Distributed Packet Buffers for High-Bandwidth Switches and Routers

Dong Lin, Student Member, IEEE, Mounir Hamdi, Fellow, IEEE and Jogesh Muppala, Senior Member, IEEE

Abstract—High-speed routers rely on well-designed packet buffers that support multiple queues, provide large capacity and short response times. Some researchers suggested combined SRAM/DRAM hierarchical buffer architectures to meet these challenges. However, these architectures suffer from either large SRAM requirement or high time-complexity in the memory management. In this paper, we present scalable, efficient and novel distributed packet buffer architecture. Two fundamental issues need to be addressed to make this architecture feasible: (a) how to minimize the overhead of an individual packet buffer; and (b) how to design scalable packet buffers using independent buffer subsystems. We address these issues by first designing an efficient compact buffer that reduces the SRAM size requirement by \((k-1)/k\). Then we introduce a feasible way of coordinating multiple subsystems with a load-balancing algorithm that maximizes the overall system performance. Both theoretical analysis and experimental results demonstrate that our load-balancing algorithm and the distributed packet buffer architecture can easily scale to meet the buffering needs of high bandwidth links and satisfy the requirements of scale and support for multiple queues.

Index Terms—Router memory, SRAM/DRAM, Packet scheduling.

1 INTRODUCTION

The phenomenal growth of the Internet has been fuelled by the rapid increase in the communication link bandwidth. Internet routers play a crucial role in sustaining this growth by being able to switch packets extremely fast to keep up with the growing bandwidth (line rate). This demands sophisticated packet switching and buffering techniques. Packet buffers need to be designed to support large capacity, multiple queues and provide short response times.

The traditional rule of thumb for Internet routers states that the routers should be capable of buffering \(RTT* R\) [6] data, where \(RTT\) is a round-trip time for flows passing through the router, and \(R\) is the line rate. In [6][9][10], this rule was challenged. However, routers manufacturers still seem to favor the use of large buffers. For instance, the Cisco CRS-1 modular service card with a 40 Gbps line rate incorporates a 2 GB packet buffer memory per line card and per side (ingress and egress) [11].

In order to support fine-grained IP quality of service (QoS) requirements, nowadays, a packet buffer usually maintains thousands of queues. For example the Juniper E-series routers [7] maintain as many as 64000 queues. Given the increasing popularity of OpenFlow [19], a packet buffer that supports millions of queues is always desired.

Furthermore, a packet buffer should be capable of sustaining continuous data streams for both ingress and egress. Given a 40 Gbps line rate and 40 Byte packets, the access time for a single packet is 4 ns in each direction.

With the ever-increasing line rate, current available memory technologies, namely SRAM or DRAM alone cannot simultaneously satisfy these three requirements. This prompted researchers to suggest hybrid SRAM/DRAM (HSD) architecture with a single DRAM [1], interleaved DRAMs [12]-[15], or parallel DRAMs [16] sandwiched between SRAMs.

In this paper, we briefly review previous work on packet buffer architectures and present scalable and efficient hierarchical packet buffer architecture. By fully exploring the advantage of parallel DRAMs, we first propose a memory management algorithm called Random Round Robin which reduces the SRAM size to \(1/k\) through a fast batch load scheme, where \(k\) denotes the number of DRAMs working in parallel. We also devise a “traffic-aware” approach which aims to provide different services for different types of data streams. This approach further reduces the system overhead. Both mathematical analysis and simulation demonstrate that the proposed architecture together with its algorithm reduce the overall SRAM requirement significantly while providing guaranteed performance in terms of low time complexity, upper-bounded drop rate and uniform allocation of resources. In one simulation, the proposed architecture reduces the size of SRAM by more than 95% and the maximal delay is only \(us\)-level, when the traffic intensity is 76%.

The rest of the paper is organized as follows. In Section 2, we briefly review the related work from the literature. We then show their architectural scalability limitations analytically in Section 3. In Section 4, we introduce the concept of traffic-aware approach in designing a packet buffer. We propose a new buffering architecture reducing the requirement of SRAM size in Section 5. Its performance is studied in Section 6. Further using it as a basic
building block, we present a distributed packet buffer architecture in Section 7. The corresponding mathematical analysis is shown in Section 8. In Section 9, the simulations are carried out to verify these results. We present some discussions in Section 10 and then conclude this paper in Section 11.

2 BACKGROUND AND RELATED WORK

2.1 SRAM and DRAM Technology

Current SRAM and DRAM cannot individually meet the access time and capacity requirements of router buffers. While SRAM is fast enough with an access time of around 2.5ns [2], its largest size is limited by current technologies to only a few MB and it is power hungry. On the other hand, a DRAM can be built with large capacity, but the typical memory access time (i.e. \( T_{rc} \)) is too large, around 40ns [3]. (The multiplexed addressing scheme of DRAM requires two-step addressing which incurs extra time in addressing different memory cells when they are located in different rows. \( T_{rc} \) is used to denotes the row cycle time. In practice, the \( T_{rc} \) contributes significantly to the access time of DRAM. Therefore, it can be simply regarded as the access time of the DRAM.) Over the last decade high bandwidth and large storage capacity DRAM have become available. However the DRAM memory access time still proves to be a stumbling block. It decreases by only 10% every 18 months [4]. In contrast, as the line-rate increases by 100% every 18 months [5], DRAM will fall further behind in satisfying the requirements of high-speed buffers.

Given a DRAM family, in order to keep the DRAM modules busy, we need to transfer a minimum size chunk (it is also called as block in [1]) of data into it to effectively utilize the bandwidth provided by the DRAM module. Large memory access time of DRAM requires the system to read/write data from/to any memory address for at least \( T_{rc} \) time units [1]. For example, the Samsung K4B4G0446B DRAM (DDR3-800) supports a maximum speed of 1.6 Gbps/pin while its minimal \( T_{rc} \) is 49.5 ns [22]. With a maximum of 32 pins in a single chip, the size of a chunk is around \((32*1.6\text{Gbps}*49.5\text{ns}) = 2534.4 \text{ bits.} \) By using a Reduced Latency DRAM (RLDRAM) chip, the chunk size can be greatly reduced. Taking Micron MT49H16M36 as an example, it supports a minimal \( T_{rc} \) around 15 ns and 1.067 Gbps/pin peak data rate [23]. With a maximum of 32 pins in a single chip, the chunk size is 511.7 bits. Therefore, we come to the conclusion that the chunk size could range from 64 Bytes to 320 Bytes.

