet loss rates. A single queue cannot satisfy the service requirement since it only supports first-come-first-served policy. In order to achieve differentiated service guarantees, per-flow queueing is necessarily in advanced high-performance routers to manage packets for different flows in separate logical queues. Scheduling algorithms are required to decide the order of service for per-flow queues at line rate. Most of the advanced scheduling techniques are based on assigning deadline timestamps to packets depending on their urgency, and on servicing flows in the earliest-deadline-first order. This earliest-deadline-first scheduling approach can be facilitated using a priority queue. Scalable priority queue implementation requires solutions to two fundamental problems. The

first is to sort queue elements in real-time at ever increasing line speeds. The second is to effectively maintain a huge number of packets upon packet arrival and schedule them on time for departure.

In general, the advancement of network processor designs and faster memory systems make the current network equipments more capable than their predecessors in processing speed, power efficiency, and latency. However, the performance gains are easily overshadowed by the growing demands pushed by the ever-increasing line rate. Sophisticated algorithms have been developed to provide for improved throughput or fairness on the network. Many of the algorithms, however, are too complex to be implemented in any commercially feasible system, or the costs of implementing them constantly outweigh the benefits of the performance improvements. This dissertation aims at bridging the gap between advanced algorithms and their implementations in real-world network systems in order to build network traffic managers with high throughput, better fairness, and worst-case performance guarantees.

1.2 Techniques for Providing Throughput, Fairness, and Worst-case Performance Guarantees

There are several key techniques adopted in this dissertation to improve the throughput, fairness, and worst-case performance of traffic managers.

1.2.1 Pipelined Architectures

The principle of pipelining has been a major architectural attribute of current computer systems, which all have pipeline processing capabilities in the form of internally pipelined instruction and arithmetic units or in the form of pipelined special purpose functional units [77]. Network equipments can benefit from pipelining as well to achieve higher throughput without reducing the latency of each operation by dividing sequential processes into subprocesses, each of which can be executed efficiently on a special dedicated module. The higher throughput is achieved by arranging the special dedicated modules to operate concurrently. An example of a three-stage pipeline with

three concurrent jobs is shown in Figure 1.2. Even though each job shown in the figure takes three cycles to finish, the amortized time complexity for the pipeline is only one cycle when it is running at full capacity.

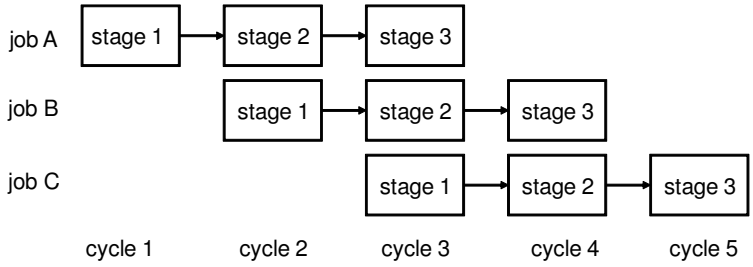


Figure 1.2: A three-stage pipeline with three concurrent jobs.

Pipelined architecture is the key technique to achieving high throughput implementation of priority queues for per-flow scheduling. On the Internet with millions of concurrent flows, a priority queue implementation needs to locate the packet with the earliest departure time for service at line rate among all flows. In the worst case with minimum size ethernet packets, as many as 150 million packets need to be scheduled for departure in every second on a 100 Gb/s link. As the line rate increases, the throughput of the packet scheduler needs to be increased accordingly to accommodate the worst-case scenario where all the packets are of the smallest size. Such a high throughput is only achievable with pipelined architectures.

1.2.2 Load-balancing and Parallelism

Load-balancing and parallelism have been widely adopted in the networking area, such as in the design of load-balanced routers [15, 16]. Parallelism refers to the practice of using multiple slow systems to match the behaviors of a much faster system in terms of throughput, fairness, and worst-case performance. Load-balancing refers to the practice of distributing work among slow systems so that the work-load to different systems are evenly distributed. Efficient and effective load-balancing algorithms and techniques are keys to building high performance and robust parallel systems with worst-case performance guarantees which can mimic the behavior of a faster system. Load-balancing and parallelism will be applied extensively in this dissertation to provide both

high throughput and guaranteed worst-case performance for statistics counter arrays, robust network memory systems, and also for high speed packet buffers.

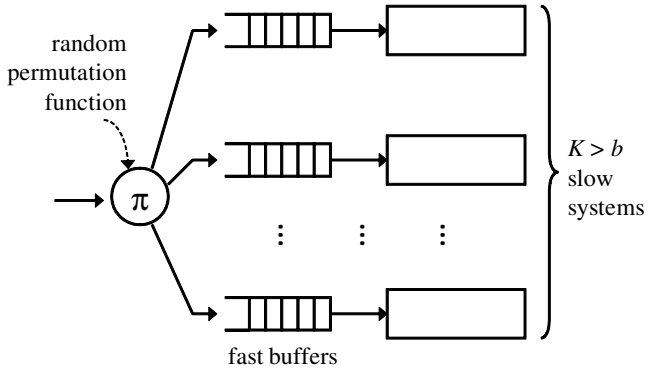


Figure 1.3: A parallel system architecture.

Figure 1.3 shows the architecture of a parallel system consisting of K slow systems and K fast buffers. Assume that a fast system is b times faster than each of the slow systems, Then by designing $K > b$, the parallel system potentially provides a higher throughput than the faster system. With an effective load-balancing algorithm which is implemented as a random permutation function π, the average work-load can be distributed into the K slow systems uniformly-at-random [14]. The K fast buffers are implemented to temporarily store the pending work to a slow system, since the work-load to different slow systems can never be perfectly equal due to randomness as exemplified by the Birthday Paradox [63]. Such a design is not robust, however, against worst-case work-load distributions, as there can be certain incoming work-load patterns that will cause more work to be sent to certain slow systems, causing system overflow by flooding the fast buffers. In this dissertation, we improve the parallel system design of Figure 1.3 by developing robust architectures for statistics counter arrays and network memory systems. We also build robust deterministic packet buffer systems with parallel memories by adding more constraints to the packet arrival processes.

1.3 Problem Statement and Contributions

In this section, we formally define the problem targeted in this dissertation:

How can one design traffic managers with throughput, fairness, and quality-of-service guarantees, while providing worst-case performance guarantees?

We solve this problem by improving the throughput, fairness, and worst-case performance of several components in a traffic manager. In particular, we design parallel statistics counter arrays with load-balancing algorithms that are robust against adversarial arriving counter updates while matching the line rate throughput. We design a pipelined memory system for network processing that is robust against even the worst-case memory access patterns. We design succinct priority indexing structures with pipelined architectures for the management of large priority queues at line rate. Also, we design reservation-based packet buffers that can guarantee deterministic packet departures with the help of both pipelined architectures and parallelism.

The main contributions of the dissertation are as follows:

- **Robust Statistics Counter Arrays:** We propose two DRAM based counter architectures that can effectively maintain wirespeed updates to large counter arrays. Both of them can harness the performance of modern commodity DRAM offerings by interleaving counter updates to multiple memory banks. The first proposed architecture makes use of a simple randomization scheme, a small cache, and small request queues to statistically guarantee a near-perfect load-balancing of counter updates to the DRAM banks. The statistical guarantee of the proposed randomized scheme is proven using a novel combination of convex ordering and large deviation theory. The second architecture is based on the observation that most flows on the Internet consist of multiple packets that are transmitted during a relatively short period of time, which are described as traffic bursts. This architecture makes use of a simple randomization scheme and a set of small, fully associative request queues to statistically guarantee a near-perfect load-balancing of counter updates to the memory banks. This architecture explores the benefits of traffic bursts to greatly reduce the size of the request queues while providing a diminishing overflow probability guarantee. We also develop a queueing model to show that, as long as the flow sizes are heavy-tailed distributed, the maximum request queue length is always bounded by a finite number. Our simulations confirm the effectiveness of the queueing model. Both of our proposed statistics

counter array architectures can support arbitrary increments and decrements at wirespeed, and can support different number representations, including both integer and floating point number representations. They can effectively maintain wirespeed updates to large counter arrays while providing a diminishing system overflow probability. The two statistics counter arrays are described in details in Chapter 2 and Chapter 3, respectively.

- **Memory Systems for Network Processing:** We propose a robust pipelined memory architecture that can emulate an ideal SRAM by guaranteeing with very high probability that the output sequence produced by the pipelined memory architecture is the same as the one produced by an ideal SRAM under the same sequence of memory read and write operations, except time-shifted by a fixed pipeline delay of Δ. Given a fixed pipeline delay abstraction, no interrupt mechanism is required to indicate when read data is ready or a write operation has completed, which greatly simplifies the use of the proposed solution. The design is based on the interleaving of DRAM banks together with the use of a reservation table that serves in part as a data cache. In contrast to prior interleaved memory solutions, this design is robust under all memory access patterns, including adversarial ones, which makes our robust pipelined memory suitable for network applications, particularly network security applications, where worst-case performance is the focus of concern. The details of this work are discussed in Chapter 4.

- **Pipelined Architectures for Per-flow Scheduling:** Priority queues are essential building blocks for implementing advanced per-flow disciplines and hierarchical QoS at high-speed network links. Scalable priority queue implementation requires solutions to two fundamental problems. The first is to sort queue elements in real-time at ever increasing line speeds (e.g., at OC-768 rates). The second is to store a huge number of packets (e.g., millions of packets). We present novel solutions by decomposing the problems into two parts, a succinct priority index in SRAM that can efficiently maintain a real-time sorting of priorities, coupled with a DRAM-based implementation of large packet buffers. In particular, we propose three related novel succinct priority index data structures for implementing high-speed priority indexes: a Priority-Index (PI), a Counting-Priority-Index

(CPI), and a pipelined Counting-Priority-Index (pCPI). We show that all three of these structures can be very compactly implemented in SRAM using only $\Theta(U)$ space, where U is the size of the universe required to implement the priority keys (timestamps). We also show that the proposed priority index structures can be implemented very efficiently as well by leveraging hardware-optimized instructions that are readily available in modern 64-bit processors. The operations on the PI and CPI structures take $\Theta(\log_W U)$ time complexity, where W is the processor word-length (i.e., $W = 64$). Alternatively, operations on the pCPI structure take amortized constant time with only $\Theta(\log_W U)$ pipeline stages (e.g., only 4 pipeline stages for $U = 16$ million). Finally, we show the application of our proposed priority index structures for the scalable management of large packet buffers at line speeds. The pCPI structure can be implemented efficiently in high-performance network processing applications such as advanced per-flow scheduling with QoS guarantee. The details are presented in Chapter 5.

- **Packet Buffers with Deterministic Packet Departure:** Existing DRAM-based architectures for supporting linespeed queue operations can be classified into three categories: prefetching-based, randomization-based, and reservation-based. They are all based on interleaving memory accesses across multiple parallel DRAM banks for achieving higher memory bandwidths, but they differ in their packet placement and memory operation scheduling mechanisms. In this dissertation, we present efficient reservation-based packet buffer architectures with interleaved memories that take advantage of the known packet departure times to achieve simplicity and determinism. We develop three new packet buffer architectures: a deterministic frame-sized packet buffer, a deterministic block-sized packet buffer, and a randomized block-sized packet buffer. The number of interleaved DRAM banks required to implement the proposed packet buffer architectures is independent of the arrival traffic pattern, the number of active flows, and the number of priority classes, yet the proposed architectures can achieve the performance of an SRAM implementation with throughput and worst-case performance guarantee. In particular, the proposed block-based solution achieves an order of magnitude reduction in the total SRAM size compared to previous solutions. It is scalable

to growing packet storage requirements in future routers while matching increasing line rates. The design, implementation, and evaluation of the proposed packet buffer architectures are described in Chapter 6.

Chapter 2

Maintaining State Information with Millions of Counters

2.1 Introduction

It is widely accepted that network measurements are 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 measurements, router management, intrusion detection, traffic engineering, and data streaming applications, there is often the need to maintain very large arrays of statistics counters at wirespeed (e.g. many millions of counters for per-flow measurements [76, 87]). 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 a 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 frame. Large counters, such as 64 bits wide, are needed for tracking accurate counts even in short time windows if the measurements take place on high-speed links as smaller counters can quickly overflow. Additionally, a practical counter array solution has to be able to handle any arbitrary sequence including adversarial incoming sequences of counter addresses, i.e. indices, to be incremented because statistics counter arrays may often be used in security applications, such as intrusion detection, and in settings where an adversary has incentives to compromise the perfor-

mance guarantees.

Although implementing large counter arrays in SRAM can satisfy the performance needs, the amount of SRAM required can be both infeasible and impractical. As reported in [108], 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 been actively seeking alternative ways to realize large arrays of statistics counters at wirespeed [20, 36, 37, 59, 65, 76, 81, 87, 91, 108].

Several designs of large counter arrays based on hybrid SRAM/DRAM counter architectures have been proposed [76, 81, 87, 108]. 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. Increments are made only to these SRAM counters. When the values of the SRAM counters come close to overflowing, they are scheduled to be "flushed" back to the corresponding DRAM counters. These schemes significantly reduce the SRAM cost. In particular, the scheme by Zhao et al. [108] achieves the theoretically minimum SRAM cost of 4 to 5 bits per counter when the SRAM-to-DRAM access latency 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 significant 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, the possibility for a counter to overflow could occur much earlier, and more SRAM counter bits would be needed to compensate, resulting in yet larger SRAM requirements. On the other hand, in our counter array scheme we only need to increase the storage for several thousand entries. To maintain 16 million active counters, the size increase in our scheme is less than the traditional SRAM/DRAM schemes by more than 3 orders of magnitude. 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 [38, 106] to maintain values such as the entropies of data streams.

2.1.1 DRAM Can Be Plenty Fast

In this chapter, we challenge the main premise behind previous hybrid SRAM/DRAM architecture proposals. Their main premise is that DRAM accesses 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 [35, 58, 101, 102] that can be exploited to make a DRAM solution practical. We propose DRAM-based counter architectures that allow for wirespeed updates to large counter arrays.

Motivated 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 continuously been driving aggressive roadmaps in terms of ever increasing memory bandwidths that can be provided at commodity pricing. For example, the Cell processor from IBM/Sony/Toshiba [32] uses two 32-bit channels of XDR memories [78] with an aggregated memory bandwidth of 25.6 GB/s. Using an approach called micro-threading [101], 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 [79] 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 [40] 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 [102].

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 [78], 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.

2.1.2 Our Approach

Our main idea is to randomly distribute the memory addresses to which consecutive counter indices are mapped across the memory banks so that a near-perfect balancing of memory access loads can be provably achieved, under both arbitrary and adversarial counter update patterns. Suppose the SRAM-to-DRAM random access latency ratio is μ, e.g. μ = 4ns/64ns = 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 scheme in use.

The randomized scheme does not require extra DRAM. It works by ensuring that the memory load is evenly distributed when different counters are updated over time, *under arbitrary counter update sequences*. However, 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. 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, a key contribution of this chapter is a mathematical one: we prove 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 2.4). This result is a *worst-case large deviation theorem* in nature [71] because it establishes a bound on the largest/worst case value among the tail probabilities of having long queueing delays under all admissible including adversarial counter update patterns. In the course of proving this result, we establish a novel general methodology for establishing worst-case large deviation bounds. Worst-case large deviation bounds, such as ours are very hard to obtain because the parameter space of all admissible counter update patterns underlying the large deviation tail bound problem is gigantic. Although given a particular parameter setting, i.e., a particular counter update sequence, establishing the tail bound in our problem is straightforward through the Chernoff technique, enumerating this procedure over the entire parameter space is computationally impossible, and finding the maximum of such bounds appears to be analytically impossible as well through tractable optimization techniques.

Our methodology to overcome this difficulty is a novel combination of convex ordering and large deviation theory. To our surprise, we found that we are able to find a parameter configuration, i.e., a counter update sequence that dominates all other configurations by the convex order. Since the exponent function is a convex function, we are able to dominate the moment generating function (MGF) of queueing delays, which is a random variable, under all other parameter settings by the MGF under the worst-case parameter setting. We can then apply the Chernoff technique to this worst-case MGF to obtain very sharp tail bounds. Using this theoretical framework, we show that only very small queues on the order of $K = 45$ entries per request queue are required to ensure a negligible overflow probability (e.g. under 10^{-14}).

2.1.3 Summary of Contributions

We make several contributions in this work:

- We propose a DRAM-based counter architecture that can effectively maintain wirespeed updates to large counter arrays. As we shall see, our proposed counter

scheme can leverage the internal independent memory banks already available inside a modern DRAM chip without the expense of multiple parallel memory channels, thus making it very cost effective.

- We develop a novel mathematical methodology for establishing worst-case large deviation bounds. As one of its applications, we present a rigorous theoretical analysis on the performance of our proposed randomized counter architecture in the worst-case.

- We present concrete evaluations of our proposed scheme using the XDR memory architecture [78, 79, 101], which has 16 internal independent memory banks in each memory chip and 256 memory banks across 16 memory chips in a single memory module. We show that wirespeed performance can be achieved without the expense of multiple memory channels.

- Compared to existing hybrid SRAM/DRAM counter architectures [76, 81, 87, 108], our randomized counter 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 SRAMs 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 [38, 106]), while hybrid SRAM/DRAM counter architectures typically can only accommodate "increment by 1" efficiently. Finally, as we shall show in Section 2.5, 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, this approach is scalable to future application scenarios in which vastly larger counter arrays are possible. 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 the low cost of DRAM[1].

[1] As of this writing, 4GB of DRAM costs under $20, over 200MB/$.

2.1.4 Outline of Chapter

The rest of the chapter is organized as follows. Section 2.2 outlines additional related work. Section 2.3 describes a proposed randomized counter architecture in details. Section 2.4 provides a rigorous analysis of the performance of our randomized counter architecture under the worst case memory access pattern. Section 2.5 presents evaluations of the proposed architecture. Finally, Section 2.6 concludes the chapter.

2.2 Related Work

In this section, we outline prior work related to our problem. As already discussed in Section 2.1, the naïve approach of storing full counters in SRAM is prohibitively expensive. Although a hybrid SRAM/DRAM architecture [76, 81, 87, 108] 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 [20, 65, 91], which are all based on the basic idea invented by Morris [65]. Approximate counting keeps the cost low by sacrificing counter accuracy. The idea is to probabilistically increment a counter based on the current counter value. However, approximate counting in general has very large error margin when the number of bits per counter used is small, and possible estimation values are very sparsely distributed in the range of possible counts. Therefore, when the counter values are small, the estimation can cause very high relative errors (well over 100%). Thus this approach is only applicable to those network measurement and data streaming applications that can tolerate such inaccuracies. The approach may not be acceptable for those network accounting and data streaming applications where small counter values are important for the overall measurement accuracy. Other such systems include the NetFlow from Cisco [27], the filter-based accounting from Juniper [86], and the sample-and-hold solution [28].

A second approach is a counter architecture called counter braids [59], which was inspired by the construction of low-density parity-check codes which keeps track of the exact counts of all flows without remembering the association between flows and counters[2]. 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, where the transformation matrix is the result of hashing all flow labels during a measurement epoch. In a way, counter braids are “more passive" than SRAM/DRAM hybrid architectures. Flow counts can be decoded through an iterative decoding process at the end of the measurement period. The decoding process incurs significantly longer delay than a DRAM access, and it can be processed off-line.

A third approach is based on an efficient variable-length counter representation called BRICK [37]. 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 naïve SRAM implementation.

In all three above SRAM-based approaches, significant amounts of SRAM are still necessary for very large counter arrays (e.g. for tens of millions of counters). In contrast, our proposed solution stores all counters in DRAM only. We believe these approaches are complementary as they have different design tradeoffs. It is worth noting that general memory systems supporting arbitrary memory read and write accesses [1, 100] are applicable for the maintenance of large counter arrays. They are not as efficient, however, and incur large delays compared to memory systems specialized for this purpose.

Several packet buffer designs [43, 99] use interleaved DRAM banks, so that packets can be written to multiple DRAM banks following a certain order and later retrieved according to their departure times. Only one bank can be addressed in each cycle to avoid bank conflicts. Even though our counter array also explores DRAM inter-

[2]Counter braids consider a more general problem that also addresses flow association.

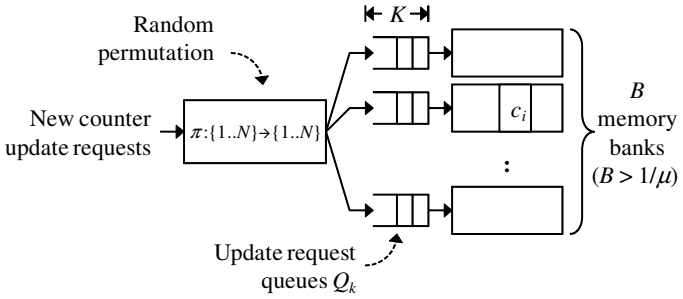
leaving, the problems of managing statistics counters and buffering packets are fundamentally different. For packet buffer design, the goal is to find one DRAM bank entry among all banks in which to store an arriving packet so that it can be retrieved in time for departure. While for statistics counters, the counters are stored at fixed locations with a guarantee that any counter update requests can be successfully processed.

Finally, the idea of using DRAM interleaving to implement large counter arrays was first proposed in [57]. We extend that work considerably in this chapter by presenting a new mathematical framework for analyzing the behavior of such architectures under practical conditions, as detailed in Section 2.4. We also introduced a cache module in the architecture to combat adversarial counter update patterns, which adds considerably to the complexity of our analysis.

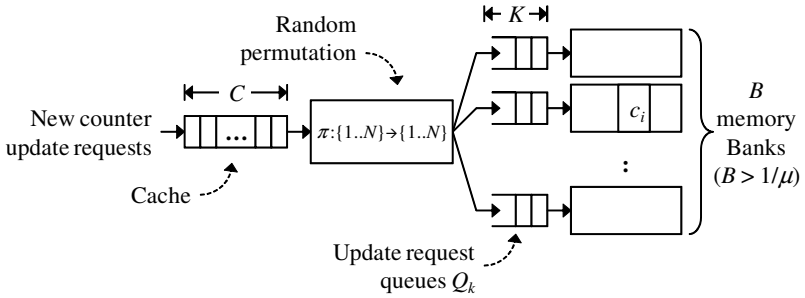
2.3 Randomized Counter Architecture

In the past, memory interleaving has been successfully used for improving the performance of computer systems [58, 73, 80], for graphics or video intensive applications [101], and for implementing routing functions like high-performance packet buffers [89]. In this section, we describe how this technique can be employed for statistics counting.

In this section, we describe our randomized counter architecture with a statistical service guarantee. Figure 2.1(a) depicts a simplified version of our randomized counter architecture. Given an SRAM-to-DRAM random access latency ratio of μ (e.g. μ = 4ns/64ns = 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 each memory bank will receive about one out of B counter updates to it on average with high probability. If the input counter update request generated by an arriving packet is a counter index, a permuted index can be achieved by applying a pseudorandom permutation function $\pi : \{1,\ldots,N\} \rightarrow \{1,\ldots,N\}$ to the counter 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 $a = \lfloor \pi(i)/B \rfloor$. Our experiments show that simple modulo function mapping the counter indexes to different DRAM banks is sufficient to guaran-



(a) Basic architecture.



(b) Extended architecture to handle adversaries.

Figure 2.1: Memory architecture for randomized DRAM-based counter schemes.

tee fairly uniform load distribution to different banks, in which case there is no need to store the pseudorandom permutation function explicitly. In fact if the input packet label (such as its 5-tuple) needs to be hashed to a counter index, we can view the hashing as random and a separate permutation is no longer necessary. Hash functions on 5-tuples in the packet headers are commonly applied in network processors, thus no extra cost will be involved.

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. Conceptually, these request queues are then serviced concurrently. The actual *read* and *write* operations are serviced alternately at the time slot level. In particular, at each memory bank k, assume a read operation is initiated at time s at the head of Q_k for the counter request. The data will be available $1/\mu$ time slots later. A write operation for the incremented counter value can then be initiated at time $s+1/\mu$, which will be completed by time $s+2/\mu$. 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, the proposed design can effectively keep count of new packet arrivals so 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. Note that I/O is not the bottleneck in our system, since in each memory channel, DRAM banks are always accessed sequentially and different banks are never accessed in the same cycle. It takes the I/O only one cycle to finish a read or write operation to a DRAM bank on the chip, even though the actual operation takes several cycles to finish inside a bank. The DRAM bank manager will handle the actual writing of data into a bank based on receiving commands and data from the I/O.

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 pseudorandom permutation function (or the key) is *unknown* 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 associate cache module (e.g. containing C cache entries) to our architecture to absorb such repetitions, as shown in Figure 2.1(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 proven in the adversarial paging literature that fancy policies such as LRU will not outperform FIFO under adversarial conditions [66, 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 look up the cache to see if there is already a pending update request to this counter. If there is, then we can just 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 accesses that is either read or write, there is no incentive (for degrading the performance

of our system) by an adversary accessing the same counter repeatedly within a sliding window of C cycles.

In the next section, we present a detailed theoretical analysis of the architecture depicted in Figure 2.1(b) under all counter update sequences.

2.4 Performance Analysis

In this section, we analyze the performance of our randomized counter architecture. We prove the main theoretical result of this chapter, 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 sequences including any adversarial counter update sequences. As explained earlier, this worst-case large deviation result is proven using a novel combination of convex ordering and large deviation theory.

The main idea of this proof follows. Given any arbitrary counter update sequence over a time period $[s,t]$ (viewed as a parameter setting), we are able to obtain a tight stochastic bound of the number of arrival counter update requests to a DRAM bank during $[s,t]$ using various tail bound techniques. Since our scheme has to work with all possible counter update sequences, our bound clearly has to be the worst case, i.e. the maximum, stochastic bound over all cases. The space of all such sequences is so large, however, that enumeration over all of them is computationally prohibitive and low complexity optimization procedures for finding the worst case do not seem to exist. Fortunately, we discover that the number of arrivals under all these counter update sequences are dominated by that under a particular, i.e., the worst-case counter update sequence, in the convex order but not in the stochastic order. Since $e^{\theta x}$ is a convex function, we can upper-bound the MGFs of the number of arrivals under all other counter update sequences by that under the worst-case update sequence. The final tail bound is obtained by simply applying the Chernoff bound to this worst-case MGF. However we will show this worst-case MGF is prohibitively expensive to compute, let alone applying a Chernoff technique to it. We solve this problem through upper-bounding this MGF by a computationally friendly formula.

In the previous section, we introduced 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 the 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. It is intuitive 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. We omit the proof of this fact since it would be a tedious application of the elementary stochastic ordering theory [83].

The rest of this section is organized as follows. In Section 2.4.1, 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 the next three sections we show how to bound each individual event $D_{s,t}$ using the aforementioned convex ordering technique. In Section 2.4.3, we establish the worst-case number of update requests $X_{s,t}$ during time interval $[s,t]$, in terms of convex order. In Section 2.4.4 we bound $X_{s,t}$ with the sum of i.i.d. random variable which can be easily computed.

2.4.1 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, and 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 $\mathtt{P}_{\mathtt{overall}}$ can be bounded by,

$$\mathtt{P}_{\mathtt{overall}} \leq B \times \Pr[\tilde{D}], \tag{2.1}$$

which is the 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,

$$\Pr[\tilde{D}] \leq \sum_{0 \leq s \leq t \leq n} \Pr[D_{s,t}]. \tag{2.2}$$

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 $[s,t]$, we have

$$D_{s,t} \equiv \{\omega \in \Omega : X_{s,t} - \mu(t-s) > K\}, \tag{2.3}$$

where implicit probability space Ω is the set of all permutations on $\{1,...,N\}$. For 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 are dequeued following an arbitrary pattern, with the only restriction being 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, each instance $\omega \in \Omega$ gives us an arrival sequence to the queues of the DRAM banks.

The inequality (2.2) 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 1. $\tilde{D} = \bigcup_{0 \leq s \leq t \leq n} D_{s,t}$

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 that 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, obviously the queue will overflow at time t or earlier, so $\omega \in \tilde{D}$. □

Remark: A similar lemma is proved in [108, Lemma 1]. Here we point out the stronger relationship of equivalence.

2.4.2 Mathematical Preliminaries

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 that can be 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 2.4.4. However, it is not possible to enumerate over all possible parameter settings, i.e., counter update sequences, to find the worst-case $\Pr[D_{s,t}]$ bound. Fortunately, convex ordering allows 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 of $X_{s,t}$ and use X instead.

In the following, we first describe the standard Chernoff method for obtaining sharp tail bounds from the MGF of a random variable, in this case X.

$$\begin{aligned}\Pr[D_{s,t}] &= \Pr[X > K + \mu\tau] \\ &= \Pr[e^{X\theta} > e^{(K+\mu\tau)\theta}] \\ &\leq \frac{E[e^{X\theta}]}{e^{(K+\mu\tau)\theta}},\end{aligned}$$

where $\theta > 0$ is any constant. The last step is due to the Markov inequality. Here τ is defined as $t-s$. Then, the overflow probability for one request queue during cycles $[0,n]$ is calculated according to (2.2). And the system overall overflow probability can be calculated as in Equation (2.1).

Since this is true for all θ, we have

$$\Pr[D_{s,t}] \leq \min_{\theta>0} \frac{E[e^{X\theta}]}{e^{(K+\mu\tau)\theta}}. \tag{2.4}$$

Then, we aim to bound the moment generating function (MGF) $E[e^{X\theta}]$ by identifying the worst-case counter update sequences. Note that we resort to convex ordering, because *stochastic order*, which is the conventional technique for establishing ordering between random variables and is stronger than *convex order*, does not hold here, as we will show shortly.

Since convex ordering techniques and related concepts are needed to establish the bound, we first present the definition of majorization, exchangeable random variables, convex function, and convex ordering as follows.

Definition 1 (Majorization [60, Definition 1.A.1])**.** *For any n-dimensional vectors a and b, let $a_{[1]} \geq \ldots \geq a_{[n]}$ denote the components of a in decreasing order, and $b_{[1]} \geq \ldots \geq b_{[n]}$ denote the components of b in decreasing order. We say a is* majorized *by b, denoted $a \leq_M b$, if*

$$\begin{cases} \sum_{i=1}^{k} a_{[i]} \leq \sum_{i=1}^{k} b_{[i]}, & \text{for } k = 1, \ldots, n-1, \\ \sum_{i=1}^{n} a_{[i]} = \sum_{i=1}^{n} b_{[i]}. \end{cases} \tag{2.5}$$

Definition 2 (Exchangeable random variables)**.** *A sequence of random variables $X_1, \ldots, X_n$ is called* exchangeable*, if for any permutation $\sigma : [1, \ldots, n] \to [1, \ldots, n]$, the joint probability distribution of the permuted sequence $X_{\sigma(1)}, \ldots, X_{\sigma(n)}$ 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 3 (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 $0 < \alpha < 1$.*

The following lemma about convex functions will be used later.

Lemma 2. *Let $b_1 \leq a_1 \leq a_2 \leq b_2$ be such that $a_1 + a_2 = b_1 + b_2$ and let $f(x)$ be a convex function. Then, $f(a_1) + f(a_2) \leq f(b_1) + f(b_2)$.*

Proof. Let $\alpha = \frac{a_1 - b_1}{b_2 - b_1} \in (0,1)$. It is not hard to verify that $a_1 = (1-\alpha)b_1 + \alpha b_2$ and $a_2 = \alpha b_1 + (1-\alpha)b_2$. Hence $f(a_1) \leq (1-\alpha) f(b_1) + \alpha f(b_2)$ and $f(a_2) \leq \alpha f(b_1) + (1-\alpha) f(b_2)$, which combine to prove the lemma. □

Definition 4 (Convex order [67, Section 1.5.1]). *Let X and Y be random variables with finite means. Then we say that X is less than Y in convex order (written $X \leq_{cx} Y$), if $E[f(X)] \leq E[f(Y)]$ holds for all real convex functions f such that the expectations exist.*

Since the MGF ($\mathrm{E}[e^{X\theta}]$) is the expectation of a convex function ($e^{x\theta}$) of X, establishing convex order will help to bound the MGF. The following result from Marshall [60] relates majorization, exchangeable random variables and convex order together. It is restated here in the language of convex ordering. It is a special case of a more general theorem [60, Proposition 11.B.2].

Lemma 3 ([60, Proposition 11.B.2.c]). *If $X_1, \ldots, X_n$ are exchangeable random variables, a and b are n-dimensional vectors, then $a \leq_M b$ implies $\sum_{i=1}^{n} a_i X_i \leq_{cx} \sum_{i=1}^{n} b_i X_i$.*

Finally, we define stochastic order, the lack of which in our case leads us to resort to the convex ordering.

Definition 5 (Stochastic order [67, Section 1.2.1]). *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.*

2.4.3 Worst Case Update Request Sequence

In this section, we 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 the probability that the i^{th} address is mapped to the DRAM bank under consideration, and N is the total number of addresses/indices in the DRAM banks. We have

$$E[X_i] = \frac{1}{B}. \tag{2.6}$$

Thanks to the random counter index permutation scheme, we can view X_i's as a result of sampling without replacement from N values, of which $\frac{N}{B}$ are 1 and the rest are 0. Therefore the X_i's are exchangeable random variables, though they are not independent.

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

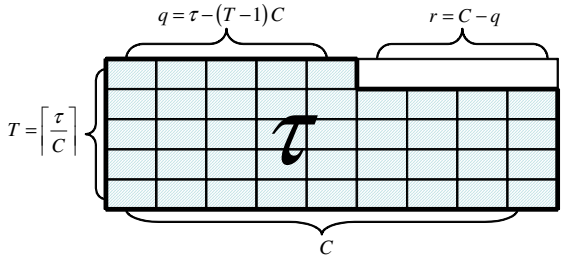


Figure 2.2: Relationship of q, r, T and τ.

to the same DRAM address should not repeat within any sliding window of C cycles. Therefore none of the counts $m_1, ..., m_N$ can exceed T, where

$$T = \lceil \frac{\tau}{C} \rceil. \tag{2.7}$$

Moreover, let $q = \tau - (T-1)C$ and $r = C - q$. Then only the first q requests could repeat with count T. Figure 2.2 will help readers understand the relationship of q, r, T as functions of τ.

We call any vector $m = \{m_1, ..., m_N\}$ a valid splitting pattern of τ, if it satisfies the following,

$$\begin{cases} 0 \le m_i \le T, \\ \sum_{i=1}^{N} m_i = \tau, \\ |\{i : m_i = T\}| \le q. \end{cases} \tag{2.8}$$

Let $\mathscr{M}$ be the set of all valid splitting patterns.[3] For simplicity, we set

$$X_m = \sum_{i=1}^{N} m_i X_i. \tag{2.9}$$

We are ready to specify the family of worst-case counter update sequences. A worst-case counter update sequence takes the following form: first come $q + r(= C)$ update requests for distinct counter indices $a_1, a_2, ..., a_{q+r}$, and then they repeat for $T-1$ times in total, finally followed by q update requests for counter indices $a_1, a_2, ..., a_q$. In other words, inside this window of τ cycles, counters $a_1, a_2, ..., a_q$ (q of them in total) are accessed T times and counters $a_{q+1}, ..., a_{q+r}$ (r of them in total) are accessed $T-1$

[3] It 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 in our analysis.

times. In Theorem 1, we show that this arrival sequence is indeed the worst-case arrival counter updates in the sense of convex ordering.

Let m^* be the aforementioned pattern for one of such counter update sequences, i.e. let $m_1^* = \ldots = m_q^* = T, m_{q+1}^* = \ldots = m_{q+r}^* = T-1, m_{q+r+1}^* = \ldots = m_N^* = 0$. We have the following theorem,

Theorem 1. *m^* is the worst case splitting pattern in terms of convex ordering, i.e. $X_m \leq_{cx} X_{m^*}, \forall m \in \mathscr{M}$.*

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

$$\begin{cases} m_{[i]} \leq T = m_i^*, & \text{for } 1 \leq i \leq q. \\ m_{[i]} \leq T-1 = m_i^*, & \text{for } q+1 \leq i \leq q+r. \end{cases} \tag{2.10}$$

Therefore majorization condition (2.5) is true for $1 \leq k \leq q+r$. Since

$$\sum_{i=1}^{q+r} m_i^* = \tau = \sum_{i=1}^{N} m_i. \tag{2.11}$$

The condition is true for $i > q+r$ as well. Therefore by definition $m \leq_M m^*$.

The theorem follows from Lemma 3 because $X_1, \ldots, X_n$ are also exchangeable. □

Remark: Note that stochastic order does not hold here in general, since $\mathrm{E}[X_m] = \mathrm{E}[X_{m^*}] = \tau/B, \forall m \in \mathscr{M}$. For stochastic order to hold between two random variables of different distributions, their expectations must differ [67, Theorem 1.2.9].

Unfortunately, it is in general not possible to apply the Chernoff bound directly to the MGF of X_{m^*}. For $\tau \leq C$, X_{m^*} is a hypergeometric random variable whose MGF $\mathrm{E}[e^{X_{m^*}\theta}]$ is a hypergeometric series [31] that is expensive to compute. For $\tau > C$, X_{m^*} 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.

2.4.4 Relaxations of X_{m^*} for Computational Purposes

We first state the main theorem of the chapter.

Theorem 2.

$$\text{For } \tau \leq C,$$

$$\Pr[D_{s,t}] \leq \min_{\theta>0} \frac{\left(\frac{1}{B}e^{\theta} + (1-\frac{1}{B})\right)^{\tau}}{e^{(K+\mu\tau)\theta}}. \tag{2.12}$$

$$\text{For } \tau > C,$$

$$\Pr[D_{s,t}] \leq \min_{\theta>0} \frac{\left(\frac{1}{B}e^{T\theta} + (1-\frac{1}{B})\right)^{q} \left(\frac{1}{B}e^{(T-1)\theta} + (1-\frac{1}{B})\right)^{r}}{e^{(K+\mu\tau)\theta}}. \tag{2.13}$$

Proof. To prove the theorem, our remaining task is to find a way to upper-bound $\mathrm{E}[e^{X_{m^*}\theta}]$ by a more computationally friendly formula, to which the Chernoff technique can be applied. The following result by Hoeffding, which bounds the outcome of sampling without replacement by that with replacement in the convex order, will be used to accomplish this task for $\tau \leq C$.

Lemma 4 ([34, Theorem 4])**.** *Let the population S consist of N values $c_1, ..., c_N$. Let $X_1, ..., X_n$ denote a random sample without replacement from S and let $Y_1, ..., Y_n$ denote a random sample with replacement from S. Let $X = X_1 + ... + X_n$, $Y = Y_1 + ... + Y_n$. Then $X \leq_{cx} Y$.*

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

Theorem 3. *Same notations as in Lemma 4. Let $a_1, ..., a_n$ 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. Thus for any convex function f, we have*

$$\mathrm{E}f\left(\sum_{i=1}^{n} a_i X_i\right) \leq \mathrm{E}f\left(\sum_{i=1}^{n} a_i Y_i\right). \tag{2.14}$$

Proof. The proof of Theorem 3 is in the Appendix. □

We consider the two cases: $\tau \leq C$, which is the scenario that the measurement window τ in number of cycles is no larger than the cache size in number of entries, and the case $\tau > C$ separately.

The Case $\tau \leq C$

When $\tau \leq C$, $X_{m^*} = X_1 + ... + X_\tau$. Let the population S consist of $c_1, ...c_N$ 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 Lemma 4 has the same distribution as our X_{m^*}, and Y_i's 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 $X_m \leq_{cx} X_{m^*} \leq_{cx} Y$ we get

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

By (2.4), we now have the following bound:

$$\Pr[D_{s,t}] \leq \min_{\theta>0} \frac{\left(\frac{1}{B}e^\theta + (1-\frac{1}{B})\right)^\tau}{e^{(K+\mu\tau)\theta}}, \quad \tau \leq C.$$

The Case $\tau > C$

For $\tau > C$, we have $X_{m^*} = T(X_1 + ... + X_q) + (T-1)(X_{s+1} + ... + X_{q+r})$, from Section 2.4.3. Lemma 4 cannot be applied due to the coefficients T and $(T-1)$. Instead, we apply Theorem 3 and get,

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

Together with (2.4), we have

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

Thus we have proved Theorem 2. □

Remark: For $\tau > C$ we need Lemma 4 alone. For the worst case counter update sequence, instead of selecting $C = q + r$ permutation destinations for the counter addresses and checking which ones are mapped to the $\frac{N}{B}$ addresses in a DRAM bank, we can treat it as selecting $\frac{N}{B}$ permutation sources, and checking which ones are among the addresses that contain the counter updates. Let the population S' be exactly the components of m^*, i.e. let $c_i = m*_i$. 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 are of value 0. Thus

$$X_{m^*} = X'_1 + ... + X'_{\frac{N}{B}}, \tag{2.15}$$

where X'_i's are a random sample without replacement from S'. By Lemma 4 we have

$$X_{m^*} \leq_{cx} Y' = \sum_{i=1}^{N/B} Y'_i, \tag{2.16}$$

where Y'_i's are a random sample with replacement from S'. Thus the Y'_i's are i.i.d. random variable with

$$\Pr[Y'_i = y] = \begin{cases} \frac{q}{N}, & \text{if } y = T; \\ \frac{r}{N}, & \text{if } y = T-1; \\ \frac{N-q-r}{N}, & \text{if } y = 0. \end{cases} \tag{2.17}$$

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

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

For the last inequality we used the inequality $(1 + \frac{a}{n})^n < e^a$. By (2.4), we now have the following bound (it's not hard to verify that it also applies to $\tau \leq C$),

$$\Pr[D_{s,t}] \leq \min_{\theta>0} e^{\left(qe^{T\theta} + re^{(T-1)\theta} - q - r\right)/B - (K+\mu\tau)\theta}. \tag{2.18}$$

However the bounds in Theorem 2 are better numerically in our setting.