The contract price for 1Gbit DDR3-1333 Spot on May 18th 2010 is 2.81 US Dollars [24], while a 576 Mbits RLDRAM Spot (Micron MT49H16M36-18) costs 139 US Dollars [25]. Given much cheaper price and higher performance, nowadays, the DDR3 family of DRAM takes a large fraction of the router buffer market, making the typical chunk size become 320 Bytes.

2.2 Packet Buffer Architectures

Bridging the speed gap between the SRAM and the DRAM becomes a major challenge. This speed mismatch does not refer to the bandwidth but the access time and the concomitant access granularity. Due to the variable packet sizes that the IP protocol allows, it is common for packet processors to segment packets into fixed size cells, to make them easier to manage and switch. A common choice for the cell size is 64 Bytes because it is the first power of two larger than the size of a minimum packet (i.e., 40Bytes). Thus, a packet buffer should be able to access data at the granularity of a cell. This requirement however is not applicable to the DRAM. In a cell-based packet buffer where a chunk is much larger than a cell, the payload efficiency and the effective throughput of the entire system are dramatically reduced. For example, the payload efficiency for the Samsung K4B4G0446B is only \( 64/320 = 1/5 \).

This prompted researchers to suggest hybrid SRAM/DRAM (HSD) architecture [1]. To conduct a quantitative analysis, a parameter called \( b \) was introduced in [14]-[18] to denote the ratio of access time between the DRAM and the SRAM. Accordingly, the access granularity of DRAM is \( b \) times that of the SRAM, i.e. \( b \) cells. This description is simple and straightforward. However, it becomes inadequate under some circumstances. First, the access time alone cannot determine the access granularity. The minimal access granularity has to be related to the other factors, such as the bandwidth, and the frequency. For example, given \( b \) equals to 10, the access time of the DRAM is ten times that of the SRAM. According to the definition of \( b \), the access granularity of the DRAM should be 10 cells. However, if the bandwidth of the SRAM is twice that of the DRAM, the access granularity of the DRAM becomes 5 cells. The odd situation also happens when a DRAM with shorter access time and higher bandwidth is introduced, where the access granularity of new DRAM depends on the product of both its bandwidth and the current access time. There is no guarantee that the speed mismatch is improved. Second, the definition of \( b \) encounters some troubles in modeling a complicated architecture that consists of multiple SRAM and DRAM devices. It becomes meaningless to compare the access time of individual memory devices. Third, the definition of \( b \) becomes inapplicable when the allowed minimal access granularity of SRAM is less than a cell. Given the same bandwidth, as long as the SRAM still adopts the cell-based access, the access granularity of DRAM has to be less than \( b \) cells, even if the access time of DRAM is indeed \( b \) times that of the SRAM.

Being aware of these drawbacks, a new description of
this problem was introduced in [1]. Using the same parameter \( b \), it directly refers to the chunk size of a single DRAM. Thus, the speed mismatch of SRAM and DRAM now changes into the size mismatch of cell and chunk. We use this definition in our latter statement and re-describe the previous work in this way for the sake of consistency.

An additional challenge in designing packet buffers is that we need to maintain multiple queues, rather than just one single FIFO queue. Intuitively, dispatching and storing packets in multiple separate queues entail significant overhead for the memory management algorithms.

In short, the fundamental problem in design a packet buffer is to find an efficient way to bridge the gap of size between the cell and the chunk. It must introduce minimal overhead while satisfying the aforementioned requirements, viz., SRAM-level access time, DRAM-level storage capacity and large-scale multiple queuing.

Generally speaking, there are three ways of organizing a packet buffer, viz., the hybrid SRAM/DRAM architecture [1], the interleaved architecture [12]-[15] and the parallel DRAM architecture [16].

### 2.2.1 Hybrid SRAM/DRAM Architecture

![Diagram of Nemo architecture](image)

Batch load would be an effective method to improve the system performance. As shown in Fig. 1, Iyer et al. [1] first introduced the basic hybrid SRAM/DRAM architecture with one DRAM sandwiched between two smaller SRAM memories, where the two SRAMs hold heads and tails of all the queues and the DRAM maintains the middle part of the queues. Shuffling packets between the SRAM and the DRAM is under the control of a memory management algorithm (MMA). The principal idea behind their MMA is to temporarily hold \( O(b) \) amount of data for each queue in both the ingress and egress SRAM, so as to change the scattered DRAM accesses into a continuous one. Since the batch loads are strictly limited within each queue, the size requirement of SRAM for the HSD architecture is \( O(bQ) \), where \( Q \) is the number of FIFO queues. Whenever a FIFO queue accumulates \( b \) amount of data, it is transferred to the DRAM through a single write. A SRAM queue size of \( 2b \) guarantees against queue overflow. In [1], the authors further suggested that the size of the tail SRAM can be further reduced to \( Q(b-1) \) by introducing a pipeline design. They also introduced a so-called the Earliest Critical Queue First (ECQF)-MMA for the egress, which reduces the size of head SRAM to \( Q(b-1) \). By introducing an extra delay (i.e. \( Q(b-1)+1 \)), the ECQF-MMA now predicts the most critical queue (the one that goes empty or bears the biggest deficit first; in case of even, pick an arbitrary one.) and fetches the corresponding \( b \)-size chunk of data from the DRAM in advance. In [1], the author proved that, for the architecture shown in Fig. 1, ECQF achieves the size of the smallest possible head-cache; i.e. no algorithm can do better than ECQF. This architecture, also known as Nemo, has been adopted by Cisco.

### 2.2.2 Interleaved DRAM Architectures

Given a certain family of DRAM, the chunk size of \( b \) is determined by two factors. They are the bandwidth of a DRAM and its \( T_{EC} \). Assume the bandwidth of a DRAM is \( BW, b=BW* T_{EC} \). Given fixed \( T_{EC} \) and cell-size, \( b \) can be reduced by replacing a fast DRAM with multiple slower DRAMs, i.e. smaller value of \( BW \) for each. In this way, \( b \) could match the size of cell, thus the batch load is no longer needed. Each DRAM is now capable of accommodating cells independently. Accordingly, the MMA now needs to coordinate the data transfers between \( Q \) queues and \( k \) DRAMs, which can be formalized as a bipartite graph for maximum matching [13] problem. Given the original \( b \) in a packet buffer with only one DRAM is \( b_p \), \( k \) should be not less than \( (b_p/64\text{Bytes}) \), in order to provide an equivalent overall bandwidth.

Shrimani et al. [12] proposed memory architecture with \((b_p/64\text{Bytes})\) interleaved DRAMs. The MMA is based on a randomized algorithm, where each cell is written/read into/from the interleaved DRAM memories randomly, thus it seriously suffers from an out-of-sequence problem.

Using \((b_p/64\text{Bytes})\) interleaved DRAMs as well, Garcia et al. [14][15] suggested that a per-queue Round-Robin dispatching scheme (i.e., if a cell is the \( i \)-th cell in a queue, then it should be dispatched into DRAM \( j \), where \( j = i \mod k \) could guarantee a maximum matching when the system is equipped with a sufficient large tail buffer, typically twice the size of Nemo. Feng et al. [13] further suggested that the tail buffer can be implemented using \( k \) distributed SRAMs, which is easier to implement in practice. With per-queue Round-Robin, the out-of-sequence problem is solved. However, the high time complexity in finding the maximum matching remains. Given the fact that a single chunk could require \( O(Q) \) iterations before finding a maximum matching, the author acknowledged that the time complexity in achieving the maximum matching could be \( O(Q) \) in the worst case [14].

### 2.2.3 Parallel DRAM Architecture

Feng et al. [16] proposed a parallel hybrid SRAM/DRAM architecture with \( k \) \((k=(b_p/64\text{Bytes}))\) DRAMs named PHSD. Compared with the interleaved architectures [12]-[15], the PHSD reduces the time complexity of MMA to
O(k) by introducing k arbiters working in parallel. By further setting a limitation on the burst size of incoming traffic, O(kb ln Q) size requirement of SRAM was derived. However, the size requirement of SRAM still accounts for O(kbQ) in the worst case.

2.2.4 Other Approaches
Designing a general-purpose packet buffer is always a difficult task. In contrast, for specific applications where buffer behavior is predictable, the task can be greatly simplified. Kabra et al. [17] introduced a parallel DRAM approach that is optimized for the fixed departure time. Lin et al. [18] found the short-term stability of the number of connections in a trunk based on the analysis on real-life traces. Taking this characteristic into account, they proposed an approximation algorithm, which serves the application of fairness queuing.

3 THE SCALING LIMITATION OF PREVIOUS ALGORITHMS

Single DRAM packet buffer architectures like Nemo [1] cannot meet the requirements of ever increasing bandwidth and storage capacity. A straightforward idea to improve its performance is to replace the single DRAM in the middle with multiple parallel DRAMs. These DRAMs share the same address bus, thus can be still regarded as a single DRAM from outside.

On the other hand, a packet buffer architecture with multiple DRAMs [12]-[16] requires the use of slow DRAMs compared to the Nemo architecture, in order to maintain the same bandwidth. Memory vendors always seek to provide DRAM modules with higher bandwidth and larger storage capacity. In order to avoid the size mismatch between a chunk and a cell, it becomes less cost-effective to choose slow DRAMs deliberately, when faster and cheaper DRAMs are available. The fast RLDRAM is not an option as well, because of its price.

To conduct a fair comparison, we assume that all the architectures [1],[12]-[16] use the same raw materials in building a packet buffer. In other words, we assume that a packet buffer always consists of k identical DRAMs and the physical features of these DRAMs remain the same no matter whichever kind of packet buffer architecture is adopted. The chunk size of a single DRAM is always b. We restrict the definition of b. b always equals to h, in the following description. We mainly focus on the scalability issue of previous algorithms; i.e. whether the overall bandwidth accounts for k^2BW or not, the required size of SRAM, the time-complexity of MMA and the other system overheads.

3.1 Buffer Behaviors
When we carefully examine the hierarchical packet buffer architectures by using the aforementioned methodology, whether the HSD architecture [1], interleaved DRAMs [12]-[17], or parallel DRAM [16], they all rely on three parameters, k, b and Q. The required size of SRAM is always O(kbQ).

To understand this phenomenon for its further study, we first examine the buffer behavior of previous hybrid SRAM/DRAM architectures and algorithms, especially in the Nemo and the PHSD architectures.

As shown in Fig. 2, the DRAM structure in this extend version of Nemo is implemented as a composition of k DRAMs that simply provides a data bus of width k times that of a single DRAM data bus. Given a fixed chunk size of b for a single DRAM, Nemo increases the scale of batch load by k times, which requires each of the Q queues to maintain kb-size of data. Whenever kb-size of data is accumulated in a queue, it will be written into k DRAMs through a mutual data bus. In this way, the size gap between cell and chunk is compensated. One major drawback of Nemo is that the first (kb-1) size of data cannot depart from the queue until the last bit arrives. This increases the buffering requirement.

Like the other independent address bus architectures, the PHSD adopts a distributed implementation that uniformly dispatches the traffic of each queue to the k DRAMs respectively; i.e. if a cell is the i-th cell in a queue, then it will be dispatched to DRAM j, where j = i mod k. By using so-called per-queue Round-Robin scheme, the previous single kb-size queues are replaced by k queues of size b each. As a result, the system maintains 2kQ queues for both ingress and egress. Fig. 3 illustrates the organization of a single queue in PHSD. After carefully analyzing its buffer behavior, we make the following observations:
1) Given that only the first \(k(b-64)\) size of data arrives to a queue, the data could stay in the SRAMs infinitely. So the maximum SRAM size in PHSD is roughly equal to that of Nemo, when \(k\) is small.

2) The departure time of each \(b\)-sized chunk in PHSD relies on the corresponding traffic pattern. Given that only the first \(k(b-64)+64\) size of data arrives in each of the multiple queues, it creates an unbalanced traffic allocation among the DRAMs, i.e. only the first DRAM receives \(b\) size of data.

3) Although the long-term output is balanced by the per-queue Round Robin scheme, short-term biased output still exists, e.g. the heads of currently active queues in the output may all be allocated to a particular DRAM. Accordingly, this DRAM becomes the bottleneck of entire system in egress.

4) Per-queue Round Robin increases the scale of queuing and maximum matching problem by \(k\) times making the corresponding MMA less efficient.