We also see that the bounds in Theorem 2 are translation invariant, i.e. they only depend on $\tau = t - s$. Therefore, the computation cost of (2.2) 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 bounds for the overflow probability under the worst-case counter update request sequences. The bounds can be computed by $O(n)$ number of numerical minimizations for one-dimensional functions expressed in Theorem 2.

2.5 Performance Evaluation

In this section, we present evaluation results for our proposed DRAM-based counter array architecture. The outline of this section is as follows. Section 2.5.1 describes the instantiation of our proposed solution. Section 2.5.2 outlines the parameters of two real-world Internet traffic traces for our evaluations. Section 2.5.3 presents numerical results derived using the analytical models presented in Section 2.4 for our randomized counter architecture. Finally, Section 2.5.4 provides a comparison of our new approach with the state-of-the-art hybrid SRAM/DRAM approach [108].

2.5.1 Implementation Details

To provide a general formulation, we have assumed in Section 2.3 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. Therefore, a single memory channel can be used to pipeline memory transactions across them at peak rates when operating in an interleaving manner.

To provide a concrete analysis of our proposed solution, we use the specifications of an actual commercial high-bandwidth memory part, namely the XDR memory from Rambus [78, 101]. As depicted in Figure 2.3, each XDR memory chip contains 16 internal memory banks. Although the XDR memory has a worst-case access latency of

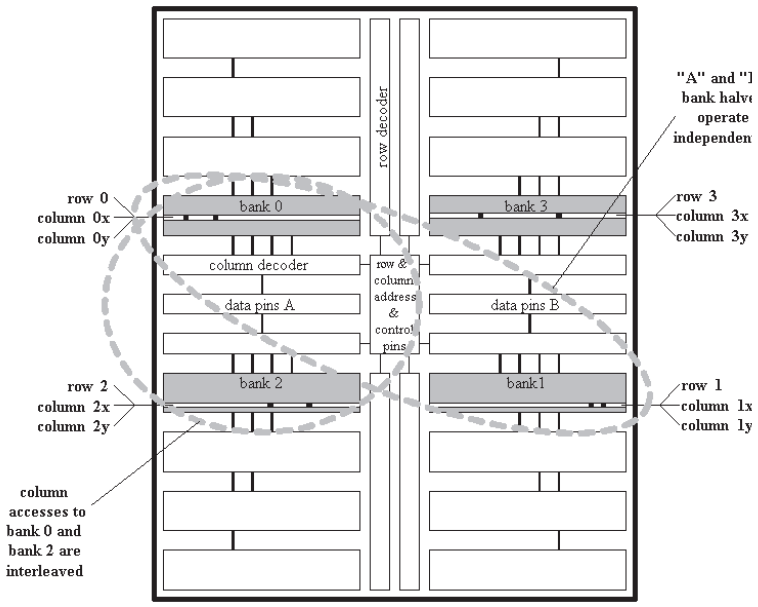


Figure 2.3: An example high-bandwidth DRAM architecture [101]. Each XDR memory IC has 16 internal memory banks that can be interleaved to achieve high-bandwidth memory access.

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 packet (40 bytes) 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 [32] and the latest mainstream Intel x86 multi-core processor [41] 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 access latency

ratio that is no smaller than $1/16$. Note that we use an ASIC implementation where we can directly interact with the external DRAMs. Such a design is different from using a general purpose CPU to implement the algorithm, since modern CPUs always use intermediate caches to access external DRAMS.

2.5.2 Traffic Traces

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

2.5.3 Tail Bounds for Randomized Counter Architecture

The randomized counter scheme avoids the need to replicate counters, and thus requires the same amount of DRAM as the hybrid SRAM/DRAM schemes.

Overflow Probabilities with $\mu = 1/16$

In this section, we present the numerical results computed from the formulae derived in Section 2.4 using MATLAB 7.11. We use $n = 10^{10}$ for all the following examples, where n is the total number of cycles for the measurement period. The overall overflow probability can be calculated from Equations (2.1) and (2.2).

In Figure 2.4, 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 the request queue length K increases, the overflow probability bound decreases. However, after K reaches certain thresholds, the overflow probability

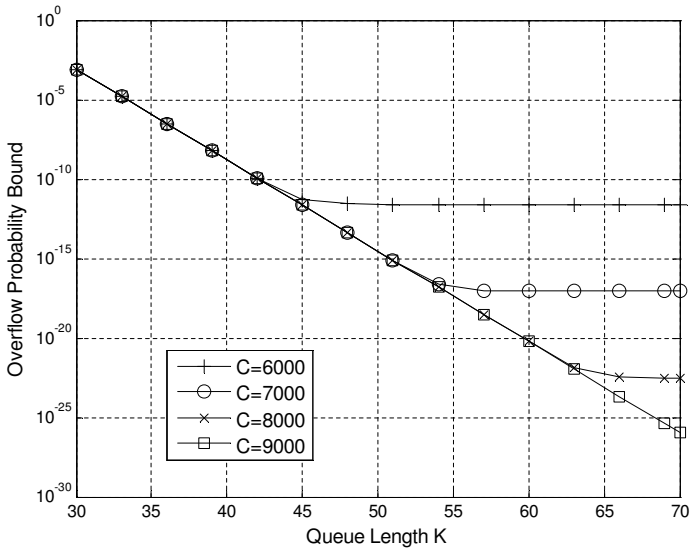


Figure 2.4: Overflow probability bounds as a function of request queue size K with $\mu = 1/16$ and $B = 32$.

bound stays practically flat. The flat level depends on the cache size C, which confirms our prediction that given a fixed size cache of size C, in the worst case an adversary can keep sending repetitive requests that is C cycles apart, thereby causing system to overflow with non-zero probability no matter how large the request queue size is. Actually, as cache size C approaches infinity, the overflow probability as a function of queue length K becomes the overflow probability of a discrete-time single server FIFO queue with Bernoulli arrivals (Geo/D/1 queue) with arrival probability $1/B = 1/32$ to a queue of limited size K. As shown in Figure 2.4, when $C \geq 8000$, the overflow probability bound has only negligible decreases as cache size C increases for the practical setting of overflow probabilities. In other words, by increasing the size of the cache, the performance of the system will not improve significantly. Therefore, we consider that $C = 8000$ is enough for practical purposes in achieving an overflow probability bound of 10^{-14} with request queue larger than 45 entries.

In Figure 2.5, the system overflow probability bounds with different numbers of memory banks B as a function of queue length K are presented, where $\mu = 1/16$ and $C = 8000$. Cache size $C = 8000$ is used here because by using a larger cache, the system performance will not achieve any significant improvement. It can be seen from this

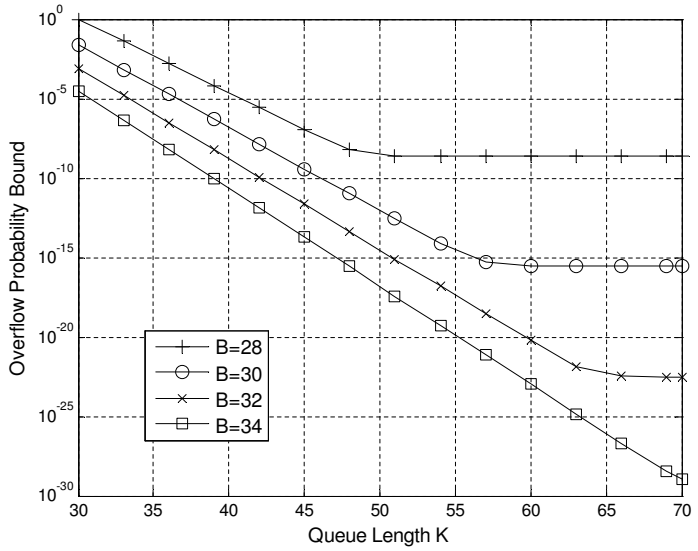


Figure 2.5: Overflow probability bounds as a function of the number of memory banks B with $\mu = 1/16$ and $C = 8000$.

figure that given the same K, as the number of memory banks B increases, the overflow probability bound decreases, which confirms our analysis that by using more memory banks, the load of arrival requests to each bank is reduced. Let's define the total queue size as M, where $M = B \cdot K$. From Figure 2.5, the optimal configuration for total queue size that can achieve a certain overflow probability bound can be calculated by choosing the curve with the smallest M that achieves this overflow probability bound.

Overflow Probabilities with $\mu = 1/12$

In this section, the overflow probability for the system with $\mu = 1/12$ is presented. Compared to a system with $\mu = 1/16$, we expect the system with $\mu = 1/12$ to achieve better/lower overflow bounds with the same setting, since the increase of value μ corresponds to a smaller gap between the DRAM and SRAM access latencies.

In Figure 2.6, the overflow probabilities as a function of cache size C and queue length K is presented, where $\mu = 1/12$ and $B = 32$, i.e., there are 32 memory banks in the system. It is easy to see from this graph that as K increases, the overflow probability decreases. For $C \geq 4000$, the overflow probability $\Pr[D_{s,t}]$ decreases exponentially.

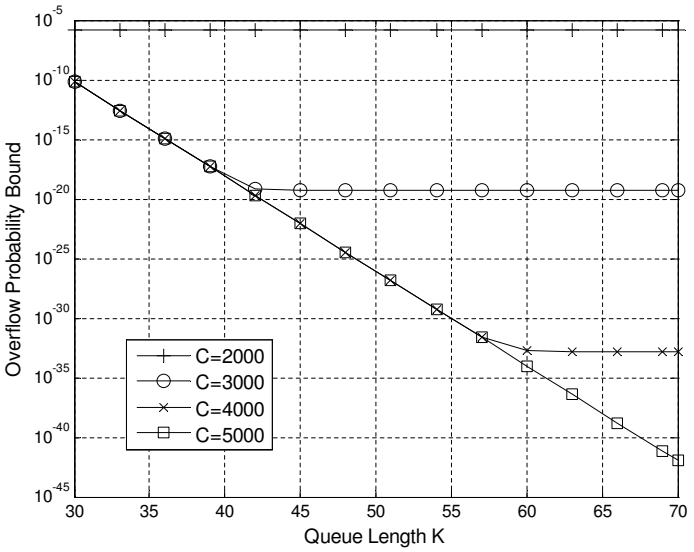


Figure 2.6: Overflow probability bounds as a function of queue size K with $\mu = 1/12$ and $B = 32$.

In Figure 2.6, it is also shown that as the cache size C grows, the overflow probability decreases. When $C \geq 4000$, the overflow probability has only negligible decreases as cache size C increases. I.e., by increasing the cache size used in the system, the performance will not be improved much for the practical setting of overflow probabilities. Therefore, $C = 4000$ is enough for the practical purposes. Compared with Figure 2.4, with a larger μ and the same request queue size, the cache size C can be significantly reduced to achieve the same overflow bounds.

In Figure 2.7, the system overflow probability with different number of memory banks B as a function of K is presented, where $\mu = 1/12$ and $C = 4000$. It can be seen from this figure that given the same K, as B increases, the overflow probability decreases. Compared with Figure 2.5, we can see that with a small increase in the SRAM-to-DRAM access latency ratio μ and the same setting of K, even with half the cache size, the overflow probability bounds are significantly reduced.

Our simulation results above show the overflow probability for the worst case counter update sequences among all possible ones. For actual traffic pattern from real-world traffic traces, the system performance is much better than the one predicted by the overflow probability bound. We run our simulations using the two Internet traces from

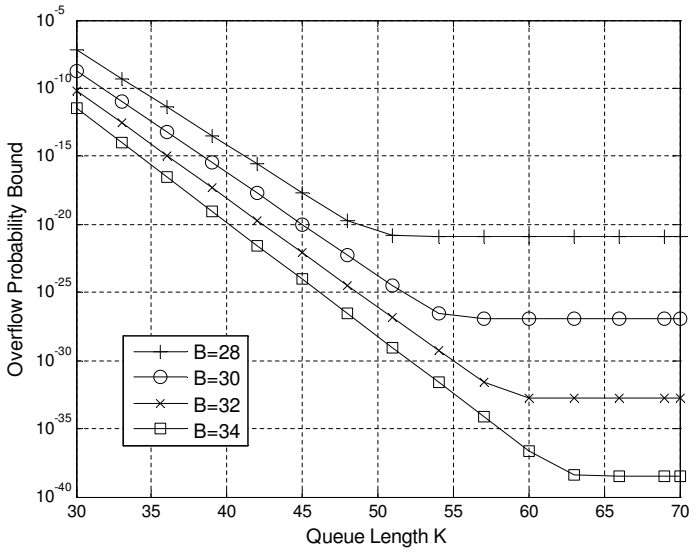


Figure 2.7: Overflow probability bounds as a function of number of memory banks B with $\mu = 1/12$ and $C = 4000$.

USC and UNC above for an extensive period of time. By setting the parameters such that C ranges from 2000 to 9000, K ranges from 30 to 70, and B ranges from 28 to 34, we observed no system overflow in either of the traces. This can be explained by the fact that the counter update patterns in real traces are much better than the worst case update patterns predicted in this chapter. Thus our statistics counter array, which is designed to handle worst case update patterns, doesn't overflow at all for the two real traces.

2.5.4 Cost-Benefit Comparison

Table 2.1 compares our proposed approach with a naïve SRAM approach as well as with the hybrid SRAM/DRAM counter architecture approaches [76, 81, 87, 108]. The naïve SRAM approach simply implements all counters in SRAM. For the hybrid SRAM/DRAM approach, we specifically compare against the state-of-the scheme proposed by Zhao et al. [108] that provably achieves the minimum SRAM requirement for this architecture class. As demonstrated in [108], in one example their approach requires almost a factor of one sixth times as much SRAM as the first hybrid SRAM/DRAM solution proposed in [87] and more than a factor of two times less SRAM than an improved

Table 2.1: Comparison of different schemes for a reference configuration with 16 million 64-bit counters. For our method, $K = 50$, $B = 32$, and $C = 7000$.

	Naïve	Hybrid SRAM/DRAM [108]	Randomized Counter
Counter DRAM	None	128 MB	128 MB
Counter SRAM	128 MB	8 MB	None
Control	None	1.5 KB SRAM	25 KB CAM and 5.5 KB SRAM

solution proposed in [76]. For an SRAM-to-DRAM latency ratio of $\mu = 1/16$, the architecture in [108] requires $w = \lceil \log 1/\mu \rceil = \lceil \log 16 \rceil = 4$ 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.

For 16 million counters, a naïve implementation would require 128 MB of SRAM, which is clearly far too expensive. For the scheme by Zhao et al., 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. Other SRAM schemes exist [36, 91], where much smaller size SRAM is required and no DRAM access is necessary to retrieve a counter value. However, they only approximate the counters to a certain extent and there is no guarantee that any counter value is exact. On the other hand, our proposed scheme, together with the naïve SRAM and hybrid SRAM/DRAM solutions always return exact counter values if there is no system overflow. For our scheme and the hybrid SRAM/DRAM solutions, the exact counters are retrieved at the expense of slow DRAM accesses.

For our proposed randomized counter solution, a small update request queue needs to be maintained at each memory bank. As shown in Figure 2.4, when we use

$B = 32$ memory banks, $K = 50$ 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 2.1(b) in Section 2.3, 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 counts[4]. 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.

It is worth noting that the overflow probability bounds are derived for our randomized counter architecture using the worst case counter update assumption. In practice, the system overflow probability for the Internet traffic is much smaller than the overflow probability bound, since real-world traffic traces contain more interleaving-friendly counter update patterns than the worst case predicted by our model. While our analysis predicts that request queues with 50 entries are required to achieve an overflow probability bound of 10^{-14} with a cache of size 7000, in our experiments the number of occupied entries in a request queue never exceeded 18.

2.6 Conclusion

In this chapter, we have addressed the problem of maintaining a large array of exact statistics counters that needs to be updated at very high speeds. We proposed a DRAM-based counter architecture that can effectively maintain wirespeed updates to large counter arrays by exploiting advanced architecture features that are readily avail-

[4] An 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.

able in modern commodity DRAM architectures. In particular, we presented a randomized counter architecture that can harness the performance of modern commodity DRAM offerings by interleaving counter updates to multiple memory banks. We presented a rigorous theoretical analysis on the performance of our proposed counter architecture under the worst-case counter update sequences using a novel combination of convex ordering and large deviation theory. Our analysis also takes into considerations adversarial counter update patterns. Among the salient features of our proposed DRAM-based counter architecture is its ability to support arbitrary increments and decrements at wirespeed as well as different number representations, including both integer and floating point number representations.

Chapter 2, in full, is a reprint of the material as it appears in the following publications:

- Hao Wang, Haiquan (Chuck) Zhao, Bill Lin, and Jun (Jim) Xu, "DRAM-Based Statistics Counter Array Architecture with Performance Guarantee", *IEEE/ACM Transactions on Networking (ToN)*, 2011.

- Haiquan (Chuck) Zhao, Hao Wang, Bill Lin, and Jun (Jim) Xu, "Design and Performance Analysis of a DRAM-based Statistics Counter Array Architecture", *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, Princeton, NJ, October 19-20, 2009.

- Bill Lin, Jun (Jim) Xu, Nan Hua, Hao Wang, and Haiquan (Chuck) Zhao, "A Randomized Interleaved DRAM Architecture for the Maintenance of Exact Statistics Counters", *ACM Special Interest Group on Performance Evaluation (SIGMETRICS)*, Seattle, WA, June 15-19, 2009.

The dissertation author was the primary investigator and author of the papers.

Chapter 3

Optimizing Statistics Counter Array for Internet Traffic

3.1 Introduction

In this chapter, we propose a DRAM-based architecture which combines the functionalities of both the cache and request queues to enhance the system performance and reduce the overflow probability for real-world Internet traffic. Our architecture is based on the observation that most flows on the Internet consist of multiple packets that are transmitted during a relatively short period of time. We refer to the packets from the same flows received within the short period time as traffic bursts. In a statistics counter tracking accurate flow statistics, each arriving packet triggers an update to the corresponding counter. Thus traffic bursts would trigger repetitive updates to the same counters within a short time, which can drastically increase the traffic loads to the memory banks storing the counter values and further cause system overflow. In our new architecture, we turn such traffic bursts to our advantage and effectively reduce the traffic loads to the memory banks by merging the repetitive counter updates in the request queues. Our proposed architecture makes use of a simple randomization scheme and small fully associative request queues to statistically guarantee a near-perfect load-balancing of counter updates to the memory banks. To mitigate the congestion caused by traffic bursts in the request queues, an incoming counter update is merged if there is

a counter update in the request queue to the same memory address. In case there is no matched update in the request queues, the new update is inserted into a request queue based on the random permutation function. The counter update requests in the request queues are constantly sent to DRAM banks to update the corresponding counter values. In choosing an counter update to be sent to the memory banks from a request queue, we present two request queue update policies: first-in-first-out (FIFO) and least-recently-used (LRU). We show that with a simple merging operation supported by the request queues and a request queue update policy, we can effectively resolve the adversarial access patterns of receiving bursts of counter updates to the same memory locations. We show that for the worst case traffic patterns such as those potentially triggered by external threats, our new architecture performs at least as well as the random counter scheme in Chapter 2. For real-world Internet traffic, we develop queueing models to show that as long as there is some traffic bursts in the arrival, the maximum request queue length is always bounded by a finite number. We also provide simulations results using Internet traffic traces to confirm the effectiveness of our queueing models. In summary, our proposed statistics counter architecture can effectively maintain line rate updates to large counter arrays while providing a diminishing overflow probability.

3.1.1 Summary of Contributions

We make several contributions in this work:

- We propose a DRAM-based statistics counter architecture that can take the advantage of traffic bursts to reduce the work load to DRAM banks and reduce the chance of system overflow. Traffic bursts which would have caused more accesses to DRAM banks can actually reduce the effective traffic loads to our counter system.

- In our counter architecture, we use request queues with merging capability to temporarily buffer pending counter updates. Unlike the design in Chapter 2, there is no separate cache. We show that by designing our request queues as large as the size of the cache in Chapter 2, we can provide the same worst case performance guarantee. Since overflow occurs only in the request queues, the overflow prob-

ability in our new architecture is lower than that in Chapter 2 due to the larger request queue sizes.

- We provide queueing models to show that as long as there are traffic bursts in the arriving traffic, the request queue size to avoid system overflow is always bounded by a finite number.
- We provide simulations using real-world Internet traces to analyze the request queue size requirement. The results of these simulations closely match the bounds predicted by our queueing models.

The rest of the chapter is organized as follows. Section 3.2 defines the notion of an ideal SRAM emulation. Section 3.3 describes a proposed randomized counter architecture in detail. Section 3.4 analyzes the performance of the proposed architecture with different traffic models. Section 3.5 compares our design with the robust counter design in Chapter 2 to show that the proposed one outperforms the previous one while using less SRAM. Section 3.6 presents simulation results using Internet traffic traces. The results predicted by our models closely match the simulation results. Finally, Section 3.7 concludes the chapter.

3.2 Ideal SRAM Emulation for Statistics Counter

In this section, we introduce a statistics counter architecture that can emulate a fast SRAM by exploiting memory interleaving using multiple DRAM banks. The definition of an SRAM emulation was first introduced in [100] for general memory systems, where both memory read and write operations are supported. For statistics counters, an SRAM emulation only needs to support frequent memory updates (write operations) and infrequent read operations to retrieve data from the counters. Memory updates need to be operating at line rate, while data retrievals occur less frequently by several orders of magnitudes. For certain applications, data can even be processed offline for further analysis, such as for the Internet traffic size distribution analysis. We assume that time is slotted, and there is at most one counter update request arriving in each time slot (simply a cycle). Suppose it takes b cycles for a DRAM bank to process

one counter update request, while it takes only one cycle for an SRAM to process the same request. An SRAM emulation that mimics the behavior of an ideal SRAM for the statistics counter array is defined as follows.

Definition 6 (Emulation)**.** *A statistics counter architecture using slower memories is said to* emulate *an ideal SRAM counter array if it can admit the arriving counter update patterns that are admissible to an ideal SRAM counter array with high probability. All the admitted counter updates are processed within a bounded delay.*

An SRAM emulation guarantees the following:

- *High Throughput*: An emulated SRAM counter array works at the same line rate and approximates the throughput of an ideal SRAM counter array.
- *Small Drop Rate (Outage)*: In an emulated SRAM counter array, the counter update requests are dropped with diminishing probability. Preferably, the drop rate is zero for all incoming counter update patterns just as in an ideal SRAM counter array system.
- *Bounded Delay*: The counter updates are processed within a bounded period of time once they are admitted into the system.

3.3 The Proposed Design

The proposed statistics counter architecture is shown in Figure 3.1. There are K interleaving DRAM banks with $K > b$ in order to match the SRAM throughput. There is one request queue for each memory bank to temporarily store the counter update requests waiting to be processed by the memory banks. The request queues are implemented as fully associative caches that employ a certain *cache update algorithm*. We implement a pseudorandom permutation function to distribute incoming counter updates into the K request queues. In the following, we present the detailed descriptions for each module in the figure.

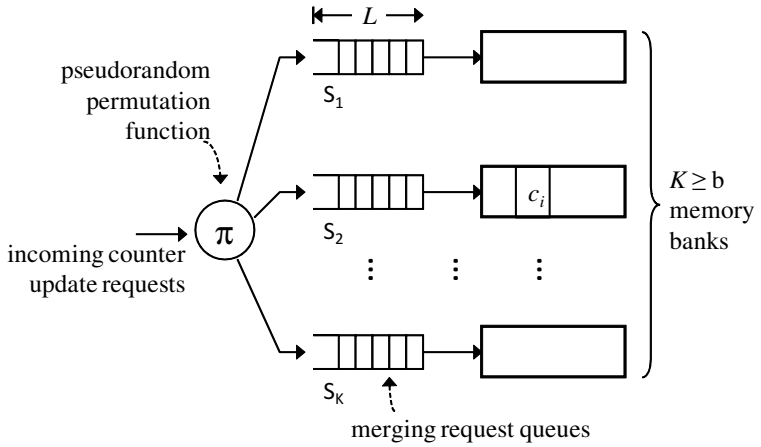


Figure 3.1: Statistics counter array architecture.

3.3.1 Counter Architecture

- *Pseudorandom Permutation Function* π: A pseudorandom permutation function is implemented to randomly distribute memory locations so that counter update requests to different counter locations are spread into K memory banks uniformly and at random (u.a.r) with equal probability $1/K$. Unless otherwise noted, when referring to a counter address, we will be referring to the address after the pseudorandom permutation. Note that incoming counter updates accessing the same memory address will be mapped to the same request queue even with the presence of the pseudorandom permutation function, since each counter is only stored in a single fixed location in the DRAM banks.

  The pseudorandom permutation function can be any linear permutation function that randomly "shuffles" counter locations inside the DRAM banks. However, once chosen, the function needs to be kept fixed for an extensive period of time, since counters have to be stored at deterministic locations in the memory banks and the cost of moving them from one memory location to another is both expensive and time consuming. However, the pseudorandom permutation function can be changed infrequently, say once every several minutes, in order to provide robustness against probe-response attempts by adversaries trying to determine the function.

Also, the pseudorandom permutation function must be kept unknown to the incoming traffic so that an adversary cannot purposely unbalance the loads to different banks by exploring the counters that are stored in any memory bank. However, it is still possible for an adversary to purposely trigger updates to the same memory bank by sending repetitive counter update requests to the same counter locations, since such requests will be processed by the same memory bank eventually.

- *DRAM Banks*: There are K DRAM banks in the counter array to match the SRAM throughput, where $K > b$ and b is the number of time slots it takes for a DRAM bank to finish processing one counter update request. We assume that a bank can process only one counter update at a time. Therefore, any counter update arriving to a memory bank will keep the bank *busy* for the next b cycles before another update can be processed by the same bank.

- *Request Queues*: There are K request queues, one for each memory bank. The request queues are denoted from S_1 to S_K. They temporarily store counter update requests waiting to be processed in the DRAM banks. Each entry in the request queue contains information on the counter index and the accumulated counter update. For a counter array supporting 16 million counters, the unique counter index for each counter can be set to a size of 3 bytes. The request at the head of a request queue will be sent to the corresponding memory bank when the bank is not busy.

  To make our counter architecture robust against adversarial counter update patterns, we require each request queue to be implemented as a fully associative cache that supports fast query and update operations. Once a new counter update enters a request queue, a query is initiated in the queue using the address of the incoming update as the key. If there is a match in the request queue, the new counter update will not generate a new entry in the request queue. Instead, the matched counter update request in the request queue is updated with the new request using simple arithmetic operations. We call this step a *merge* operation. We assume that the request queues can complete three operations in one time slot:

a query operation, an insert operation (merging or creating a new entry), and a removal operation to send a request already in the request queue to the corresponding memory bank when the bank is not busy. Therefore in each time slot, one new request may enter a request queue and another request in the queue may depart to update the corresponding bank.

Several cache update algorithms can be implemented in the request queues. A simple first-in-first-out (FIFO) rule always chooses the earliest request in a request queue. Least-recently-used (LRU) cache update policy can also be adopted to remove the request with the oldest update time in a request queue. In Section 3.6, we will show that LRU rule is slightly more efficient at merging incoming counter updates and reducing the loads in the request queues, even though it is more complex than the FIFO rule.

We will provide rigorous analysis of the performance of our statistics counter architecture in Section 3.4 to show that even with diminishing merging probability in the request queues, the total occupied queue size is always bounded, as long as there are traffic bursts in the arrival requests, and the flow sizes follow a certain heavy-tailed distribution. Such a result is contrary to the general M/D/1 queueing model, where the queue length can grow to any size with non-zero probability no matter what the arrival rate is. We provide simulation results using real-world Internet traces for our architectures implementing FIFO or LRU request queue update policy.

3.3.2 Why Merging Request Queues are Necessary?

With the help of a pseudorandom permutation function, we can safely assume that the counter update requests are distributed into the memory banks uniformly at random in the absence of adversaries. We expect that each memory bank receives roughly an equal share of the total requests in the long run. However, in a short period of time, the counter update requests to the DRAM banks may not be balanced due to the Birthday Paradox [63]. Therefore, there is no guarantee that the arriving counter updates are distributed into the memory banks truly uniformly at any moment, such as in a round-robin pattern. Request queues are necessary to temporarily store counter update

requests pending access the memory banks. Moreover, certain adversarial counter update patterns may further overload a memory bank with bursty arrivals. For example, although the random permutation function is unknown to attackers so that they cannot figure out how the counter addresses are mapped to DRAM banks, an attacker can still trigger repeated updates to the same DRAM bank by sending packets with the same flow record, which will generate counter updates to the same counter stored in the same memory location. Therefore, the number of pending updates in a request queue may grow indefinitely. Flows containing large files being transferred over the network may also cause a sudden increase in the arriving packets with the same flow record in a short period of time, which may overwhelm the fixed sized request queue it is mapped to. To mitigate such situations, in this design it is necessary to have the request queues merging incoming counter update requests with the ones in the request queue with the same flow record. We will show that the proposed statistics counter outperforms the robust counter design in Chapter 2 under both adversarial and non-adversarial arrivals.

3.4 Performance Analysis

In this section, we provide a queueing model for our statistics counter with merging request queues. We will show that with a reasonable merging probability assumption, the maximum queue size is always bounded.

3.4.1 Preliminaries

We assume at most one counter update request arrives in each cycle. Even though more counter updates may arrive in a multi-core multi-port network processor (NP) systems, which would complicate the analysis, they can be easily handled by using an input queue to temporarily buffer the arrival traffic and thus to provide an admissible arrival rate to the system. With $K \geq b$, we can ensure that the counter array is rate-stable, which means that the arrival rate to the counter array is no larger than the total throughput. More specifically, the maximum arrival rate to the counter array is at most one request per cycle, while the rate of requests leaving all the request queues, i.e. the service rate can be as large as $K/b \geq 1$ when all of the K queues are non-empty. We

adopt the terminologies in [89] to first define the arrival and departure processes for a request queue. Then, we provide a detailed analysis of our queueing model with merging request queues.

Let's assume that time is slotted. We have defined the smallest unit of time as a slot or a *cycle*. Thus the following analysis will be carried out in the discrete time domain. We assume that there are a total of N distinct counters in the DRAM banks, so counter update requests are mapped to at most N different memory locations. We denote by $A[t]$ the cumulative number of requests arriving at the statistics counter during cycles $[0,t]$, and $A_j[t]$ the cumulative number of requests for counter j during the same time $[0,t]$. The arrival processes of counter update requests $A_j[t]$ can be shown to be stationary and ergodic [14, 89]. So we have

$$A[t] = \sum_{j=1}^{N} A_j[t]. \tag{3.1}$$

We denote the arrival rate to counter j as λ_j such that the overall arrival rate is not larger than 1, so

$$\sum_{j=1}^{N} \lambda_j \leq 1. \tag{3.2}$$

We also assume all $A_j[t]$ to be independent of each other. This assumption is based on the fact that each counter in the statistics counter array keeps track of a specific flow or a specific group of flows. On the Internet, traffic consists of millions of flows generated by thousands of independent sources. The counter update requests generated by these flows can be assumed to be independent processes.

We denote by $RA^i[t]$ the cumulative number of requests arriving at request queue S_i during cycles $[0,t]$. Each incoming counter update request is processed by the pseudorandom permutation function and assigned to one of the K request queues uniformly at random. We denote by π the pseudorandom permutation function. The following holds,

$$RA^i[t] = \sum_{\pi(j)=i} A_j[t]. \tag{3.3}$$

Each counter update request is assigned to one of the request queues. Each request queue contains distinct counter update requests that are not in any other request queue. Since $A_j[t]$ are independent of each other, $RA^i[t]$ are independent processes.

For simplicity we assume that $A_j[t]$ are independent and identically distributed (i.i.d.) random processes. Later in the section, we will deal with the cases when the assumption does not hold.

When $A_j[t]$ are i.i.d. and distributed into K request queues uniformly at random, the arrival process to a request queue S_i is also i.i.d. For a request queue S_i, its arrival rate λ^i is

$$\lambda^i = \frac{\sum_{j=1}^{N} \lambda_j}{K} \leq \frac{1}{K}, \tag{3.4}$$

which states that due to the pseudorandom permutation function, each request queue receives about $1/K$ of the total counter updates.

Similar to the arrival processes, we denote by $D[t]$ the cumulative number of departures from the request queues during cycles $[0,t]$. Each one of the request queues is being serviced deterministically at a rate of μ^i=$1/b$, since it takes a DRAM bank b cycles to finished updating one counter. Combining all the K request queues, therefore, the overall maximum departure rate for the statistics counter array is K/b.

3.4.2 Union Bound - System Overflow Probability

Our statistics counter array overflows when any one of the K queues overflows. Since the arrivals to request queues are i.i.d., it is reasonable to set the request queues to be of equal length L. We denote the counter array system overflow probability as $P_{overflow}$ and the overflow probability for a request queue S_i of length L as $P^i_{overflow}(L)$. Thus we have,

$$P_{overflow} \leq \sum_{i=1}^{K} P^i_{overflow}(L), \tag{3.5}$$

which is the union bound.

Therefore it suffices to analyze the overflow probability of one request queue. We start by obtaining the steady-state probability of the occupancy for a request queue of infinite size exceeding L as a surrogate for the overflow probability of a finite request queue of size L. The reason is that this surrogate is shown to be an upper bound to the actual overflow probability [14, 89]. Let ℓ be the number of pending requests in the request queue S_i, then

$$P^i_{overflow}(L) \leq P(\ell^i > L). \tag{3.6}$$

Combining (3.5) and (3.6), we have

$$P_{overflow} \leq \sum_{i=1}^{K} P(\ell^i > L). \tag{3.7}$$

Since the arrival processes to the request queues are i.i.d., and the request queues are serviced deterministically at the same rate $1/b$, we expect the occupancies $\ell^i(t)$ of any request queue at cycle t to be i.i.d. Therefore we can drop the superscript i and rewrite (6.12) as

$$P_{overflow} \leq K \times P(\ell > L). \tag{3.8}$$

Therefore, in the following we will analyze the probability that the number of requests ℓ in a request queue of infinite size exceeds L, and use (6.23) to bound the counter array system overflow probability.

3.4.3 Request Merging Probability

The merging request queues in the statistics counter are implemented as fully associative caches. In case of merging, the incoming request does not generate any new entry in the request queues. We define the *average merging probability* (AMP) function as $\Theta_{\text{amp}}(\ell)$, which is the probability that an incoming counter update request is merged into a request queue of length ℓ upon its arrival. The function $\Theta_{\text{amp}}(\ell)$ has the following properties:

1. $\Theta_{\text{amp}}(0) = 0$;
2. $\Theta_{\text{amp}}(\ell) > 0$, for $\ell > 0$;
3. $\Theta_{\text{amp}}(\ell_i) \leq \Theta_{\text{amp}}(\ell_j)$, for $0 \leq \ell_i \leq \ell_j$.

Property 1) states that the merging probability is zero when the request queue is empty. Property 2) states that the merging probability is positive when the request queue is non-empty. Property 3) follows from the fact that as the occupancy of a request queue increases, on average it is more likely to find a request in the request queue that has the same address as the incoming request. We will analyze the behavior of a request queue given a specific average merging probability function. Our model can be regarded as a

general birth-death process with variable birth rate and a fixed death rate. Here, a birth is the event of a counter update request creating a new entry in a request queue, and a death is the event that an entry in a request queue is removed to update the corresponding memory bank.

Now let's derive the steady-state probability of the occupancy of a request queue. For simplicity, we denote by λ the arrival rate and μ the service/departure rate of a request queue. For each request queue we have the following,

- arrival rate: $\lambda \leq 1/K$.
- service rate: $\mu = 1/b$.
- average merging probability: $\Theta_{\text{amp}}(\ell)$, where ℓ is the queue length observed by an arrival request.

To develop the worst case overflow probability of a request queue, we assume that the arrival rate is at its maximum value with $\lambda = 1/K$. We denote by P_ω the steady-state probability of a request queue with ω occupied entries (denoted as state ω). The balance equations for our model are as follows:

$$\lambda P_0 = \mu P_1; \tag{3.9}$$

$$\left(\lambda - \Theta_{\text{amp}}(\omega) + \mu\right) P_\omega = \left(\lambda - \Theta_{\text{amp}}(\omega - 1)\right) P_{\omega-1} + \mu P_{\omega+1},\ \omega > 0. \tag{3.10}$$

The above equations are due to the fact that at the steady-state, the rate at which a request queue leaves a state ω should be the same as the rate it enters the state. In Equation (3.9), the request queue can leave state 0 only by an arrival, since clearly there cannot be a departure when the queue is empty. Likewise, the request queue enters state 0 only by departure of one entry from state 1. On the left of Equation (3.10), the request queue leaves state $\omega > 0$ by one arrival or one departure, minus the cases when it is merged upon arrival. On the right side of Equation (3.10), the request queue enters state ω by one arrival when it is at state $\omega - 1$ if it is not merged, or by a departure when it is at state $\omega + 1$.

By solving Equations (3.9) and (3.10), we get the steady-state probabilities of the system as,

$$P_\omega = \frac{\prod_{i=0}^{\omega-1}\left(\lambda - \Theta_{\text{amp}}(i)\right)}{\mu^\omega} P_0,\ \omega > 0. \tag{3.11}$$

To understand how request merging helps reducing the request queue length, let's compare the steady-state probabilities of a system with request merging with a system without request merging. Let's denote the steady-state probabilities of a system without merging as Q_ω, where ω is its queue length at state ω. For the fairness of comparison, we assume the same arrival rate λ and service rate μ, so

$$\lambda Q_0 = \mu Q_1 \tag{3.12}$$

$$(\lambda + \mu) Q_\omega = \lambda Q_{\omega-1} + \mu Q_{\omega+1},\ \text{for } \omega > 0. \tag{3.13}$$

This is an M/D/1 queuing model with steady-state probability $Q_\omega = (\frac{\lambda}{\mu})^\omega Q_0$ [82].

Theorem 4. *$\exists\ \beta > 0$, such that $P_\omega < Q_\omega$, $\forall\ \omega > \beta$.*

Before proving Theorem 4, we present the following lemmas.

Lemma 5. *$P_{\omega+1} \leq \frac{\lambda}{\mu} P_\omega$, $\forall\ \omega \geq 0$.*

Proof. We know that:

$$P_{\omega+1} = \frac{\lambda - \Theta_{\text{amp}(\omega)}}{\mu} P_\omega,\ \text{for } \omega \geq 0 \tag{3.14}$$

$$\Theta_{\text{amp}} \geq 0. \tag{3.15}$$

Therefore $P_{\omega+1} \leq \frac{\lambda}{\mu} P_\omega$. □

Lemma 6. *$P_0 \geq Q_0$.*

Proof. From Lemma 5, we have

$$P_\omega \leq (\frac{\lambda}{\mu})^\omega P_0, \forall\ \omega \geq 0. \tag{3.16}$$

So,

$$1 = \sum_{\omega=0}^{\infty} P_\omega \leq \sum_{\omega=0}^{\infty} (\frac{\lambda}{\mu})^\omega P_0 \tag{3.17}$$

Thus

$$P_0 \geq \frac{1}{\sum_{\omega=0}^{\infty}(\frac{\lambda}{\mu})^{\omega}} = Q_0. \tag{3.18}$$

□

Now we shall prove Theorem 4.

Proof. From Lemma 6, we have $P_0 \geq Q_0$. We claim that there exists β such that $P_\beta \leq Q_\beta$, for $\beta \geq 0$. It clearly holds when $P_0 = Q_0$. Let's consider when $P_0 > Q_0$. If $\forall\ i$, $P_i > Q_i$, then

$$\sum_{\omega=0}^{\infty} P_\omega > \sum_{\omega=0}^{\infty} Q_\omega.$$

However, we know $\sum_{\omega=0}^{\infty} P_\omega = 1 = \sum_{\omega=0}^{\infty} Q_\omega$ which is a contradiction. So there is a β such that $P_\beta \leq Q_\beta$. Let's consider the smallest β. So we have that $P_k > Q_k$, where $0 \leq k < \beta$. We need to show that starting from β we have $P_j \leq Q_j$ for $\forall\ j \geq \beta$, which can be proved as follows.

$$P_{\beta+1} = \frac{\lambda - \Theta_{\text{amp}}(\beta)}{\mu} P_\beta. \tag{3.19}$$

Thus,

$$\begin{aligned} P_{\beta+1} &\leq \frac{\lambda - \Theta_{\text{amp}}(\beta)}{\mu} Q_\beta, \text{ since } P_\beta \leq Q_\beta, \\ &< \frac{\lambda}{\mu} Q_\beta = Q_{\beta+1}. \end{aligned} \tag{3.20}$$

Using the same argument, we have $P_{\beta+2} < Q_{\beta+2}$, etc. Therefore, $P_j < Q_j$ for $\forall\ j > \beta$, which proves Theorem 4. □

Theorem 5. *If $\Theta_{\text{amp}}(l)$ is an increasing function of l and upper-bounded by 1, then there exists a γ such that $\forall\ \omega > \gamma$, $P_\omega = 0$.*

Proof. The average merging probability function $\Theta_{\text{amp}}(l) \leq 1$. From Lemma 1, we know if $\lambda - \Theta_{\text{amp}}(\omega) \leq 0$, then $P_{\omega+1} \leq 0$. Since $\Theta_{\text{amp}}(l)$ is a non-negative increasing function and $\lambda \leq 1$, then for any fixed λ, there exists a γ such that $1 \geq \Theta_{\text{amp}}(\gamma) \geq \lambda$. Thus $P_{\gamma+1} \leq 0$. However, since $P_{\gamma+1}$ is the steady-state probability of request queue of length $\gamma+1$, $P_{\gamma+1} \geq 0$. Therefore $P_{\gamma+1} = 0$, which means the request queue will never grow to a length $\gamma+1$. As a result, the request queue will never grow beyond length $\gamma+1$, which leads to $P_\omega = 0$, $\forall\ \omega > \gamma$. □

For a statistics counter array, it is intuitive that as the number of requests in a request queue increases, the probability for an arrival request accessing the same memory address as one of the requests already buffered in the request queue increases. Therefore, we expect the lengths of the request queues to be bounded when request merging occurs with request merging probabilities following properties 1), 2), and 3).