5) The same Round Robin sequence adopted by multiple queues may create synchronization effect that overwhelms the first DRAM in the initial stage.

6) Each DRAM maintains exactly \(Q\) queues and the entire system maintains \(kQ\) queues in total. In practice, maintaining so many dynamic queues on such a big scale is a real challenge. For queues bearing modest traffic, this incurs significant overhead, even though there is no practical reason to maintain queues among multiple DRAMs to support it.

According to the aforementioned analysis, it seems that the per-queue Round-Robin does more harm than good, and both Nemo and PHSD fail to exploit the advantage of having parallel DRAMs. The requirement of SRAM size in the worst case is always \(O(kbQ)\).

### 3.2 Exponentially Increasing SRAM Size

Due to the sluggish advance in the access delay of DRAM, \(b\) will remain more or less constant, making \(k\) and \(Q\) dominate the SRAM size of a packet buffer.

In order to increase the bandwidth and the storage capacity of a packet buffer, one may need to introduce more DRAMs working in parallel. Assuming the line rate of Internet increase at a speed of \(N\), \(k\) must also increase at the same speed. On the other hand, with the increasing bandwidth of a packet buffer, larger scale of multiple queuing is naturally required. According to the \(N^2\)-hypothesis, the number of connections is proportional to the square root of line rate \([8]\). Since \(Q\) is usually related to the number of data streams (the aggregation of single connection or multiple connections) that need to be processed independently, \(Q\) has to be increased continuously at a speed of \(N^{1/2}\). Hence, the size of SRAM has to be increased exponentially at a speed of \(N^{3/2}\), in contrast to a linear increase in the line rate.

### 4 Distributed Packet Buffer Architecture

In our view, all packet buffering techniques so far have adopted a traffic-agnostic approach while designing the packet buffering algorithms. We must clarify that even though existing approaches do use \(Q\) queues, each queue is treated the same by the buffer management algorithms. No effort is made to exploit the inherent characteristics of the corresponding traffic patterns like the arrival rate, burst sizes, transit time requirements through the router etc. However a traffic-aware approach to the problem, we believe, will yield new possibilities for conquering the scalability problem. Furthermore, the hierarchical approaches adopted so far including interleaving and parallel buffering, are too strait-jacketed where regularity of the structures put in artificial constraints on the possibilities for exploring alternatives to scalability.

In this paper, we investigate a new dimension to the problem, viz. how to extend the packet buffer architectures by using independent packet buffer sub-systems. The overall packet buffer now takes the form of a distributed system composed of several compact packet buffers. We profess that the only real requirement for a packet buffer is that it should be able to absorb incoming traffic at a given rate, and maintain the outgoing traffic at the same rate, while still supporting the requirements for the different data streams transiting through the buffer.

To make the idea more concrete, we present a numerical example next. Assume a single packet buffer with \(bQ\) size of SRAM is capable of supporting \(BW\) bandwidth and \(Q\) queues. To increase its bandwidth by \(3\) times and double the number of queues, the required SRAM size now increases to \(4bQ\). Alternatively, we can build two subsystems, each supporting \(2BW\) bandwidth and \(Q\) queues while together providing \(4BW\) bandwidth and \(2Q\) queues. Consequently, the overall size of the SRAM required is reduced to \(4bQ\), while its performance could roughly match the previous solution as long as an efficient load-balancing algorithm is applied. By introducing a fine-grained design in which more subsystems are introduced, the SRAM size can be further reduced. For example, a distributed packet buffer consisting of four sub-buffers, each of which supports \(BW\) bandwidth and \(0.5Q\), the entire system requires only \(2bQ\) of SRAM in total.

![Distributed packet buffer](image_url)

As shown in Fig. 4 the distributed packet buffer is implemented as a composition of multiple compact packet buffers. Such distributed system where \(k\) independent pack-
et buffers work together providing increasing performance but at a linearly increasing scale needs to be designed. Each compact packet buffer acts as an independent unit implementing its own packet buffer architecture and memory management algorithm. The selection of the packet buffer architecture and the MMA for a compact packet buffer may be dictated by buffering characteristics and requirements for the data streams being buffered there. For example, data streams requiring short transit time through the buffer may be accommodated by a compact packet buffer that supports such requirement. On the other hand, data streams that require large amount of buffer space, but with less stringent timing requirements could be accommodated by another compact buffer specifically designed with these characteristics, or be distributed over multiple compact buffers. Incoming cells are distributed to the packet buffers by the distributor based on their own requirements. On the output side, an aggregator collects cells from the compact buffers based on the output schedule and dispatches them accordingly. This immediately raises the following questions that need to be addressed in order to successfully design a distributed buffer architecture:

- How do we design a compact packet buffer to provide certain buffer characteristics and meet requirements of the data streams being buffered while at the same time harness them to meet the overall buffer design goals?
- How do we design a suitable distributor algorithm and an aggregator algorithm?
- How do we achieve good load-balance in the distributed system whereby we achieve balanced utilization of all the resources in the compact buffers?

Deciding on a suitable distributed system architecture that achieves the best overall performance while incurring minimal overhead is not straightforward. We must carefully choose the granularity of a distributed system.

In [26], we have proposed the basic idea of the distributed packet buffer architecture. However, a fine-grained distributed system which consists of several simple compact buffers may yield a cost-effective system in theory but not necessarily in practice. A large number of sub-systems may introduce coordination overhead making the load-balancing less efficient. This includes extra storage of state maintenance and the difficulty in designing a high-speed distributor/aggregator with several ports. Therefore, designing a distributed packet buffer requires several tradeoffs. As a starting point, we need a well-designed compact packet buffer which can easily be tailored to meet different requirements.

## 5 Compact Packet Buffer Design

Addressing the drawbacks of the previous algorithms, we propose a new memory architecture that reduces the system overhead in term of SRAM size while relying on non-specific traffic pattern. In [27], we have demonstrated that with a fast batch load scheme and random round robin MMA, the required size of SRAM in the ingress (also known as tail cache in [1]) can be reduced to $1/k$ that of in Nemo. Given the required size of SRAM in the egress (also known as head cache in [1]) remains the same, the overall size of SRAM accounts for $(k+1)/2k$ that of in Nemo. In this paper, we will further prove the head cache can be also reduced to $1/k$ that of in Nemo, making the overall size of SRAM becomes $1/k$ that of in Nemo.