3.4.4 Average Merging Probability for Internet Flows

Mergings in the request queues occur when an arriving update and an update in the request queues are accessing the same counters with the same memory addresses. Traffic bursts caused by multiple packets from the same flows being sent across the network from senders to receivers during a short period time contribute greatly to the mergings in the request queues. On the Internet, the flow sizes (in the number of packets) of the traffic have been known to follow certain heavy-tailed (or long-tailed, power-law tailed) distribution [24, 30]. Unlike a Gaussian distribution, a heavy-tailed distribution has no "typical" scale and is hence frequently called "scale-free". Such a feature implies that for any fix-sized time window, the flow size distribution of all active flows still follows a certain heavy-tailed distribution. For a counter array maintaining flow statistics, such a distribution would lead to frequent updates to the same counters, which would necessarily be mapped to the same memory banks. More update requests may be generated by traffic bursts than the throughput of a counter array during even a relatively short period of time, due to the "scale-free" nature of the heavy-tailed distribution, which would cause a system with fix-sized request queues to overflow.

To solve the above problem, in our statistics counter array with merging request queues, traffic bursts within a short time window can be successfully combined in the request queues without generating extra accesses to the memory banks. In this section, we will analyze the average merging probability in a request queue under the assumption that the flow sizes follow a certain heavy-tailed distribution.

Let X be a variable representing the flow size. The flow size distribution on the Internet is heavy-tailed and it can be expressed as follows [24, 30],

$$\texttt{Prob}(X > x) \sim x^{-\alpha}, \text{ as } x \to \infty, \tag{3.21}$$

where X is a random variable representing the number of packets in a flow and α is a shape parameter. For a heavy-tailed distribution, we have that $0 < \alpha < 2$ and the distribution has infinite variance. The shape parameter α determines how frequently large flows appear. A smaller α means that the distribution has a larger tail and large flows appear more frequently, which would result in a smaller request queue size. As α becomes larger, large flows appear less frequently, which would result in a larger request queue size for the same amount of arriving traffic since request merging becomes less likely.

If the flow sizes on the network follow the heavy-tailed distribution in Equation (3.21), then the probability mass function for a flow to be of size $X = x$ packets can be expressed as,

$$\mathtt{p}(X = x) = c\alpha x^{-\alpha-1}, \tag{3.22}$$

where c is a constant such that the sum of all the probabilities over X equals 1.

Let's assume that the size of the largest flow is F_{max}. The average number of active flows within a time window T is defined as N_{active}. For a randomly chosen packet arriving within the same time window T, the probability that it belongs to a flow of size x is denoted as $\mathtt{Prob}_x$, which is:

$$\mathtt{Prob}_x = \frac{x}{\sum_{y=1}^{F_{max}} y \cdot \mathtt{p}(X = y) \cdot N_{active}}, \tag{3.23}$$

where $\mathtt{p}(X = y) \cdot N_{active}$ is the total number of flows of size y within the time window T. The denominator in Equation (3.23) is the total number of packets on the network during the time period T. Within the same time window T, the maximum number of total packets arriving is denoted as S. Then the number of active flows can be expressed as,

$$N_{active} = \frac{S}{\sum_{y=1}^{F_{max}} y \cdot \mathtt{p}(X = y)}. \tag{3.24}$$

For a request queue currently containing ℓ pending requests, the probability that a newly arriving counter update request is merged with one of the ℓ requests in the request queue can be expressed as,

$$\mathtt{Prob}_{\mathtt{merge}}(\ell) = \sum_{x=2}^{F_{max}} \ell \cdot \mathtt{Prob}_x^2 \cdot \mathtt{p}(X = x) \cdot N_{active}. \tag{3.25}$$

Equation (3.25) can be explained as follows. A new counter update request is merged with the one already stored in the request queue only if they are from the same flow and thus sharing the same counter address. For an arriving packet from a flow of size x, each pending request in the request queue is generated by a packet from the same flow with probability $\texttt{Prob}_x$. There are ℓ pending requests in the request queue. So the merging probability for a request generated by a packet from a flow of size x is denoted as $\ell \cdot \texttt{Prob}_x^2$. Among all the active flows, there are $\texttt{p}(X = x) \cdot N_{active}$ flows of size x. Merging is only possible for flows with more than two packets. Thus to calculate the overall merging probability, the summation is from the flows of size two packets to F_{max} packets. Equation (3.25) is also the average merging probability as a function of the number of requests ℓ in a request queue.

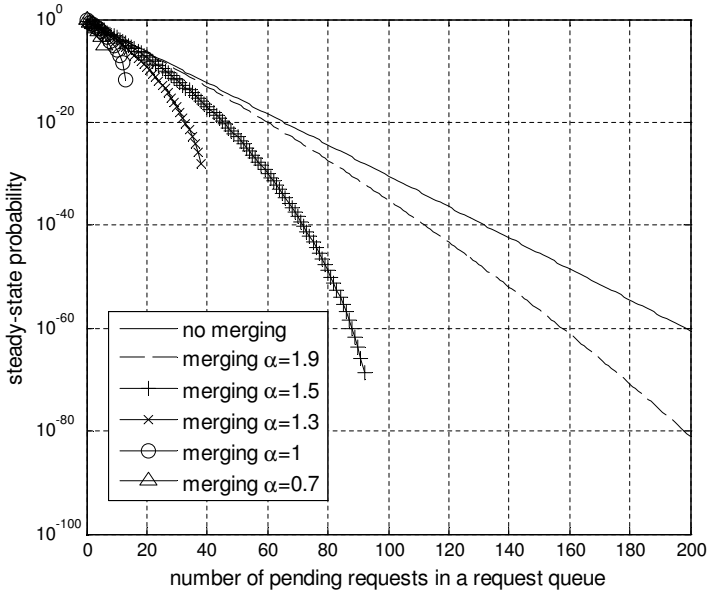


Figure 3.2: Steady-state probability for merging request queue under heavy-tailed traffic with $b = 16$ and $K = 32$ banks.

The steady-state probability for a merging request queue under heavy-tailed traffic with DRAM-to-SRAM access latency ratio $b = 16$ and a total of $K = 32$ DRAM banks is shown in Figure 3.2. It is easy to see from this figure that for request queues with no merging capability, the maximum queue length is unbounded, since the steady-state probability for the request queue to reach any queue length is positive. However,

for our merging request queues under heavy-tailed traffic with $\alpha = 1.5$, the maximum queue length is bounded by about 100 entries. With $\alpha = 1.3$, the maximum queue length becomes 40. With $\alpha = 1$, the maximum queue length is bounded by less than 20 entries.

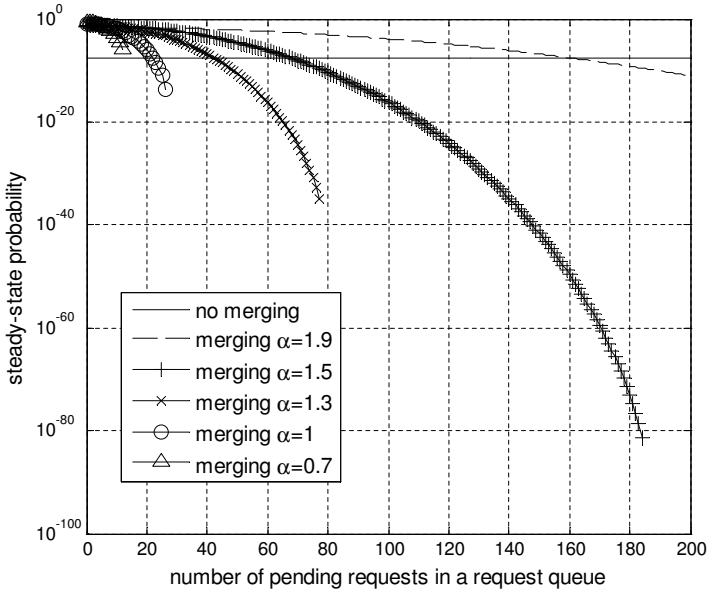


Figure 3.3: Steady-state probability for merging request queues under heavy-tailed traffic with $b = 16$ and $K = 16$ banks.

Figure 3.3 shows the steady-state probability for a merging request queue under heavy-tailed traffic with parameters $b = 16$ and $K = 16$. For request queues with no merging capability, the maximum queue length is unbounded. The request queue can never reach steady state, since the arrival rate is the same as departure rate. On the other hand, for merging request queues under heavy-tailed traffic with $\alpha = 1.5$, the maximum queue length is bounded by about 185 entries. When $\alpha = 1.3$, the maximum queue length becomes 80. For heavy-tailed traffic with parameter $\alpha = 1$, the maximum queue length becomes only about 25.

3.5 Robustness Against Adversaries

For real-world Internet traffic, the arriving processes $A_j[t]$ to different counters may not be i.i.d. Certain flows may trigger the corresponding counters to be updated

more frequently than the others, which means that in our system, counter update requests to certain counters may arrive at higher probability than average. However, higher arrival flow rate generates more incoming requests in a short time, therefore the requests are actually more likely to be merged to a request already in the request queue. The merging request queues effectively smooth the traffic bursts in the arrival process.

An adversary can only attack the system by sending the same counter update request as infrequently as possible, thereby effectively reducing the average merging probability in the request queues. In the robust statistics counter architecture in Figure 2.1(b), the cache module is separated from the request queues. The request queues only buffers the pending requests to memory banks. The cache module merges the repetitive requests to the same memory location. In Chapter 2, a worst case performance evaluation is provided. To achieve a overflow probability of 10^{-14}, the cache is of size about 7000 entries, each request queue has 50 entries, and a total of $K = 32$ memory banks. The key problem with such a design is that in order to lower the overflow probability for the worst case arrival traffic pattern (with adversaries), the performance of the system under regular traffic is not optimized. To compare our statistics counter array with merging request queues to the one in Chapter 2, let's consider the design where the total size of the request queues is the same as the size of the cache in Figure 2.1(b) of 7000 entries. Therefore, there are about 218 entries in each request queue. From Section 3.4.4, we can achieve zero overflow probability for arriving traffic with a heavy-tailed distribution, while the example in Chapter 2 only achieves an overflow probability of about 10^{-14}, which is due to the much smaller request queue size.

In the presence of adversaries sending repetitive counter updates to overflow the system, our statistics counter array can be modified to be more robust by allowing different DRAM banks to share their request queues as shown in Figure 3.4. When one request queue is full and the arriving counter update cannot be merged to any entry in the request queue, the arbiter will pick another request queue to store the arriving update. Since a request queue can finish three operations in one cycle as stated in Section 3.3.1, including a query, an insertion and a removal, query operations can be initiated instantaneously in different request queues to look for a match for the arriving counter update request. If there is a match in any request queue, the newly arrived request will be

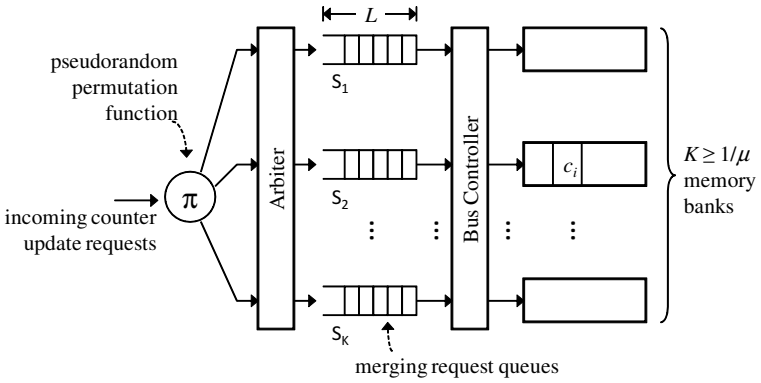


Figure 3.4: Statistics counter array with shared request queues.

merged with the one in the request queue and no new entry will be generated in the request queues. The arbiter effectively snoops all the request queues for a match upon a request arrival. When a counter update request departs from a request queue, the bus controller decides which bank the request should be sent to based on its counter index. Therefore, the repetitive counter updates separated by up to $K \cdot L$ cycles can be merged by the request queues when the queues are almost full. When the request queues have empty entries, repetitive updates will not cause an overflow. So the statistics counter with shared request queues performs at least as good as the scheme in Chapter 2 in the presence of adversaries. To see this, let's first consider that the best strategy for an adversary is by sending same counter update requests every C cycles which is is proven in Chapter 2, where C is the size of the cache. All repeated requests within C cycles will be merged to another request already in the cache. In Figure 3.4, when the request queues are full, we effectively have a cache of size $K \cdot L$. All the repeated requests within $K \cdot L$ cycles are merged. When the request queues are not full, the arbiter will always be able to find an empty entry in which to store the request. By setting $K \cdot L = C$, we can achieve the same overflow probability as in Figure 2.1(b) for the worst case traffic pattern. Moreover, the size of each merged request queue L is typically much larger than L' in Figure 2.1(b), therefore we also achieve lower overflow probability for incoming traffic in absence of adversaries since system overflows only occur in the request queues.

3.6 Performance Evaluation

3.6.1 Traffic Traces

In this section, we present simulation results of our merging statistics counter architecture with two real-world Internet traffic traces from USC and UNC in Chapter 2. 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. To test the performance of our statistics counter array under heavy traffic loads, we speed up the traffic traces so that one counter update request arrives at the system in each time slot.

3.6.2 Experimental Results

We are interested in the maximum queue length (occupancy) in the request queues for various queue update policies. In a statistics counter design, if the maximum queue length is bounded by a constant B for a specific traffic trace, then request queues guarantee no overflow for the traffic trace for each queue of length B. Our queueing model predicts that for request queues without a merging operation, the maximum queue length is unbounded since the steady-state probability for any queue length is non-zero, for all arrival rates of the incoming traffic. On the other hand, the maximum queue length is bounded for our merging request queue design, if the incoming traffic follows a certain heavy-tailed distribution.

Typically, DRAM access latency is 10 to 20 times larger than the SRAM access latency. In our experiment, we assume that the SRAM is working at line rate, so that one counter update takes one cycle to process. On the other hand, it takes a DRAM bank b cycles to process a counter update. In the following, we assume $b = 16$.

With K DRAM banks, each bank needs to store $16/K$ millions of counters. We assume that each DRAM access takes $b = 16$ cycles. The maximum request queue lengths for $K = 32$ memory banks using the USC trace is shown in Figure 3.5. The results for both FIFO and LRU update rules are presented together with the maximum queue length of a simple request queue with no request merging for comparison. The

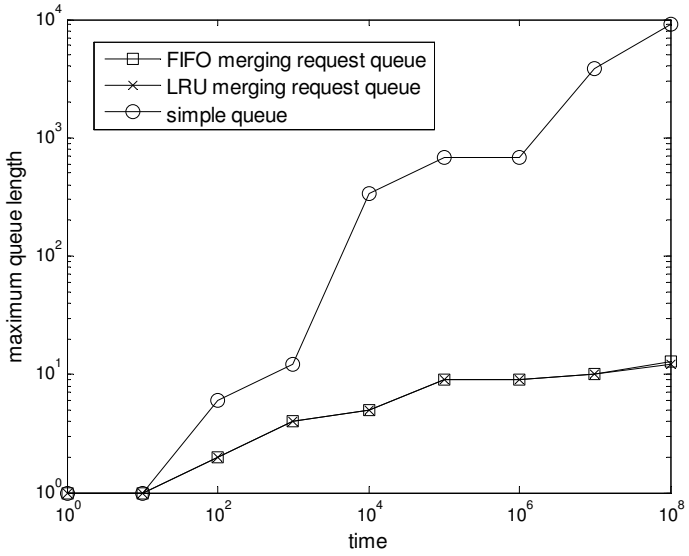


Figure 3.5: Maximum queue length, $K = 32$, USC trace.

maximum queue lengths with the FIFO update rule and LRU update rule are about the same, while the LRU scheme achieves slightly smaller queue size. After 10^8 packets arrivals, the maximum queue length with FIFO update rule is 13 entries, while it's only 11 entries for request queues with LRU update rule. The maximum queue length of the simple request queue with no merging operation is in the order of 10^4, which is significantly larger than that of the FIFO or LRU scheme. Also, the maximum queue length of the simple request queue increases as more updates arrive, since during a larger period of time there are more traffic bursts generating updates to the same memory locations, which increases the maximum queue length considerably. Over time the occupancies in different request queues will drift apart [51]. It is worth noting that in this experiment, we use enough DRAM banks to provide a potential throughput that is twice the line rate by using 32 memory banks while a single bank is only 16 times slower. Such a scenario can also be translated to a reduced arriving traffic where the arrival rate is only 50% of the service rate. However, even with the pseudorandom permutation function, there is no guarantee that the loads to the memory banks are uniformly distributed at all times due to the existence of traffic bursts. So by building a counter array with more DRAM banks to provide higher equivalent throughput or reducing the arrival rate, there is still no guarantee of a bounded request queue size. However, with the merging operations,

most traffic bursts are merged in the request queues. The maximum queue length stays bounded in our statistics counter array as predicted by the models in Section 3.4.4.

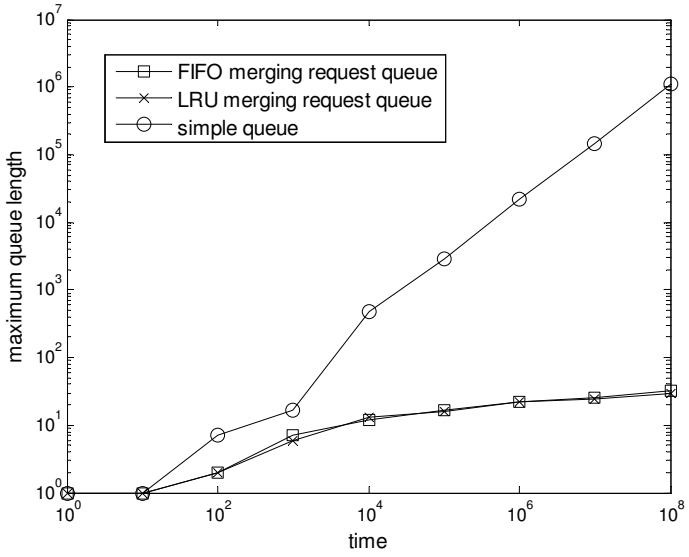


Figure 3.6: Maximum queue length, $K = 16$, USC trace.

To further stress the system and examine its performance, we present the maximum queue lengths for different designs using only $K = 16$ memory banks and $b = 16$ in Figure 3.6. For a simple queue without merging, the maximum queue length quickly explodes as more updates arrive since on average the arrival rate to a request queue is the same as the service rate. Any bursts of updates with the same memory addresses will cause the request queues to increase dramatically. However, in our merging queue design, the maximum queue lengths are only increased by roughly a factor of 2 compared to the result shown in Figure 3.5 where 32 banks are used. For a FIFO update rule scheme, the maximum queue length is 29. For an LRU update rule scheme, the maximum queue length is only 27. As the maximum queue lengths increase, on average there are more counter updates stored in a request queue at any time and the average merging probability for arriving updates is also increased, which in return effectively reduces the maximum queue lengths.

We observe that FIFO and LRU queue update schemes achieve similar performances with LRU scheme slightly better. Compared with the result for a simple queue with no merging, the maximum queue lengths for FIFO and LRU schemes are greatly

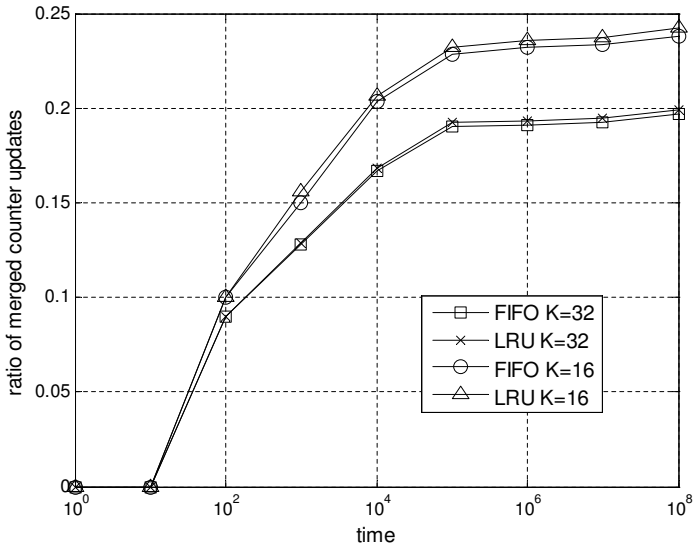


Figure 3.7: Ratio of merged counter updates, USC trace.

reduced. Figure 3.7 shows the ratio of merged counter update requests with different configurations for FIFO and LRU schemes. We can see that about 20% of the incoming counter updates are merged with $K = 32$ for both schemes. By reducing the number of memory banks, the ratio of merged incoming counter updates grows to 25%, which further increases the average merging probability Θ_{amp}. The increased ratio of merged updates explains why the maximum queue lengths only increase by a factor of 2 as shown in Figure 3.6.

The maximum request queue lengths for $K = 32$ memory banks and each DRAM access takes $b = 16$ cycles using the UNC trace is shown in Figure 3.8. The maximum queue length of the simple request queue with no merging operation is significantly larger than the maximum queue length with merging request queues. With merging operation, for both FIFO and LRU schemes the maximum queue lengths stay stable below 12 entries. Figure 3.8 also shows that the maximum queue length for the simple queue scheme with UNC traffic is not always increasing, such as from time 10^3 to 10^4 cycles. This is due to the fact that UNC trace has 13.5 million active flows, which is much larger than the 8.6 million active flows from USC trace. During time such as 10^3 to 10^4 cycles, there are more active flows with very few traffic bursts. With the help of the pseudorandom permutation function, the arriving requests are effectively distributed

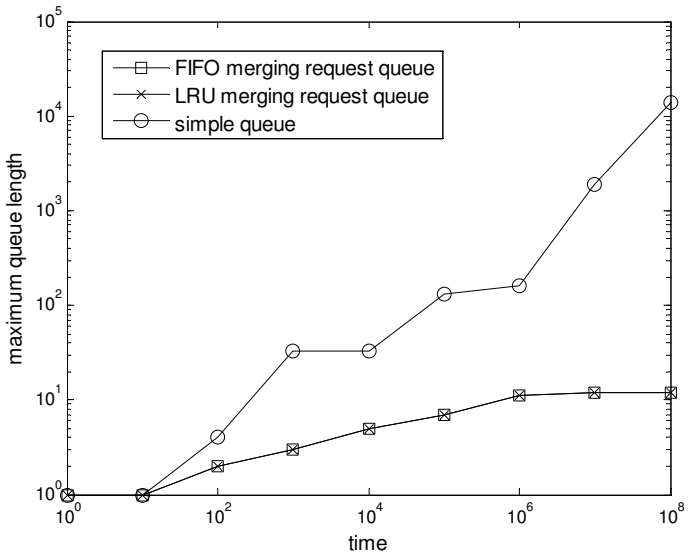


Figure 3.8: Maximum queue length, $K = 32$, UNC trace.

more uniformly without the heavy presence of traffic bursts. Since the average service rate is twice the average arrival rate to any request queue, the maximum request queue length is actually decreasing during this time.

The maximum request queue lengths for $K = 16$ memory banks and $b = 16$ using the UNC trace is shown in Figure 3.9. With fewer memory banks, the maximum queue length of the simple request queue with no merging operation grows even faster. The maximum queue length for merging request queues with FIFO update rule stays stable below 52 entries. The maximum queue length for merging request queues with LRU update rule is only 49 entries.

Similar to the results using USC trace, compared with a simple queue with no merging, the maximum queue lengths for both FIFO and LRU schemes are greatly reduced. In Figure 3.10, we can see that about 8% of the incoming counter updates are merged with $K = 32$. If we reduce the number of memory banks, the maximum queue length increases and the ratio of merged incoming counter updates increases to 12% for $K = 16$. It is also shown in Figure 3.10 that during the time 10^3 to 10^4 cycles, the ratio of merged counter updates decreases, which is caused by the sudden increase in the number of active flows and the decrease in number of traffic bursts in the UNC trace during that time. Even with a reduced number of traffic bursts, our statistics counter

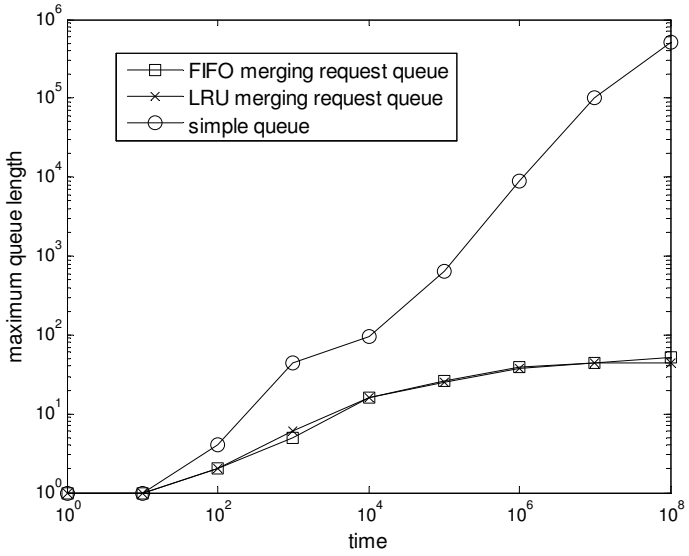


Figure 3.9: Maximum queue length, $K = 16$, UNC trace.

array still outperforms a system with simple queue significantly by providing a bounded maximum request queue size as shown in Figure 3.8 and Figure 3.9.

3.6.3 Cost-benefit Comparison with Other Approaches

For the proposed solution, using $K = 32$ DRAM banks and request queues of size $L = 20$ entries each are sufficient for the traffic traces using either FIFO or LRU queue update policy. The fully associative cache needed for request queues is of size $M = K \cdot L$ entries. In the DRAM banks there are 16 millions counters. Each entry in the request queues requires 3 bytes to encode the counter indices of 16 million counters and 4 bits for accumulated counts. So the total cache size is about 2.2 KB. Our experiments show that a simple modulo function mapping the counter indexes to different DRAM banks is sufficient to guarantee fairly uniform load distribution to different banks in absence of traffic bursts, in which case there is no need to store the pseudorandom permutation function explicitly. In fact if the input packet label (such as its 5-tuple) needs to be hashed to a counter index, we can view the hashing as random and a separate permutation is no longer necessary. Hash functions on the 5-tuple in the packet header are commonly applied in network processors, thus no extra system cost will be involved.

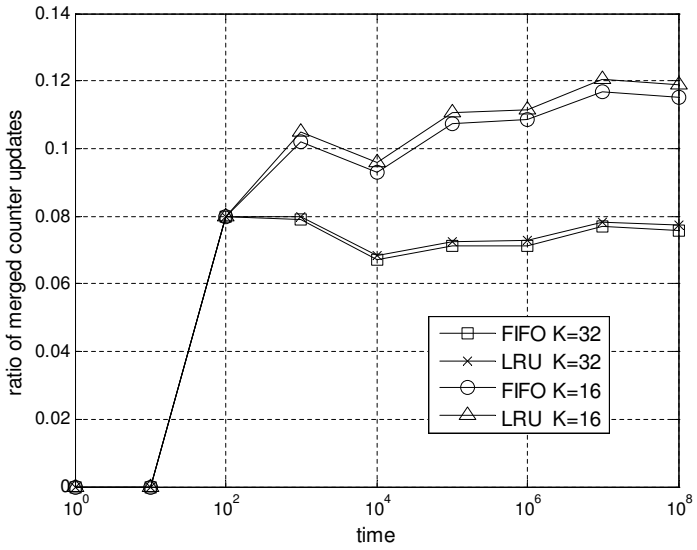


Figure 3.10: Ratio of merged counter updates, UNC trace.

In the counter architecture of Chapter 2, it requires 25 KB in cache and 5.5 KB in SRAM to achieve an overflow probability of 10^{-14}. For the statistics counter array in this chapter, we use only 25 KB in request queues as cache to achieve the same worst case performance guarantee as discussed in Section 3.5. Additionally, our scheme provides zero overflow probability for normal traffic without adversaries. As a comparison, the hybrid SRAM/DRAM counter architecture approaches [108] would require more than 8 MB of SRAM, and the SRAM size grows linearly with the number of counters while providing no worst case service guarantee. Moreover, repetitive counter updates are merged by our merging request queues while they cause system overflow in hybrid SRAM/DRAM schemes. The detailed comparison is shown in Table 3.1.

3.7 Conclusion

We proposed a robust DRAM-based statistics counter architecture that can effectively maintain exact wirespeed updates to large counter arrays. To take advantage of the fact that most flows on the Internet consist of multiple packets that are being sent within a short time, our proposed architecture makes use of a simple randomization scheme plus a set of small fully associative request queues to statistically guarantee a near-

Table 3.1: Comparison of different schemes for a reference configuration with 16 million 64-bit counters. For our method, $K = 32$ and $L = 20$.

	Hybrid SRAM/ DRAM ([108])	Counter with Cache (Chapter 2)	Counter with Merging Queues (this Chapter)
Counter DRAM	128 MB	128 MB	128 MB
Counter SRAM	8 MB	None	None
Control	1.5 KB SRAM	15 KB CAM and 5.5 KB SRAM	2.2 KB CAM

perfect load-balancing of counter updates to the memory banks. This effective turns the bursty arriving traffic in our favor by reducing the total number of counter update requests to be processed by the DRAM banks in the counter array. We also developed queueing models and showed that with traffic bursts in the arriving counter updates, the maximum request queue length is always bounded by a small value. Our simulation results proved our model by showing that with merging request queues, the maximum queue length can never grow over a fixed size. Therefore, our proposed statistics counter architecture can effectively maintain wirespeed updates to large counter arrays while providing a diminishing overflow probability for Internet traffic.

Chapter 3, in part, has been submitted for publication of material as it may appear in IEEE Transactions on Parallel and Distributed Systems, Hao Wang, Bill Lin, and Jun (Jim) Xu, "Robust Statistics Counter Arrays with Interleaved Memories". The dissertation author was the primary investigator and author of the paper.

Chapter 4

Robust Memory Systems for Network Processing

4.1 Introduction

Modern Internet routers often need to manage and move a large amount of packet- and flow-level data. Therefore, it is essential for the memory system of a router to be able to support both read and write accesses to such data at link speeds. As link speeds continue to increase, router designers are constantly grappling with the unfortunate tradeoffs between the speed and cost of SRAM and DRAM. While fitting all such data into SRAM with access latencies typically between 4 and 8 ns is fast enough for the highest link speeds, the huge amount of SRAM needed renders such an implementation prohibitively expensive, as hundreds of megabytes or even gigabytes of storage may be needed. On the other hand, although DRAM provides inexpensive bulk storage, the prevailing view is that DRAM with access latencies typically between 50 and 100 ns is too slow for providing wirespeed updates. For example, on a 40 Gb/s OC-768 link, a new packet can arrive every 8 ns, and the corresponding read or write operation to the data structure needs to be completed within this time frame. In this work, we do away with this unfortunate tradeoff entirely, by proposing a memory system that is almost as fast as SRAM, *for the purpose of managing and moving packet and flow data*, and almost as affordable as DRAM.

4.1.1 Motivation

To illustrate our problem, we list here three massive data structures that need to be maintained and/or moved inside high-speed network routers.

Network flow state: Maintaining precise flow state information [19] at an Internet router is essential for many network security applications, such as stateful firewalls, intrusion and virus detection through deep packet inspection, and network traffic anomaly analysis. For example, a stateful firewall must keep track of the current connection state of each flow. For signature-based deep packet inspection applications such as intrusion detection and virus detection, per-flow state information is kept to encode partial signatures (e.g., current state of the finite state automata) that have been recognized. For network traffic anomaly analysis, various statistics and flow information, such as packet and byte counts, source and destination addresses and port information, flow connection setup and termination times, routing and peering information, etc, are recorded for each flow during a monitoring period for later offline analysis. Indeed, the design of very large flow tables that can support fast lookups and updates has been a fundamental research problem in network security area.

Precise flow state also needs to be maintained for many other network applications unrelated to security. For example, in packet scheduling algorithms (for provisioning network quality-of-service), for each *flow*, the flow table needs to maintain a *timestamp* that indicates whether the flow is sending faster or slower than its fair share, which in turn determines the priority of the packets currently waiting in the queue that belongs to this flow.

The management of precise per-flow state information is made challenging by at least two factors. First, the number of flows can be extremely large. Studies have shown that millions of concurrent flows are possible at an Internet router in a backbone network [50]. Second, the lookup and updating of flow records must be performed at very high speeds. As mentioned, although SRAM implementations can support wirespeed access, they are inherently not scalable to support millions of flow records (potentially hundreds of millions in the future). On the other hand, although DRAM provides huge amounts of inexpensive, naïve usage of DRAMs would lead to worst-case access times that are well beyond the packet arrival rates on a high-speed link. As a result, most

network security applications have typically been deployed in the access network close to the customers to reduce both the number of flows that need to be managed as well as the wirespeed. However, with the proliferation of mobile devices, the number of concurrent flows is expected to increase rapidly even at the edge of the network. In addition, there are a number of reasons for deploying some network security functions deeper into the backbone network where the number of concurrent flows is substantially higher. For example, the backbone network carries a far more diverse traffic mix that can potentially pose a wider range of security problems. Other solutions for mitigating the need to maintain large amounts of precise state information at wirespeed include data streaming [68] and sampling methods, such as Cisco's sampled NetFlow [27], Juniper's filter-based accounting [86], and the sample-and-hold solution [28]. However, these methods cannot keep track of accurate flow states, which imposes limitations on their effectiveness.

High-speed packet buffers: The implementation of packet buffers at an Internet router is another essential application that requires wirespeed access to large amounts of storage. Historically, the size of packet buffers have been increasing with link speeds. To tackle this problem, several designs of packet buffers based on hybrid SRAM/DRAM architectures or memory interleaved DRAM architectures have been proposed [29, 43, 46, 51, 89, 99]. Although the designs are both effective and practical, these solutions are problem-specific. Ideally, one would like a general memory architecture that is applicable to both the packet buffering problem as well as to the various network flow state implementation problems.

Statistics counter arrays: Network measurement is essential for the monitoring and control of large networks. For implementing 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 wirespeed. Thus the design of memory systems specialized for statistics counting has received considerable research attention in the recent years. The problem of maintaining statistics counters is simpler in nature compared to the design of a general memory system since the statistics counter only needs to support wirespeed updates (write operations), whereas a general memory system needs to support arbitrary memory read and write operations at wire-

speed.

4.1.2 Our Approach

In this work, we design a DRAM-based memory architecture that can emulate a fast massive SRAM module by exploiting memory interleaving. In this architecture, multiple DRAM banks run in parallel to achieve the throughput of a single SRAM. The proposed architecture provides fixed delay for all the read requests while ensuring the correctness of the output results of the read requests. More specifically, if a memory read operation is issued at time t, its result will be available at output exactly at time $t+\Delta$, where Δ is a fixed pipeline delay. In this way, a processor can issue a new memory operation every cycle, with a deterministic completion time Δ cycles later. No interrupt mechanism is required to indicate when read data is ready or a write operation has completed for the processor that issued the requests. With a fixed pipeline delay, a processor can statically schedule other instructions during this time. Although this Δ is much larger than the SRAM access latency, router tasks can often be properly implemented to work around this drawback due to the deterministic nature of this delay.

For example, in packet scheduling algorithms for provisioning quality-of-service, the flow table needs to maintain for each *flow* a *timestamp* that indicates whether the flow is sending faster or slower than its fair share, which in turn determines the priorities of the packets belonging to this flow that are currently waiting in the queue. Using weighted fair queueing [21], when a new packet *pkt* arrives at time t, a read request will be issued to obtain its flow state such as the virtual finish time of the last packet in the same flow. At time $t+\Delta$, when the result comes back, the virtual finish time of *pkt* (a per-flow variable) and the system virtual time (a global variable) can both be precisely computed and updated because they depend only on packet arrivals happening before or at time t, all of which are retrieved at time $t+\Delta$. In other words, given any time t, the flow state information needed for scheduling the packets that arrive before or at time t are available for computation at time $t+\Delta$. Although all packets are delayed by Δ using the proposed design, we will show that this Δ is usually in the order of tens of microseconds, which is three orders of magnitude shorter than end-to-end propagation delays (tens of milliseconds across the entire US).

While multiple DRAM banks can potentially provide the necessary raw bandwidth that high-speed routers need, traditional memory interleaving solutions [58,80] do not provide consistent throughput reliably because certain memory access patterns, especially those that can conceivably be generated by an adversary, can easily get around the load balancing capabilities of interleaving schemes by overloading a single DRAM bank. Another solution proposed in [1] is a virtually-pipelined memory architecture that aims to mimic the behavior of a single SRAM bank using multiple DRAM and SRAM banks under average arrival traffic with no adversaries. All the pending operations to the memory system are provided with fixed delay using a cyclic buffer. However, this architecture assumes perfect randomization in the arrival requests to the memory banks. Under adversarial arrivals, the buffer will overflow and start dropping requests, which greatly degrades the system performance.

To guard against adversarial access patterns, memory locations are randomly distributed across multiple memory banks so that a near-perfect balancing of memory access loads can be provably achieved. This random distribution is achieved by means of a random address permutation function. Note that an adversary can conceivably overload a memory bank by sending traffic that would trigger the access of the same memory location. They will necessarily be mapped to the same memory bank. However, this case can be easily handled with the help of a reservation table (which serves in part as a data cache) in our memory architecture. With a reservation table of C entries, we can ensure that repetitive memory requests to the same memory address within a time window C will only result in at most two memory accesses in the worst-case, with one read request followed by one write request. All the other requests will only be stored in the reservation table to provide a fixed delay.

4.1.3 Outline of Chapter

The rest of the chapter is organized as follows. Section 4.2 presents the related work on the memory system design. Section 4.3 defines the notion of SRAM emulation. Section 4.4 describes the basic memory architecture. Section 4.5 extends our proposed memory architecture by providing robustness against adversarial access patterns. Section 4.6 provides a rigorous analysis on the performance of our architecture in

the worst-case. Section 4.7 presents an evaluation of our proposed architecture. Finally, Section 4.8 concludes the chapter.

4.2 Related Work

The problem of designing memory systems for different network applications has been addressed in many past research efforts. We will first present the development of specialized memory systems for statistics counter arrays. Then we will show memory systems designed for allowing only restricted memory access patterns, and also the memory systems with techniques to accommodate unrestricted memory access patterns.

4.2.1 Statistics Counter Arrays

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 [108], 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. 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 quickly overflow. Thus a total of 108 MB of SRAM would be needed for just the counter storage, which is prohibitively expensive. Researchers have been actively seeking alternative ways to realize large arrays of statistics counters at wirespeed [20, 37, 59, 65, 76, 81, 87, 91, 107, 108].

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 of 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 are scheduled to be “flushed" back to the corresponding DRAM counters. These schemes all significantly reduce the SRAM cost. However, such designs are not suitable for arbitrary increments to the counters which would cause the counters in SRAM to be flushed back to DRAM arbitrarily more frequently. In addition, they do not support arbitrary decrements and are based on an in-

teger number representation. Another statistics counter array design proposed in [107] utilizes several DRAM banks to mimic a single SRAM. There is a request queue for each DRAM bank to buffer the pending counter update requests to the bank. The counter updates are distributed into different memory banks with the help of a random permutation function. Also a cache module is implemented to reduce the memory accesses caused by repetitive updates to the same counters. Such a design supports both counter increments and decrements, as well as floating point number representation and other formats. However, it is suitable only for the maintenance of statistics counters where memory write operations to update the counters occur much more frequently than read operations to retrieve the counter values, and such counter retrieval read operations are sparsely distributed in time or even processed off-line. Special purpose memory architectures are also developed for IP lookup [7, 45, 52], which is a fundamental application supported by all routers. In general, specialized memory systems for maintaining statistics counters cannot be easily adapted to support arbitrary data accesses and updates.

4.2.2 Memory Systems with Restricted Accesses

To design memory systems with bulk storage working at wirespeed, multiple DRAM are often implemented in parallel to achieve SRAM throughput. Since the access latency of a DRAM bank is much larger (several orders of magnitude) than that of SRAM, various approaches have been developed to hide such a latency. Memory interleaving solutions are proposed in [58, 80] where prefetching and other bank-specific techniques are developed. In these schemes, the memory locations are carefully distributed into several memory banks in a pseudorandom fashion. If the memory accesses are evenly distributed across memory banks, the number of bank conflicts can be reduced. Therefore, the supported memory access patterns are greatly restricted which limits the applications of such designs. For example, in the application of a packet buffer for one traffic flow, the arriving packets can simply be stripped sequentially across memory banks in the FIFO order using the memory interleaving solution. However, consider a high speed packet buffer supporting millions of flows. The order in which packets are retrieved from the memory is typically determined by a packet scheduler implementing a specific queueing policy. The packets across the memory banks may be retrieved in

an unbalanced fashion, which can cause the system performance to degrade drastically. Other applications such as lookups of global variables will trigger accesses to the same memory locations repeatedly and all of the accesses are mapped to the same memory banks for sure, since the memory locations in the memory banks are fixed after the mapping. Such events can potentially cause unbounded delay for pending memory accesses to the memory banks. Moreover, in any memory system the buffer size is always limited, therefore eventually memory access requests will be dropped when a buffer overflows. For network routers, dropped TCP packets may require frequent retransmissions by the senders, which would further reduce the available network bandwidth and drain the available buffers. In network security applications, buffer overflows may introduce unaccounted for network traffic which would cause failures in tracking down suspicious flows. This can lead to an increase of miss rate in identifying malicious traffic patterns. In summary, memory systems with restricted access patterns are not robust to unrestricted traffic patterns and have very limited applications.