Instead of waiting for $kb$-size of data before doing a batch load, we adopt a per-queue Random Round Robin (RRR) scheme to achieve a fast batch load. As shown in Fig. 5, the system now dispatches $b$-sized chunks of data whenever it is accumulated in each queue. These $b$-sized chunks are dispatched to multiple DRAMs on a per-queue Round Robin basis. However, unlike previous algorithms where each queue follows the same Round Robin sequence, the RRR scheme randomly selects the starting DRAM of each queue and keeps updating it for every $k$ rounds. As demonstrated in Fig. 5, $i, j, m$ and $n$ are random numbers that determine the dispatching sequence of a queue. Accordingly, $log_b k$ bits information has to be attached to the head of every $kb$-sized chunk for the sake of reordering it in egress. In this way, RRR avoids the synchronization effect while still guaranteeing the in-order processing and uniform allocation of $kb$-size data within each queue.

![Fig. 5. Fast batch load & Random Round Robin](image)

![Fig. 6. The architecture of a compact buffer in the ingress.](image)
buffer.

At the egress we maintain a $bQ$-size SRAM serving as the head cache which is $1/k$ that of in Nemo. (The actual size of head cache relies on the configuration of ECQF-MMA[1]. It could be $(bQ+\ln Q)$ in the worst case. However, it is always $1/k$ that of in Nemo. Detailed analysis will be presented in Section 6.2.) Whenever the scheduler requests data from a queue, its corresponding $b$-sized data (i.e. the first chunk of this queue in the order of FIFO) is fetched from the corresponding DRAM arbitrary. When multiple chunks need to be fetched out from the same DRAM (i.e. the heads of current active queues locate at the same DRAM chip), conflict happens. An arbiter is introduced to solve this problem.

The internal structure of the arbiter is shown in Fig. 7. For each DRAM, the arbiter maintains a separated request list and $k$ request lists in total. Whenever a new request arrives at the arbiter, it will first be identified which specific queue it belongs to. Then, its current RRR sequence will be derived according to the $\log k$ bits information attached to the head of every $kb$-sized chunk. Referring to its round robin counter, the location of its head chunk and the corresponding request list can be determined. For every round, only the oldest request of each request list can be issued to the DRAMs. Since the queuing length of the request lists could be different, the delay that a request is satisfied varies depending on current workload and the conflict status. In the next section, we will prove that such kind of request queuing delay and the queuing delay in the inner-front-buffers can be upper-bounded, given satisfying certain conditions.

![Fig. 7. The architecture of an arbiter.](image)

To prevent any blocking of data during the initial stage, as we start to write into the packet buffer, packets are written to the head cache first; so they are available immediately if a queue is read. To accomplish this, the architecture in Fig. 6 has a direct-write path for packets from the writer, to be written directly into the head cache.

### 6 Performance Analysis and Simulation Study for Compact Packet Buffer with RRR

#### 6.1 Mathematically Guaranteed Performance

We present a detailed analysis showing that the proposed compact buffer design provides guaranteed performance in terms of upper-bounded delay and drop rate, when a small speedup factor is provided.

Since the signal transferring delay between SRAMs and DRAMs depends on the design of specific circuit board, we omit it in this paper. This assumption applies for both mathematical analysis and simulation unless otherwise stated. Therefore, the overall delay of the proposed system in real-life has to be slightly higher than that of derived in this paper, e.g. less than $10T_{RC}$.

As a tradeoff between storage and uniform allocation of incoming traffic, the RRR adopts a fast batch load scheme that may lead to a short-term unbalanced traffic allocation among DRAMs. However, for a specific single queue, the RRR guarantees the traffic allocation gap between any pair of DRAMs to be no more than $b$, as $kb$-size data of each queue is always uniformly distributed among $k$ DRAMs. Given a finite depth of the inner-front-buffers, it provides the system with a fixed time window that allows unbalanced traffic to be temporarily buffered.

Given the difficulty of modeling real-life traffic, our analysis is based on the worst case. We assume that the buffer time window is so small that none of the queues has the chance to spread their traffic among multiple DRAMs. In other words, if the queue does contribute some traffic in a small time window, it always contributes exactly one $b$-sized chunk to a single DRAM that results in the most unbalanced traffic allocation. Further considering the random sequence generated by the RRR, the proposed architecture can be modeled as $k$ independently and identically distributed finite capacity $M/D/1/n$ queuing systems.

Because of space limitations, details are omitted here. The traffic intensity of each queue is defined as $\rho$. Let $Q$ be the random variable of $Q_i := Q(t_i)$, where $Q(t_i)$ denotes the number of the chunks in the inner-front-buffer at time of $t_i$.

**Proposition:** Given all the parameters including the maximum buffer depth of $n$, the loss probability at a queue can be computed as follows.

$$Pr(Q > n) = 1 - \sum_{i=0}^{n} Pr(Q = i)$$

Where $Pr(Q = n)$ is defined as follows.
Pr(Q = n) = (1 - ρ) \sum_{i=0}^{n} \frac{e^{\rho(-1)^{n-i}(\rho + n - i)\rho^{n-i-1}}}{(n - i)!} 

(2)

Proof:
The probability generating function, in terms of the Laplace transform of the service time distribution.

Since the service times are iid, we take a random variable B with distribution function B(x) and density function b(x) as representative for all service times. As usual, the arrival process is a Poisson process with parameter λ. Next, the family of random variables V_i denotes the respective number of chunk arrivals in the i-th service time. Clearly, V_i depends on the length of the i-th service period, which always has the distribution B(x). Thus, it is reasonable to assume that all V_i have the same distribution and thus form an iid sequence of random variables.

Define V as the random variable of V_i, its probability generating function in terms of the Laplace transform is defined as:

\[ G_V(z) = \sum_{i=0}^{\infty} \int_{0}^{\infty} p(i, \lambda x)b(x)dx \cdot z^i = L_B(\lambda(1 - z)) \] (3)

According to the Pollazek-Khintchine transform equation:

\[ G_N(z) = \frac{(1 - \rho)(1 - z)}{L_B(\lambda(1 - z)) - z} \] (4)

Then we get

\[ G_V(z) = (1 - \rho)(1 - z) \sum_{i=0}^{\infty} b^i \rho(1 - z) \] (5)

Change (5) in a form of \( \sum_{i=0}^{\infty} b^i \) by using the definition of the exponential series, we finally arrive at (2).