4.2.3 General Memory Systems with Unrestricted Accesses

Extensive research efforts have taken place in the design of high speed packet buffers to support unrestricted packet retrievals. For example, in order to implement packet buffers at an Internet router, the buffers need to be able to work at wirespeed and to provide bulk storage. As the link speed increases, the size of these packet buffers increase also. To tackle this problem, several designs of packet buffers based on hybrid SRAM/DRAM architectures or on memory interleaved DRAM architectures have been proposed [29, 33, 43, 46, 51, 89, 99]. The basic idea of these architectures is to use multiple DRAM banks to achieve SRAM throughput, while reducing [33, 51, 89] or eliminating [29,43,46,99] the potential bank conflicts. One of the designs, the prefetching-based solutions described in [29, 43] support linespeed queue operations by aggregating and prefetching packets for parallel DRAM transfers using fast SRAM caches. These architectures require complex memory management algorithms for real-time sorting and a substantial amount of SRAM for caching the head and tail portions of logically separated queues to handle worst-case access patterns. Another type of the packet buffer design, randomization-based architectures [51, 89] are based on a random placement of

packets into different memory banks so that the memory loads across the DRAM banks are balanced. While effective, these architectures only provide statistical guarantees as there are deterministic packet sequences that will cause system overflows. The third type of the packet buffer design is the reservation-based architectures [46, 99] where a reservation table serves as a memory management module to select one bank out of all available banks to store an arriving packet such that it will not cause any conflicts at the time of packet arrival or departure. An arrival conflict occurs when an arriving packet tries to access a busy bank. A departure conflict occurs when a packet in a busy bank needs to depart. In order to avoid these conflicts, these architectures need to use about three times the number of DRAM banks and to deploy a complicated algorithm in the memory selection step. In general, while the above approaches are effective in the design of packet buffers, the substantial amount of SRAM requirement, the vulnerability towards certain arrival patterns, or the lack of providing a fixed delay for memory read or write accesses renders them unsuitable for many network applications.

Ideally, we would like a general memory architecture that is applicable to both the packet buffering problem as well as to various other network applications. The idea of a pipelined memory system with a fixed pipelined delay was proposed by Agrawal and Sherwood in [1], where a virtually pipelined network memory (VPNM) architecture is presented. In this system, a universal hashing function is applied on all arriving memory access requests to distribute them into multiple DRAM banks. Each memory access request is delayed by a fixed D cycles. The VPNM works quite effectively in the average case, as verified by both their analytical and empirical results. However, network operators are often concerned about worst-case performance, particularly in the case of network security applications. Indeed, in the literature, most proposed algorithms [95] for packet classification, regular expression matching, and weighted fair queueing, just to name a few, are all designed for the worst-case, even though worst-case situations may not be common. For example, it is well-known that most IP longest prefix matches can in practice be resolved with the first 16 to 24 bits, but yet the IP lookup algorithms that have been adopted commercially have all been designed to operate at wirespeed to match all 32 bits of an IPv4 address in the worst-case. Similarly, it has also been shown that most regular expression matches fail after a small number of symbols, but in the

worst-case, monitoring very long sequences of partial matches may be necessary. In general, network operators would like solutions that are robust under widest variety of often unforeseen operating conditions, although these are often dependent on uncertainties that are beyond the control of the operator.

In this work, we design a provably robust pipelined memory system that performs well under all operating conditions, including adversarial ones. We provide a novel mathematical model to analyze the overflow probability of such a system using the combination of convex order and large deviation theory. We establish a bound on the overflow probability and identify the worst-case memory access patterns. To our best knowledge, it is the only proven bound on a memory system for any admissible and even adversarial memory access patterns. The overflow probability for any real-world Internet traffic is always upper bounded by the bounds predicted in our model.

4.3 Ideal SRAM Emulation

In this chapter, we aim to design a DRAM-based memory architecture that can emulate a fast SRAM by exploiting memory interleaving. Suppose that a new memory access request is generated by each arriving packet. We define a cycle to be the minimum time it takes for a new packet to arrive at the router linecard. E.g. at line rate 40 Gb/s (OC-768) with a minimum packet size of 40 bytes, a cycle is 8 ns. A larger packet takes multiple cycles to arrive and thus generates a memory access request at a lower rate. We assume that it take a DRAM bank $1/\mu$ cycles to finish one memory transaction. We use $B > 1/\mu$ DRAM banks and randomly distribute the memory requests across these DRAM banks so that when the loads of these memory banks are perfectly balanced, the load factor of any DRAM bank is $\frac{1}{B\mu} < 1$. We also assume that an ideal SRAM can complete a read or write operation in the same cycle that the operation is issued. An SRAM emulation that mimics the behavior of an ideal SRAM is defined as follows:

Definition 7 (Emulation)**.** *A memory system is said to* emulate *an ideal SRAM if under the same input sequence of reads and writes, it produces exactly the same output sequences of* read *results, except time-shifted by some fixed delay* Δ.

More specifically, our emulation guarantees the following semantics:

- *Fixed Pipeline Delay*: If a read operation is issued at time t to an emulated SRAM, the data is available from the memory controller at exactly time $h = t + \Delta$ (instead of in the same cycle), where h is the *completion time*.

- *Coherency*: The read operations output the same results as an ideal SRAM system, except for a fixed time-shift.

Figure 4.1 illustrates the concept of SRAM emulation. In Figure 4.1, a series of six read and write accesses to the same memory location are initiated at times 0, 1, 2, ..., 5 respectively. If the memory is SRAM, then the read accesses at times 1, 4, 5 should return values a, c, c respectively. With our SRAM emulation, exactly the same values a, c, c will be returned, albeit at times $1+\Delta$, $4+\Delta$, $5+\Delta$ respectively.

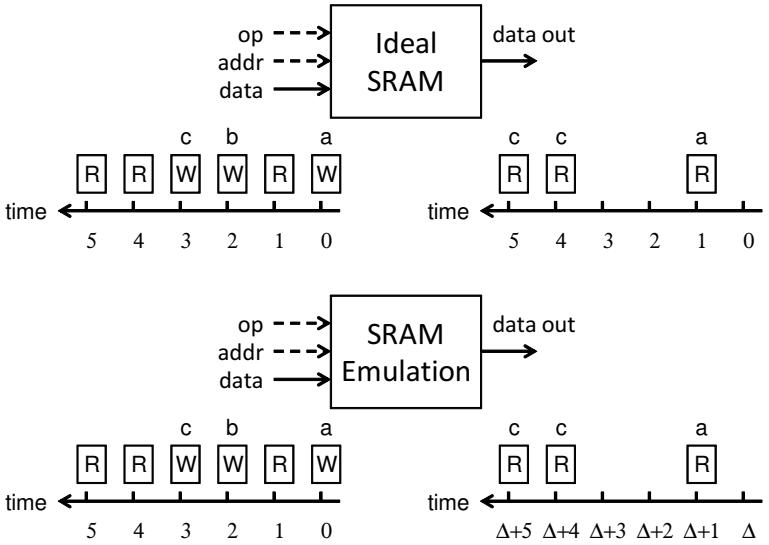


Figure 4.1: SRAM emulation of memory systems for network processing.

Note the definition of emulation does not require a specific completion time for a write operation, because nothing is returned from the memory system to the issuing processor. As to the read operations, the emulation definition only requires the output sequence (i.e. sequence of read results) to be the same, except time-shifted by a fixed Δ cycles. The definition does not require the *snapshot* of the memory contents to be the same. Therefore, the contents in the DRAM banks do not necessarily have to be the same as in an ideal SRAM at all times. For example, as we shall see later in our extended memory architecture, with the two back-to-back write operations in Figure 4.1,

only the data of the second write operation is required to update the memory to provide coherency, whereas both write operations correspond to actual memory updates in an ideal SRAM.

4.4 The Basic Memory Architecture

The basic memory architecture is shown in Figure 4.2. It consists of a reservation table, a set of DRAM banks with a corresponding set of request buffers, and a random address permutation function.

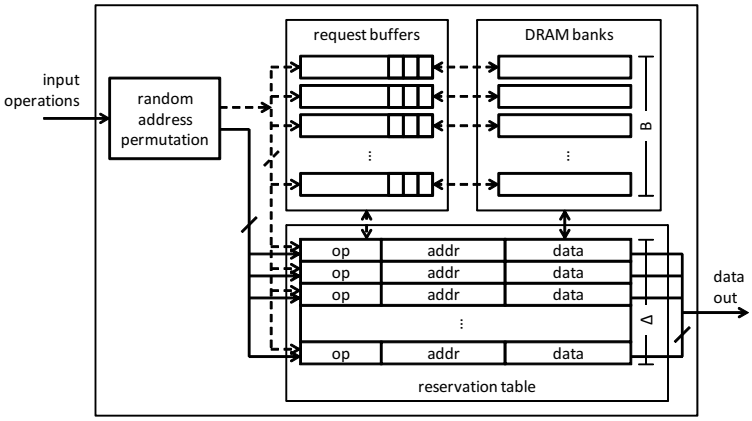


Figure 4.2: Basic memory architecture.

4.4.1 Architecture

- *Reservation Table*: The reservation table with Δ entries is implemented in SRAM to provide for a fixed delay of Δ for all incoming operations. For each operation arriving at time t, an entry (a row in Figure 4.2) is created in the reservation table at location $(t+\Delta) \mod \Delta$. Each reservation table entry consists of three parts: a 1-bit field to specify the operation (read or write), a memory address field, and a data field. The data field stores the data to be written to or to be retrieved from the memory.

- *DRAM Banks and Request Buffers*: There are $B > 1/\mu$ DRAM banks in the system to match the SRAM throughput. Also, there are B request buffers, one for each

DRAM bank. Memory operations to the same DRAM bank are queued at the same request buffer. Each request buffer entry is just a *pointer* to the corresponding reservation table entry, which contains the information about the memory operation. Given the random permutation of memory addresses, we can statistically size the request buffers to ensure an extremely low probability of overflow. We defer to Section 4.6 the discussion of this analysis. Let K be the size of a request buffer. Then the B request buffers can be implemented using multiple memory channels. In each channel the request buffers are serviced in a round-robin order and different channels are serviced concurrently. For each channel there is a dedicated link to the reservation table. For example, with $\mu = 1/16$ and $B = 32$ the request buffers can be implemented on two channels of 16 banks on each channel. Therefore, each request buffer takes $1/\mu$ cycles to be serviced. A memory operation queued at a request buffer would take at most $\Delta = K/\mu$ cycles to complete. We set the fixed pipeline delay in cycles for the external memory interface to this Δ.

- *Random Address Permutation Function*: The goal of the random address permutation function is to randomly distribute memory locations so that memory operations to different memory locations are uniformly spread over B DRAM banks with equal probability $1/B$. The function is a one-to-one repeatable mapping. Therefore the function is pseudorandom and deterministic. Unless otherwise noted, when referring to a memory address, we will be referring to the address after the random permutation. Note that if incoming operations access the same memory location, they will still generate entries to the same request buffer.

4.4.2 Operations

For a read operation issued at time t, its completion time is $h = t + \Delta$. A reservation table entry is created at location $h \mod \Delta$. The data field is initially pending. A DRAM read operation gets inserted into the corresponding request buffer. When a read operation at the head of a request buffer is serviced, which may be earlier than its completion time h, the corresponding data field of its reservation table entry gets updated. In

this way, the reservation table effectively serves as a *reorder* buffer. At the completion time h of the read operation, data from corresponding reservation table entry gets copied to the output. A reservation table entry is removed after Δ cycles.

For a write operation issued at time t, a reservation table entry is created at location $h \mod \Delta$, where $h = t + \Delta$. Also, a DRAM write operation is inserted into the corresponding request buffer. When the write operation at the head of a request buffer is serviced, which may be earlier than its completion time h, the write data is updated to the DRAM address. Its reservation table entry is removed Δ cycles after its arrival time t.

By sizing the reservation table to have Δ entries, the lifetime of a read or write reservation entry is guaranteed to be longer than the time it takes for the DRAM read or write to occur respectively.

4.4.3 Adversarial Access Patterns

Even though a random address permutation is applied, the memory loads to the DRAM banks may not be balanced due to some adversarial access patterns as follows. First, many applications require the lookup of global variables. Repeated lookups to a global variable will trigger repeated operations to the same DRAM bank location regardless of the random address permutation implemented. Second, although attackers cannot know how memory addresses are mapped to DRAM banks, they can still trigger repeated operations to the same DRAM bank by issuing memory operations to the same memory locations. Due to these adversarial access patterns, the number of pending operations in a request buffer may grow indefinitely. To mitigate these situations, our extended memory architecture in Section 4.5 effectively utilizes the reservation table as a cache to eliminate unnecessary DRAM operations queued in the request buffers.

4.5 The Extended Memory Architecture

Figure 4.3 depicts our proposed extended memory architecture. As with the basic architecture, there is a reservation table, a set of DRAM banks, and a corresponding set of request buffers. For each memory operation, its completion time is still exactly Δ

cycles away from the perspective of the issuing processor. However, in contrast to the basic memory architecture, we do not necessarily generate a new DRAM operation to the corresponding request buffer for every incoming memory operation. In particular, we can avoid generating new DRAM operations in many cases by using the reservation table effectively as a data cache. We will describe in more details in our *operation merging rules* later in this section.

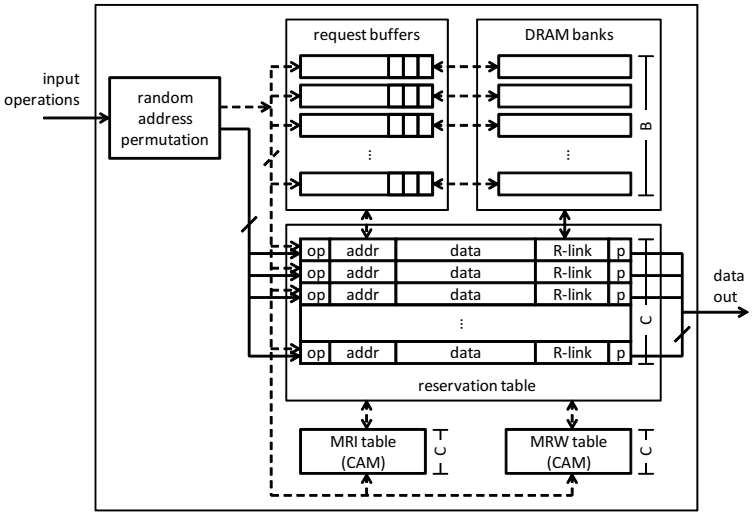


Figure 4.3: Extended memory architecture.

At a high-level, our goal is to *merge* operations that are issued to the same memory location within a window of C cycles. This is achieved by extending the basic architecture in the following ways. First, the size of the reservation table can be set to any $C \geq \Delta$ entries to catch mergeable operations that are issued at most C cycles apart. Second, each reservation table entry is expanded to contain the following information:

- *Pending Status p*: A memory read operation is said to be pending with $p = 1$ if there is a corresponding entry in the request buffer or if it is linked to an operation that is still pending. Otherwise, the read operation is non-pending with $p = 0$. The memory write operations in the reservation table are set as pending with $p = 1$.

- *R-link*: Pending read operations to the same memory address are linked together into a linked list using the R-link field, with the earliest/latest read operations at the head/tail of the list respectively.

In addition to the extra fields in the reservation table, two lookup tables are added to the extended architecture: a Most Recently Issued (MRI) lookup table and a Most Recent Write (MRW) lookup table. Both of these tables can be implemented using Content-Addressable-Memories (CAM) to enable direct lookup. The MRI table keeps track of the most recently issued operation (either read or write) to a memory address. When a new read operation arrives, an MRI lookup is performed to find the most recent operation in the reservation table. The MRW table keeps track of the most recent write operation to a memory address. When a write operation in the reservation table is removed, an MRW lookup is performed to check if there is another more recent write operation to the same memory address. For an MRI lookup, if there is a matched entry, it returns a pointer to the row in the reservation table that contains the most recent operation to the same memory address. Similarly, an MRW lookup returns a pointer to the row in the reservation table corresponding to the most recent write operation for a given memory address.

In the extended architecture, each entry in the request buffers needs a data field for write operations. For read operations in the request buffers, a request buffer entry serves as a pointer to the corresponding reservation table entry. But for write operations in the request buffers, a request buffer entry stores the actual data to be written to a DRAM bank.

4.5.1 Reservation Table Management

A reservation table entry becomes freed when the operation in the entry stays in the reservation table for C cycles. For an operation that arrived at time t, its reservation table entry will be freed at time $g = t + C$. To achieve this, we maintain two pointers: pointer h as a completion time pointer, and pointer g as a garbage collection time pointer. When the current time is g, the corresponding reservation table entry is freed, meaning the entry is cleared. On an MRI (or MRW) lookup, if it returns a reservation table entry with a different address (*addr*), then we also remove this MRI (or MRW) entry.

4.5.2 Memory Operations

When a read operation (R) arrives, a new entry in the reservation table is created for R. Then we have the following three cases:

- *Case 1*: If the MRI lookup fails, which means there is no previous operation to the same memory address in the reservation table, we then generate a new read operation to the corresponding request buffer. A new MRI entry is created for this operation also, and its pending status p is set to 1. When the actual DRAM read operation finishes, the data field of R in the reservation table entry is updated, and p is reset to 0.

- *Case 2*: The most recent operation to the same memory address returned by the MRI lookup is a pending read operation. In this case, we follow these steps:

  1. We create an R-link (shown as the dashed line in Figure 4.4) from this most recent read operation to R and set its pending status p to 1. We do not create a new read operation in the request buffers.
  2. The MRI entry is updated with the location of R in the reservation table.

  Essentially, a linked list of read operations to the same address is formed, with head/tail corresponding to the earliest/latest read, respectively. At the completion time of a read operation, read data is copied to the next read operation by following its R-link, and we reset p to 0. Therefore, only the first read operation in the linked list generates an actual read operation in the request buffer. The remaining read operations simply copy the data from the earlier read operation in the linked list order. An example of the management of R-links for read operations is shown in Figure 4.4.

- *Case 3*: The most recent operation to the same address returned by the MRI lookup is a write operation or a non-pending read operation. In this case, we copy the data from this most recent write or non-pending read operation to the new R location. In this way, R is guaranteed to return the most recent write or non-pending read data. The MRI entry is updated by pointing to this new operation.

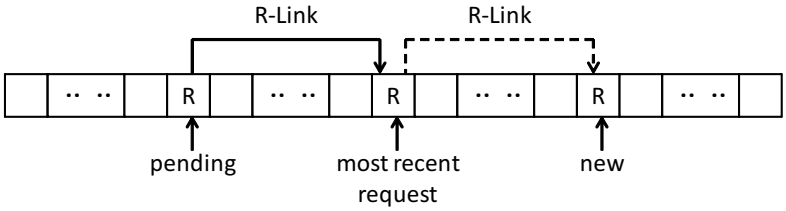


Figure 4.4: R-link for read operations.

When a read operation reaches its fixed delay Δ and departs from the system, its data is sent to the external data out. The corresponding reservation table entry is removed after C cycles. Also, if there is an entry in MRI that points to the reservation table entry, then this MRI entry is removed also.

When a new write operation (W) arrives, a new entry in the reservation table is created for W. Also, we have the following two cases:

- *Case 1*: If the MRI lookup fails, we create a new write operation in the corresponding request buffer, and we create a new MRW and MRI entry for this new operation.

- *Case 2*: If the MRI lookup returns an entry in the reservation table, an MRW lookup is also performed. If the MRW lookup returns an entry in the reservation table, then the corresponding entries in the MRI and MRW are updated by pointing to the new operation W in the reservation table. Otherwise, if the MRW lookup does not find an entry, then the corresponding entry in MRI is updated to W and a new entry is created in MRW for W.

A write operation W is removed from the reservation table after C cycles. An MRW lookup is performed to check if there is a more recent write operation in the reservation table. If the MRW lookup returns a different entry in the reservation table for the same address, no request buffer entry will be created for W. Otherwise, there is no other write operation to the same address in the reservation table, and the MRW lookup only returns the address of W. Then a new entry in the request buffer is generated for W to update the corresponding DRAM bank location. Also, the entry in MRW pointing to W is deleted.

To guarantee that the reservation table can provide a cache of size C instead of $C-1$, as will be shown in Section 4.5.3, a new operation arriving at the reservation table will first check against the address of the expiring reservation table entry before rewriting it. For a new read operation, if the reservation table entry location it will occupy contains the same address field, then the data field is preserved and reused by the new read operation. For a new write operation, if the reservation table entry it shall occupy is a write operation of the same address, then the expiring write operation will simply be discarded without being inserted into the request queues. Both processes described above can be easily supported by requiring the reservation table to read its expiring entry first and then update it with the new memory operation.

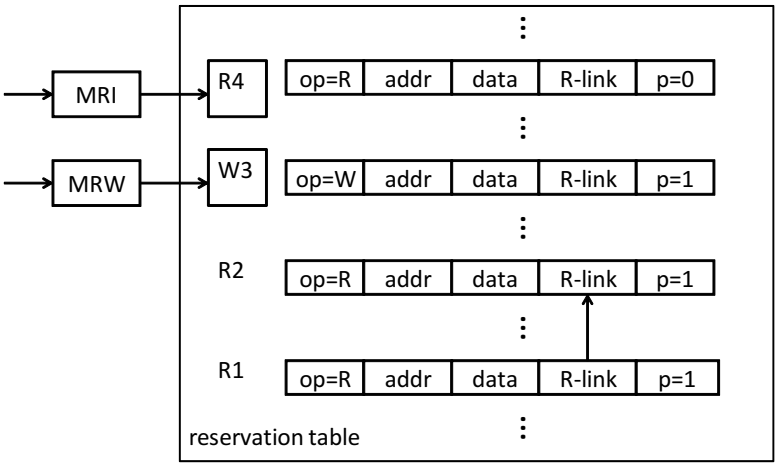


Figure 4.5: Reservation table snapshot.

A snapshot of the reservation table entries accessing the same DRAM address is shown in Figure 4.5. The memory operations are read (R1), read (R2), write (W3), and read (R4), with R4 being the most recent operation. The R-link of the pending read operation R1 points to the second read R2. Since the most up-to-date data is available in W3, the data is copied to the read operation R4 directly. Therefore, only R1 and W3 generate entries in the request buffers. Operations R2 and R4 are never inserted into the request buffers.

4.5.3 Operation Merging Rules

Based on the preceding descriptions of the read and write operations, we effectively are performing the following merging rules to avoid introducing unnecessary DRAM accesses. The operations arrive in the order from right to left on the left-hand-side of the arrows. The actual memory operation(s) generated in the request buffers are shown on the right-hand-side of the arrows.

1. RW $\rightarrow$ W: The R operation copies data directly from the W operation stored in the reservation table, thus it is not inserted into the request buffers.

2. WW $\rightarrow$ W: The earlier (right) W operation does not generate a new entry in the request buffers.

3. RR $\rightarrow$ R: The latter (left) R operation is linked to the earlier R operation by its R-link. No entry is created in the request buffer for the latter R operation.

4. WR $\rightarrow$ WR: Both the R operation and the W operation have to access the DRAM banks. They can not be merged. Entries in the request buffer need to be created for both W and R operations.

For the last rule, WR $\rightarrow$ WR, even though the operations cannot be merged, the future incoming operation to the same memory address can be merged. Consider the following two cases (inputs are sequenced from right to left):

- (RW)R $\rightarrow$ (W)R: The last two (RW) are merged to (W) using the RW $\rightarrow$ W rule.

- (WW)R $\rightarrow$ (W)R: The last two (WW) are merged to (W) using the WW $\rightarrow$ W rule.

With the above merging rules, data coherency is maintained. Since the reservation table has already provided a fixed delay Δ for all the read operations, to show data coherency, we need to prove that all the read operations will output the correct data. We will prove this by showing in the following that the relative ordering of the read and write operations are preserved in our architecture, which means that the read operations output the same data as in an SRAM system. In particular, we focus on a write operation

and show that operation merging rules will not affect the read operations before or after the write operation. Consider the following cases:

1. A newly arrived write operation W will not affect the data retrieved by the read operations before it, since the write operation W will not be inserted into the request buffer until C cycles later, by which time all the read operations before W have already finished.

2. For read operations arriving after a write operation W in the reservation table, with no other writes in between, they will read the correct data directly from W.

3. For read operations arriving after a write operation W in the reservation table, with other writes in between, they will read the correct data from the most recent write operation W'.

4. A write operation W is removed from the reservation table C cycles after it arrives. Upon removal, if there is another most recent write operation W' in the reservation table, the newly arrived read operations will read the correct data from W'. Therefore, the removal of W will not affect the data of future read operations.

5. Upon the removal of a write operation W in the reservation table, if there is no other more recent write operation, but there is a most recent read operation R, an entry is generated in the request buffer for W to update the data in the corresponding DRAM bank. According to the memory operations described in Section 4.5.2, R copies the data directly from W since R arrives later than W. All future incoming read operations will read the correct data from R directly or from the DRAM bank. Since W is already in the request buffer, future incoming read operations will generate entries in the request buffer only after W, and thus they will read the correct data from the memory banks.

6. Upon the removal of a write operation W in the reservation table if there are no other operations accessing the same DRAM location in the reservation table, then an entry is generated in the request buffer for W to update the memory banks. All the future read operations will only generate entries in the request buffer after W, thus they will retrieve the correct data.

In summary, the read operations always return the correct data with a fixed delay Δ, therefore the system is data coherent.

Further, using the proposed merging rules, we can ensure the following properties:

- There can be only one write operation in a request buffer every C cycles to a particular memory address. A write operation is generated in the request buffer only when there is a write operation in the reservation table for C cycles and there are no more recent write operations in the reservation table during this period of time. Thus a write operation is generated in the request buffer at most once every C cycles.

- There can be at most one read operation in a request buffer every C cycles to a particular address. When there is more than one arriving read operation to the same address within C cycles, we are guaranteed that all except the first read are merged using R-links.

- There can be at most one read operation followed by one write operation in a request buffer every C cycles to a particular address. If there is a read operation following a write operation, then the read operation can get data directly from the write operation. In this case, no new entry will be generated in the request buffer for the read operation.

4.5.4 Arrivals to Request Buffers

In this section we further analyze the arrival patterns to one request buffer. In particular, we will present the worst case arrival pattern to a request buffer in one cycle.

Essentially, all the write operations are delayed in our memory system by C cycles to be inserted in to a request buffer if they are not merged with other write operations. This doesn't violate our fixed delay Δ, since we only provide the fixed delay Δ for read but not write operations. For a read operation, if it is not merged in the reservation table, a new entry in the request buffer is created immediately upon its arrival. On the other hand, the return data of the read operation will only leave the system after Δ cycles. Therefore, even though in our memory system there is at most one arriving

read or write operation every cycle, it may happen in the request buffer that more than one entry is created in a cycle, or no entry is created at all in a cycle. Consider that the reservation table entries are occupied by pending write operations to different addresses in the same memory bank, and the arrivals are read operations to a new set of different addresses in the same memory bank. In this case, in each cycle, a new read operation entry needs to be created in the request buffer. Also, a write operation which has been in the reservation table for C cycles will be removed and thus it generates a new entry in the same request buffer. Therefore, in one cycle at most two new entries can be created in the request buffer, with one entry for the read operation and the other for the write operation. Since the read and write operations are to different memory locations, they are mapped to the same memory bank with probability $1/B$ due to the random address permutation function, where B is the total number of request buffers. In the worst case a specific request buffer experiences a bursty arrival of two requests per cycle with probability of $1/B$. Such a bursty arrival pattern lasts at most C cycles, which is the maximum number of write requests stored in the reservation table.

It is worth noting that since there is at most one arrival read or write operation to our memory system every cycle, on average there is strictly less than one entry created in the request buffers every cycle due to our operation merging rules. Also, in the system we use extra memory banks to match the SRAM throughput, i.e. $B > 1/\mu$. So in the long run, our system is stable, since we have more departures than arrivals in the request buffers. However, in a short period of time, a request buffer may still overflow due to certain bursty arrivals. In the next section, we will present probability bounds on such overflow events under the worst case memory access patterns.

4.6 Performance Analysis

In this section, we prove the main theoretical result of this chapter, which bounds the probability that any of the request buffers will overflow for all arrival sequences, including adversarial sequences of read/write operations. We use the same techniques as in Section 2.4, with the system overflow probability defined as in inequalities (2.1) and (2.2).

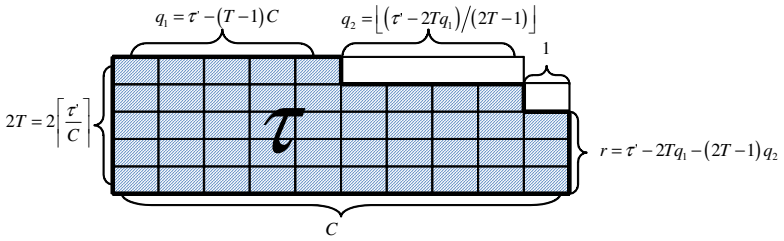


Figure 4.6: Relationship of q_1, q_2, r, T and τ'.

4.6.1 Worst-Case Parameter Setting

In this section we find the worst-case read/write operation sequence for deriving tail bounds for individual $\Pr[D_{s,t}]$ terms, where $\Pr[D_{s,t}]$ is the overflow probability for one request queue from time s to t as defined in inequality (2.2). Suppose ℓ distinct memory addresses are read or written back during time interval $[s,t]$, and the counts of their appearances are $m_1,...,m_\ell$, with $\sum_{i=1}^{\ell} m_i = \tau'$, where τ' is defined earlier as $\tau' = \tau + \min(\tau, C)$. We have,

$$X = \sum_{i=1}^{\ell} m_i X_i, \tag{4.1}$$

where X_i is the indicator random variable for whether the i^{th} address is mapped to the DRAM bank. We have

$$E[X_i] = \frac{1}{B}. \tag{4.2}$$

The X_i's are not independent since we are doing selection without replacement.

With the help of the reservation table, the read operations in the request buffers to the same DRAM address should not repeat within any sliding window of C cycles, and the write back operations in the request buffers to the same DRAM address should not repeat within any sliding window of C cycles. Therefore none of the counts $m_1,...,m_l$ can exceed $2T$, where $T = \lceil \frac{\tau'}{C} \rceil$. Moreover, let $q_1 = \tau' - (T-1)C$, then only q_1 addresses could have count $2T$. Figure 4.6 will help readers understand the relationship of q, r, T as functions of τ'.

We call any vector $m = \{m_1,...,m_\ell\}$ a valid splitting pattern of τ' if the following

conditions are satisfied.

$$\begin{cases} 0 < m_i \leq 2T, \\ \sum_{i=1}^{\ell} m_i = \tau', \\ |\{i : m_i = 2T\}| \leq q_1. \end{cases} \tag{4.3}$$

Let $\mathscr{M}$ be the set of all valid splitting patterns. [1] Let $X_m = \sum_{i=1}^{\ell} m_i X_i$. We note that if we reorder the coordinates in m, X_m still maintains the same distribution due to symmetry between the X_i's.

First we establish a partial order among $\mathscr{M}$.

Lemma 7. *Let $m = \{m_1, ..., m_\ell\}$ and $m' = \{m'_1, m'_2, m_3 ..., m_\ell\}$ differ only in the first two coordinates, and $0 \leq m'_1 \leq m_1 \leq m_2 \leq m'_2$ be such that $m_1 + m_2 = m'_1 + m'_2$. Then $X_m \leq_{cx} X_{m'}$.*

Proof. Let's consider the distribution of (X_1, X_2) conditioned on $X_3, ..., X_\ell$. Let $p_{x_1 x_2} = \Pr[X_1 = x_1, X_2 = x_2 | X_3, ..., X_\ell]$, and $c = m_3 X_3 + ... + m_l X_l$. We have $p_{01} = p_{10}$ due to symmetry of X_1 and X_2. So for any convex function $f(x)$,

$$\begin{aligned} & \mathrm{E}[f(X_m)|X_3, ..., X_l] \\ = \; & \mathrm{E}[f(m_1 X_1 + m_2 X_2 + c)|X_3, ..., X_l] \\ = \; & p_{00} f(c) + p_{10} f(c + m_1) + p_{01} f(c + m_2) \\ & + p_{11} f(c + m_1 + m_2) \\ \leq \; & p_{00} f(c) + p_{10} f(c + m'_1) + p_{01} f(c + m'_2) \\ & + p_{11} f(c + m'_1 + m'_2) \\ = \; & \mathrm{E}[f(X_{m'})|X_3, ..., X_l]. \end{aligned}$$

The inequality is due to Lemma 2 applied to the sequence $c + m'_1, c + m_1, c + m_2, c + m'_2$, and the fact that $p_{01} = p_{10}$ and $m'_1 + m'_2 = m_1 + m_2$.

Now applying the law of total expectation,

$$\begin{aligned} \mathrm{E}[f(X_m)] &= \mathrm{E}[\mathrm{E}[f(X_m|X_3, ..., X_l)]] \\ &\leq \mathrm{E}[\mathrm{E}[f(X_{m'}|X_3, ..., X_l)]] \\ &= \mathrm{E}[f(X_{m'})], \end{aligned}$$

[1] Not all valid splitting patterns may have plausible read/write sequences matching them. But this does not affect our bounds.

which proves the lemma. □

Let $q_1 = (\tau' - (T-1)C$, $q_2 = \lfloor(\tau' - 2Tq_1)/(2T-1)\rfloor$, $r = \tau' - 2Tq_1 - (2T-1)q_2$. Let m^* be such a splitting pattern: q_1 memory addresses are accessed $2T$ times, q_2 addresses are accessed $2T-1$ times, and one address is accessed r times. We have the following theorem:

Theorem 6. *m^* is the worst case splitting pattern in terms of convex ordering, i.e. $X_m \leq_{cx} X_{m^*}, \forall m \in \mathscr{M}$.*

Proof. We will first show that if m cannot be reordered to become m^*, there exists $m' \in \mathscr{M}$ such that $X_m \leq_{cx} X_{m'}$.

Suppose $m = \{m_1, ..., m_l\}$. If m has less than q_1 items with count $2T$, then it has at least two items with count less than $2T$, say m_1 and m_2. Let $m'_1 = m_1 - 1, m'_2 = m_2 + 1, m' = \{m'_1, m'_2, m_3, ..., m_l\}$. It's easy to see that m' has at most q_1 items with count $2T$, so $m' \in \mathscr{M}$. By Lemma 7, $X_m \leq_{cx} X_{m'}$. (If $m'_1 = 0$, we can now drop it from m'.).

If m has exactly q_1 items with count $2T$, it must have less than q_2 items with count $2T-1$, so it has at least two items with count less than $2T-1$. Using similar arguments as above, we show that there exists $m' \in \mathscr{M}$ such that $X_m \leq_{cx} X_{m'}$.

As we pointed out earlier, if we reorder the coordinates, X_m will maintain the same distribution. By repeatedly applying the above adjustment and reordering, we see that we can finally reach m^*. Since convex ordering is transitive, $X_m \leq_{cx} X_{m^*}, \forall m \in \mathscr{M}$. □

4.6.2 Request Queue Overflow Probability

For a memory system with worst case memory access patterns as shown in Theorem 6, the system overflow probability can be bounded by the following Theorem.

Theorem 7.

$$\text{For } \tau \le C,$$

$$\Pr[D_{s,t}] \le \min_{\theta>0} \frac{(\frac{1}{B}e^{2\theta} + (1-\frac{1}{B}))^{\tau}}{e^{(K+\mu\tau)\theta}}. \tag{4.4}$$

$$\text{For } \tau > C,$$

$$\Pr[D_{s,t}] \le \min_{\theta>0} \frac{1}{e^{(K+\mu\tau)\theta}} \cdot [\frac{1}{B}e^{2T\theta} + (1-\frac{1}{B})]^{q_1}$$
$$\times [\frac{1}{B}e^{(2T-1)\theta} + (1-\frac{1}{B})]^{q_2} \cdot [\frac{1}{B}e^{r\theta} + (1-\frac{1}{B}))]. \tag{4.5}$$

Proof. To prove the theorem, we can apply Lemma 4 and Theorem 3. We consider the two cases $\tau \le C$ (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.

The case $\tau \le C$

When $\tau \le C$, $X_{m^*} = X_1 + ... + X_\tau$. Let the population S consist of $c_1, ...c_N$ such that $\frac{N}{B}$ of c_i's are of value 2 and the rest of them are of value 0. If we let $n = \tau$, then X in Lemma 4 has the same distribution as our X_{m^*}, 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 $X_m \le_{cx} X_{m^*} \le_{cx} Y$ we get

$$\begin{aligned}
\mathrm{E}[e^{X_m\theta}] &\le \mathrm{E}[e^{X_{m^*}\theta}] \le \mathrm{E}[e^{Y\theta}] \\
&= \mathrm{E}[e^{(Y_1+...+Y_\tau)\theta}] \\
&= \mathrm{E}[e^{Y_1\theta}]^{\tau} \\
&= (\frac{1}{B}e^{2\theta} + (1-\frac{1}{B}))^{\tau}.
\end{aligned}$$

The case $\tau > C$

For $\tau > C$, we have $X_{m^*} = 2T(X_1 + ... + X_{q_1}) + (2T-1)(X_{q_1+1} + ... + X_{q_1+q_2}) + rX_{q_1+q_2+1}$. We can similarly apply Theorem 3 and get

$$
\begin{aligned}
&\mathrm{E}[e^{X_m\theta}] \le \mathrm{E}[e^{X_{m^*}\theta}] \le \mathrm{E}[e^{Y\theta}] \\
&= \mathrm{E}[e^{(2T(Y_1+\ldots+Y_{q_1})+(2T-1)(Y_{q_1+1}+\ldots+Y_{q_1+q_2})+rY_{q_1+q_2+1})\theta}] \\
&= \mathrm{E}[e^{2TY_1\theta}]^{q_1}\mathrm{E}[e^{(2T-1)Y_1\theta}]^{q_2}\mathrm{E}[e^{r\theta}] \\
&= [\frac{1}{B}e^{2T\theta}+(1-\frac{1}{B})]^{q_1}\cdot[\frac{1}{B}e^{(2T-1)\theta}+(1-\frac{1}{B})]^{q_2} \\
&\times[\frac{1}{B}e^{r\theta}+(1-\frac{1}{B})].
\end{aligned}
$$

Together with (2.4), we have Theorem 7. □

In conclusion, we have established the bound for the overflow probability under any read or write sequences. It can be computed through $O(n)$ number of numerical minimizations for one-dimensional functions expressed in Theorem 7.

4.7 Performance Evaluation

In this section, we present evaluation results for our proposed robust memory system in Section 4.5 using Matlab with parameters set according to the current DRAM and SRAM technology under the worst case memory access patterns that are identified in Section 4.6. In particular, we use parameters derived from two real-world Internet traffic traces, with one from University of Southern California (USC) and the other from University of North Carolina (UNC). 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 data storage for both traces, we set the number of addresses in the DRAM banks to be $N = 16$ million. The performance of our robust memory system on real traffic traces, including the two above, is guaranteed to be better than the bounds shown in this section.

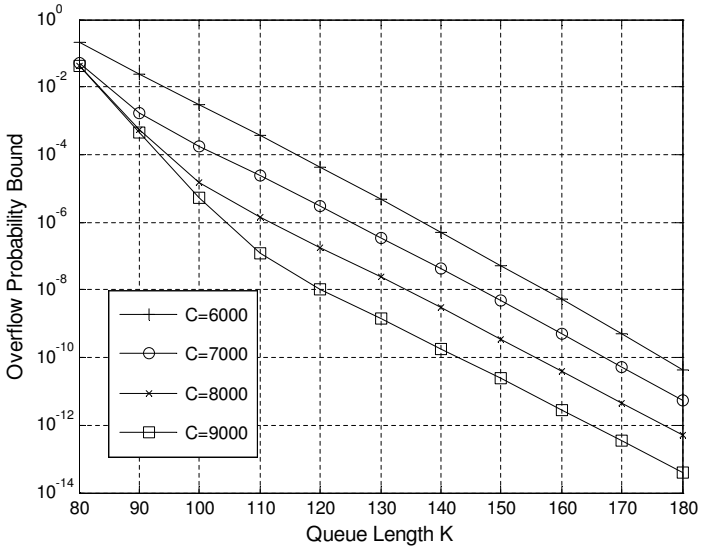


Figure 4.7: Overflow probability bound as a function of request buffer size K with $\mu = 1/10$ and $B = 32$.

4.7.1 Numerical Examples of the Tail Bounds

In Figure 4.7, the overflow probability bounds with different reservation table sizes C as a function of request buffer sizes K are presented, where $\mu = 1/10$ and $B = 32$. As K increases, the overflow probability bound decreases. With $C \geq 8000$ the overflow probability bound of 10^{-12} is achieved starting from a request buffer of size $K = 180$.

In Figure 4.8, the overflow probability bounds with different reservation table sizes C as a function of request buffer sizes K are presented, where $\mu = 1/10$ and $B = 64$. 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 process with arrival probability $1/B = 1/64$ to a queue of length K. By increasing the number of memory banks, with a moderate $C = 3000$ we can achieve an overflow probability bound of 10^{-30} with only a request buffer of size $K = 120$.

In Figure 4.9, the system overflow probability bounds with different numbers of memory banks B as a function of queue length K are presented, where $\mu = 1/10$ and $C = 8000$. This figure shows that by fixing K, the overflow probability bound decreases as B increases.

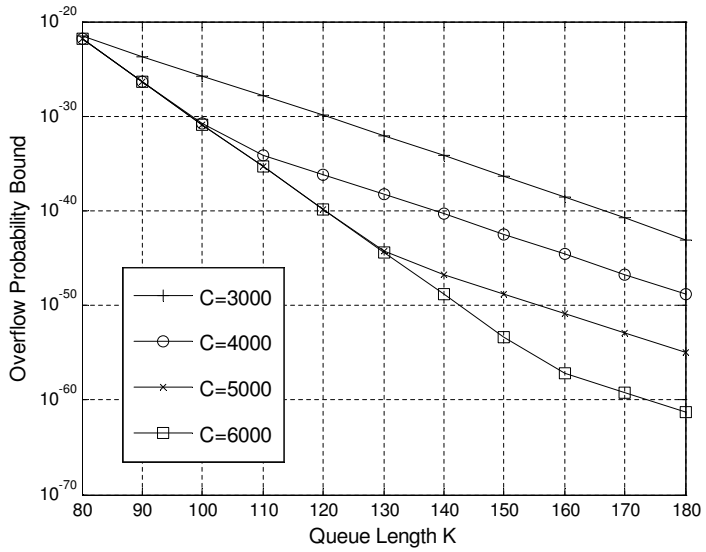


Figure 4.8: Overflow probability bound as a function of request buffer size K with $\mu = 1/10$ and $B = 64$.