In TABLE 1, we list the values of n needed to guarantee a steady state buffer (i.e., 10^6 loss probability). For example, with 90% traffic intensity, we choose n=101b as a typical value of inner-front-buffer size to achieve a loss probability less than 10^{-9}. Consequently, the corresponding response time is upper-bounded by 101T_{RC} with a probability of (1-10^{-9}).

<table>
<thead>
<tr>
<th>TRAFFIC INTENSITY vs FIFO SIZE FOR 10^6 LOSS PROBABILITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traffic Intensity</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>n</td>
</tr>
</tbody>
</table>

**Proposition:** Any output requests can be satisfied within 202T_{RC} with a probability of (1-10^{-9}), given (10/9) speedup factor to a compact buffer.

**Proof:**
Due to the symmetric architecture of the proposed system, the conclusion we derived for the ingress can be directly applied for the egress. The corresponding queuing delay for any request is upper-bounded by 101T_{RC} with a probability of (1-10^{-9}), given (10/9) speedup factor at the egress.

Therefore, by introducing a constant delay that lasts for (101 T_{RC} + 101 T_{RC}) = 202T_{RC} for all output requests, the probability that the cells are out of order is less than 10^{-9} as well.

Recall that the typical values of b for currently available DRAMS are around 64Bytes to 320Bytes. When b=320Bytes, an inner-front-buffer requires about 32KB storage and the maximum delay is about 49.5ns*202 < 1*10^5. By using a reduced latency DRAM chip, when b equals to 64Bytes, an inner-front-buffer costs only 6.4KB storage, and the maximal delay is reduced to 3.03us. Therefore, the maximum delay of a compact buffer is always a few microseconds.

For a system with a temporary buffer in the ingress, the pattern of input traffic can get distorted, thus creating a different pattern of traffic being injected to the inner-front-buffers. For example, with a modest intensity of traffic and a relatively large size of tail cache, there could be no input to the inner-front-buffers during the initial period. Since b is relatively small, we expect that such kind of distortion will not change the system performance significantly. However, in some extreme cases when the system is injected with modest traffic, this distortion could be amplified creating an illusion that the system performs beyond the theoretical bounds. We will present detailed analysis based on concrete examples in latter simulations.

**TABLE 2** compares the key features of the algorithm proposed in this paper with the conventional ones.

<table>
<thead>
<tr>
<th>COMPARISON AMONG DIFFERENT ALGORITHMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM Number</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>SRAM Size</td>
</tr>
<tr>
<td>SRAM Bandwidth</td>
</tr>
<tr>
<td>Effective Throughput</td>
</tr>
<tr>
<td>Rely on Traffic Pattern</td>
</tr>
</tbody>
</table>

6.2 Influence of Extra Delay
In a compact buffer, any request must be postponed for additional 202T_{RC} to ensure in-order processing. Since such delay lasts for only a few microseconds, it will not decline the quality of service significantly. However, it may change the time complexity of MMA and affect the scalability of the entire compact buffer.

For both ingress and egress, the dispatching of cells and requests requires nothing other than an O(1)-
complexity mapping. However, in order to ensure the tail cache never under-runs, a so-called lookahead request window is required by the ECQF-MMA, in order to predict the most critical queue and reduce the required SRAM size. According to the conclusions presented in [1], the relationship between the sizes of head cache and the lookahead window are shown in TABLE 3. Given enough requests are accumulated; a feasible scheduling can be found which reduces the required size of head cache. Consequently, the time complexity of ECQF-MMA is strictly related to the size of lookahead window.

TABLE 3

<table>
<thead>
<tr>
<th>Head Cache Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lookahead</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>Q(b−1)+1</td>
</tr>
</tbody>
</table>

Due to the additional delay, our compact buffer requires to increase the lookahead window by 20b. Apparently, when Q is much larger than 202, it affects the time complexity and the delay of the overall system very little. The overall time complexity of the ECQF-MMA still ranges from O(1) to O(Q) depending on the chosen size of lookahead window.

Although many previous algorithms [1][13]-[16] accept such Q(b−1)+1 lookahead windows by default, we must acknowledge that the ECQF-MMA itself may not scale when the number of queues is very large. Therefore, the actual size of head cache should be much larger than bQ when Q is quite large. E.g., O(bQlnQ). The RRR and its corresponding architecture we proposed in this paper do not reduce the time complexity of MMA, but reduces the size of SRAM to 1/k that of in Nemo.

6.3 Simulation Study and Results

We simulate a compact packet buffer with ten DRAMs based on the system architecture shown in Fig. 6. The simulation is based on 64 Byte cells.

We test the system performance with four different chunk sizes, viz., 64, 128, 320 and 640 Bytes, each of which consists from one to ten cells respectively. We define a timeslot as the minimum working span where each DRAM is capable of processing exactly one cell. Since there are ten DRAMs, at most ten cells are generated in each timeslot depending on the traffic intensity.

Since we consider fixed size cells in the simulation, we generate only two kinds of traffic patterns: uniform traffic and unbalanced (hotspot) traffic. For the uniform traffic, incoming cells are uniformly distributed across all the Q queues. Analysis of real-life traces indicated that the top 10% of flows account for over 90% of the packets and the bytes transmitted [21]. Therefore, for the unbalanced (hotspot) traffic, 90 percent of the cells are destined to only 10 percent of the queues. For both traffic patterns, there are 10^b active queues.

At the egress, we simulate two kinds of output behaviors: random scheduling and round robin scheduling. Due to the similarity of the results, only the result under random scheduling is revealed.

Fig. 8(a) and (b) show the number of backlogged cells in inner-front-buffers under different traffic patterns. We clearly observe that the average queue length is always less than 50, while the single inner-front-buffer still attains its peak size of around 160. Fig. 8(c) illustrates a clear comparison of system performance under both traffic patterns. The results are almost the same except that the system with uniform traffic suffers from more dramatic buildup during the initial period (i.e., first 10^3 time slots). The simple reason is that the uniform traffic distorts the traffic intensity. In the beginning of the simulation when the tail cache is empty, uniform traffic evenly allocates cells among all queues creating less “dispatchable” chunks than the unbalanced traffic. Thus, it actually leads to a lighter workload in the first 3*10^3 time slots as shown in Fig. 8(d). However, when the backlogged cells reside in the tail cache are finally released, it creates a burst which causes a sudden buildup in the inner-front-buffers. Because of space limitations, we only show the simulation results with the uniform traffic as it is the most rigorous traffic pattern for the proposed compact buffer architecture.