In Figure 4.10, the system overflow probability bounds with different numbers of memory banks B as a function of queue length K are presented, where $\mu = 1/10$ and $C = 4000$. As expected, the overflow probability bound is greatly reduced by using more memory banks or larger request buffers.

Now let's consider the size of the reservation table. In each of its entries, one bit is used for *op* to distinguish read and write operations. With $N = 16$ millions entries in the DRAM banks, *addr* of size $\log_2 N = 24$ bits is sufficient to address every memory location. The size of the *R-link* is $\log_2 C = 13$ bits, with $C = 8000$. The pending status p takes 1 bit. Let the data size be 40 bytes or 320 bits. Altogether the total size of an entry in the reservation table is 359 bits. With $C = 8000$ entries, its total size is about 351 KB, which can easily fit in an SRAM.

For the MRI and MRW tables, only pointers are stored to enable fast searching on the reservation table entries. Each entry in the MRI or MRW table is of size $\log_2 N = 24$ bits, where N is the number of addresses in the DRAM banks. There can be at most C entries in the MRI or MRW, where C is the size of the reservation table. With $C = 8000$, the total size of MRI or MRW is about 24 KB, which can be easily implement in CAMs.

In the request buffers, each entry is a pointer to an entry in the reservation table

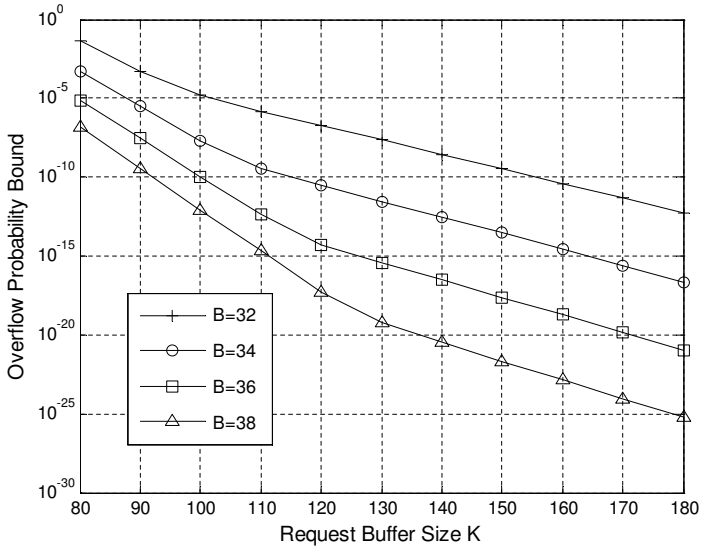


Figure 4.9: Overflow probability bound as a function of number of memory banks B with $\mu = 1/10$ and $C = 8000$.

plus a data field for a write operation. A pointer in the request buffers is of size $\log_2 C$. For $C = 8000$, the size of the pointer in the request buffer is 13 bits. Let the size of the data field be 8 bytes. Let $B = 32$ and $K = 180$ to provide with 10^{-12} overflow probability, the total size of the request buffers is only about 55 KB.

It is worth noting that these evaluations are based on the assumption of the worst case scenarios where the requests to the same memory locations are repeated every C cycles. For real-world traffic, this assumption is far too pessimistic. We expect that much smaller request buffers (K) and a much smaller reservation table size (C) will be sufficient for most real-world Internet traffic. This will result in a much smaller delay Δ, where $\Delta = K/\mu$.

4.7.2 Cost-Benefit Comparison

Table 4.1 compares our proposed approach with a naïve SRAM approach as well as with the VPNM architecture approach [1]. The comparison is for $\mu = 1/10$, which is the case for the XDR memory [78, 101]. We assume that there are 16 million entries in the DRAM banks with each entry 40 bytes wide. The naïve SRAM approach

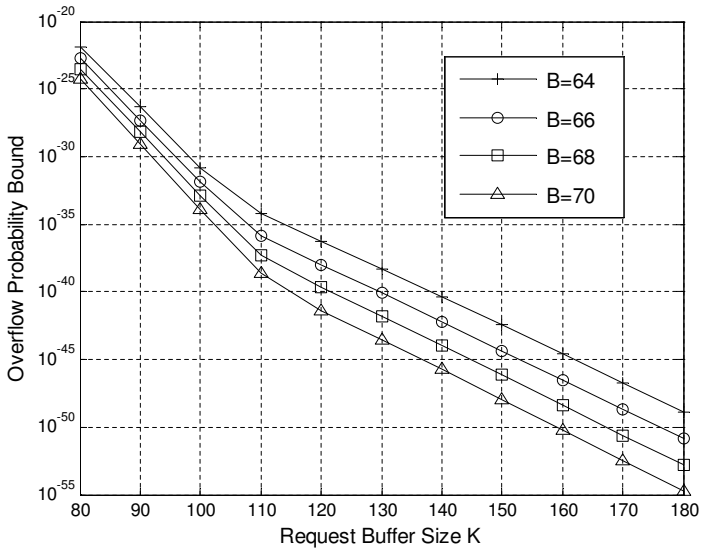


Figure 4.10: Overflow probability bound as a function of number of memory banks B with $\mu = 1/10$ and $C = 4000$.

Table 4.1: Comparison of different schemes for a reference configuration with 16 million addresses of 40 bytes data, $\mu = 1/10$, $B = 32$, and $C = 8000$.

	naïve	VPNM [1]	ours
DRAM	none	640 MB	640 MB
SRAM	640 MB	200 KB	451 KB
CAM	none	176 KB	48 KB
worst case overflow probability bound	0	no guarantee	$< 10^{-12}$

simply implements all entries in SRAM. In the VPNM, the delay storage buffer has to be implemented in CAM in order to eliminate redundant memory accesses. Our new robust pipelined memory uses less CAM than the VPNM and a comparable amount of SRAM. However, with the parameter setting, the overflow probability of our robust pipelined memory is bounded by 10^{-12}, while there is no such guarantee using the VPNM.

4.8 Conclusion

We proposed a memory architecture for high-end Internet routers that can effectively maintain wirespeed read/write operations by exploiting advanced architecture features that are readily available in modern commodity DRAM architectures. In particular, we presented an extended memory architecture that can harness the performance of modern commodity DRAM offerings by interleaving memory operations to multiple memory banks. In contrast to prior interleaved memory solutions, our design is robust to adversarial memory access patterns, such as repetitive read/write operations to the same memory locations, by using only a small amount of SRAM and CAM. We presented a rigorous theoretical analysis of the performance of the proposed architecture in the worst-case using a novel combination of convex ordering and large deviation theory. Our architecture supports arbitrary read and write patterns at a wirespeed of 40 Gb/s or beyond.

Chapter 4, in part, is a reprint of the material as it appears in the following publications:

- Hao Wang, Haiquan (Chuck) Zhao, Bill Lin, and Jun (Jim) Xu, "Robust Pipelined Memory System with Worst Case Performance Guarantee", *IEEE Transactions on Computers (TC)*, 2011.

- Hao Wang, Haiquan (Chuck) Zhao, Bill Lin, and Jun (Jim) Xu, "Design and Analysis of a Robust Pipelined Memory System", *IEEE International Conference on Computer Communications (INFOCOM)*, San Diego, CA, March 15-19, 2010.

The dissertation author was the primary investigator and author of the papers.

Chapter 5

Priority Queues for High Speed Scheduling

5.1 Introduction

For advanced per-flow service disciplines at high-speed network links, it is essential to maintain priority values in sorted order. A single queue system cannot satisfy the service requirements since it only supports a first-come-first-served policy. In order to achieve differentiated service guarantees, per-flow queueing is necessarily applied in advanced high-performance routers to manage packets for different flows in separate logical queues [54]. Different flows may have different QoS requirements that correspond to different data rates, end-to-end latencies, and packet loss rates, thus they are assigned different priority values accordingly. When multiple service classes are also supported, packets in a flow are further split into separate queues that correspond to the different service classes. Scheduling algorithms are required to decide the order of service for per-flow queues at line rate. A number of advanced scheduling techniques have been proposed, such as weighted fair queueing (WFQ) [21], worst-case fair weighted fair queueing (WF^2Q) [8], and deficit round-robin (DRR) [88], all of which approximate the performance of the general processor sharing policy (GPS) [72]. They differ in the way the service time of a flow is updated after being serviced, and in the way the service time of a new active flow is determined. A good review of scheduling algorithms

can be found in [105]. Most of these advanced scheduling techniques are based on assigning *deadline timestamps* to packets, depending on their urgency, and servicing flows in the *earliest-deadline-first* order. This earliest-deadline-first scheduling approach can be facilitated using a priority queue [10, 12, 42, 64, 97, 98] with index keys that sort flows in increasing deadlines, where the deadlines correspond to keys in the priority queue.

The scalable priority queue implementation requires managing a large number of queues. On the Internet, there can be tens of thousands of concurrent flows at any time [90]. Furthermore, each of these flows may contain packets belonging to different service classes. Both the different flows and different classes may have to be stored in separate queues in order to meet QoS requirements. Therefore, to provide for fine-grained per-flow queueing and scheduling, hundreds of thousands of queues may need to be maintained. Hierarchical scheduling [9] is commonly applied to aggregate flows into a number of flow aggregates to ease the design. To schedule a flow to service, a flow aggregate is first picked, and then a flow within the aggregate is serviced. However, in some levels of the hierarchy there can still be several thousands of flows, while others contain only a few flows. It is reasonable to expect the number of flows in an aggregate to keep growing as more fine-grained QoS is promised.

The scalable priority queue implementation also requires the ability to store a huge number of packets at ever-increasing line rates. For example, various network processing applications require the management of memory banks to buffer packets. Especially in high-performance routers, a large number of packets need to be temporarily stored at each linecard in response to congestion in the network. One of the critical challenges is to keep up the speed of the memory banks to match the ever-increasing line rates and at the same time be able to store an ever-increasing number of packets. In any time slot, one arrival packet may need to be written to the buffer and one packet in the buffer may need to be read for departure. To provide sufficient buffer at the time of congestions, a widely used buffer sizing rule is to size the packet buffer to be the product of the average round-trip-time (RTT) and the line rate [96]. While some research shows that this rule is an overestimate on the total buffer size [5] under certain traffic assumptions, results such as [22] show that even larger buffers may be required under other traffic patterns. For a linecard operating at a line rate 40 Gb/s (OC-768) with

RTT = 250 ms [13], the buffer sizing rule translates to a buffer of size 10 Gb. With the minimum TCP packet size of 40 bytes, a new packet may be inserted into the buffer and a buffered packet may be removed every 8 ns. Thus priority queues have to update the corresponding priority keys within this time frame. During a round-trip-time, there can be tens of thousands of active flows sending packets to the packet buffer, with each flow containing packets of several classes. So hundreds of thousands of queues may need to be maintained to provide for per-flow queueing.

In this chapter, we propose novel solutions to solve the scalable priority queue implementation problem by decomposing it into two parts, a succinct priority indexing module in SRAM that can efficiently maintain the real-time sorting of priority keys/indexes, coupled with an architecture to manage a large number of priority queues at line speeds using the succinct priority indexing module. In particular, we propose three related novel succinct priority index data structures for implementing high-speed priority indexes: a Priority-Index (PI), a Counting-Priority-Index (CPI), and a pipelined Counting-Priority-Index (pCPI). We show that all three structures can be compactly implemented in SRAM using only $\Theta(U)$ space, where U is the size of the universe required to implement the priority keys (deadline timestamps). Compared to other pipelined priority indexes [10,42,97,98], the pCPI has the advantage that it can be implemented very efficiently by leveraging hardware-optimized instructions that are readily available in modern 64-bit microprocessors. Moreover, operations on the pCPI structure take constant time using only $\Theta(\log_{64} U)$ pipeline stages, which makes it suitable for the maintenance of a large number of priority queues at line rate. For example, to maintain 16 million indexes for 256 K priority queues, a total of 4 pipeline stages would be sufficient. The solutions to the problems of mapping packet departure times into priority indexes, as well as the problem of locating the priority queue that contains the packet with any departure time represented by a priority index are provided in this chapter. Therefore, the pCPI can be integrated as an essential part of the packet scheduler. Finally, we show that our proposed priority index structures can be combined with a DRAM-based architecture to provide scalable storage for large packet buffers at line speeds. Our memory management techniques are motivated by the hybrid SRAM/DRAM FIFO memory architectures in [43,89,109]. In particular, in [43,89] management of packet buffers using

DRAM banks with SRAM queues are developed without the detailed implementation of the packet scheduler. In [109], the authors proposed a hybrid SRAM/DRAM priority queue system using a pipelined heap as scheduler. In our approach, we implement our pCPI as an essential unit for scheduling packets for per-flow queueing systems.

The remainder of this chapter is organized as follows. In Section 5.2, we review related work. In Section 5.3, we illustrate a scheduling algorithm for per-flow queues. In Section 5.4, we introduce a high-level abstraction for a succinct priority indexing structure. In Section 5.5, we describe an extended data structure called Counting-Priority-Index (CPI) that supports top-down index operations. In Section 5.6, we show a pipelined version of the succinct priority indexing structure, which is referred to as a pipelined Counting-Priority-Index (pCPI). In Section 5.7, we present the implementation of pCPI for per-flow queueing. In Section 5.8, we discuss the application of per-flow queue management using pCPI for large packet buffers. In Section 5.9, we evaluate the complexity of our per-flow queue management solutions. Finally, we conclude in Section 5.10.

5.2 Related Work

For small priority queues with at most tens to hundreds of entries, specialized shift-register architectures or systolic-array architectures may be used [18,64], but these architectures are inherently not scalable. Calendar queues [12] are a viable alternative for priority queues with thousands of entries, but this approach remains too expensive and does not provide deterministic response times due to hashing.

For these reasons, in the advanced QoS scheduling literature, often a binary heap data structure is assumed for the implementation of priority queues, which is known to have $\Theta(\log_2 n)$ time complexity for heap operations, where n is the number of heap elements. The heap data structure only requires $\Theta(1)$ processing hardware versus $\Theta(n)$ processing hardware assumed in architectures such as comparator-tree or systolic-array architectures. However, to support high-performance network processing applications, e.g. advanced QoS scheduling on a 40 Gb/s (OC-768) link and beyond, the algorithmic complexity of conventional heap operations is not fast enough for large priority queues

when n is large.

To overcome the size-dependent algorithmic complexity of conventional heaps, pipelined heaps have been proposed [10, 42, 97] that can achieve $\Theta(1)$ amortized time complexity. This constant time complexity makes it possible to support network processing applications like advanced QoS scheduling at extremely high-speeds. However, these pipelined heap structures have a hardware complexity of $\Theta(\log_2 n)$ pipelined stages, where n is the number of priority keys. This hardware complexity can be substantial for a large n. For example, with $n = 256$ K per-flow queues, 18 pipelined stages are required. To reduce the number of pipelined stages requirement, the pipelined van Emde Boas tree [98] based on the van Emde Boas tree data structure [94] has been proposed. It achieves amortized constant time complexity with hardware complexity of a total $\Theta(\log_2 \log_2 U)$ pipelined stages, where U is the size of the universe representing the maximum range of the priority keys to be considered. Even with a universe of size $U = 16$ million , only 5 pipelined stages are required, which is a substantial reduction from the pipelined heap structures. However, compared to our proposed priority indexing structures in this chapter, the pipelined van Emde Boas tree requires considerably more complicated operations and larger storage. Also, our priority indexing structures are faster to implement thanks to the hardware-optimized instructions provided by the modern microprocessors. At different levels of a pipelined van Emde Boas tree, the nodes are of different sizes. So operations such as finding the first value 1 bit in a node takes a linear time proportional to the size of the node, which greatly reduces the speed of the pipelined operations.

5.3 Scheduling in Per-flow Queueing

As shown in the previous section, per-flow queueing is required to provide satisfying QoS for future networks. The management of per-flow queues demands an effective scheduler. Priority queues are effective data structures to compute and update schedules at line rate, and to minimize service time jitters. In this section, we review scheduling algorithms using priority queues as essential elements. Specifically, we consider a weighted round robin scheduler as an example.

5.3.1 Scheduling Algorithm

A weighted round robin (WRR) scheduler is shown in Figure 5.1. The scheduler handles a total of five flows, which are labeled from letter A to E. Among these flows, flow C is empty and thus not eligible for service. The relative weight factors for different flows are shown in the figure. For simplicity, we assume all the packets in the per-flow queues are of the same size, so the service intervals for different flows are inversely proportional to their weight factors.

A naïve WRR scheduler may generate bursty service even under low traffic load. It may service the flows in a round robin order by serving a number of packets proportional to the flow's weight factor on each visit. Such a service pattern leads to burstiness for packets from heavy flows, which would cause undesirably large service time jitters.

Instead, a more effective WRR scheduler services different flows one packet at a time. The scheduler maintains the service times of the first packet in each flow. As shown in Figure 5.1, flow A has service time at $t = 7$, flow B at $t = 18$, flow D at $t = 10$, and flow E at $t = 15$. The scheduler uses a virtual clock to schedule service times for flows. Let's assume that the flow of weight factor 1 (flow E) has a service interval of 100 cycles. The service intervals of the other flows can be calculated according to the inversely proportional relationship of the service intervals and the weight factors. The results are shown in Figure 5.1. First, the virtual clock is advanced to $t = 7$ and the first packet in flow A is serviced. Since flow A has a service interval of 10, its second packet becomes the head of flow A which is of service time $7 + 10 = 17$. Then the virtual clock advances to time 10 and services flow D which has the next earliest service time. In summary, the scheduler maintains the weight factors for the different flows. Only the service time of the first packet in a flow is calculated and stored by the scheduler. The service times of the other packets in a flow are only calculated based on the service time of the corresponding previous packet and the service interval for the flow. The calculation is done only when a packet becomes the head of the flow, thereby eliminating the storage requirement for maintaining the service times for all the packets in a flow explicitly. At each cycle, the flow with the earliest service time is picked by the scheduler for service from all per-flow queues.

The adoption of a virtual clock enables the scheduler to be both work-conserving

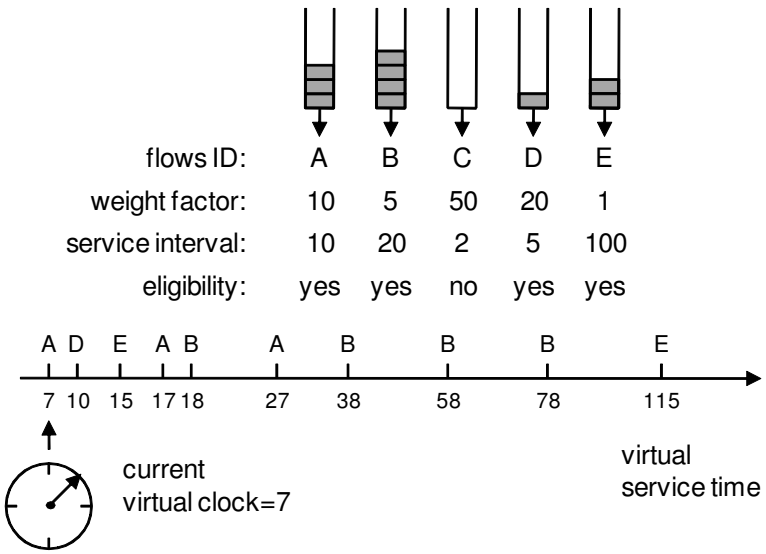


Figure 5.1: Example of weighted round robin (WRR) service schedules.

and less computationally intensive. With the help of a virtual clock, the next flow is always serviced immediately by advancing the virtual clock to the flow service time. As long as there is an active flow in the system, the scheduler can always schedule a packet. If a real clock is used instead, the scheduler has to wait until it reaches the service time of the next earliest flow to service a packet, which reduces the overall throughput. Moreover, the modeling of real service time is computationally intensive. The real service time for different flows has to be recomputed every time there is a packet arrival to any flow or a packet departure from any flow. By using a virtual clock, only the ordering of the service times are accurately maintained. The real service time is no longer relevant, since the scheduler advances the virtual clock to service the next flow no matter how far into the future that time is.

5.3.2 Per-flow Queueing Operations

In a scheduler, there is a crucial operation which locates the next earliest service time among all of the flows, which is commonly supported by priority queues. To effectively manage per-flow queues, a priority queue must at least support the operations as follows:

- *Insert Priority:* When the service time of a new packet is calculated, a new entry

has to be inserted into the priority queue for the packet in order to schedule it for service. The priority of a packet is directly related to its service time.

- *Extract Minimum Priority:* A priority queue supports the operation of finding and removing the entry with the minimum priority, which corresponds to the earliest departure time of all packets. The packet will then be scheduled for service at its service time by the scheduler.

In this chapter, the priority queue is implemented using a pipelined Counting Priority Index, which represents a priority key using a one bit index. To support the operations required for per-flow queueing, we develop the mapping from the departure timestamp of a packet to a priority index for priority insertions. To support the operations of extracting minimum priority, we present the mapping from a priority index back to a departure timestamp, and further to a per-flow queue.

5.4 Succinct Priority Indexing Abstraction

In this section, we outline the high-level abstraction of a priority queue implementation that we call a succinct priority indexing structure. Compared to other types of priority queues, it takes the advantage of fast instructions provided by modern processors. Although the priority indexing structure is presented in the context of advanced per-flow scheduling, we note that it can also be applied to other networking problems, such as statistics counting [87, 107], which also relies on priority queues as a fundamental building block.

5.4.1 Abstraction

The basic structure of the abstraction, which we call a Priority-Index (PI), is a perfect W-way tree. It exploits built-in hardware instructions that are readily available in modern x86 processors to achieve fast indexing speed. Suppose we have a fixed universe of size $U = W^h$, and we wish to represent a set S of N elements from this universe, where $N \leq U$. This basic abstraction can be viewed as a bitmap that records the elements of the universe in the set S. Each element i of the universe is associated with a binary bit

b_i in the bitmap. The bit b_i is set to 1 if $i \in S$, and 0 otherwise. In a PI structure, a leaf node contains a total of W bits, with each bit representing an element in the universe. For a non-leaf node there are also W bits, with each bit associated with a corresponding child node. The bits in the non-leaf nodes serve as *summaries* of their child nodes, i.e., a bit in a non-leaf node is set to 1 if there is at least one non-zero bit in its child node, or 0 otherwise. A PI of 3 levels is shown in Figure 5.2. With each node of size $W = 64$ bits, it can represent a set from a universe of size $U = 2^{18}$.

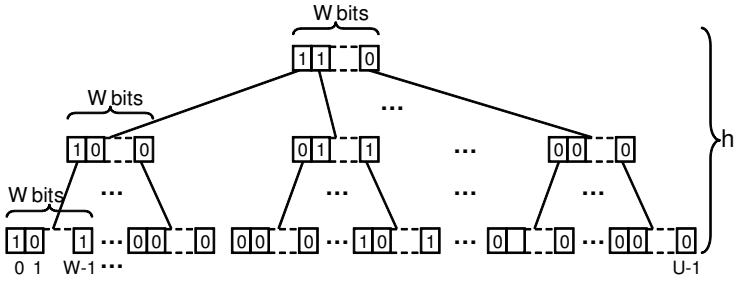


Figure 5.2: Data structure of a PI with $h = 3$ levels.

5.4.2 Index Update Operations

The succinct priority index abstraction supports the operations shown in Table 5.1. The advantage of a succinct priority index over pipelined heap and pipeline van Emde Boas tree structures is its fast index update operations supported by modern processors. Modern 64-bit x86 processors (from both Intel and AMD [3, 39]) all have built-in instructions to compute the position of the *most-significant-bit* of value 1 in a 64-bit word using `BSR` (bit-scan-reverse) or *least-significant-bit* of value 1 in a 64-bit word using `BSF` (bit-scan-forward) as a single step operation. If there is no such bit, the `BSR` and `BSF` operations will set a flag `ZF` (zero flag) to 1. In addition, AMD also has a `LZCNT` (leading-zero-count) instruction [3] that returns the number of leading 0's in a word. The position of the most-significant-bit of value 1 in a word is just `LZCNT` $+ 1$, unless `LZCNT` is 64, in which case all bits are 0's and there is no value 1 bit in the word. Therefore, the position of the first value 1 bit in a word from right or left can be located by one instruction. We will show later that such constant time operations enable our priority indexing structures to support fast line rate updates.

Table 5.1: Operations supported by a succinct priority indexing structure.

`insert(i)`	Insert a new index i to S
`delete(i)`	Delete index i from S
`findmin`	Find the smallest index in S
`extractmin`	Retrieve and remove the smallest index in S
`successor(i)`	Find the successor of index i in S
`extractsucc(i)`	Retrieve and remove the successor of index i in S

For advance per-flow queue management, the architecture supports the operation of retrieving the smallest index. In our priority indexing structure, priorities are represented by a finite number of indexes, which correspond to the departure times of the packets stored in the per-flow queues. The departure times are mapped to indexes using modulo operation as shown in Section 5.7.1. So it may happen that packets with later departure times are mapped to smaller indexes. The `extractsucc(i)` operation is essential for skipping the rollover indexes correspond to later departure times to locate the index for the earliest departure time that is after the previous extracted index. For applications requiring the retrieval of the largest index, `extractmax`, `predecessor`, and `extractpred` operations are supported instead.

5.5 Counting-Priority-Index

In this section, we present the CPI structure which has time complexity of $\Theta(\log_W U)$ for index update operations.

5.5.1 Structure of CPI

All operations traverse the CPI from root to leaf with the help of extra counters at non-leaf nodes. A total of W counters are required at each non-leaf node. Let the counters for a non-leaf node be `counter`$[0]$, `counter`$[1]$, $\cdots$, `counter`$[W-1]$, respectively. The `counter`$[i]$ of a node is used to track the number of value 1 bits in its $(i+1)^{th}$ child node from the left. A counter is 0 if there is no value 1 bit in its child node. Its corresponding bit in the W-bit word is also 0. Otherwise the counter should be

a non-zero value and the corresponding bit in the W-bit word is 1. The W-bit word at each non-leaf node is utilized to take the advantage of the fast instructions provided by modern processors. To locate the first non-zero counter in a node, instead of checking the counters one by one which would result in $O(W)$ time complexity, `BSR` or `BSF` can be applied to get the position of the counter from its W-bit word using only one instruction. The data structure of the CPI with height $h = 3$ is shown in Figure 5.3.

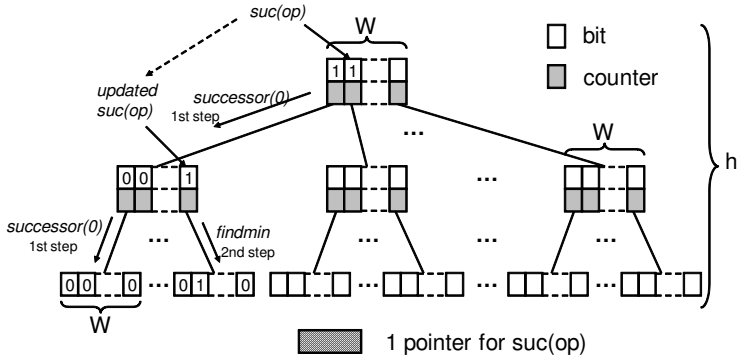


Figure 5.3: Example of a `successor(0)` operation in a CPI.

5.5.2 Operations in CPI

An `insert(i)` operation increases the counters on its way from the root to a leaf node and updates the corresponding bits and counters. A `delete(i)` operation starts from the root node and moves down until it reaches the index i. The counters on its path are decreased by one. If a counter becomes zero, its corresponding bit in the W-bit word is reset to 0. At the leaf node, the bit corresponding to index i is set to 0. The `extractmin` operation finds the lowest index and removes it from the CPI. It goes top-down by first checking the root node to find the first value 1 bit to the left. Then it moves to the child node of such a bit and checks again until a leaf node is reached. The counters on its path are decreased by one. It is worth noting that in each node, to find the first value 1 bit to the left takes only one instruction if $W = 64$. The `findmin` operation is similar to the `extractmin` operation without decreasing the counters or removing the index.

In a CPI, the `successor(i)` operation consists of two steps. In the first step,

it starts from root node and works top-down to find the path to index *i*. At the same time, a pointer `suc(op)` stores the bit that can potentially lead to the successor of index *i*. The parameter `op` is the *operation id* number in the system that is unique for each operation. `suc(op)` is initialized to `NULL`. At each level, the node on the path leading to index *i* is examined for its first value 1 bit to the right of the path. If there is such a bit in the node, the `suc(op)` is updated by pointing to the bit. Otherwise, the `suc(op)` is left unchanged. After the index *i* is reached at the leaf-level, the operation continues to the second step by first checking if the successor of *i* is in the same leaf node as *i*. If it is, the operation is finished and the successor of index *i* is found. Otherwise, the operation resumes from the location kept in the pointer `suc(op)` to extract the minimum index in CPI using `suc(op)` as its root node. The operation then outputs the index as the successor of *i*. Therefore the second step is the same as `findmin(> i)` using `suc(op)` as its root node. At the leaf nodes, if the pointer `suc(op)` is still `NULL`, a new `findmin` operation is initiated to find the minimum index in the CPI. Its result is regarded as the output of `successor(i)`, unless it returns `NULL` when the CPI is empty.

To show that the `successor(i)` operation described above can find the successor of index *i*, let's consider the following. First assume that the successor of index *i* is larger than *i*. Iff the successor of *i* is in the same leaf node as *i*, it can always be found by the algorithm. Let's assume that the successor of *i* is not in the same leaf node of *i*. In non-leaf nodes, the *W*-bit words are designed to remember whether there are existing elements, i.e. value 1 bits, in the corresponding child nodes. So there exists at least one non-leaf level where there is a value 1 bit in a *W*-bit word representing the existence of index *i*. There is another value 1 bit in the same *W*-bit word representing the existence of the successor to *i*. To see that there can be no other bit of 1 in between these two bits of 1's, consider the opposite. There is an other value 1 bit/bits in between these two bits. Then the successor of *i* should be in the child node that belongs to the first of these bits from the left, which contradicts the fact that the successor of *i* is in the child node that belongs to another value 1 bit. The `successor(i)` operation can always find such a node in the level of the tree containing the two value 1 bits shown above. On the other hand, when the successor of index *i* is smaller than index *i* due to the clock rollover shown later in Section 5.7.1, it can be found by the `findmin` operation when

successor(i) returns NULL. Thus the successor of index i is always correctly retrieved if it exists. An example of the successor(i) operation in CPI with $i = 0$ is shown in Figure 5.3. Here suc(op) is updated twice, once at the root level, and the second time at the second level, where there are value 1 bits in the same node to the right of the path leading to index i.

In the extractsucc(i) operation, the counters on the path from the root node to the index of the successor of i need to be updated. Thus, the first step of the extractsucc(i) operation is the same as the first step of the successor(i) operation. In the second step, the extractmin(> i) operation is implemented using suc(op) as the root node. If the operation returns NULL, then a new extractmin operation is initiated to remove the smallest index in CPI due to clock rollover.

5.5.3 Memory and Complexity

Consider a CPI of size $U = W^h$ with $W = 64$. There are U bits in the leaf level nodes. For the upper next level, there are U/W bits for the W-bit words. There are U/W counters with each counter of size $\log_2 W$ to maintain the W indexes in its child node. Altogether there are $6U/W$ bits for the counters in this level. In fact, the number of value 1 bits in a leaf node has $\log_2 W + 1$ states, including the all zero state and the all one state. With the help of the single bit summary index, a counter in the second level node from the bottom of a CPI need only to distinguish $\log_2 W$ states, which requires $\log_2 W$ bits. The all zero state and the all one state can be distinguished by the summary index. Therefore altogether there are $U(1+\log_2 W)/W$ bits at this level, including both the bits for the counters and the W-bit word. Similarly, the next upper level needs $U(1+2\log_2 W)/W^2$ bits, with each counter of size $2\log_2 W$ to track its W^2 descendant nodes at the leaf-level. A pointer is required to store suc(op). To calculate the size of the pointer, it is important to realize that only $\log_2 h$ bits are needed to store the level of the node in the CPI and $\log_2 W$ bits are needed to store the location of the bit within a node. The node containing the bit to be stored in the pointer is on the path of operation successor(i) or extractsucc(i). Therefore knowing the level and the position of the bit within a node is enough to accurately locate it in a CPI. So the pointer is only of size $\log_2 h + \log_2 W$ bits. At any moment, there is one operation in CPI, so one pointer is

sufficient. For $U = W^h$ and $W = 64$, its memory requirement $\mathtt{M}_{\mathtt{CPI(W,h)}}$ is as follows,

$$\begin{aligned}\mathtt{M}_{\mathtt{CPI(W,h)}} =& U + \frac{U}{W}(1+\log_2 W)+ \\ & \cdots + \frac{U}{W^{h-1}}(1+(h-1)\log_2 W)+ \\ & \log_2 h + \log_2 W \\ \approx & 1.11U, \text{ for } W = 64. \end{aligned} \tag{5.1}$$

The required memory to support a CPI is only of size $1.11U$ for a universe of size U. In a CPI, `insert`, `delete`, `findmin`, and `extractmin` operations take $\Theta(\log_W U)$ time, by using `BSR` and `BSF` instructions provided by x86 processors to achieve a time complexity of one time slot at each level of CPI. The `successor(i)` and `extractsucc(i)` operations may need to traverse the CPI from the root node to the leaf-level nodes twice in the worst case. Thus their time complexity is also $\Theta(\log_W U)$.

5.6 Pipelined Counting-Priority-Index

In this section, the pipelined Counting-Priority-Index (pCPI) is presented. In a pCPI, a new operation can be initiated in each time slot.

5.6.1 Structure of pCPI

By pipelining all operations, a pCPI can achieve h times the access rate of a regular CPI, where h is the number of pipeline stages. The data structure of a pCPI with $h = 3$ is shown in Figure 5.4.

5.6.2 Operations in pCPI

The `insert(i)`, `delete(i)`, `findmin` and `extractmin` operations only traverse the pCPI once, so they can be easily pipelined. However, `successor(i)` and `extractsucc(i)` operations need to be modified in order to be pipelined.

The `successor(i)` operation is different from the one in CPI. Once the pointer `suc(op)` is updated, the `successor(i)` operation is split into two parallel sub-operations

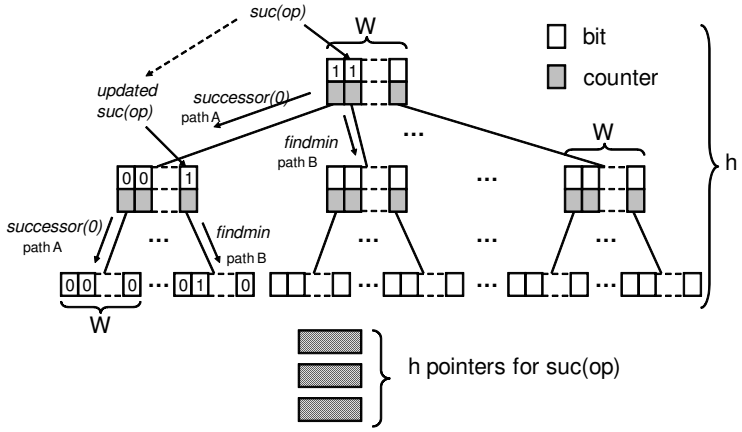


Figure 5.4: Example of a successor(0) operation in a pCPI.

following two different paths. One path (path A) continues the original successor(i) operation to locate index i. The other path (path B) is the path of the findmin($>$ i) operation which uses the suc(op) as its root node. If the suc(op) is updated in path A, path B stops its current process and continues the findmin($>$ i) operation using the new suc(op) as its root node. As a consequence, the two sub-operations always work at the same level from the root to leaf-level nodes. At leaf nodes, if findmin($>$ i) locates an index, it is then used as output of successor(i). As in a CPI, if the pointer suc(op) is still NULL at a leaf node, then a new findmin operation is initiated to find the minimum index in the CPI. Its result is used as the output. An example of the pipelined successor(i) operation with $i = 0$ is shown in Figure 5.4. Here the suc(op) is updated twice, once at the root level, and the second time at the second level. After each update, the findmin($>$ i) operation resumes from the new suc(op). The flow graph of a successor(i) operation is shown in Figure 5.5.

The extractsucc(i) operation has to be split into two operations in order to be pipelined. For a extractsucc(i) operation, the counters on its path B need to be updated. However, part or all of the entries on path B may not be on the path that leads to the actual successor of index i. Therefore it is necessary to separate one extractsucc(i) operation into a succeccsor(i) operation with a returned index j for $j > i$ and a delete(j) operation, or a succeccsor(i) operation with return *mathttNULL* and a extractmin(j) operation in case of clock rollover such that the next timestamp is mapped to a smaller index. In this way, a counter is not updated unless it is indeed

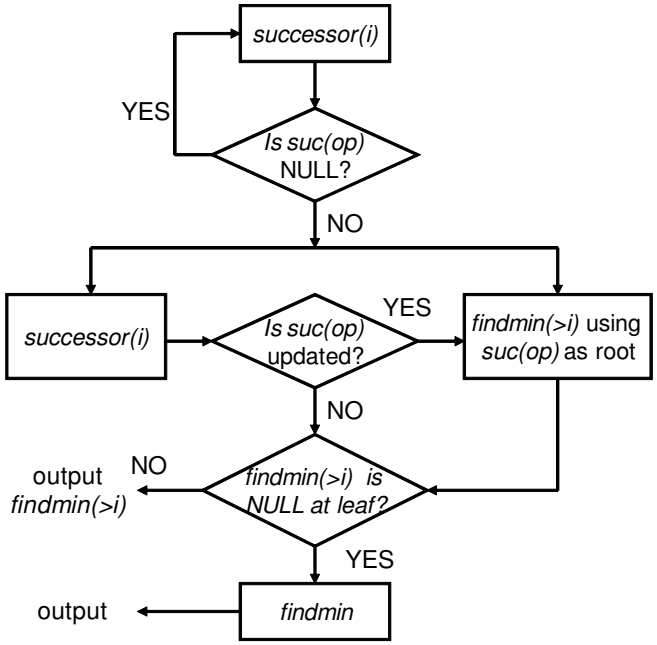


Figure 5.5: Flow graph of `successor(i)` operation in a pCPI.

on the path leading to index j. It is worth noting that the counters at non-leaf nodes cannot be used to help locate the successor of index i, since a nonzero counter can also represent the existence of value 1 bits that are smaller than index i but correspond to larger timestamps, even though the successor of index i cannot be reached from the same counter.

5.6.3 Memory and Complexity

Compared with non-pipelined CPI, in a pCPI extra pointers are needed for `suc(op)`. As in CPI, $\log_2 h$ bits are required to distinguish the level the pointer is pointing to and $\log_2 W$ bits are required to store the location of the pointer within a node. There are at most h operations in the pCPI at the same time, so h such pointers are enough for the pipeline operations. For $U = W^h$ and $W = 64$, the following holds for

the total memory size $\mathtt{M}_{\mathtt{pCPI(W,h)}}$.

$$\begin{aligned}\mathtt{M}_{\mathtt{pCPI(W,h)}} =& \mathtt{M}_{\mathtt{CPI(W,h)}} + (h-1)\{\log_2 h + \log_2 W\} \\ \approx& 1.11U + h\log_2 h + \log_2 W^h \\ \approx& 1.11U + \log_2 U, \text{ since } h \ll W = 64 \\ \approx& 1.11U \text{ if } U > 2^{10} \end{aligned} \tag{5.2}$$

So a pCPI requires essentially the same amount of memory as a CPI, which is about 11% larger than the size of the universe U.

For a pCPI, in each time slot one new operation can be initiated, thus enabling fast operations that are essential for per-flow queueing in network applications. At each level of the pCPI, two sub-operations may exist at the same time on different nodes for a `successor(i)` operation. Therefore if all the nodes in the same level of the pCPI are in a single SRAM, it needs to support *double* access rate per time slot. When the pCPI is employed to achieve a constant processing speed, different levels should be stored in independent SRAMs to achieve the best performance. An `extractsucc(i)` requires a `delete(j)` operation (or `extractmin` with clock rollover) to successfully remove the correct index. In the worst case, a `successor(i)` operation requires a `findmin` operation when the successor of index i is smaller than i due to clock rollover. However, clock rollovers are rare since the total number of index U is huge (e.g. 2^{24}). Both `successor(i)` and `extractsucc(i)` operations still take an amortized constant time to finish.

5.7 pCPI for Per-flow Queueing

The pCPI can be implemented as an essential part of packet scheduler by sorting packets in their departure order at line rate based on the adopted scheduling algorithm. A pCPI maintains the ordering of the indexes that correspond to the timestamps of the first packets in per flow queues. It is crucial to find an efficient and effective way to map the index back to its corresponding packet when the packet is being scheduled for departure. In this section, we present the mapping from a departure timestamp of a packet to an index in the pCPI, and vice versa. We also present schemes to locate

the packet with the departure timestamp when the index corresponding to the departure timestamp is scheduled for departure. Therefore, we can effectively locate a packet in the per-flow queues for departure when its corresponding index is reset to 0 in the pCPI. The structure for per-flow queue scheduling using a pCPI is shown in Figure 5.6. The solid line represents the path for inserting a timestamp for scheduling. The dotted line represents the path for retrieving the packet record when it is scheduled for departure.

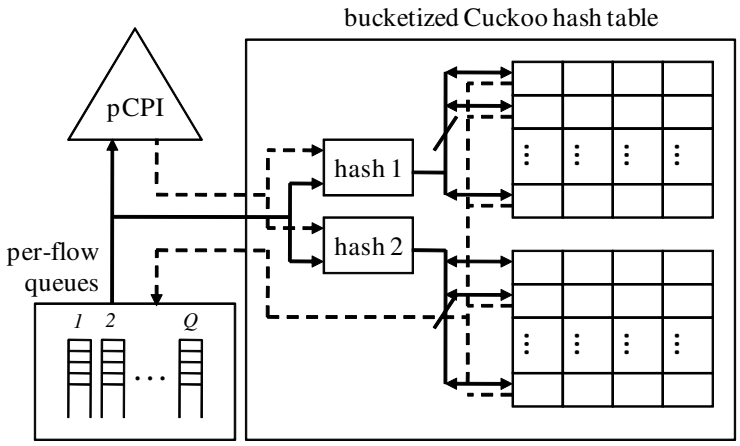


Figure 5.6: pCPI for per-flow queue scheduling using hash tables.

5.7.1 Index Mapping

In a pCPI, a timestamp is represented by a one-bit index. We define the order of an index based on its location among all the indexes in the leaf-level nodes. If there are i bits to the left of an index, it is of order i. The index of order i is set to 1 if there is a packet with timestamp t that is mapped to the index. If the timestamp is t, it is mapped to an index using the following modulo function,

$$\mathbb{T}(t) = t \mod U, \tag{5.3}$$

where U is the total number of indexes in a pCPI. We call Equation (5.3) the *index mapping*. If there are two or more *different* timestamps mapped to an index, a collision would occur. Due to the modulo operation in the index mapping, such collisions can only be caused by clock rollover, where the timestamps mapped to the same index are separated by kU cycles, where k is an integer. Unfortunately, in a pCPI clock rollover

is not easily supported, since a timestamp is represented using only one bit in the leaf nodes. A collision involving two or more different timestamps cannot be distinguished using one bit. Even though more bits can be added at each leaf-level index for clock rollover protection, operations such as `extractsucc(i)` which can be used to find the next timestamp for departure, will not notice that an index in the pCPI is not ready for departure in the next kU cycles until it reaches the leaf-level node and checks the rollover protection bits. Such an `extractsucc(i)` operation would introduce extra delay which is also unpredictable. As a consequence, it makes the operations difficult to pipeline due to the unpredictable and non-fixed processing time for each operation, if clock rollover protection bits are added for resolving timestamp collisions.

5.7.2 Timestamp Mapping

To implement pCPI for packet scheduling, we propose to size the number of indexes in a pCPI large enough so that collisions caused by clock rollover are negligible or even impossible. Consider a packet buffer operating at line rate of 40 Gb/s with a round-trip-time of 250 ms. A packet buffer of size 10 Gb is required following the buffer sizing rule in Section 5.1. If we assume there are less than 256 K per-flow queues at any moment and the average packet size is 80 bytes, then a total of up to 16 million packets would have to be stored in the packet buffer. In this example, we can safely assume that any arriving packet has a departure time less than 250 ms away, which is the round-trip-time, since a packet is stored in the packet buffer for at most 250 ms according to the buffer sizing rule. So by designing a pCPI to support 16 million indexes, it is guaranteed that two packets in the packet buffer with different departure timestamps are always mapped to different indexes. In this case, the mapping function in Equation (5.3) is invertible when the timestamp of the first index (left-most index in the leaf-level nodes in Figure 5.4) is known. The inverse function $\mathbb{T}^{-1}$ is thus well defined as follows,

$$\mathbb{T}^{-1}(i) = i + t_0. \tag{5.4}$$

where t_0 is the timestamp of the first index in the pCPI. We call Equation (5.4) the *timestamp mapping*, which maps a priority index in pCPI back to its corresponding timestamp. Once an index is removed from the pCPI for departure, its corresponding

timestamp can be calculated using the timestamp mapping. Such a pCPI supporting 16 million indexes requires only four pipeline stages if the width of each node is 64 bits. Its size is only 2.22 MB (16 million$\times 1.11/8$).

5.7.3 Queue Mapping using Hash Functions

When a timestamp is calculated from an index in the pCPI, the next step is to locate the per-flow queue that contains the packet with the departure timestamp among all per-flow queues. We call this step the *queue mapping*. For per-flow queueing, most scheduling algorithms [8,21,88] only accurately calculate the departure timestamp of the first packet in each queue. The departure times of all the other packets are either inaccurate and subject to change by the time they become the head of the queues, or unavailable and only calculated when the packets move to the head of the queues. Therefore, only the departure timestamps of the first packets in the per-flow queues are admitted to the pCPI to set the corresponding indexes using index mapping $\mathbb{T}(\cdot)$. It is worth noting that two or more packets from different queues may have the same departure timestamp, which has to be taken into account in the queue mapping.

We propose to use a bucketized Cuckoo hash table [26] to implement the queue mapping. Consider as an example the case with 256 K queues. Since only the departure times of the first packets of the per-flow queues set the indexes in the pCPI, there can be at most 256 K indexes of value 1 in the leaf nodes. Therefore, the hash table need only to accommodate 256 K hash keys, with each hash key corresponding to a timestamp. A new entry is created in the hash table when a packet at the head of the per-flow queue enters the pCPI. In each hash table entry, both the hash key and its hash value are stored, where a hash value is the per-flow ID of the flow that contains the packet with the timestamp. Since there is no collision in the index mapping, we can use the order of an index instead of its full timestamp as a hash key. So each hash key is from a pool of 16 million timestamps, which is represented using only 24 bits for each key. In this example, a hash value which contains the per-flow ID is of size 18 bits to distinguish a total of 256 K per-flow queues.

It is difficult if not impossible to build a perfect hash function, due to the dynamic nature of per-flow queueing. Different hash keys may be mapped to the same hash

table entry. Therefore, it is important that our hash table can effectively handle hashing collisions. Classic Cuckoo hashing [70] is an open-address hash table that uses two tables and two independent hash functions. It is equivalent to a bucketized Cuckoo hash table with two hash functions and only one entry/cell per bucket. Each cell stores a hash key and its payload data (hash value). The two hash tables can be constructed by dividing equally a single hash table. A new key K_1 is hashed to one of the hash tables using one hash function. In case of a hash collision, the key K_2 that occupies the hash table entry is evicted and rehashed to the other hash table using the other hash function. The current entry is occupied by K_1 instead. If there is a collision when K_2 is being inserted into the other hash table, the key K_3 that occupies the entry is evicted and rehashed to the first table. The eviction process continues until all keys are hashed into the hash table without collision. It is proved in [70] that with a maximum load factor below 49%, the worst-case cost for a key insertion is a constant. The maximum load factor is the largest occupancy ratio in the hash table such that the average key insertion time is a constant. To increase the maximum load factor and save storage, we use a variant called a bucketized Cuckoo hash table, where a new key is hashed to a bucket and there are multiple cells/entries in a bucket. Hash collisions to the same bucket are stored in different cells within the bucket, until the bucket is full. Rehashing caused by hash collisions can be greatly reduced by storing multiple keys in a bucket, since rehashing is not necessary until a bucket is full. As shown in [26], with a bucket size of four cells and two hash functions, the maximum load factor increases to 93%. i.e., below 93% load, the hash table is almost always able to store a new hash key in constant time. Above 93% it is likely to encounter an irresolvable collision where a new hash key cannot be inserted into the hash table.

Another advantage of using a bucketized Cuckoo hash table in our design is as follows. In rare events, multiple packets may have the same departure time assigned by the scheduler based on the QoS requirements for the flows containing the packets and the departure times of the previous packets, even though in practice only one packet may arrive at or depart from the packet buffer in any time slot. A bucketized Cuckoo hash table can easily fit the keys of such packets into its buckets by increasing the number of cells allowable in a bucket. It is worth noting that such packets will set the index

in the pCPI only once. When the index is reset for departure, all the packets with the same departure time can be scheduled for departure using the queue mapping. Also, the size of the hash table can be increased when its performance decreases due to key deletions [49].

The queue mapping can also be implemented using a d-left hash table [11]. In a d-left hash table, each hash table entry is a bucket that can hold several keys. A total of d hash functions hash a key to d buckets, with each function hashing the key to one bucket. The bucket with the least number of keys is chosen to store the current key, which effectively balances the load in the table. By designing the bucket large enough, the probability that a key insertion causes a bucket to overflow can be made arbitrarily small. However, a d-left hash table requires the maintenance of the numbers of keys already stored in each bucket, which adds to the hardware complexity.

Compared to other hashing schemes, bucketized Cuckoo hashing with two hash functions has the advantage that a key lookup takes only two operations to two different hash tables, which can be further parallelized in hardware. A new key is inserted into the table using hashing and rehashing in case of collision. The probability of having an insertion failure due to an irresolvable collision is negligible if the load factor is kept below the maximum load factor for the hash table. Hash keys can be inserted into the table on average in constant time. On the other hand, hashing schemes such as the standard chain hash table, clustered chain hash table, array hash table, and linear probing suffer from the problem that in the worst case the key lookup time is unbounded [6], which is unsuitable for supporting pipelined operations necessary for managing large numbers of per-flow queues at line rate.

In summary, after the timestamp of a packet at the head of the per-flow queues is calculated, the corresponding index in the pCPI is set to 1 using the index mapping, i.e. Equation (5.3). Meanwhile, a new entry is created in the hash table for the packet using its timestamp as the hash key and the per-flow queue ID as the hash value. Both the hash key and hash value are stored in the hash table entry. When the packet is removed for departure from the pCPI, its timestamp is calculated using the timestamp mapping, i.e. Equation (5.4). The per-flow queue that contains the packet is located using the queue mapping by a hash table lookup. The corresponding entry is then removed from the

hash table entry and the packet is removed from the per-flow queue for departure. The hash table can be effectively implemented using a bucketized Cuckoo hash table with two hash functions and four or eight cells per bucket as shown in Section 5.9.

5.7.4 CAM for Queue Mapping

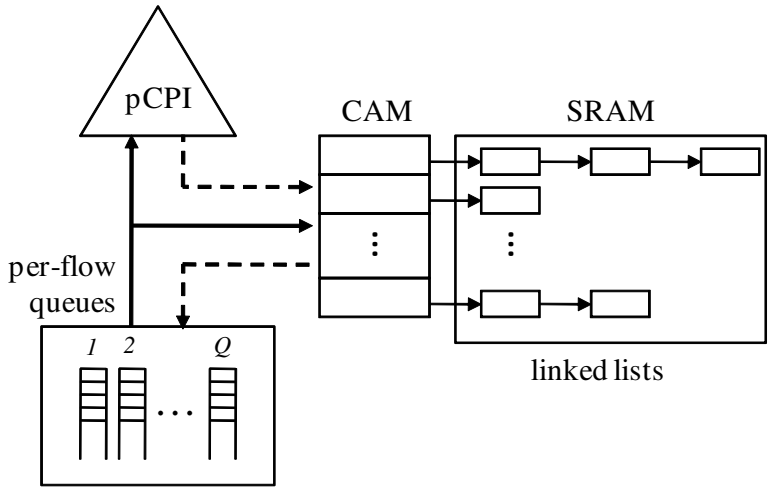


Figure 5.7: pCPI for per-flow queue scheduling using CAM.

Queue mapping can be implemented directly using CAM (content addressable memory) with the help of linked lists as shown in Figure 5.7. The timestamps of the packets at the head of the per-flow queues together with the corresponding flow IDs are stored in the CAM when they are inserted into the pCPI for scheduling. The CAM has to be large enough to maintain all the packets represented by the pCPI. In the example of 256 K per-flow queues, there can be at most 256 K entries in the CAM. Each entry in the CAM has a different timestamp. Packets with the same departure timestamps are connected together using linked lists to the same entry in the CAM. When a timestamp is calculated from the index in the pCPI, the corresponding entry in the CAM is accessed. All the packets with the same departure timestamp will then depart in order by following the linked list. Even though using CAM for queue mapping in the per-flow queue scheduling achieves significantly simpler design than using a bucketized Cuckoo hash table in SRAM, the relatively larger size of CAM and the higher cost makes the hash table design much more attractive. In a CAM, for each bit there is a comparison circuit,

which adds to the physical size and the associated cost of the CAM chip significantly. On the other hand, in the bucketized Cuckoo hash table design, only SRAM is required to store the hash keys and their values. Simple hash functions such as modulo functions which are easy to implement have been shown to perform well in practice [49]. Therefore, the bucketized Cuckoo hash table design can effectively accomplish the searching speed of a CAM design at lower cost.

5.8 Implementing Priority Queues in Packet Buffer

Per-flow queues can be implemented in packet buffers to provide for differentiated QoS guarantees. In this section, we present how a pCPI can be utilized as part of a packet scheduler for managing the packets in a packet buffer.

5.8.1 Packet Buffer Abstraction

In a network router, to efficiently process packets at high line rate, it is common to divide packets into fixed-length segments before storing them in the packet buffer and later forwarding them to the next node. Thus, we assume that all incoming variable-size packets are segmented into fixed-size packets, and reassembled when leaving the router. We also define a time slot as the time it takes for the minimum size packet to arrive in the packet buffer at a certain line rate. i.e., a time slot is the time period from the first bit of a minimum size packet arriving at the ingress port to the last bit of the packet arriving at the same port continuously. For example, at a line rate of 40 Gb/s a time slot for a 40-byte packet is 8 ns.

In a packet buffer, at each time slot at most one new packet arrives at the packet buffer with a future departure time, and at most one existing packet departs from the packet buffer in the current time slot, with both occurring in the same time slot in the worst-case. The time slots are mapped to the leaf level nodes in the pCPI module with clock rollover. The pCPI has to provide enough indexes so that clock rollovers will not lead to collisions caused by multiple departure times being mapped to the same priority index. Even though multiple packets in the per-flow queues may be assigned the same departure timestamps, they must depart from the packet buffer at different time slots

sequentially.

Consider the following example. At a line rate 40 Gb/s with an RTT of 250 ms, the average packet size is 80 bytes. According to the buffer sizing rule, we need a packet buffer of 10 Gb that can store up to 16 million packets during the round-trip-time delay. So we set the size of the universe represented by the pCPI as $U = 16$ million, which requires 4 pipelined stages. There can be tens of thousands of active flows arriving, with several service classes in each flow. We assume that there are at most 256 K per-flow queues. The mapping from flows to indexes in a pCPI is as shown in Section 5.7.

5.8.2 Packet Buffer Architecture with pCPI

To provide bulk storage at line rate, interleaved DRAM banks are developed for packet buffers [43, 100]. If a DRAM transaction (read or write) takes b time slots, more than b DRAM banks are necessary to support line rate operations. To handle repetitive accesses to the same DRAM banks, write packet queues and read request queues are required to temporarily store pending memory accesses. A pCPI can be implemented to support various scheduling algorithms for the packet buffers.

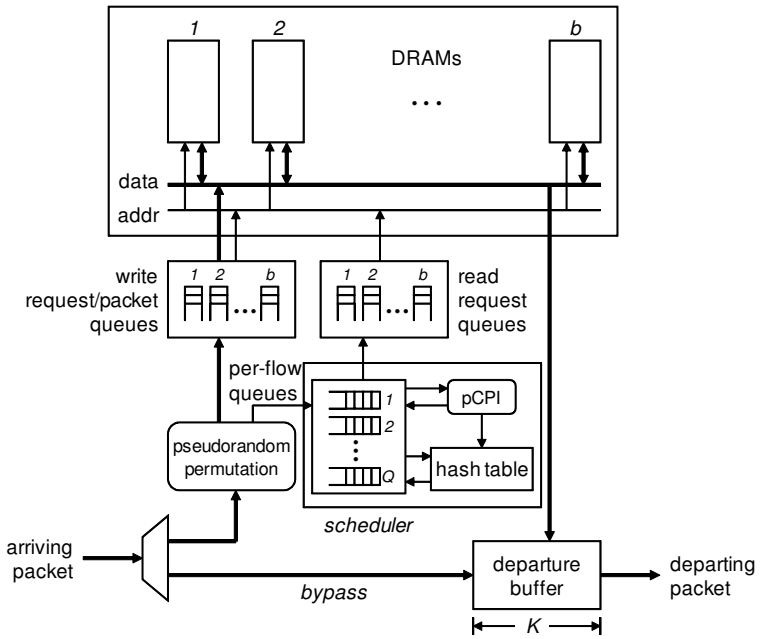


Figure 5.8: Per-flow scheduling for large packet buffers using pCPI.

An example of large packet buffer in DRAM using pCPI for packet scheduling is shown in Figure 5.8. Arrival packet information is stored in Q different per-flow queues to provide for differentiated QoS guarantee. The departure ordering of the packets in the packet buffer is managed using a pCPI together with a hash table, such as the bucketized Cuckoo hash table in Section 5.7.3. Let's assumes that the packets in the buffer are serviced in the earliest-deadline-first order. Upon packet arrival, the packet information is inserted into a per-flow queue based on its flow ID and QoS requirement. The first packets in the per-flow queues are inserted into the pCPI using index mapping. A packet stored in the packet buffer is scheduled for departure when its corresponding index is cleared to 0 in the pCPI. The per-flow queue storing the packet is located using timestamp mapping together with queue mapping. Then a packet read request based on the flow ID is sent to the read request queues to retrieve the packet from the DRAM banks. Various schemes can be applied to locate the packet in the DRAM banks. The detailed algorithm will be treated later. The storage requirement in the per-flow queues can be reduced by storing in the Q queues only the packet header which includes flow ID and packet sequence number, but not the payload data, since the payload data in a packet is not required for packet scheduling. It is worth noting that using index mapping, timestamp mapping, and queue mapping, a pCPI can also be implemented as an essential part of the scheduler for any other packet buffer scheme requiring per-flow queueing.

5.9 Performance Evaluation

In this section, the advantage of using pCPI and memory management for large priority queues is discussed. Scalable priority queue implementation requires solving two fundamental problems: efficient sorting in real-time and storage of a large number of packets. These two problems are solved in this chapter by decomposing the system into two parts, a priority index in SRAM and an efficient DRAM-based implementation of fine-grained logical queues. For example, with a 24-bit priority index, our pCPI can manage a total of 16 million per-flow queues. On a link of 40 Gb/s, it can allow us to manage per-flow rates down to $1/2^{24} \times 40$ Gb/s or about 2.5 Kbits/s. Thus extremely fine-grained QoS has been achieved. Using the 24-bit indexes, the size of the universe is

$U = 16$ million. Therefore only about 2.22 MB are required in the SRAM for the pCPI structure, since it takes only about $1.11U$ memory space.

The advantage of the pCPI compared to the pipelined heap is its dramatic reduction in the number of pipeline stages required. To match the ever-increasing line rate, different pipeline stages are necessarily stored in separate SRAMs, which determines the complexity of the implementation. As shown in Figure 5.9, the number of pipeline stages in a pipelined heap grows linearly with the width of the priority index. To support 256 K indexes, the pipelined heap needs 18 stages. In our pCPI, 3 stages are sufficient to support 256 K indexes. With 4 stages, a pCPI can support up to 16 million indexes. Even though asymptotically the pipelined van Emde Boas tree performs better than our pCPI, for all practical network applications with no more than $2^{30} = 1$ G priority indexes, the pCPI outperforms the pipelined van Emde Boas tree by using fewer stages as shown in Figure 5.9. Also, all the operations in the pCPI in each level can be finished in one instruction using the hardware optimized instructions provided by 64-bit x86 processors. The operations in a pipelined van Emde Boas tree require more instructions due to the variable size of its nodes at the different levels.

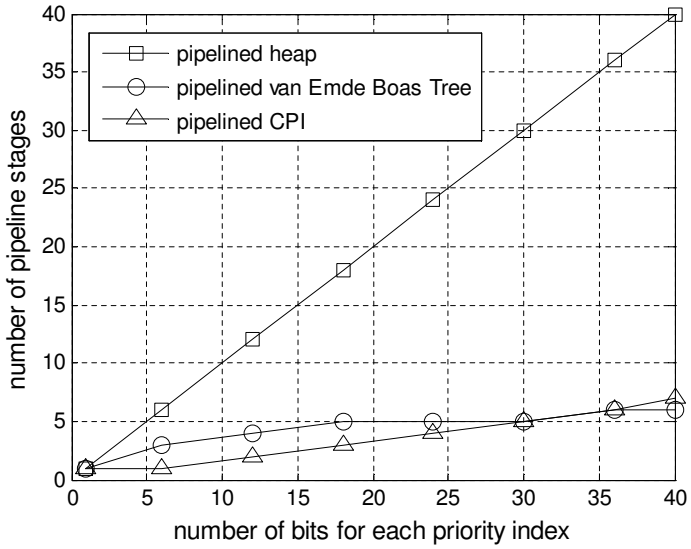


Figure 5.9: Number of pipeline stages in three data structures.

Now consider the memory requirement for implementing the queue mapping. The memory requirement for different types of bucketized Cuckoo hash tables to support

Table 5.2: Sizes of bucketized Cuckoo hash tables to store 256 K keys.

number of hash functions	number of cells per hash bucket			
	1 cell	2 cells	4 cells	8 cells
4 hashes	1.36 MB	1.33 MB	1.32 MB	1.31 MB
3 hashes	1.45 MB	1.36 MB	1.34 MB	1.32 MB
2 hashes	2.68 MB	1.53 MB	1.42 MB	1.37 MB
1 hash	2187.5 MB	218.75 MB	43.75 MB	10.94 MB

a total of 256 K per-flow queues is shown in Table 5.2 as examples. The hash tables vary in the total number of hash functions being used and the number of cells per bucket for storing the hash keys. We first analyze the SRAM size for a bucketized Cuckoo hash table with two independent hash functions and four cells in each bucket. Each hash key is of size 24 bits to distinguish a total of 16 million indexes. The hash value is of size 18 bits to represent a total of 256 K per-flow queues. So each hash entry is of size 42 bits and a bucket is of size 21 bytes. For a key lookup, all the cells in a bucket can be accessed in one step since the CPU cache-line is typically between 64 and 256 bytes, which is larger than the bucket size. Therefore in a hash table lookup operation for retrieving a flow ID there is no cache miss, which is essential for the hash table to match the line rate throughput. Consider the example that there are at most 256 K keys stored in the hash table, with one key for each packet at the front of the queues. The maximum load factor for such a table is 93%, so the bucketized Cuckoo hash table has to hold at least 276 K keys to keep the load factor below 93%. The total size of the table is only 1.42 MB. In fact, a bucketized Cuckoo hash table with 8 keys in each bucket has bucket of size 42 bytes, which can still fit in the CPU cache-line. Such a hash table has a maximum load factor of 96%. So the total size of a bucketized Cuckoo hash table with two hash functions and eight keys in each bucket is of size only 1.37 MB. As shown in Table 5.2, by using more than two hash functions which requires an increase in hardware complexity, the memory size is only slightly reduced.

Therefore, we can use bucketized Cuckoo hashing with two hash functions and eight cells in each bucket. In our example, such a hash table storing 256 K hash keys requires a total of 1.37 MB of SRAM. The total storage requirement is thus 3.59 MB

including the pCPI structure. For the pipelined heap implementation in [42], in each entry, both the timestamp and the per-flow ID are stored. To support 256 K priority keys, the total storage is 1.32 MB. The SRAM needed for our pCPI implementation is larger than the pipelined heap. Nevertheless, the total SRAM required is very small for our implementation. It can be easily pipelined. On the other hand, the pipelined heap design requires at least 18 separate SRAM devices to store the 18 pipeline stages for the 256 K per-flow queues, which adds significantly to complexity.

The memory requirement for implementing queue mapping using CAM can be calculated as follows. In our example with 24 bits timestamp and 18 bits flow ID to support 256 K per-flow queues, the total size of the CAM is about 1.32 MB. The linked lists can be dynamically allocated from a single SRAM. To support the worst case where all 256 K packets at the head of the per-flow queues have the same departure timestamp, the linked lists have to share 256 K nodes. Accordingly, the pointer in the linked list is of size 18 bits to distinguish the 256 K nodes in the linked list. In each node in the linked list, both flow ID and the next pointer are maintained. Thus the total SRAM requirement to implement the linked lists is 256 K$\times$ $(18+18) \approx 1.13$ MB.

The total memory requirement for implementing queue mapping using CAM is 1.32 MB CAM and 3.35 MB SRAM including the pCPI structure, compared to the hashing design which requires only 3.59 MB of SRAM. Given the much higher cost per bit and higher power consumption of CAM compared to SRAM, implementing the queue mapping using a bucketized Cuckoo hash table is more cost efficient and thus preferable.

5.10 Conclusion

In this chapter, fast and scalable succinct data structures for the implementation of per-flow priority queues maintenance in networking applications are presented. These structures are the Priority-Index (PI), the Counting-Priority-Index (CPI), and the pipelined Counting-Priority-Index (pCPI). Also, novel solutions on the management of a large number of priority queues using the proposed pCPI structure are developed. The presented data structures and the associated algorithms are well-suited for modern 64-

bit x86 processors from Intel and AMD with that have hardware-optimized instructions. These structures can be very compactly implemented in SRAM using only $\Theta(U)$ space, where U is the size of the universe required to implement the priority keys. such as the deadline timestamps of the packets. Our pCPI data structure can effectively support constant time priority management operations. In addition to being very fast, this architecture also scales well to a large number of priority values and to large queue sizes. In particular, the hardware complexity is only $\Theta(\log_W U)$, where U is the size of the universe that represents the range of priority keys and W is the word size. We provided a scheme to map a departure timestamp of a packet to a priority index in the pCPI. We also provided schemes to locate the per-flow queue that stored the packet when the index in the pCPI is cleared when the packet departs. Two solutions for mapping a departure timestamp back to a per-flow queue are described. One uses a bucketized Cuckoo hash table implemented in SRAM. The other one uses a CAM with linked lists storing multiple packets with the same departure timestamp. The cost of both schemes are analyzed. We show that the pCPI structure can be effectively implemented in high-performance network processing applications such as advanced packet scheduling maintaining a large number of per-flow priority queues with QoS guarantee at OC-768 (40 Gb/s) rate and beyond.

Chapter 5, in part, is a reprint of the material as it appears in the following publications:

- Hao Wang and Bill Lin, "Per-flow Queue Scheduling with Pipelined Counting Priority Index", *IEEE Symposium on High-Performance Interconnects (HOTI)*, Santa Clara, CA, August 24-26, 2011.

- Hao Wang and Bill Lin, "Succinct Priority Indexing Structures for the Management of Large Priority Queues", *IEEE International Workshop on Quality of Service (IWQoS)*, Charleston, SC, July 13-15, 2009.

Chapter 5, in full, has been submitted for publication of material as it may appear in IEEE Transactions on Parallel and Distributed Systems, Hao Wang and Bill Lin, "Per-flow Queue Management with Succinct Priority Indexing Structures for High Speed Packet Scheduling". The dissertation author was the primary investigator and author of the papers.

Chapter 6

Packet Buffers in High Speed Routers

6.1 Introduction

High-performance routers need to temporarily store a large number of packets at each linecard in response to congestion. One of the most difficult challenges in designing high-performance routers is the implementation of packet buffers that operate at extremely fast line rates. That is, at every time slot, a newly arriving packet may be written to the packet buffer, and an existing packet may be read from the packet buffer for departure. To provide sufficient packet buffering at times of congestion, a common buffer sizing rule is $B = \text{RTT} \times C$, where B is the size of the buffer, RTT is the average round-trip-time of a flow passing through the link, and C is the line rate [96]. At an average round-trip-time of 250 ms on the Internet [13] and a line rate of 40 Gb/s (OC-768), this rule of thumb translates to a memory requirement of over 1.25 GB at each linecard, which clearly makes an SRAM implementation impractical. While some research shows that this rule is an overestimate on the total buffer size [5] under certain traffic assumption, other results such as [22] show that even larger buffers may be required under different traffic patterns. The state-of-the-art commercial SRAM holds 36 Mb [85], with a random access latency below 4 ns and a power consumption of 1.8 W. A packet buffer using only SRAM would require 285 SRAM devices and consume approximately 513 W of power. Moreover, the number of SRAM devices required and the power consumption grow linearly as the line rate grows. On the other hand, DRAMs have much larger capacity and consume significantly less power. The state-of-the-art

commercial DRAM has 4 Gb of storage [84]. It has a random access latency of 36 ns and consumes only 1.76 W of power. A packet buffer using only DRAM would require merely 3 DRAM devices and consume approximately 5.28 W of power. However, the worst-case random access times of DRAM are not fast enough to match line rates at 40 Gb/s and beyond, making a naïve DRAM-based solution inadequate. Therefore, researchers have explored prefetching-based [29, 43] and randomization-based [51, 89] memory systems that aim to provide the memory access speeds of SRAM, but with the density of DRAM. These architectures can be utilized to implement a packet buffer with logically separate FIFO queues.

For example, virtual output queues (VOQs) are used in crossbar routers, where each VOQ corresponds to a logical FIFO queue for buffering packets to a particular output port. To enable the random servicing of VOQs at SRAM speeds, the prefetching-based solutions described in [29, 43] employ hybrid SRAM/DRAM designs. These architectures support linespeed queue operations by aggregating and prefetching packets for parallel DRAM transfers using fast SRAM caches. However, these architectures require complex memory management algorithms for real-time sorting and a substantial amount of SRAM for caching the head and tail portions of logically separated queues to handle worst-case access patterns. On the other hand, randomization-based architectures [51, 89] are based on a random placement of packets so that the memory loads across the DRAM banks are balanced. While effective, these architectures provide only statistical guarantees.

Although the assumption of servicing VOQs in random order is reasonable for crossbar-based router architectures, this assumption is unnecessarily general for several important router architectures. In particular, deterministic packet service models exist for those architectures where the *departure times* of packets from the packet buffers can be *deterministically* calculated exactly in advance, before the packet insertion into a packet buffer, when best-effort routing is considered. One such router architecture is the switch-memory-switch router architecture [44, 74]. This architecture can efficiently mimic an output queueing switch. The switch-memory-switch architecture is based on a set of physically separated packet buffers that are sandwiched between two crossbar switches. When a new packet arrives at an input port, the departure time of the packet

is deterministically calculated to exactly mimic the packet departure order of an output queueing switch. A matching problem is then solved to select a packet buffer to store the packet. When the packet gets inserted into a packet buffer, its departure time has already been decided.

Another router architecture is the load-balanced router architecture [15,48]. This architecture has been shown to have interesting scalability properties and throughput guarantees. The load-balanced router architecture is also based on a set of physically separated packet buffers that are sandwiched between two switches. However, in contrast to the switch-memory-switch architecture, there is no central scheduler, and the two switches have a fixed configuration that is independent of the arrival traffic. Instead, in this architecture each packet buffer maintains a set of VOQs that are serviced deterministically in a round-robin order. Therefore, the departure time of a packet from its packet buffer can also be calculated deterministically in advance of writing the packet into the packet buffer.

In this chapter, we propose reservation-based packet buffer architectures. These new architectures require a simple *deterministic* memory management scheme that exploits the known departure times of packets to achieve the performance of an SRAM packet buffer implementation by using multiple DRAM modules in an interleaved manner. With interleaved memories, a high effective memory bandwidth is achieved by accessing K slower memories independently. Using the known departure times of packets that need to be stored in the packet buffers, we can deterministically assign packets to interleaved DRAM modules in such a manner that access conflicts are avoided and packets are guaranteed to be retrieved before their departure times. Our proposed packet buffer architectures can support an arbitrary number of logical FIFO queues with the only assumption being that the departure times of packets can be determined upon the packet arrivals. We prove that the number of interleaved DRAM modules required is a constant with respect to the number of logical queues. A simple bypass scheme is employed using a small amount of SRAM to buffer packets that have departure times less than the round-trip latency of writing to and reading from a DRAM bank. The size of this SRAM is again a constant independent of the number of logical queues. Although interleaved memories have been used in previously proposed fast packet buffer

schemes [29, 43, 51, 89, 100, 107], our proposed techniques take advantage of known departure times to achieve simplicity and determinism.

We will first describe a frame-sized reservation-based approach, of which an early summary was presented in [46]. Although this approach is effective, it has one drawback in that the size of the bitmap grows *linearly* with the size of the packet buffer, therefore requiring potentially non-trivial amount of SRAM for book-keeping. To reduce the memory requirement, we further present an efficient block-sized reservation-based packet buffer architecture, based on the concept of *blocks* that supports deterministic packet departures, on which a much shorter paper of the present work was presented in [99]. This refined solution aggregates packets into blocks so that the amount of book-keeping information in SRAM is minimized. In particular, the total size of the reservation table only grows *logarithmically* with respect to the total size of the packet buffer, which results in an order of magnitude reduction in the SRAM requirement for current implementations. For both schemes, we prove that the required number of interleaved DRAM banks is only a small constant independent of the arrival traffic patterns, the number of flows, and the number of priority classes. Therefore, the reservation-based designs are scalable to growing packet storage requirements in routers while matching increasing line rates. Furthermore, we provide a variation of our designs called a randomized block-based packet buffer that achieves the performance of randomization-based schemes by sacrificing reliability to further reduce the SRAM requirement.

The rest of the chapter is organized as follows. In Section 6.2, we show the related work on memory designs. In Section 6.3, we describe the packet buffer abstraction. In Section 6.4, we present the architecture and operations of the frame-sized reservation-based packet buffer. In Section 6.5, we present the architecture and operations of the block-sized reservation-based packet buffer. In Section 6.6, we present the architecture of the randomized block-sized packet buffer. The performance analysis of these architectures is presented in Section 6.7. Finally, we conclude in Section 6.8.

6.2 Related Work

The problem of designing memory systems for different network applications has been addressed in many past research efforts. Several solutions have been offered to build memory systems specialized in serving as packet buffers. In an Internet router, packets buffers must operate at wirespeed and provide bulk storage. As the link speed increases, the sizes of the packet buffers have been increasing as well, which makes a direct DRAM implementation impossible. To tackle the problem, several designs of packet buffers based on hybrid SRAM/DRAM architectures or memory interleaved DRAM architectures have been proposed [29, 33, 35, 43, 51, 58, 80, 89]. Simple memory interleaving solutions are proposed in [35, 58, 80], where bank-specific techniques are developed. In these schemes, the memory locations are carefully mapped to memory banks using a certain pseudorandom function, so that when memory accesses are evenly distributed into different memory banks, the number of bank conflicts is minimized. The effectiveness of these solutions relies heavily upon how the arriving memory accesses are distributed into the memory banks, which greatly restricts the allowable memory access patterns and limits the applications of the schemes. For example, in a high speed packet buffer supporting millions of flows, the order in which packets are retrieved from memory banks is typically determined by a packet scheduler implementing a specific queueing policy, which may cause packets to be retrieved in an unbalanced fashion and lead to dramatic system performance degradation.

Advanced memory interleaving schemes share the idea of utilizing multiple DRAM banks to achieve SRAM throughput, while reducing [33, 51, 89] or eliminating [29,43] potential bank conflicts. Previous designs can be categorized into two types. The first type is the prefetching-based packet buffer design [29, 43]. Prefetching-based architectures support linespeed queue operations by aggregating and prefetching packets for parallel DRAM transfers using fast SRAM caches. These packet buffer architectures assume a very general model where a buffer consists of many logically separated FIFO queues that may be accessed in random order. However, these architectures require complex memory management algorithms based on real-time sorting and a substantial amount of SRAM for caching the head and tail portions of logically separated queues to handle worst-case access patterns. The other type is the randomization-based packet

buffer design. Different architectures proposed in [51, 89] are based on random placements of packets into different memory banks so that memory loads across DRAM banks are balanced. These architectures also support a very general queueing model in which logically separated queues may be accessed in random order. Various memory management algorithms are proposed to handle packets pending access to a memory bank. In-order packet departures across different flows are provided in [51], while in [89] there is no such guarantee. In general, while effective, they only provide statistical guarantees as there are deterministic packet sequences that can cause system to overflow.

In [44], a number of crossbar-based single-buffered (SB) routers are analyzed. The sizes of the middle stage memories for routers with centralized shared memory are investigated. It is shown that for several such routers to emulate the behavior of a FCFS shared memory router, the total memory bandwidth of the middle stage memories has to be between $2NR$ and $4NR$, where N is the number of input-output pairs in the router, and R is the line rate. In this chapter, we adapt the Constraint Sets method described in [44], which is essentially the pigeonhole principle, for the problem of designing deterministic packet buffers inside a single linecard at high line rate for a large number of active flows. More specifically, in this chapter, we present novel reservation-based architectures where a reservation table serves as a memory management module to select one bank out of all available banks to store an arriving packet such that it will not cause any conflicts at the time of packet arrival or departure. In order to avoid memory conflicts, these architectures need to use about three times the number of DRAM banks, which is a moderate increase considering the extremely low price of current DRAM banks. [1] Moreover, the number of interleaved DRAM banks required to implement the proposed packet buffer architecture is *independent* of the number of logical queues, yet the proposed architecture can achieve the performance of an SRAM implementation and scale to growing packet storage requirements in routers while matching increasing line rates.

[1] As of this writing, 4GB of DRAM (4Gb/bank$\times$8 banks) costs under \$20, i.e. over 200MB/\$.

6.3 System Model - Packet Buffer Abstraction

In this section, we describe the packet buffer system model. First, we introduce a packet buffer abstraction. Throughout this chapter we assume that all incoming variable-size packets are segmented into fixed-size packets/cells, or simply packets/cells, and reassembled when leaving the router, which is a function supported by most current routers. We also assume that time is slotted, where a time slot is defined to be the amount of time that it takes for a (fixed-size) packet/cell to arrive (or depart) at a given line rate. For example, at 40 Gb/s the time slot for a 40-byte packet is 8 ns. For the rest of the chapter, we will simply use packets to represent fix-sized packets/cells. A packet buffer implemented in a linecard temporarily stores arriving packets from its input ports in order to mitigate congestions on the network links and also provides fast retransmission when a packet is lost on its egress link. DRAM banks are commonly utilized to provide storage for packet buffers in linecards. However, the DRAM access rate is much lower than the line rate. Our packet buffer takes the advantage of the deterministic packet buffer assumption, so that by providing a small amount of redundant DRAM banks, it guarantees that all arriving packets to a packet buffer can be admitted and all departing packets from a packet buffer can be sent out on time.

The deterministic packet buffer assumption is as follows. At each time slot t, at most one new packet p arrives at the packet buffer with a future departure time $T_d(p)$, and at most one existing packet q departs from the packet buffer in the current time slot (i.e. $T_d(q) = t$), with both occurring in the same time slot in the worst case. The departure times of packets are assumed to be *known* and *fixed* at the time of packet arrivals, which is a common assumption in building input queued systems that can emulate output queued systems. There are no restrictions on what departure times are assigned, except that the departure times for different packets are unique to ensure that only one packet departs per time slot. Like previous work on fast packet buffers [29, 43, 51, 89], this packet buffer consists of an arbitrary number of logical queues. But instead of assuming random access, we assume deterministic access based on pre-determined departure times. We refer to this packet abstraction as the *single-write-single-read deterministic packet buffer* model, or *deterministic packet buffer* model for short. A fundamental problem for buffers with deterministic departure is that packets arriving or departing at

about the same time may need to access memory banks that conflict at *write* or *read* time, since it takes a DRAM bank several time slots to finish a write or read transaction. In our proposed reservation-based packet buffer, we shall prove that with a small number of DRAM banks and an efficient bank selection algorithm, all memory conflicts can be resolved.

The stability of a buffer architecture requires that the amount of buffers needed to provide reliable storage is finite. The stability of the reservation-based packet buffers are achieved by ensuring that all the packets depart at their pre-determined departure times. Therefore, only a finite number of packets are stored in the packet buffer until their departure times. If the arrival traffic is admissible, which means that the packet arrival rate is no larger than the departure rate, then our deterministic reservation-based packet buffer architectures are stable. In the rare event that the incoming traffic becomes inadmissible by overshooting with a large number of packets at certain input ports due to oversubscribing bursty arrivals, more DRAM banks are required to buffer the overloaded traffic at a linecard. It will be evident later in the chapter that if the maximum arrival overshooting factor is $\mathtt{H_{os}}$, the total number of DRAM banks required at the line card is increased by a factor of $\mathtt{H_{os}}$ as well.

6.4 Deterministic Frame-sized Packet Buffer

We next describe the deterministic frame-sized reservation-based packet buffer architecture, as depicted in Figure 6.1. In the frame-sized packet buffer, the packets stored in DRAM banks are managed in groups called frames. The size of a frame is the same as the number of time slots it takes for a DRAM bank to finish a memory transaction.

6.4.1 Architecture

To provide bulk storage, our architecture uses an interleaved memory architecture with multiple slower DRAM modules. In particular, let there be K memories. We denote the DRAM banks as D_1, D_2, $\cdots$, D_K, respectively. DRAM access latencies are usually much slower than the line rate. Hence, multiple time slots are required to com-

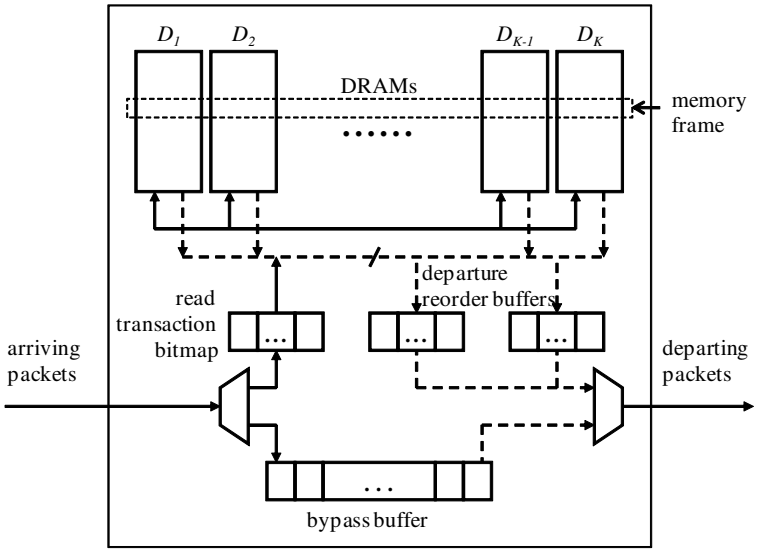


Figure 6.1: Deterministic frame-sized packet buffer architecture.

plete a memory transaction. Let b be the number of time slots required to complete a single DRAM *write* or *read* transaction. Once a memory transaction has been initiated to a DRAM, that memory will remain *busy* for b time slots until the memory transaction completes. Specifically, if a memory transaction is initiated to a particular memory bank D_i at time slot t, then the memory bank D_i is said to be *busy* from time slot t to time slot $t+b-1$. We assume that SRAM can write an arriving packet and read a departing packet in each time slot, which is a common assumption in packet buffer designs. A time interval that covers b time slots is defined as a *frame*. A time slot t is said to belong to frame $f = \lceil \frac{t}{b} \rceil$. For example, time slots $t = 1,2,\ldots,b$ belong to frame $f = 1$, time slots $t = b+1, b+2, \ldots, 2b$ belong to frame $f = 2$, etc. Each of the K memory banks is segmented into several cells, with each cell the size of one packet. The j^{th} cell in bank D_i is denoted as $C_i(j)$. We define a stripe or a *memory frame* as

$$\mathtt{S}_{\mathtt{DRAM}}(j) = \bigcup_{i=1}^{K} C_i(j). \tag{6.1}$$

Thus the j^{th} memory frame, which is the collection of the j^{th} cells from all DRAM banks, is of size K packets. Packets with the same departing time frame will be stored in the same memory frame.