By further prolonging the simulation to 10^6 timeslots, we test the system performance under different configurations. With b = 1 cell, the tail cache is abolished. In Fig.
9(a), we observe that the corresponding practical result is always well upper-bounded by the theoretical value. By doubling the size of $b$, the distortion effect appears again. With low traffic intensity injected to the system, its maximum delay could go beyond the theoretical estimation. For example, with 30% traffic intensity and $b=2$ cells, a maximal delay of 26 timeslots is recorded which is 18.2% higher than the theoretical value of 22 timeslots. Detailed packet traces are shown in Fig. 9(b). The tail buffer distorts the traffic pattern dramatically creating a lot of new bursts. There are more red points than the black ones in the region of high traffic intensity, e.g. >40%.

With the increasing traffic intensity and the growing size of $b$, the distortion effect has been greatly weakened. In contrast, the system benefits from the increasing size of $b$ that provides positive reshaping of traffic. From both Fig. 9(a) and (c), we clearly find that the practical system performance under higher traffic intensity is always better than the theoretical value. With 70% traffic intensity and $b=5$ cells, a maximal delay of 132 time slots is recorded, while the theoretical value is 160 time slots. Meanwhile, with 90% traffic intensity and $b=10$ cells, a maximum delay of 839 time slots is recorded. In contrast to the theoretical value of 1010 time slots, the practical performance gains 16.9% improvement.

As illustrated in Fig. 9(d), the practical maximum response time for the output request is also well upper-bounded by the theoretical value. Besides the maximum response time, we also monitor its distribution. As illustrated in Fig. 9(e), when $b$ equals to 320 Bytes, more than 98.9% output request can be answered within 100 timeslots.

7 Distributed Memory Hierarchy

Compared with the previous approaches, the fast batch load scheme and the RRR algorithm reduce the SRAM size requirement to $1/k$ when a small amount of speedup is provided. Moreover, it does not rely on any traffic pattern, thus is capable of serving any application. However, it is still a traffic-agnostic approach which requires $O(bQ)$ size of SRAM. The scaling limitation remains.

To overcome this limitation, we first consider a simple architecture. Fig. 10 illustrates an example of a distributed packet buffer system consisting of three compact packet buffers. Queues are mapped to the compact packet buffers and this information is tracked in the queue table. A dispatching module located between the queue table and packet buffers delivers cells according to the tags. A FIFO queue in front of each compact buffer deals with short-term bursty traffic, and further forms a subsystem together with this compact buffer. For the sake of ease of description, this FIFO queue is named as outer-front-buffer.

At the outset, we must clarify a few important issues.

Fig. 9. System performance under different configurations
First, a compact packet buffer is injected with a maximum of 90% traffic intensity, in order to fulfill the speedup requirement. In other words, the output speed of an outer-front-buffer never exceeds 90% of the capacity of a single compact buffer.

Second, a compact buffer is capable of forwarding data in cells instead of chunks, because of its internal SRAM/DRAM hybrid architecture and the corresponding MMA. It provides a short access time in the ingress, however suffers from relatively large delay in avoiding the out-of-sequence problem. As we have proved that such delay can be upper-bounded, a compact buffer can be simply regarded as an array of multiple FIFO queues which is able to process individual cells with a constant delay (i.e. the upper-bound of delay in Section 6). Accordingly, a physical-queue is referred to a FIFO queue of a compact buffer and there are Q physical-queues in a single compact buffer.

Third, regarding a packet buffer as a black box, a logical-queue must be established whenever a stream of data (e.g. one flow or an aggregation of multiple flows) needs to be processed in a manner of FIFO. Given multiple compact buffers, we need to figure out a feasible scheme to map a logical-queue to single/multiple physical-queue(s). When a single physical-queue cannot satisfy the buffering needs of a logical-queue (e.g. the compact buffer where this physical-queue located has no resource to spare), then the packet distributor will map this logical-queue to multiple physical-queues that belong to different compact buffers. Similarly, mapping multiple active logical-queues to the physical-queues located at the same compact buffer may overwhelm this compact buffer. The packet distributor implements a suitable load-balancing scheme that keeps track of the information of each compact buffer, including the utilization of storage and bandwidth, and the number of active physical-queues. Using this information, we need to devise a load-balancing algorithm that can figure out the best configuration of the queue table and queue mapping which uses the smallest number of physical-queues to support the largest number of logical-queues.

In order to achieve the above, we first build a basic framework that allows logical-queues to dynamically switch from one physical-queue to another without any blocking. Unlike the simple linked-list based scheme in [14], in our distributed system, any logical-queue can be mapped to:

1) A single physical-queue: we refer to such logical-queues as "small" logical-queue.

2) K physical-queues that allocate in K subsystems respectively (K denotes the total number of subsystems in a distributed packet buffer): we refer to such logical-queues as "large" logical-queue.

This distinction for a logical queue can be applied both at the ingress (distributor) and egress (aggregator), thus leading to combinations of states. e.g. a logical-queue is in the state of "large-small" if it is served by all subsystems in ingress and served by only single subsystem in egress. Accordingly, a single bit should be introduced to mark this turning point as shown in Fig. 11, where each color represents a physical-queue in each subsystem and each rectangle represents a logical-queue.

Fig. 12 shows the state machine we have defined. Although there could be thousands of combinations, we only reserve six critical states. They are "unallocated", "large", "large-small", "large-medium", "large-large".
“small” and three intermediate states “large-small”, "small-large" and "large-small-large". Any logical-queue can switch its state between “small” and “large” smoothly with certain constraints. Meanwhile, it is strictly controlled that any logical-queue can only possess no more than three serving states at any time, i.e. at most 2 turning points. This helps the system minimize the overhead of state maintenance.

Based on the state machine above, we devise a load-balancing algorithm. The pseudo code of our algorithm is shown in Fig. 13. The algorithm is naturally separated into three tasks that are implemented at the distributor, compact packet buffer subsystem and the aggregator respectively. The tasks communicate with each other through the centralized queue table.