6.4.2 Packet Access Conflicts

For each arriving packet we must ensure that it can be assigned to a DRAM bank. To make sure that there is always an available DRAM bank to write such a packet, we must consider three types of conflicts.

- *Arrival write conflicts*: At the current frame f, there can be at most b packets that need to be written into the DRAM banks. Each packet must be written to a different memory. We refer to this type of conflicts as *arrival write conflicts*. For each arriving packet that needs to be written in the current frame f, there can be at most $b-1$ arrival write conflicts.

- *Arrival read conflicts*: At the current frame f, there can be at most b packets that must be read from the DRAM banks for departure. The b DRAM banks assigned to the newly arriving packets cannot already hold packets that must be read in the current frame for departure. We refer to this type of conflicts as *arrival read conflicts*. Since there can be at most b packets that must be read in the current frame for departure, there can be at most b arrival read conflicts.

- *Departure read conflicts*: For each arriving packet p that must be written into the DRAM banks in the current frame f, that packet will need to be read eventually in a future frame d for departure. At this future frame d, there may be other packets with the same departure frame. Therefore, the memories assigned to these other packets cannot be assigned to the current packet p. We refer to this type of conflicts as *departure write conflicts*. Since there can be at most b packets departing in any frame, there can be at most $b-1$ departure write conflicts.

Therefore, by the pigeonhole principle, we need at least $[[(b-1)+b+(b-1)]+1]$ memories to avoid all three types of conflicts. That is, we must have $K \geq 3b-1$, which is a small constant that is independent of traffic arrival and the number of logical queues. Thus we have the following theorem.

Theorem 1. *With at least* $3b-1$ *DRAM banks, where* b *is the number of time slots it takes for a DRAM bank to finish one memory transaction, it is always possible for a deterministic frame-sized reservation-based packet buffer to admit all the arrival packets and write them into DRAM banks based on their departure times.*

Proof. Since each packet has a unique arrival time and a unique departure time, only b packets need to be written to the DRAM banks at each frame, and only b packets need to be read from the DRAM banks at each frame. Therefore, there can only be $b-1$ arrival write conflicts, b arrival read conflicts, and $b-1$ departure read conflicts. By the pigeonhole principle, we need one more memory to resolve all three conflicts. Therefore, $K \geq 3b-1$ will suffice. □

The packets in a memory frame are moved to one of the departure reorder buffers implemented in SRAM before their departure times. It is guaranteed that the read transactions do not conflict with the write transactions since the free memory banks for write operations are determined after the read operations in the same time slot. There are two departure reorder buffers, each of which is capable of buffering up to b packets, which is the maximum number of departing packets in a frame. The departure reorder buffers operate alternatively, with one buffer reading packets from a memory frame in DRAM banks for later departure while the second buffer departing packets it stores. In the next frame, the second buffer becomes empty and is rotated to read packets from the next memory frame in DRAM banks, while the first buffer begins to depart the packets it stores. With the help of the deterministic packet buffer model, the packets departing in a certain time frame are known and deterministic. The entries in a departure reorder buffer are circularly indexed corresponding to packet departure times, so that a packet in a memory frame is deterministically written to a specific location in a departure reorder buffer based on its departure time. Later the packets in a departure reorder buffer can depart in the order of earliest-departure-time-first.

Based on this operational model, it is easy to see that the minimum round-trip latency for storing and retrieving a packet to and from one of the DRAM modules is 3 frames or $3b$ time slots. One frame is for writing the packet to a DRAM bank. Another frame for writing into one of the departure reorder buffers. Finally, one more frame for departing from the departure reorder buffer, since there are at most b packets stored in a departure reorder buffer. Thus, excluding the arrival time slot, a packet with a departure time less than $3b$ time slots away *may not* be retrieved on time for departure. Therefore, we will only write a packet into a DRAM module if its departure time is at least $3b$ time slots away. The other packets with departure time less than $3b$ time lots away are written

to a bypass buffer in order to guarantee on time departure. The structure of the bypass buffer is shown in Figure 6.2. It is implemented as a single circular linked list. Due the the deterministic packet buffer design, each entry in the bypass buffer corresponds to a specific time slot. For an arriving packets with departure time earlier than $3b$ slots away, it is deterministically stored in a bypass buffer location based on its departure time. Such an operation is managed by the packet locator implementing simple arithmetic logic to circular-index the bypass buffer entries. A head pointer locates the entry corresponding to the current departure time. Upon packet departure, the packet pointed by the head pointer is removed to the egress port, and the pointer moves to the next entry in the bypass buffer. It is worth noting that an entry in the bypass buffer may be empty if the packet with the corresponding departure time is actually stored in a departure reorder buffer, in which case the packet arrived at the packet buffer more than $3b$ time slots ago.

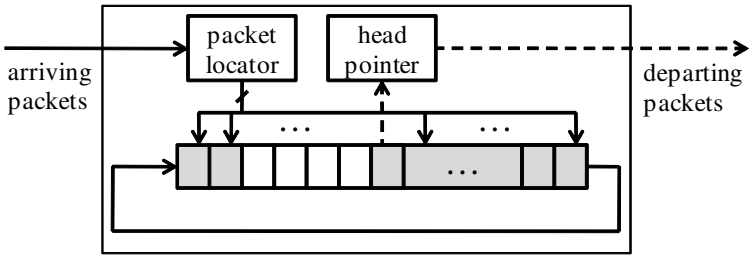


Figure 6.2: Bypass buffer architecture.

6.4.3 Memory Management Implementation

In this section, we address implementation issues for the frame-sized packet buffer architecture.

DRAM Selection Logic

We first consider the implementation of the memory management control logic for selecting an available DRAM module for storing a packet.

To find a compatible memory for inserting an incoming packet, we maintain a two-dimensional *read transaction* bitmap I to indicate which DRAM module has al-

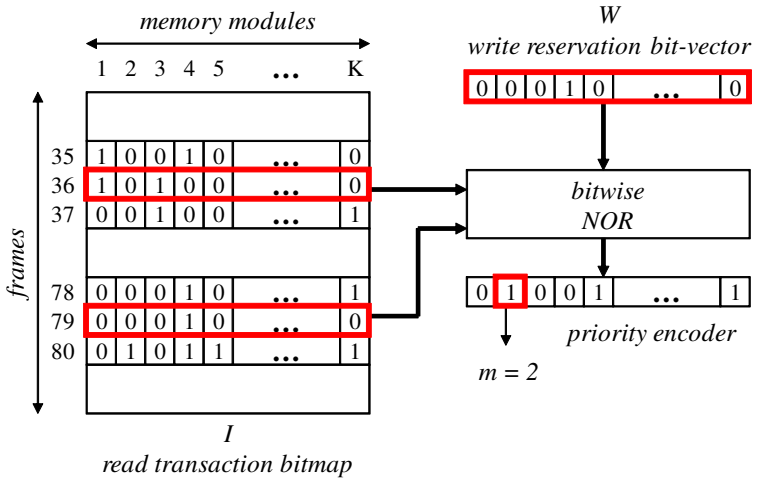


Figure 6.3: DRAM selection logic for frame-sized packet buffer.

ready stored a packet that must be *read* at the corresponding frame for departure. Each *row* in the bitmap corresponds to a frame, and each *column* of the bitmap corresponds to a DRAM module. There are exactly $3b-1$ columns in the bitmap, corresponding to the $3b-1$ DRAM modules. To avoid an infinite number of rows, we limit the furthest departure frame into the future that we need to consider, and we use *circular indexing* to map a frame to a row in the bitmap. In particular, let X be the furthest number of departure frames into the future that we consider. Then, via circular indexing, a frame f is mapped to the r^{th} row in the bitmap where $r = f \mod X$. We defer to the next section to discuss the maximum X that we should consider. For the ease of discussion in this section, we assume there is a one-to-one correspondence between a frame and a row in the bitmap.

Therefore, each row in the read transaction bitmap corresponds to a frame and each column corresponds to a memory module. We use $I(f,m)$ to indicate whether or not the m^{th} DRAM module already has a stored packet that must be read at the f^{th} frame slot. That is,

$$I(f,m) = \begin{cases} 1, & \text{if yes;} \\ 0, & \text{otherwise.} \end{cases} \tag{6.2}$$

We use the notation $I(f)$ to denote the row-vector at the f^{th} row of the bitmap. All bitmap locations are initialized to 0's. This bitmap structure is depicted in Figure 6.3.

Given this definition of the bitmap, there is a natural mapping to an SRAM implementation, for example using a $3b-1$ wide SRAM word and associating SRAM address location f to $I(f)$.

In addition to the read transaction bitmap I, we maintain a *write reservation* bit-vector W. This bit-vector has $3b-1$ bit locations. At the current frame, we use $W(m)$ to indicate if the m^{th} memory module has been assigned to an arriving packet. This bit-vector can be maintained in registers. It is initialized to all 0's at the beginning of a frame to indicate that no memory selection has been made yet in the frame.

At each frame f, we may need to write at most b newly arriving packets into DRAM banks. For each of these b packets, we need to find a compatible DRAM module that avoids the three types of conflicts discussed in Section 6.4.1, namely arrival write conflicts, arrival read conflicts, and departure read conflicts. We use a simple greedy algorithm that selects the first compatible memory module for each packet. In particular, for each newly arriving packet p that must be written in the current frame f, we do the following:

- We use W to check for arrival write conflicts, and we use $I(f)$ to check for arrival read conflicts.
- Let d be the future frame slot when packet p needs to be read for departure. Then, we must also use $I(d)$ to check for departure read conflicts.
- To determine if a memory module m is compatible for storing the packet, we use a Boolean check

$$\overline{W(m)+I(f,m)+I(d,m)}, \tag{6.3}$$

  which corresponds to a 3-input NOR operation. Instead of checking one memory module at a time, we can use *bitwise*-NOR operations on the bit-vectors,

$$\overline{W+I(f)+I(d)}, \tag{6.4}$$

  and feed the result into a priority encoder, which would return the index to the first compatible memory module, which is also shown in Figure 6.3.

- Let n be the memory module selected for p. Then we set $W(n) = 1$ to indicate that the memory module has been selected and cannot be selected for another packet to ensure no arrival write conflicts. We also set $I(d,n) = 1$ to indicate that this memory module will be busy with a read transaction at frame d.

Note that the above memory selection logic can be performed on-the-fly within one time slot as each new packet arrives. We do not have to wait for the whole frame of packets to arrive before finding a compatible memory module for each packet. By Theorem 1, we know that with $K = 3b - 1$ memory modules, a compatible memory can always be found for an arriving packet.

Size of the Bitmap

As mentioned in the previous section, we limit the number of rows in the bitmap to correspond to the furthest number of departure frames into the future that we wish to consider. The limit is set relative to the size of the packet buffer. One common rule of thumb is to size the packet buffer according to the product of the average round-trip-time and the line rate [96] of the maximum admissible traffic. Suppose the size of the packet buffer is equivalent to T packets, then it is reasonable to limit the furthest departure time slot relative to the current time slot to be T, which in turn corresponds to limiting the furthest departure frame relative to the current frame to be

$$\texttt{num}_{\texttt{FRAME}} = \left\lceil \frac{T}{b} \right\rceil . \tag{6.5}$$

Each row in the bitmap has $3b-1$ bits. Thus the total size of the bitmap can be calculated as

$$\texttt{size}_{\texttt{BITMAP}} = \texttt{num}_{\texttt{FRAME}} \times (3b-1) = \left\lceil \frac{T(3b-1)}{b} \right\rceil , \tag{6.6}$$

which in turn is only about 3 bits per packet (about 1% of the size of a 40-byte packet). Assuming a round-trip-time of 250 ms and a line rate of 40 Gb/s, the common rule of thumb for sizing packet buffers would correspond to a memory requirement of about $T = 3 \times 10^7$ packets, which corresponds to a bitmap with approximately 12 MB. Such a bitmap requires several external SRAMs. In the next section, we will provide a block-sized packet buffer that dramatically reduces the SRAM requirement for managing packets in the packet buffer, so that on-chip SRAM would be sufficient.

Location of Packets in a DRAM Module

In this section, we address several additional details that must be considered. First, when a packet p is assigned to a memory module m, we need a way to assign the actual memory location to store the packet. Then, later at a future frame f, we need a way to determine

1. which packets must be read (up to b packets),
2. which memory modules to retrieve the packets, and
3. which memory location within each memory module to read.

At frame slot f, we can use the read transaction bitmap I to answer (2). That is, we need to retrieve a packet from memory module m if $I(f,m) = 1$. By construction, there can be at most b memory modules with $I(f,m) = 1$.

To answer (1) and (3), we need to consider the method we use to allocate memory locations within each memory module. One approach would be to maintain a lookup table that stores for each time slot a pointer to the memory location where the corresponding packet is stored. The lookup table could be indexed by time slots when a packet must be read for departure. However, such a lookup table may be too large and expensive. Instead, we *partition* the packet locations in DRAM banks by memory frames, with each memory frame corresponding to a departure time frame. Suppose a packet has been assigned to the memory module m, with departure time frame f. Then it is stored in a cell in DRAM bank m that is part of a memory frame that matches the departure time frame f. By ensuring that all conflicts have been resolved in the memory selection step, we are guaranteed that at most one packet p will occupy a cell in a memory bank. Then, for packet retrieval at frame f, we look up the read transaction bit-vector $I(f)$ to determine which memory bank contains a packet that must be read at this frame, i.e., where $I(f,m) = 1$, and read the packet location in the memory frame corresponding to f in the bank m. We defer to Section 6.5.4 to discuss the total size of the DRAM banks.

6.5 Deterministic Block-sized Packet Buffer

The frame-size packet buffer in Section 6.4 suffers from the problem that the total SRAM required grows linearly with line rate, which makes it less attractive for high line rates, such as OC-768. In this section we describe the deterministic block-sized reservation-based packet buffer architecture that minimizes the total SRAM requirement, as shown in Figure 6.4.

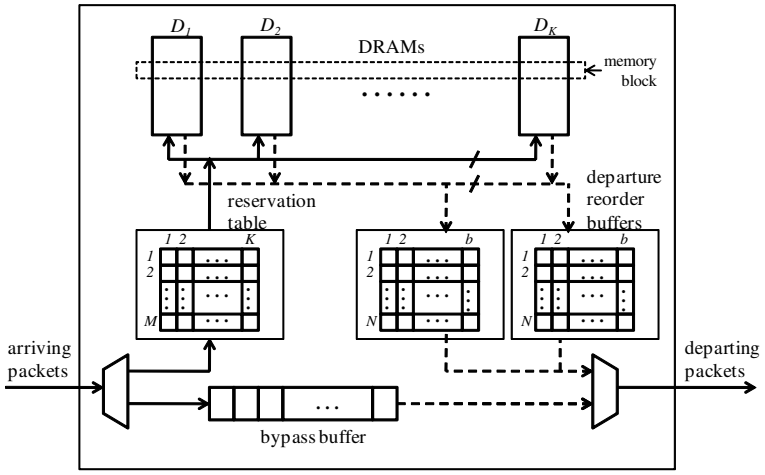


Figure 6.4: Deterministic block-sized packet buffer architecture.

6.5.1 Architecture

We use the same notations as in Section 6.4. Furthermore, in a block-sized packet buffer, each of the K memory banks is segmented into several *sections*. One section can hold up to N packets. The j^{th} section in the DRAM bank D_i is denoted as $D_i(j)$. We define a *memory block* $\mathtt{B}_{\mathtt{DRAM}}(j)$ as follows,

$$\mathtt{B}_{\mathtt{DRAM}}(j) = \bigcup_{i=1}^{K} D_i(j). \tag{6.7}$$

Thus the j^{th} memory block, which is the collection of the j^{th} sections from all DRAM banks, is of size $N \cdot K$ packets.

In the frame-sized scheme, one bit is needed to track each packet location in DRAM banks. Therefore the total SRAM requirement grows linearly with the line rate

due to the buffer sizing rule $B = \text{RTT} \times C$, which is not scalable to future high line rates. In the block-sized scheme we use a module, namely a reservation table for bookkeeping purposes. Each entry in the reservation table keeps track of the number of packets currently stored in a section in the DRAM banks. If the size of a DRAM section is N, then only $\log_2 N$ bits are required to track the number of packets in a section. On the other hand, a total of N bits are required to pinpoint exactly which packets are currently stored in a section as in the frame-sized scheme. To avoid memory conflicts during packet arrivals and departures, in the block-sized scheme the packets in a DRAM block are accessed collectively. The detailed function of the reservation table is presented in the next section.

An incoming packet is stored in a section in a DRAM bank belonging to a memory block based on its departure time. There is a one-to-one mapping from a reservation table entry to a memory section. Once a packet is stored in a memory section, the corresponding reservation table entry is updated by adding one to its current value. The arrival packets with departure times within N frames (i.e. $N \cdot b$ time slots) are written to the same memory block, where N is also the size of a DRAM section in number of packets. So there can be at most $N \cdot b$ arriving packets stored in a memory block which can potentially hold up to $N \cdot K$ packets. Upon departure, the packets in a memory block are moved to a departure reorder buffer. The algorithm for selecting a DRAM bank to store an incoming packet based on its departure time using the reservation table will be introduced in Section 6.5.4.

6.5.2 Reservation Table

The reservation table is implemented in on-chip SRAM for fast lookups and updates. The reservation table is necessary for keeping track of the packets in the DRAM banks for the following reasons. First, our architecture needs to guarantee that the packets in the DRAM banks will depart at their pre-determined departure times. Second, a finite-size DRAM bank must not admit too many arriving packets in a short time, which would otherwise cause it to overflow.

For the bitmap module in Section 6.4, although only a single bit is required per packet location, which is substantially smaller than storing whole packets, the size of the

bitmap nonetheless grows *linearly* with the number of DRAM packet buffer locations. Larger packet buffers would be required in the future to match increasing line rates, and the bitmap would increase proportionally as a consequence.

Instead of using N bits as a bitmap to represent N packet locations in a memory section, the block-sized packet buffer uses a counter of size $\log_2 N$ bits to keep track of the actual number of packets present in these N packet locations. Since N can be represented using $\log_2 N$ bits, $\log_2 N$ bits are sufficient for maintaining a total of N packet locations. These packets will be reordered upon departure in a departure reorder buffer. We denote the size of the reservation table as $\texttt{size}_\texttt{R}$. Let the packet buffer be of size N_{max} packets. Then the linear bitmap module in Section 6.4 would require a size of N_{max} bits. However, our reservation table is only of size

$$\texttt{size}_\texttt{R} = N_{max} \cdot \frac{\log_2 N}{N}. \tag{6.8}$$

If we choose N to be $N = \alpha N_{max}$, where α is a constant and $\alpha < 1$, (i.e., the size of a memory section is a fixed portion of the packet buffer, which is indeed the case), then

$$\texttt{size}_\texttt{R} = N_{max} \cdot \frac{\log_2 (\alpha N_{max})}{\alpha N_{max}} = \frac{\log_2 N_{max}}{\alpha} + \frac{\log_2 \alpha}{\alpha}. \tag{6.9}$$

Therefore the size of the reservation table grows logarithmically as the DRAM bank size grows.

The reduction of the size of the reservation table is achieved at the cost of larger departure reorder buffers implemented in SRAM. Intuitively, since we only count the number of packets in a memory section, information on the relative orderings of the packets is lost. For a fixed size buffer, the larger N is, the larger the constant α becomes. Therefore the size of the reservation table $\texttt{size}_\texttt{R}$ decreases as N increases as evident in Equation (6.9). On the other hand, as N increases the size of the departure reorder buffers increases since there are more packets in a memory block to be reordered before departure. In Section 6.7, we will show that for a fixed size buffer there is an optimal value of N such that the total SRAM requirement is minimized.

6.5.3 Packet Access Conflicts

As in the frame-sized packet buffer, there needs to be enough DRAM banks in the system to store all the arriving packets and to ensure packets depart at their departure

times. First, let's consider the three kinds of memory conflicts.

- *Arrival write conflicts:* See Section 6.4.2.
- *Arrival read conflicts:* See Section 6.4.2.
- *Departure read conflicts (or overflow conflicts):* When a new packet arrives, it will be stored in a memory block based on its departure time. Incoming packets with departure times within N frames are stored in the same memory block. There are at most $N \cdot b$ packets departing within N frames. However, a memory section can store only N packets at most. Without an effective algorithm to choose DRAM banks to store the incoming packets, in the worst case more than N packets may be admitted to the same memory section, which would cause the bank to overflow.

Theorem 2. *With at least* $3b-1$ *DRAM banks, where* b *is the number of time slots it takes for a DRAM bank to finish one memory transaction, it is always possible for a deterministic block-sized reservation-based packet buffer to admit all the arrival packets and write them into memory blocks based on their departure times.*

Proof. It is already clear that we need $b-1$ banks to resolve the arrival conflicts and b banks to resolve the departure conflicts. We only need to prove that $b-1$ extra DRAM banks are sufficient to resolve the overflow conflicts.

With $K = 3b-1$, a memory block can hold a total of up to $N \cdot K = (3b-1) \cdot N$ packets. For any DRAM bank, one of its sections can hold at most N packets. On the other hand, there are at most $N \cdot b$ incoming packets within N frames. At any given time slot, there are at most $b-1$ arrival conflicts and b departure conflicts. So there are at least b free DRAM banks to store the current incoming packet. For choosing a bank to store the current incoming packet out of the b free banks, we implement the following algorithm.

Water-Filling Algorithm: *For an incoming packet* p *with departure time slot* $T_d(p)$*, we check the corresponding memory block. The number of packets stored in the sections of the block in the* b *free DRAM banks are denoted as* $\Psi(D_{\gamma_1})$, $\Psi(D_{\gamma_2})$, $\cdots$, $\Psi(D_{\gamma_b})$ *respectively. We choose a DRAM bank* D_{γ_i} *to store the packet* p *based on the*

following,

$$i = \arg \min_{1 \le j \le b} \Psi(D_{\gamma_j}). \tag{6.10}$$

To prove the theorem, let's consider the following scenario. After the current incoming packet is written to one of the b free DRAM banks, the bank stays busy for the next b time slots. After b time slots the same b DRAM banks are free again for the newly arriving packet, which happens with non-zero probability. If a packet with departure time corresponding to the same memory block arrives, the packet will then be written to the same block in one of the b free banks (but maybe to a different bank among the b banks depending on which one contains the fewest packets). Therefore, the worst case happens when all of the packets with departure times corresponding to the same memory block are to be stored in the same b DRAM banks. In this case, there are $N \cdot b$ packets and the memory block in the b banks can store up to $N \cdot b$ packets. The water-filling algorithm above guarantees that with a storage space of size equal to $N \cdot b$ packets, we can always store all of the $N \cdot b$ arriving packets, since the most empty bank is filled first. So in the worst case, $b-1$ extra DRAM banks are sufficient to solve the overflow conflicts, but fewer banks cannot solve such conflicts. Therefore, with the water-filling algorithm, a total of $3b-1$ DRAM banks is sufficient to store all arriving packets. The memory block to store a packet is decided by the packet departure time. □

As in the frame-sized packet buffer, there are two departure reorder buffers in the block-sized packet buffer. Each buffer is capable of buffering up to $N \cdot b$ packets, which is the maximum number of packets in a memory block. The packets are read from DRAM banks to a departure reorder buffer following a simple algorithm: We choose the bank with the largest number of packets in its corresponding section among all of the free banks based on the departure time. There are at most N packets in a memory section. On the other hand, there are $N \cdot b$ time slots to read the packets from a memory block to a departure reorder buffer. With the assumption that a read transaction in DRAM takes b time slots, it is always possible to read all the packets from a memory section even if the section is full. As in the frame-sized buffer, the packets are written to deterministic locations in a departure reorder buffer based on their departure times. The packet with the earliest departure time leaves the buffer first.

Based on the operation model, in the worst case the minimum round-trip latency for storing and retrieving a packet to and from one of the DRAM banks is $(2N+1)\cdot b$ time slots, including the current time slot. First, b time slots are needed to write a packet into a DRAM bank, since it takes a DRAM bank b time slots to finish a write transaction. Upon packet departure, at most $N\cdot b$ time slots are needed for reading packets from a memory block into one of the departure reorder buffers, since there are at most N packets stored in a memory section. Finally, at most $N\cdot b$ time slots are needed to depart the packets from a departure reorder buffer. Therefore for an incoming packet with a departure time within $(2N+1)\cdot b$ time slots away, it *may not* be retrieved in time for departure if written to a memory bank. To guarantee deterministic packet departures, a bypass buffer in SRAM is required to store these packets. The structure of the bypass buffer is similar to the one in the frame-sized packet buffer shown in Figure 6.2.

6.5.4 Memory Management Implementation

In this section, several memory management issues in the block-sized packet buffer are addressed.

DRAM Selection Logic

The DRAM selection logic is shown in Figure 6.5. There are K columns and M rows in the reservation table. Each column corresponds to one DRAM bank and each row corresponds to one memory block. We denote the entry in the i^{th} row and j^{th} column of the reservation table as $R(i,j)$, which represents the number of packets in bank D_j that belongs to the i^{th} block, i.e. section $D_j(i)$. For each DRAM bank D_j, we use a single bit $W(j)$ to maintain its activity. If D_j is busy, we set $W(j)=1$; otherwise $W(j)=0$. These bits form a write candidate vector W.

As an example, in Figure 6.5 for any arriving packet p with arrival time belonging to block s and departure time belonging to block $s+u$, the entries in both row s and row $s+u$ in the reservation table are retrieved. We use $R(s)$ to represent the s^{th} row in the reservation table. From $R(s)$, the write candidate vector W is constructed. For selecting a DRAM bank to store the arriving packet, we follow these steps:

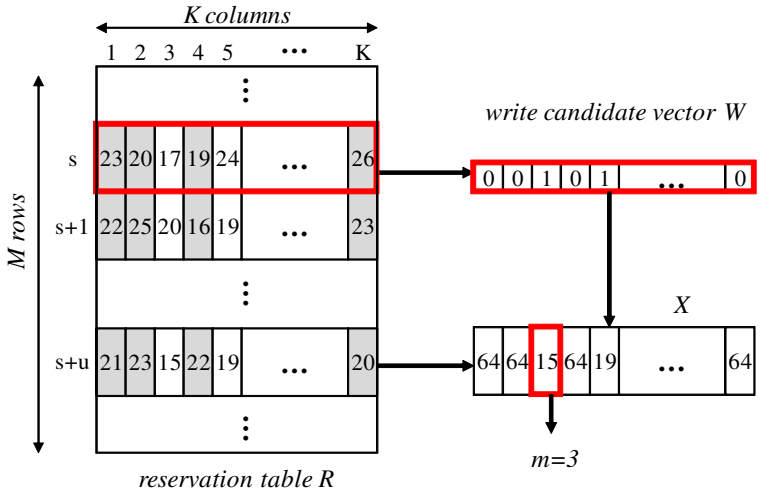


Figure 6.5: DRAM selection logic for block-sized packet buffer.

- Use the write candidate vector W to check arrival write conflicts and arrival read conflicts in the DRAM banks. The vector W can be maintained separately or constructed when necessary from $R(s)$.

- The row in the reservation table $R(s+u)$ is retrieved.

- To find the memory bank for storing the incoming packet, we compare W and $R(s+u)$ to find the most empty free bank.

$$X(i) = \begin{cases} R(s+u,i), & \text{if } W_i = 1; \\ N, & \text{if } W_i = 0, \end{cases} \tag{6.11}$$

  where $X(i)$, $R(s+u,i)$, and W_i are the i^{th} entries of vectors X, $R(s+u)$, and W respectively. N is the maximum number of packets that can be stored in a DRAM section, which is assumed to be 64 in Figure 6.5.

- Find the smallest entry $X(m)$ in vector X with ties broken arbitrarily,

$$m = \arg\min_{\phi} X(\phi). \tag{6.12}$$

- According to Theorem 2, the incoming packet p will then be stored in memory bank D_m.

Size of DRAM Banks

In this section, the size of the DRAM banks and the issue of locating packets in the DRAM banks upon departure are addressed. For an incoming packet, after updating the entry in the reservation table based on its departure time, the packet is written to the corresponding block in the DRAM banks. Since the packets are assumed to be of fixed sizes, we partition memory banks into cells with each cell the size of a packet. Upon departure, all the packets within a block of the DRAM banks are moved to one of the two departure reorder buffers. The packets depart in order of their departure times from the departure reorder buffers. The total size of the DRAM banks can be calculated as follows. Let T_{max} be the number of time slots into the future we need to consider. Circular indexing is used to map time slots to memory blocks. Let the average packet size be P. Then the size of all DRAM banks, $\texttt{size}_{\texttt{DRAM}}$, is

$$\texttt{size}_{\texttt{DRAM}} = T_{max} \cdot P \cdot \frac{3b-1}{b}. \tag{6.13}$$

There should be enough memory blocks to cover the furthest departure time slot we need to consider. Using the buffer sizing rule in Section 6.4.3 to size the packet buffer in accordance to the product of the average round-trip-time (RTT) and the line rate (C), we have

$$T_{max} = \text{RTT} \cdot C/P. \tag{6.14}$$

So,

$$\texttt{size}_{\texttt{DRAM}} \approx 3 \cdot \text{RTT} \cdot C. \tag{6.15}$$

For example, assuming RTT $=$ 250 ms and $C = 40$ Gb/s, the total size of DRAM banks is 30 Gb or about 3.75 GB to store packets of total size up to 1.25 GB, which is a modest amount of DRAM with current technology.

SRAM Size

First, let's calculate the size of the reservation table. Each row of the reservation table represents N frames, each of which consists of b time slots. Also, there are K columns in the reservation table representing K DRAM banks. Let the number of

DRAM banks be $K = 3b - 1$ which is sufficient to resolve all memory conflicts. The maximum number of packets in the packet buffer $\texttt{num}_{\texttt{PACKET}}$ is

$$\texttt{num}_{\texttt{PACKET}} = T_{max} = \frac{\text{RTT} \times C}{P}. \tag{6.16}$$

The total number of frames in the packet buffer $\texttt{num}_{\texttt{FRAME}}$ is

$$\texttt{num}_{\texttt{FRAME}} = \frac{\text{RTT} \times C}{bP}. \tag{6.17}$$

The number of blocks in the packet buffer $\texttt{num}_{\texttt{BLOCK}}$ is

$$\texttt{num}_{\texttt{BLOCK}} = \frac{\text{RTT} \times C}{bPN}. \tag{6.18}$$

It is reasonable to limit the number of rows in the reservation table to the number of memory blocks. Each entry in the reservation table is of size $\log_2 N$ bits. So the size of the reservation table $\texttt{size}_{\texttt{R}}$ can be calculated as

$$\begin{aligned} \texttt{size}_{\texttt{R}} &= \texttt{num}_{\texttt{BLOCK}} \cdot K \cdot \log_2 N \\ &= \frac{\text{RTT} \cdot C}{bPN} \cdot (3b-1) \cdot \log_2 N \\ &\approx 3 \cdot \frac{\log_2 N}{N} \cdot \frac{\text{RTT} \cdot C}{P}. \end{aligned} \tag{6.19}$$

The size of each departure reorder buffer is the same as the maximum number of packets in a memory block. We use $\texttt{size}_{\texttt{D}}$ to denote the size of each buffer. Thus,

$$\texttt{size}_{\texttt{D}} = N \cdot b \cdot P. \tag{6.20}$$

A packet is only written to a bypass buffer if its departure time is less than or equal to $(2N+1) \cdot b$ time slots away. All packets in a bypass buffer will depart in the next $(2N+1) \cdot b$ time slots. Thus the size of the bypass buffer $\texttt{size}_{\texttt{BP}}$ is

$$\texttt{size}_{\texttt{BP}} = (2N+1) \cdot b \cdot P. \tag{6.21}$$

For example, let the round-trip-time be 250 ms [13] and the line rate be 40 Gb/s. We choose $N = 64$ and average packet size P is 40 bytes. Then the total size of the reservation table is about 9 Mb or just 1.2 MB. The total size of the departure reorder buffers is about 80 KB. The size of the bypass buffer is only 81 KB. They can be easily implemented with small SRAMs. Moreover, the sizes of these SRAMs are independent of the arrival traffic patterns, the number of flows and the number of priority classes in each flow.

6.6 Randomized Block-sized Packet Buffer

In this section a new architecture, randomized block-sized packet buffer, is presented. It sacrifices the deterministic service guarantee to further reduce the SRAM required. As shown in Figure 6.6, there is no reservation table to manage the arrival packets. Instead, there are write request queues to temporarily buffer the incoming packets before writing them to DRAM banks. In the randomized packet buffer architecture, only statistical guarantees are provided. However, we will later show that the statistical guarantees are strong enough for most networking applications.

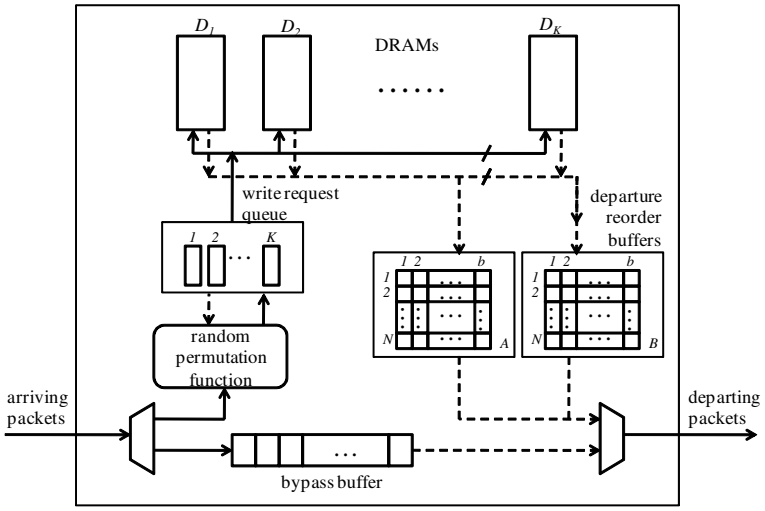


Figure 6.6: Randomized block-sized packet buffer system architecture.

In the randomized block-sized packet buffer system, an incoming packet p is assigned to a DRAM bank out of the K banks by a random permutation function. We denote the permutation function as π. The output of the function π is an integer from 1 to K. For each DRAM bank, there is a write request queue buffering all the packets waiting to be written to the DRAM bank, adding to a total of K queues. If $\pi(p) = i$, the incoming packet will be inserted to the end of the i^{th} write request queue. The packet will be written to DRAM bank D_i when there is no packet ahead of it in the write request queue and D_i is idle. The block in a DRAM bank that stores the packet p is decided by the departure time of the packet as in a block-sized packet buffer.

We cannot simply *stripe* incoming packet locations across the memory banks based on their departure time order. i.e., we cannot map a packet with departure time t

to the k^{th} DRAM bank where $k = t \mod K$. The random permutation function π is necessary for the following reason. The departure time of arriving packets may not follow striping order since the corresponding packets may have different service requirements. Therefore, there are pathological cases in which consecutively arriving packets with different departure time may be written to the same DRAM bank. In these cases, the packets will enter the same write request queue, causing the queue to grow indefinitely in order to provide a satisfactory overflow probability. It is worth noting that the same pathological cases can be generated by adversaries to attack the packet buffer system. With a random permutation function, we can safely assume that the pathological cases only happen with negligible probability for non-adversarial arrivals [100]. For adversarial arrivals, the probability of a successful attack is also negligible if the random permutation function is unknown to the attackers.

The random permutation function randomly distributes arriving packets based on their departure time across the K memory banks so that with high probability each memory bank will receive one out of K arriving packets on average. This is achieved by applying a pseudo-random permutation function

$$\pi : \{1,\ 2, \ldots\ , T_{max}\} \rightarrow \{1,\ 2, \ldots\ , K\} \tag{6.22}$$

to the departure time of packet to obtain a permuted memory location, where T_{max} is the furthest departure time slot considered in the function π. The actual departure time slot t can be mapped to a virtual time slot $t' \in [1, T_{max}]$ according to $t' = t \mod T_{max}$. A packet with virtual departure time t' will be stored in the i^{th} DRAM bank, if $i = \pi(t') + 1$, and its memory block location is $\mathtt{B}_{\mathtt{DRAM}} = \lceil \frac{\pi(t')+1}{bN} \rceil$.

The write request queues are implemented as cyclic FIFO queues. When a queue is full and there are more packets arriving, the queue overflows. The length of the queues should be large enough to guarantee a low overflow probability. With the help of an effective random permutation function, we assume each arriving packet is buffered into any one of the K write request buffers uniformly-at-random with an equal probability of $1/K$. Therefore the arrival packet traffic coming into each queue follows a geometric distribution with the same probability. Let L be the number of packets in a queue. First assume the size of the queue is large enough to hold all the packets and that there is one new packet arriving in any time slot. In m frames there are mb arrival packets, where b

is the number of time slots for a DRAM bank to finish a write or read transaction. Let the number of DRAM banks be $K = 3b - 1$. Let's assume that in any time slot a packet departs from a DRAM bank to a departure reorder buffer with equal probability $1/K$, and it takes a bank b times slots to read a packet. So a DRAM bank will be available for writing packets only $mb - mb^2/K$ time slots in every m frames (i.e. mb time slots), since the read transactions take mb^2/K time slots during this period of time. Thus in every mb time slots, a write request queue can write at most $m - mb/K$ or about $2m/3$ packets to the corresponding DRAM bank. Therefore, the number of packets in a queue can be regarded as a geometric arrival queue with a constant departure rate of $2m/3$ packets per mb time slots. The probability for a queue to have l packets after m frames can be calculated as follows.

$$\texttt{Prob}\{L = l\} = \binom{mb}{l + \lceil 2m/3 \rceil} \left(\frac{1}{K}\right)^{l+\lceil 2m/3 \rceil} \left(1 - \frac{1}{K}\right)^{mb-l-\lceil 2m/3 \rceil}. \tag{6.23}$$

Equation (6.23) shows that in order to have l packets left in a write request queue after m frames, there has to be $l + 2m/3$ packets arriving at the queue. Let the overflow probability $\texttt{Prob}_{over}$ be the probability that $l > L_{max}$, where L_{max} is the maximum number of packets that can be stored in a write request queue. Thus,

$$\begin{aligned}
\texttt{Prob}_{over} &= \sum_{l=L_{max}+1}^{\infty} \texttt{Prob}\{L = l\} \\
&= \sum_{l=L_{max}+1}^{mb-\lceil 2m/3 \rceil} \texttt{Prob}\{L = l\} \\
&= \sum_{l=L_{max}+1}^{mb-\lceil 2m/3 \rceil} \binom{mb}{l + \lceil 2m/3 \rceil} \left(\frac{1}{K}\right)^{l+\lceil 2m/3 \rceil} \times \left(1 - \frac{1}{K}\right)^{mb-l-\lceil 2m/3 \rceil} \\
&= \sum_{l=L_{max}+\lceil 2m/3 \rceil+1}^{mb} \binom{mb}{l} \left(\frac{1}{K}\right)^{l} \left(1 - \frac{1}{K}\right)^{mb-l}.
\end{aligned} \tag{6.24}$$

The overflow probability for geometric queue with departures as a function of time is shown in Figure 6.7. In this figure, L_{max} is the maximum number of packets a write request queue can store (i.e. queue size). For a fixed L_{max}, the overflow probability first increases, since it takes time for packets to aggregate in a queue. Then the overflow probability slowly decreases, since the arrival rate to the queue is about $1/K \approx 1/3b$ packet per time slot, while the queue writes packets to a DRAM bank at a rate of about

$2/3b$ packet per time slot. Thus with high probability there are more departing packets than arriving packets to a write request queue within a time window, and the overflow probability decreases. With queue of size 14 packets, the overflow probability is guaranteed to be less than 10^{-10}, which is sufficient for most networking applications. It is worth noting that the curves shown in the figure are not *smooth* due to the round up in Equation (6.24).

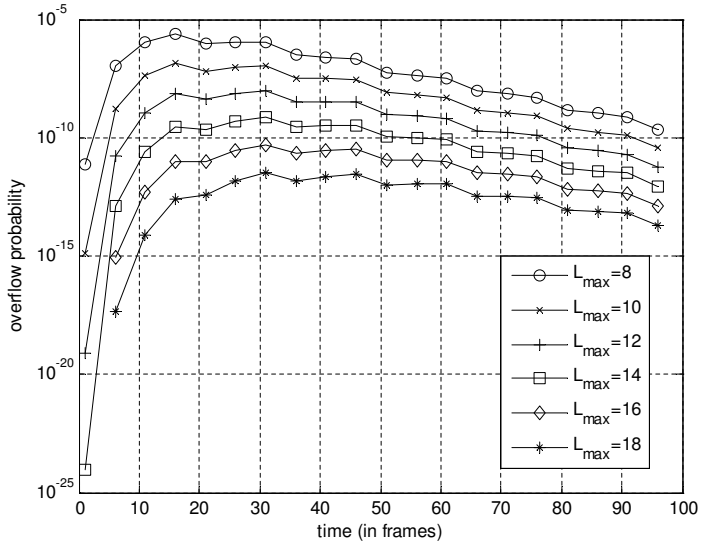


Figure 6.7: Overflow probability for a write request queue.

The total size of the write request queues $\texttt{size}_{\texttt{WQ}}$ is,

$$\texttt{size}_{\texttt{WQ}} = K \cdot L_{max} \cdot P. \tag{6.25}$$

Let the packet size be $P = 40$ bytes, $b = 16$, $K = 3b - 1$, and $L_{max} = 14$. A total of 26 KB SRAM is sufficient to support an overflow probability below 10^{-10} in the write request queues. In the worst case the write request queue contains $L_{max} - 1$ packets when a packet arrives. It takes an incoming packet $L_{max} \cdot b$ time slots to be written to a DRAM bank. Upon departure, it takes at most $N \cdot b$ time slots to move a packet to a departure reorder buffer and another $N \cdot b$ time slots to leave the departure reorder buffer. So in the worst case, the round-trip delay for a packet can be $(2N + L_{max}) \cdot b$ time slots. A bypass buffer is thus necessary for arrival packets with departure time less than $(2N + L_{max}) \cdot b$ time slots away. There can be at most one arrival packet in each time slot, so a bypass

Table 6.1: SRAM sizes (in MB) for different N with a line rate of 40 Gb/s for a deterministic block-sized packet buffer.

N	reservation table	departure buffers	bypass buffer	TOTAL
1	12	0.01	0.01	12.01
32	1.88	0.04	0.04	1.96
64	1.13	0.08	0.08	1.29
128	0.66	0.16	0.16	0.97
256	0.38	0.32	0.32	1.01
512	0.22	0.63	0.63	1.47
1024	0.12	1.25	1.26	2.62

buffer of size $\texttt{size}_{\texttt{BP}}$ where

$$\texttt{size}_{\texttt{BP}} = (2N + L_{max}) \cdot b \cdot P, \tag{6.26}$$

is sufficient to buffer all the packets. For $N = 64$, $b = 16$, $L_{max} = 14$ and $P = 40$ bytes, the bypass buffer requires only 89 KB of SRAM.

6.7 Performance Analysis

In this section, we first investigate the optimal value of N, the number of packets in a DRAM section so that the size of the SRAM is minimized in the deterministic block-based packet buffer. As shown in Section 6.5.4, the sizes of the reservation table, the departure reorder buffers, and the bypass buffers are all functions of N. As N grows, the size of the reservation table decreases, and the sizes of the departure reorder buffers and bypass buffer increase. In Table 6.1, the line rate is 40 Gb/s. We choose RTT = 250 ms, $b = 16$, and $K = 3b - 1$. The fixed packet size $P = 40$ bytes. The optimal value of N that minimizes the SRAM requirement is $N = 128$. The total SRAM needed in this optimal case is about 0.97 MB.

In Table 6.2, the line rate is 100 Gb/s. We choose RTT = 250 ms, $b = 16$, $K = 3b - 1$ and $P = 40$ bytes. The SRAM size is minimized with $N = 256$. The total

SRAM needed is about 1.57 MB. The optimal value of N that minimizes the SRAM size increases as the line rates increase.

Table 6.2: SRAM sizes (in MB) for different N with line rate at 100 Gb/s for deterministic block-sized packet buffer.

N	reservation table	departure buffers	bypass buffer	TOTAL
1	30	0.01	0.01	30.01
32	4.69	0.04	0.04	4.77
64	2.82	0.08	0.08	2.97
128	1.65	0.16	0.16	1.96
256	0.94	0.32	0.32	1.57
512	0.53	0.63	0.63	1.78
1024	0.30	1.25	1.26	2.80

It is worth noting that when $N = 1$, the reservation table degrades to the read transaction bitmap in the frame-sized packet buffer.

In Table 6.3, we compare the memory requirements for the block-sized packet buffer and the frame-sized one. The line rates are 10 Gb/s, 40 Gb/s, and 100 Gb/s. Let RTT $= 250$ ms, $b = 16$, $K = 3b - 1$, and $P = 40$ bytes. In general, the required SRAM

Table 6.3: Comparison of SRAM size requirements.

C	deterministic block-sized $N = 128$	deterministic block-sized $N = 256$	deterministic frame-sized	ratio of the optimal block-sized scheme over the frame-sized scheme
10 Gb/s	0.48 MB	0.72 MB	3 MB	16%
40 Gb/s	0.97 MB	1.01 MB	12 MB	8.1%
100 Gb/s	1.96 MB	1.57 MB	30 MB	6.5%

bank size in the block-sized packet buffer grows only logarithmically with the line rates, while it grows linearly for the frame-sized packet buffer. At a line rate of 10 Gb/s, the

optimal block-sized packet buffer demands only 16% of the total SRAM required by the frame-sized one. This ratio of the SRAM requirement decreases to 8.1% and to 6.5% at a line rate of 40 Gb/s and 100 Gb/s, respectively. The ratio only decreases as line rate grows, which is a great advantage of the block-sized packet buffer over the frame-sized packet buffer for future high line rates.

Table 6.4: SRAM requirement comparison with a prefetching-based packet buffer.

prefetching-based [43]	deterministic frame-sized	deterministic block-sized
64 MB	12 MB	1 MB

In Table 6.4, we compare the SRAM requirement of the prefetching-based packet buffer [43] with the deterministic packet buffers in this chapter. We use RTT = 250 ms and a line rate of 40 Gb/s. Other values are $b = 16$, $K = 3b - 1$, and $P = 40$ bytes. The SRAM requirement for our block-sized scheme is based on Section 6.5.4. We assume the same setting for our frame-sized scheme where one bit of SRAM is required in the bitmap for each packet. The total SRAM size in our frame-sized packet buffer is only 18.8% of the state-of-the-art SRAM/DRAM prefetching buffer scheme, while the total SRAM size in our block-sized packet buffer is only 1.5% of the prefetching buffer scheme. In the prefetching-based scheme, SRAM is required to implement head and tail caches whose sizes grow linearly with the number of logical queues. Commercial routers today support many logical queues to implement classes-of-service. With Juniper routers supporting up to 64K logical queues, the minimum SRAM required is 64 MB assuming packets are written to bulk DRAM in blocks of size 1000 bits [43]. Further, the prefetching-based schemes require complex control logic for managing the head and tail caches. Although these schemes are more general because they can handle random packet departures, we show in this chapter that the reservation-based designs are significantly more SRAM-efficient for router applications where the deterministic packet departure setting is valid.

In Table 6.5, we compare the proposed randomized block-sized packet buffer with other randomized packet buffer schemes. We assume RTT = 250 ms, the line rate

$C = 40$ Gb/s, and $b = 16$. With an average packet size of 40 bytes, the total size of DRAM banks is 30 Gb or about 3.75 GB to store packets of total size up to 1.25 GB in our randomized block-sized packet buffer scheme. The memory shown in the table are for providing an overflow probability smaller than 10^{-10}.

Table 6.5: Comparison of randomized packet buffer schemes.

schemes	[89]	[51]	randomized block-sized (this chapter)
DRAM	2.76 GB[2]	2.76 GB[2]	3.75 GB
SRAM	150 KB	239 KB	115 KB

It is worth noting that the randomized scheme in [89] doesn't guarantee in order departures across flows, since there is no reorder mechanism to ensure packet departures in their designated orders. On the other hand, both our randomized block-sized packet buffer and the scheme in [51] provide in order packet departure across different flows. Our randomized block-sized packet buffer is essentially a tradeoff between using slightly more DRAM banks to reduce the total amount of SRAM required compared to [51]. While the efficiency of such a tradeoff is debatable depending on the relative prices of DRAM and SRAM, we present our randomized scheme to show that our deterministic packet buffer architecture can be adapted to provide statistical guarantees while requiring less SRAM storage.

All the results above are based on the assumption of admissible arrivals at any moment. For non-admissible arrivals, the number of DRAM banks required can be easily derived. Consider that the traffic is overshooting at some input ports in a linecard by a factor of $\mathtt{H_{os}}$ due to oversubscribing bursty arrivals. i.e., during a short period of time, the number of packets arriving at a packet buffer is $\mathtt{H_{os}}$ times the number of packets for the maximum admissible traffic. It is desirable to design a packet buffer to be able to handle such oversubscribed bursty arrivals as long as the arrival traffic is admissible in the long term (since otherwise the packet buffer would be unstable), to

[2]The DRAM sizes are calculated for a fair comparison, since in [51, 89] a DRAM bank is assumed to work at double rate with one read and one write every b time slots, while in our randomized block-sized buffer only one memory transaction is allowed in a DRAM bank every b time slots.

provide more robust services. For bursty arrivals overshooting a linecard by a factor of $\mathtt{H_{os}}$, the size of a time slot is effectively reduced by a factor of $\mathtt{H_{os}}$ and the effective line rate is increased by a factor of $\mathtt{H_{os}}$. If the DRAM access latency stays unchanged, it would take a DRAM bank $b\mathtt{H_{os}}$ time slots to finished a memory transaction. From Equation (6.15), the total size of the DRAM banks required is increased by a factor of $\mathtt{H_{os}}$, by using $3b\mathtt{H_{os}} - 1$ memory banks. Also, the size of the reservation table, bypass buffer, and departure reorder buffers are increased by $\mathtt{H_{os}}$, in order to guarantee that no packet is dropped from the temporarily oversubscribed bursty traffic.

Our block-sized packet buffer architectures are readily implemented for 100 Gb/s line rates. With a minimum packet size of 64 bytes, we have a 5 ns packet time slot in which to process each packet. Given the small SRAM size of our architecture (about 1.6 MB at 100 Gb/s), it can fit entirely in a single on-chip SRAM. Each packet requires 4 clock cycles to read two reservation table entries, select the memory bank, and update a reservation table entry. At a 1 GHz clock speed, these 4 clock cycles correspond to 4 ns, which is less than the 5 ns time slot budget. Once the reservation table has been updated, the packet can be written to the DRAM banks or to the bypass buffer, which can occur concurrently with the reservation table processing for the next packet.

6.8 Conclusion

In this chapter, we described novel reservation-based packet buffer architectures that take advantage of the known departure times of arriving packets to achieve simplicity and determinism. The switch-memory-switch router architecture [44,74] and the load-balanced router architecture [15,48], amongst others, are examples of architectures in which departure times for packets can be deterministically calculated in advance of packet insertion into packet buffers when best-effort routing is considered. We develop three new packet buffer architectures: a deterministic frame-sized packet buffer architecture, a deterministic block-sized packet buffer architecture, and a randomized block-sized packet buffer. In particular, a key contribution of this chapter is the reservation table in the deterministic block-sized packet buffer design that grows only logarithmi-

cally with line rates, leading to more than an order of magnitude reduction in SRAM requirements compared to the frame-sized packet buffer design. Furthermore, in our proposed architectures, the number of required interleaved DRAM banks is a small constant independent of the arrival traffic pattern, the number of flows, and the number of priority classes, making it scalable to the growing packet storage requirements in future routers while matching increasing line rates.

Chapter 6, in part, is a reprint of the material as it appears in the following publications:

- Hao Wang and Bill Lin, "Block-Based Packet Buffer with Deterministic Packet Departures", *IEEE International Conference on High Performance Switching (HPSR)*, Dallas, TX, June 13-16, 2010.

- Hao Wang and Bill Lin, "A Block-Based Reservation Architecture for the Implementation of Large Packet Buffers", *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, Princeton, NJ, October 19-20, 2009.

The dissertation author was the primary investigator and author of the papers.

Chapter 7

Conclusions

This dissertation aims to bridge the gap between advanced networking algorithms and their applications in future network equipments. The advancements in both hardware and software design make current network equipments more capable than their predecessors. The significant improvement in their performance, however is easily overshadowed by today's growing demands for diverse traffic spectrum on limited and expensive bandwidth. In order to build a manageable network router with strategic control, network routers are typically implemented. A router consists of two major components: a set of switch fabric and multiple linecards. The functionalities of a linecard can be categorized into three parts: packet classification, statistics accounting, and packet scheduling. In this dissertation, we focus on improving the performance of the statistics accounting and packet scheduling, which are essential for network traffic management. In particular, we propose novel DRAM-based statistics counter array architectures with worst-case performance guarantees for the maintenance of packet and flow information for statistics accounting. We propose a novel robust pipelined memory system with worst-case performance guarantee for network processing that matches the line-rate throughput. We propose succinct priority indexing structures for the management of large priority queues to enable the implementation of hierarchical quality-of-service per-flow queueing. This provides high throughput and fairness for the traffic managers. We also propose reservation-based packet buffers that offer reliable storage for packets temporarily stored in a router to provide robustness against network retransmission and congestion. Throughout the dissertation, the principles of pipelined architectures,

load-balancing algorithms, and parallelism have been applied in order to provide higher throughput, improved fairness, and worst-case performance guarantees for the network traffic managers.

There are still many questions to be answered and several approaches to be further explored in the design of better network traffic managers for future routers. For example, not only the throughput but also the end-to-end delay is often of concern for a pipelined system. Is it possible to build a robust pipelined memory system with significantly shorter pipelined delay? For packet scheduling, is it possible to design even simpler algorithms to manage large numbers of priority queues at line rate, which also require less fast memory? For packet buffering inside a router, is it possible to simplify the design of packet buffers in routers where the packet departure times are unknown? Especially, there is certainly scope for developing new and efficient algorithms and architectures for network traffic managers in network routers where certain arrival traffic patterns are known a priori or can be reliably predicted.

Appendix A

Proof of Theorem 3

Remark: The theorem can be shown to be false if a_i's can have mixed signs. E.g., $n = N = 2, c_1 = 1, c_2 = 2, a_1 = 1, a_2 = -1$, and f is strictly convex, then

$$\begin{aligned} &\mathrm{E}[f(\sum a_i X_i)] - \mathrm{E}[f(\sum a_i Y_i)] \\ &= \frac{1}{2}[f(-1) + f(1)] - \frac{1}{4}[f(-1) + f(1) + 2f(0)] \\ &= \frac{1}{2}[\frac{1}{2}f(-1) + \frac{1}{2}f(1) - f(0)] > 0, \end{aligned} \tag{A.1}$$

by Jensen's inequality. On the other hand, the signs of the c_i's do not matter since we shift them to make all c_i's positive.

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

$$\mathrm{E}g(X_1, \ldots, X_n) = \frac{1}{N^{(n)}} \sum_{N,n} g(c_{i_1}, \ldots, c_{i_n}), \tag{A.2}$$

$$\mathrm{E}g(Y_1, \ldots, Y_n) = \frac{1}{N^n} \sum_{i_1=1}^{N} \cdots \sum_{i_n=1}^{N} g(c_{i_1}, \ldots, c_{i_n}), \tag{A.3}$$

where $N^{(n)} = N(N-1)\ldots(N-n+1)$, and $\sum_{N,n}$ is taken over all n-tuples $i_1, \ldots, i_n$ of distinct positive integers not exceeding N. The goal is to find a function $\bar{g}$ such that the N^n terms of g in (A.3) can be rewritten into $N^{(n)}$ terms of $\bar{g}$, and each term of $\bar{g}$ will dominate each term of g in (A.2) for

$$g(x_1, \ldots, x_n) = f(a_1 x_1 + \ldots + a_n x_n). \tag{A.4}$$

Therefore, we want

$$\begin{aligned} \mathrm{E}g(Y_1,\ldots,Y_n) &= \frac{1}{N^n}\sum_{i_1=1}^{N}\cdots\sum_{i_n=1}^{N} g(c_{i_1},\ldots,c_{i_n}) \\ &= \frac{1}{N^{(n)}}\sum_{N,n}\bar{g}(c_{i_1},\ldots,c_{i_n}) \\ &= \mathrm{E}\bar{g}(X_1,\ldots,X_n). \end{aligned} \tag{A.5}$$

For $n=2$, we can use

$$\bar{g}(x_1,x_2) = \frac{N-1}{N}g(x_1,x_2) + \frac{1}{N}\left(\frac{a_1}{a_1+a_2}g(x_1,x_1) + \frac{a_2}{a_1+a_2}g(x_2,x_2)\right). \tag{A.6}$$

In general, let $[n]$ denote the set $\{1,\cdots,n\}$. Let

$$P = \{\rho = \{A_1,\cdots,A_k\} | A_i \subset [n], \cup_i A_i = [n]\} \tag{A.7}$$

be the set of all partitions of $[n]$. Let

$$H = \{h : [n] \to [n] | h \circ h = h\} \tag{A.8}$$

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 $(h(1),...,h(n))$. 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 $\bar{g}$ be linear combinations of $g(x_{h(1)},\cdots,x_{h(n)})$ 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

$$\begin{aligned} \bar{g}(x_1,\ldots,x_n) &= \sum_{h\in H} p_h g(x_{h(1)},...,x_{h(n)}), \\ \text{where } p_h &= p_{\rho_h}\Pi_{j\in Im(h)}\frac{a_j}{\sum_{i\in h^{-1}(j)} a_i}, \end{aligned} \tag{A.9}$$

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

$$\sum_{\{h|\rho_h=\rho\}} p_h = p_\rho, \quad \forall \rho \in P, \tag{A.10}$$

$$\sum_{h\in H} p_h = \sum_{\rho\in P} p_\rho. \tag{A.11}$$

$$\bar{g}(c_1,\alpha,c_2,\beta) = \frac{\mathbf{p_\rho a_1 a_3 g(c_1,c_1,c_2,c_2)}}{(\mathbf{a_1}+\mathbf{a_2})(\mathbf{a_3}+\mathbf{a_4})} + \frac{p_\rho a_2 a_3 g(\alpha,\alpha,c_2,c_2)}{(a_1+a_2)(a_3+a_4)} + \frac{p_\rho a_1 a_4 g(c_1,c_1,\beta,\beta)}{(a_1+a_2)(a_3+a_4)} + \frac{p_\rho a_2 a_4 g(\alpha,\alpha,\beta,\beta)}{(a_1+a_2)(a_3+a_4)} + \ldots \quad \text{(A.12)}$$

$$\bar{g}(\alpha,c_1,c_2,\beta) = \frac{p_\rho a_1 a_3 g(\alpha,\alpha,c_2,c_2)}{(a_1+a_2)(a_3+a_4)} + \frac{\mathbf{p_\rho a_2 a_3 g(c_1,c_1,c_2,c_2)}}{(\mathbf{a_1}+\mathbf{a_2})(\mathbf{a_3}+\mathbf{a_4})} + \frac{p_\rho a_1 a_4 g(\alpha,\alpha,\beta,\beta)}{(a_1+a_2)(a_3+a_4)} + \frac{p_\rho a_2 a_4 g(c_1,c_1,\beta,\beta)}{(a_1+a_2)(a_3+a_4)} + \ldots \quad \text{(A.13)}$$

$$\bar{g}(c_1,\alpha,\beta,c_2) = \frac{p_\rho a_1 a_3 g(c_1,c_1,\beta,\beta)}{(a_1+a_2)(a_3+a_4)} + \frac{p_\rho a_2 a_3 g(\alpha,\alpha,\beta,\beta)}{(a_1+a_2)(a_3+a_4)} + \frac{\mathbf{p_\rho a_1 a_4 g(c_1,c_1,c_2,c_2)}}{(\mathbf{a_1}+\mathbf{a_2})(\mathbf{a_3}+\mathbf{a_4})} + \frac{p_\rho a_2 a_4 g(\alpha,\alpha,c_2,c_2)}{(a_1+a_2)(a_3+a_4)} + \ldots \quad \text{(A.14)}$$

$$\bar{g}(\alpha,c_1,\beta,c_2) = \frac{p_\rho a_1 a_3 g(\alpha,\alpha,\beta,\beta)}{(a_1+a_2)(a_3+a_4)} + \frac{p_\rho a_2 a_3 g(c_1,c_1,\beta,\beta)}{(a_1+a_2)(a_3+a_4)} + \frac{p_\rho a_1 a_4 g(\alpha,\alpha,c_2,c_2)}{(a_1+a_2)(a_3+a_4)} + \frac{\mathbf{p_\rho a_2 a_4 g(c_1,c_1,c_2,c_2)}}{(\mathbf{a_1}+\mathbf{a_2})(\mathbf{a_3}+\mathbf{a_4})} + \ldots \quad \text{(A.15)}$$

We need to argue that there exist a p_ρ such that (A.5) 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 left-hand-side of (A.5). It corresponds to $\rho = \{\{1,2\},\{3,4\}\}$. On the right-hand-side, for any distinct $\alpha,\beta \in C$ that are different from c_1,c_2, we have Equations (A.12) to (A.15). 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 consider the case

$$g(x_1,\ldots,x_n) = f(a_1x_1+\ldots+a_nx_n).$$

If we set $f(x)=1$, we see from (A.5) and (A.9) that

$$\sum_{h\in H} p_h = 1. \quad \text{(A.16)}$$

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 $\bar{g}(x_1,\cdots,x_n)$ is a linear combination of $x_1,\cdots,x_n$. 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 them directly. We have

$$\begin{aligned}&\sum_{\{h|\rho_h=\rho\}} p_h(a_1x_{h(1)}+\ldots+a_nx_{h(n)})\\ &\qquad = p_\rho(a_1x_1+\ldots+a_nx_n),\quad \forall\rho\in P.\end{aligned}$$

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

$$\begin{aligned}&\sum_{\{h|\rho_h=\rho\}} p_h(a_1x_{h(1)}+\ldots+a_nx_{h(n)})\\ &= \frac{p_\rho a_1a_3(a_1x_1+a_2x_1+a_3x_3+a_4x_3)}{(a_1+a_2)(a_3+a_4)}\\ &+ \frac{p_\rho a_2a_3(a_1x_2+a_2x_2+a_3x_3+a_4x_3)}{(a_1+a_2)(a_3+a_4)}\\ &+ \frac{p_\rho a_1a_4(a_1x_1+a_2x_1+a_3x_4+a_4x_4)}{(a_1+a_2)(a_3+a_4)}\\ &+ \frac{p_\rho a_2a_4(a_1x_2+a_2x_2+a_3x_4+a_4x_4)}{(a_1+a_2)(a_3+a_4)}.\end{aligned}$$

For x_1 the first and third terms yield

$$\frac{p_\rho a_1a_3x_1}{x_3+x_4}+\frac{p_\rho a_1a_4x_1}{x_3+x_4}=p_\rho a_1x_1,$$

and similarly for x_2,x_3,x_4. Therefore

$$\begin{aligned}&\sum_h p_h(a_1x_{h(1)}+\ldots+a_nx_{h(n)})\\ &\quad=\sum_\rho p_\rho(a_1x_1+\ldots+a_nx_n)\\ &\quad=a_1x_1+\ldots+a_nx_n.\end{aligned}\tag{A.17}$$

From (A.16), (A.17), and Jensen's inequality, we get

$$\begin{aligned}\bar{g}(x_1,\cdots,x_n) &= \sum_{h\in H} p_h f(a_1 x_{h(1)} + \ldots + a_n x_{h(n)}) \\ &\geq f(\sum_{h\in H} p_h (a_1 x_{h(1)} + \ldots + a_n x_{h(n)})) \\ &= f(a_1 x_1 + \ldots + a_n x_n). \end{aligned} \tag{A.18}$$

Hence $\mathrm{E}\bar{g}(X_1,\cdots,X_n) \geq \mathrm{E}f(a_1X_1 + \ldots + a_nX_n)$, which together with (A.5) completes the proof. □

Bibliography

[1] B. Agrawal and T. Sherwood, "High-bandwidth network memory system through virtual pipelines," in *IEEE/ACM Transactions on Networking*, vol. 17, no. 4, Aug. 2009, pp. 1029–1041.

[2] M. Allman, V. Paxson, and E. Blanton, "RFC 5681: TCP congestion control," Sep. 2009.

[3] AMD, "Software optimization guide for AMD family 10h processors." [Online]. Available: http://www.amd.com/

[4] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker, "High speed switch scheduling for local area networks," in *ACM Transactions on Computer Systems*, vol. 11, Nov. 1993, pp. 319–352.

[5] G. Appenzeller, I. Keslassy, and N. McKeown, "Sizing router buffers," in *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, Aug.-Sep. 2004, pp. 281–292.

[6] N. Askitis, "Fast and compact hash tables for integer keys," in *Proceedings of Australasian Computer Science Conference (ACSC)*, Jan. 2009, pp. 113–122.

[7] F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh, "A tree based router search engine architecture with single port memories," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, Jun. 2005.

[8] J. C. R. Bennett and H. Zhang, "WF^2Q: worst-case fair weighted fair queueing," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, Mar. 1996, pp. 120 –128.

[9] ——, "Hierarchical packet fair queueing algorithms," in *IEEE/ACM Transactions on Networking*, vol. 5, Oct. 1997, pp. 675–689.

[10] R. Bhagwan and B. Lin, "Fast and scalable priority queue architecture for high-speed network switches," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, Mar. 2000, pp. 538–547.

[11] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting Bloom filters," in *Proceedings of ESA Annual European Symposium on Algorithms*, Sep. 2006, pp. 684–695.

[12] R. Brown, "Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem," in *Communications of the ACM*, vol. 31, Oct. 1988, pp. 1220–1227.

[13] CAIDA, "Round-trip time measurements from CAIDA's macroscopic Internet topology monitor." [Online]. Available: http://www.caida.org/analysis/performance/rtt/walrus0202.

[14] J. Cao and K. Ramanan, "A Poisson limit for buffer overflow probabilities," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, Jun. 2002, pp. 994–1003.

[15] C.-S. Chang, D.-S. Lee, and Y.-S. Jou, "Load balanced Birkhoff-von Neumann switches, part i: One-stage buffering," in *Computer Communications*, vol. 25, no. 6, 2002, pp. 611–622.

[16] C.-S. Chang, D.-S. Lee, and C.-M. Lien, "Load balanced Birkhoff-von Neumann switches, part ii: Multi-stage buffering," in *Computer Communications*, vol. 25, no. 6, 2002, pp. 623–634.

[17] C.-S. Chang, D.-S. Lee, and Y.-J. Shih, "Mailbox switch: a scalable two-stage switch architecture for conflict resolution of ordered packets," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, vol. 3, Mar. 2004, pp. 1995–2006.

[18] H. J. Chao, "A novel architecture for queue management in the ATM network," in *IEEE/ACM Transactions on Networking*, vol. 9, no. 7, Sep. 1991, pp. 1110–1118.

[19] K. Claffy, H. W. Braun, and G. Polyzos, "A parameterizable methodology for Internet traffic flow profiling," in *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 8, Oct. 1995, pp. 1481–1494.

[20] A. Cvetkovski, "An algorithm for approximate counting using limited memory resources," in *Proceedings of ACM Special Interest Group on Performance Evaluation (SIGMETRICS)*, vol. 35, no. 1, Jun. 2007, pp. 181–190.

[21] A. Demers, S. Keshav, and S. Shenkar, "Analysis and simulation of a fair queueing algorithms," in *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, Aug. 1989, pp. 1–12.

[22] A. Dhamdhere and C. Dovrolis, "Open issues in router buffer sizing," in *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, vol. 36, Sep. 2006, pp. 87–92.

[23] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep packet inspection using parallel Bloom filters," in *Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, vol. 24, no. 1, Jan.-Feb. 2004, pp. 52–61.

[24] A. B. Downey, "Evidence for long-tailed distributions in the Internet," in *Proceedings of ACM SIGCOMM Conference on Internet Measurement (IMC)*, Nov. 2001, pp. 229–241.

[25] A. I. Elwalid and D. Mitra, "Effective bandwidth of general markovian traffic sources and admission control of high speed networks," in *IEEE/ACM Transactions on Networking*, vol. 1, no. 3, Jun. 1993, pp. 329–343.

[26] U. Erlingsson, M. Manasse, and F. McSherry, "A cool and practical alternative to traditional hash tables," in *Proceeding of Workshop on Distributed Data and Structures (WDAS)*, Jan. 2006.

[27] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better NetFlow," in *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, Aug.-Sep. 2004, pp. 245–256.

[28] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, vol. 32, no. 4, Aug. 2002, pp. 323–336.

[29] J. García, J. Corbal, L. Cerdà, and M. Valero, "Design and implementation of high-performance memory systems for future packet buffers," in *Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec. 2003, pp. 372–384.

[30] M. Garetto and D. Towsley, "Modeling, simulation and measurements of queuing delay under long-tail Internet traffic," in *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, Aug. 2003, pp. 47–57.

[31] R. Graham, D. Knuth, and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Addison-Wesley, 1994.

[32] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic processing in Cell's multicore architecture," in *Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, vol. 26, no. 2, Dec. 2006.

[33] J. Hasan, S. Chandra, and T. N. Vijaykumar, "Efficient use of memory bandwidth to improve network processor throughput," in *ACM SIGARCH Computer Architecture News*, vol. 31, no. 2, 2003, pp. 300–313.

[34] W. Hoeffding, "Probability inequalities for sums of bounded random variables," in *Journal of the American Statistical Association*, vol. 58, no. 301, 1963, pp. 13–30.

[35] S. I. Hong, S. A. McKee, M. H. Salinas, R. H. Klenke, J. H. Aylor, and W. A. Wulf, "Access order and effective bandwidth for streams on a direct Rambus memory," in *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*, Jan. 1999, pp. 80–89.

[36] C. Hu, B. Liu, H. Zhao, K. Chen, Y. Chen, C. Wu, and Y. Cheng, "DISCO: Memory efficient and accurate flow statistics for network measurement," in *Proceedings of International Conference on Distributed Computing Systems (ICDCS)*, Jun. 2010.

[37] N. Hua, B. Lin, J. Xu, and H. Zhao, "BRICK: A novel exact active statistics counter architecture," in *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, Nov. 2008, pp. 89–98.

[38] P. Indyk, "Stable distributions, pseudorandom generators, embeddings, and data stream computation," in *Proceedings of Annual IEEE Symposium on Foundations of Compututer Science (FOCS)*, Nov. 2000, pp. 189–197.

[39] Intel, "Intel 64 and IA-32 architectures software developer's manual, volume 2B." [Online]. Available: http://www.intel.com/

[40] ——, "Intel IXP 465 network processor product brief." 2006. [Online]. Available: http://www.intel.com/

[41] ——, "Intel Lynnfield processor." 2010. [Online]. Available: http://www.intel.com/

[42] A. Ioannou and M. G. H. Katevenis, "Pipelined heap (priority queue) management for advanced scheduling in high-speed networks," in *IEEE/ACM Transactions on Networking*, vol. 15, Apr. 2007, pp. 450–461.

[43] S. Iyer, R. R. Kompella, and N. McKeown, "Designing packet buffers for router linecards," in *IEEE/ACM Transactions on Networking*, vol. 16, no. 3, 2008, pp. 705–717.

[44] S. Iyer, R. Zhang, and N. McKeown, "Routers with a single stage of buffering," in *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, vol. 32, no. 4, Aug. 2002, pp. 251–264.

[45] W. Jiang, Q. Wang, and V. Prasanna, "Beyond TCAMs: An SRAM-based parallel multi-pipeline architecture for terabit IP lookup," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, Apr. 2008, pp. 1786–1794.

[46] M. Kabra, S. Saha, and B. Lin, "Fast buffer memory with deterministic packet departures," in *Proceedings of IEEE Annual Symposium on High Performance Interconnects (Hot Interconnects)*, Aug. 2006, pp. 67–72.

[47] I. Keslassy and N. McKeown, "Maintaining packet order in two-stage switches," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, vol. 2, Jun. 2002, pp. 1032–1041.

[48] I. Keslassy, S.-T. Chuang, K. Yu, D. Miller, M. Horowitz, O. Solgaard, and N. McKeown, "Scaling Internet routers using optics," in *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, Oct. 2003, pp. 189–200.

[49] A. Kirsch, M. Mitzenmacher, and G. Varghese, "Hash-based techniques for high-speed packet processing," in *Algorithms for Next Generation Networks*, 2010.

[50] A. Kumar, J. Xu, L. Li, and J. Wang, "Space-code Bloom filter for efficient traffic flow measurement," in *Proceedings of ACM SIGCOMM Conference on Internet Measurement (IMC)*, Oct. 2003, pp. 167–172.

[51] S. Kumar, P. Crowley, and J. Turner, "Design of randomized multichannel packet storage for high performance routers," in *Proceedings of IEEE Annual Symposium on High Performance Interconnects (Hot Interconnects)*, Aug. 2005, pp. 100–106.

[52] S. Kumar, M. Becchi, P. Crowley, and J. Turner, "CAMP: fast and efficient IP lookup architecture," in *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, Dec. 2006, pp. 51–60.

[53] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, Sep. 2006, pp. 339–350.

[54] V. P. Kumar, T. V. Lakshman, and D. Stiliadis, "Beyond best effort: Router architectures for the differentiated services of tomorrow's Internet," in *IEEE Communications Magazine*, vol. 36, 1998, pp. 152–164.

[55] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for advanced packet classification with ternary CAMs," in *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, Aug. 2005, pp. 193–204.

[56] B. Lin and I. Keslassy, "The concurrent matching switch architecture," in *IEEE/ACM Transactions on Networking*, vol. 18, no. 4, Aug. 2010, pp. 1330–1343.

[57] B. Lin and J. Xu, "DRAM is plenty fast for wirespeed statistics counting," in *Proceedings of ACM Workshop on Hot Topics in Measurement and Modeling of Computer Systems (HotMetrics)*, Jun. 2008, pp. 45–50.

[58] W. Lin, S. K. Reinhardt, and D. Burger, "Reducing DRAM latencies with an integrated memory hierarchy design," in *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*, Jan. 2001, pp. 301–312.

[59] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter braids: A novel counter architecture for per-flow measurement," in *Proceedings of ACM Special Interest Group on Performance Evaluation (SIGMETRICS)*, vol. 36, no. 1, Jun. 2008, pp. 121–132.

[60] A. W. Marshall and I. Olkin, *Inequalities: Theory of Majorization and Its Applications*. Academic Press, 1979.

[61] A. J. McAuley and P. Francis, "Fast routing table lookup using CAMs," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, Mar. 1993, pp. 1382–1391.

[62] N. McKeown, "The iSLIP scheduling algorithm for input-queued switches," in *IEEE/ACM Transactions on Networking*, vol. 7, no. 2, Apr. 1999, pp. 188–201.

[63] E. H. McKinney, "Generalized birthday problem," in *American Mathematical Monthly*, vol. 4, no. 73, Apr. 1966, pp. 385–387.

[64] S.-W. Moon, J. Rexford, and K. G. Shin, "Scalable hardware priority queue architectures for high-speed packet switches," in *IEEE Transactions on Computers*, vol. 49, no. 11, Nov. 2000, pp. 1215–1227.

[65] R. Morris, "Counting large numbers of events in small registers," in *Communications of the ACM*, vol. 21, no. 10, Oct. 1978, pp. 840–842.

[66] R. Motwani and P. Raghavan, *Randomized Algorithms*. Cambridge, 1995.

[67] A. Muller and D. Stoyan, *Comparison Methods for Stochastic Models and Risks*. Wiley, 2002.

[68] S. Muthukrishnan, *Data Streams: Algorithms and Applications at Foundations and Trends in Theoretical Computer Science*. NOW Publisher Inc., 2005.

[69] H. Obara and T. Yasushi, "An efficient contention resolution algorithm for input queuing ATM cross-connect switches," in *International Journal of Digital and Analog Cabled Systems*, vol. 2, no. 4, 1989, pp. 261–267.

[70] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *Journal of Algorithms*, vol. 51, no. 2, 2004, pp. 122 – 144.

[71] C. Pandit and S. Meyn, "Worst-case large-deviation asymptotics with application to queueing and information theory," in *Stochastic Processes and Their Applications*, vol. 116, no. 5, 2006, pp. 724–756.

[72] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case," in *IEEE/ACM Transactions on Networking*, vol. 1, no. 3, 1993, pp. 344–357.

[73] D. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach*, 2nd ed. Morgan Kaufmann, 1996.

[74] A. Prakash, S. Sharif, and A. Aziz, "An O($\log_2 n$) parallel algorithm for output queuing," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, vol. 3, Jun. 2002, pp. 1623–1629.

[75] J. Qian, S. Hinrichs, and K. Nahrstedt, "Acla: A framework for access control list (ACL) analysis and optimization," in *Proceedings of the IFIP TC6/TC11 International Conference on Communications and Multimedia Security Issues of the New Century*, May 2001, pp. 4–18.

[76] S. Ramabhadran and G. Varghese, "Efficient implementation of a statistics counter architecture," in *Proceedings of ACM Special Interest Group on Performance Evaluation (SIGMETRICS)*, vol. 31, no. 1, Jun. 2003, pp. 261–271.

[77] C. V. Ramamoorthy and H. F. Li, "Pipeline architecture," in *Computing Surveys*, vol. 9, no. 1, Mar. 1977, pp. 61–102.

[78] Rambus, "XDR datasheet," 2005-2006. [Online]. Available: http://www.rambus.com/

[79] ——, "XDR-2 datasheet," 2008-2009. [Online]. Available: http://www.rambus.com/

[80] B. R. Rau, "Pseudo-randomly interleaved memory," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, May 1991, pp. 74–83.

[81] M. Roeder and B. Lin, "Maintaining exact statistics counters with a multi-level counter memory," in *Proceedings of Global Telecommunications Conference (GLOBECOM)*, vol. 2, Nov. 2004, pp. 576–581.

[82] S. M. Ross, *Introduction to Probability Models*, 9th ed. Academic Press, 2006.

[83] ——, *Low-Density Parity-Check Codes*. Wiley, 2nd Edition, 1995.

[84] Samsung, "Samsung K4B4G0446A DDR3 SDRAM." [Online]. Available: http://www.samsung.com/

[85] ——, "Samsung K7S3236U4C QDRII SRAM." [Online]. Available: http://www.samsung.com/

[86] C. Semeria and J. Gredler, "Juniper networks solutions for network accounting." in *Juniper White Paper*, no. 200010-001, 2001.

[87] D. Shah, S. Iyer, B. Prahhakar, and N. McKeown, "Maintaining statistics counters in router line cards," in *Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, vol. 22, no. 1, Jan. 2002, pp. 76–81.

[88] M. Shreedhar and G. Varghese, "Efficient fair queueing using deficit round-robin," in *IEEE/ACM Transactions on Networking*, vol. 4, Jun. 1996, pp. 375–385.

[89] G. Shrimali and N. McKeown, "Building packet buffers using interleaved memories," in *Proceedings of IEEE International Conference on High Performance Switching (HPSR)*, May 2005, pp. 1–5.

[90] S. Sinha, S. Kandula, and D. Katabi, "Harnessing TCPs burstiness using flowlet switching," in *ACM SIGCOMM Workshop on Hot Topics in Netwworks (HotNets)*, Nov. 2004.

[91] R. Stanojevic, "Small active counters," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, May 2007, pp. 2153–2161.

[92] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, Aug. 2001, pp. 149–160.

[93] Y. Tamir and H. C. Chi, "Symmetric crossbar arbiters for VLSI communication switches," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, 1993, pp. 13–27.

[94] P. van Emde Boas, "Design and implementation of an efficient priority queue," in *Mathematical Systems Theory*, vol. 10, 1977, pp. 99–127.

[95] G. Varghese, *Network Algorithmics*. Elsevier, 2005.

[96] C. Villamizar and C. Song, "High performance TCP in ANSNET," in *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, vol. 24, no. 5, Aug.-Sep. 1994, pp. 45–60.

[97] H. Wang and B. Lin, "On the efficient implementation of pipelined heaps for network processing," in *Proceedings of IEEE Global Communications Conference (GLOBECOM)*, Nov. 2006, pp. 1–5.

[98] ——, "Pipelined van Emde Boas tree: Algorithms, analysis, and applications," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, May 2007, pp. 2471 –2475.

[99] ——, "Block-based packet buffer with deterministic packet departures," in *Proceedings of IEEE International Conference on High Performance Switching (HPSR)*, Jun. 2010, pp. 38–43.

[100] H. Wang, H. Zhao, B. Lin, and J. Xu, "Design and analysis of a robust pipelined memory system," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, Mar. 2010, pp. 1–9.

[101] F. A. Ware and C. Hampel, "Micro-threaded row and column operations in a DRAM core," in *Rambus White Paper*, Mar. 2005.

[102] ——, "Improving power and data efficiency with threaded memory modules," in *Proceedings of IEEE International Conference Compututer Design (ICCD)*, Oct. 2006, pp. 417–424.

[103] R. Yavatkar, D. Pendarakis, and R. Guerin, "RFC 2753: a framework for policy-based admission control," 2000.

[104] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, Dec. 2006, pp. 93–102.

[105] H. Zhang, "Service disciplines for guaranteed performance service in packet-switching networks," in *Proceedings of the IEEE*, vol. 83, no. 10, Oct. 1995, pp. 1374–1396.

[106] H. Zhao, A. Lall, M. Ogihara, O. Spatscheck, J. Wang, and J. Xu, "A data streaming algorithm for estimating entropies of OD flows," in *Proceedings of ACM Internet Measurement Conference (IMC)*, Oct. 2007, pp. 279–290.

[107] H. Zhao, H. Wang, B. Lin, and J. Xu, "Design and performance analysis of a DRAM-based statistics counter array architecture," in *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, Oct. 2009, pp. 84–93.

[108] Q. Zhao, J. Xu, and Z. Liu, "Design of a novel statistics counter architecture with optimal space and time efficiency," in *Proceedings of ACM Special Interest Group on Performance Evaluation (SIGMETRICS)*, vol. 34, no. 1, Jun. 2006, pp. 323–334.

[109] X. Zhuang and S. Pande, "A scalable priority queue architecture for high speed network processing," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, Apr. 2006, pp. 1–12.