Here are some typical behaviors of the load-balancing algorithm. Whenever the first cell of a new logical-queue arrives, the distributor maps it to a subsystem that is currently the lightest loaded. For this new logical-queue, the destination subsystem reserves an empty physical-queue and updates the queue table and changes the state of this logical-queue from “unallocated” to “small”. For a single logical queue, the state of “small” could last for quite a long time. When the logical-queue becomes empty, the state of logical-queue is changed back to “unallocated”.

If a subsystem is temporarily overloaded, (i.e. the backlogged cells residing at the outer-front-buffer is beyond a threshold.) the subsystem can divert the newly arriving cells to other subsystems. The diverting can be achieved by randomly changing the state of any new arrival cells to “large-small” if it is originally served as “small”. To be more precise, the subsystem will still accept any new arrival cells. But if the new arrival cells belongs to a logical-queue which is originally served by this subsystem only (i.e. its state is “small”), the subsystem will mark the cell, and update the queue table by changing its corresponding state from “small” to “large-small”. At the ingress, if the serving state of a logical-queue is changed to “large”, the cells of this logical-queue will be dispatched to all subsystems in a per-flow round-robin manner. In this fashion, given a distributed system consisted with K subsystems, (K-1)/K of the traffic of this “small” logical-queue can immediately be diverted to other subsystems which helps to relieve the burden of the overloaded subsystem. As the output continues, the “large-small” logical-queue will be updated to “large” logical-queue when the previously marked cell is fetched.

Now this large logical-queue can be absorbed by lightly loaded subsystems. Generally speaking, it is a reverse process of the diversion described above where an intermediate state called “small-large” is introduced. To be more comprehensive, there is one additional state called “large-small-large” in our state machine. The purpose of introducing this state is to prevent any subsystem from absorbing “poison” logical-queues. A poison logical-queue is a logical-queue that bears considerable amount of traffic that cannot be easily handled by a single subsystem. When a poison logical-queue is absorbed by a subsystem, it will cause a rapid buildup of the outer-front-buffer in this subsystem forcing it to divert several of “small” logical-queues, which is highly inefficient. In some extreme cases when the only feasible assignment is to serve this “poison” logical-queue by all subsystems, absorbing a poison logical-queue blindly will lead to unnecessary drops due to finite capacity of outer-front buffers. In contrast, with the state of “large-small-large”, it provides us with an alternative choice that any logical-queue can be diverted immediately at any time.

8 Mathematical Analysis of Distributed Packet Buffer

In this section, we present a detailed explanation about the parameters of our load-balancing algorithm, such as MaxDivertTimes, THRESHOLD and DelayFactor. In particular, how these parameters affect system performance. It is difficult to estimate the traffic of a logical-queue in real life. Our load-balancing algorithm introduces a probabilistic method which distinguishes the type of logical-
queue dynamically. Assume a logical-queue bears $P$ (0 $\leq P \leq 1$) unit of traffic and a subsystem is capable of serving 1 unit of traffic at most (after considering the speedup), this logical-queue has a probability of $P$ to be diverted when it is served as “small”. As the queue state changes dynamically, any long-lasting active logical-queue is finally served as a large logical-queue. Because the logical-queue which has been diverted for no less than MaxDivertTimes cannot be absorbed any longer according to the load-balancing algorithm. Our key observation is that a logical-queue bearing more traffic has higher probability to be diverted. Hence, the expected time that a logical-queue achieves its final state could vary greatly.

**Proposition:** The expected number of timeslots that a logical-queue can be absorbed is given as:

$$\text{ExpectedTime} = \sum_{i=0}^{\infty} C_i^m \cdot (1 - P)^i (m + i)$$  

where $m$ is equal to MaxDivertTimes.

**Proof:**

A logical-queue is diverted for the $m$-th times at the $(m+i)$-th timeslot, its probability can be formulated as:

$$Pr(m, m + i) = C_i^m \cdot (1 - P)^i$$  

Thus, the expected number of timeslots that a logical-queue is diverted for exactly $m$ times can be formulated as:

$$\text{ExpectedTime} = \sum_{i=0}^{\infty} Pr(m, m + i) \cdot (m + i)$$

Set $m$ to the MaxDivertTimes.

The ExpectedTime decreases exponentially with $P$ while increasing linearly with $m$. Thus, given the average lifetime of logical-queues, we can achieve approximation of distinguished services for different logical-queues when $m$ is appropriately chosen. E.g., the state of a logical-queue bearing modest traffic varies between large and small rarely and remains small for the most of the time. In contrast, a logical-queue bearing intensive traffic quickly stays at “large” after a series of frequent state alternations.

As the first version of our distributed packet buffer design, the load-balancing algorithm only takes the bandwidth into consideration. In particular, the algorithm only monitors the loading conditions of outer-front-buffers (i.e., the number of backlogged cells in the outer-front-buffers). One straightforward concern is that the number of active physical-queues in each subsystem could vary greatly and further result in unbalanced mapping and allocation of physical queues. Here, we prove that such kind of unbalanced mapping and allocation can be neglected when the number of active logical-queues is large enough.

First, we must clarify that those “large” logical-queues do not affect this conclusion, because their workloads are always uniformly distributed.

**Proposition:** Considering a simple condition where no subsystem exceeds its maximal capacity (after considering the speedup). Despite the variable traffic intensity of logical-queues, our load-balancing can be seen as a randomized scheduling as long as no cell is backlogged in the outer-front-buffers.

**Proof:**

Since the number of backlogged cells in the outer-front-buffers is always zero, the destinations of incoming cells are randomly selected. Meanwhile, when no subsystem exceeds its maximal capacity, the traffic intensity of logical-queues is irrelevant.

**Proposition:** In a distributed system with $K$ subsystems (each subsystem with $k$ DRAMs), the probability that none of the subsystems received $1.1$ times the average number of logical-queues (denoted by $\mu$) can be formulated as follows.

$$\Pr[X_i < (1 + 0.1)\mu \text{ for any } i \leq K] > 1 - K^{-1 - c \delta 2 / 400}$$

**Proof:**

According to the Chernoff bound [20], in a distributed system with $K$ subsystems (each subsystem with $k$ DRAMs), the probability that the $i$-th subsystem received more than $(1 + \delta)\mu$ logical-queues can be formulated as follows.

$$\Pr[X_i > (1 + \delta)\mu] < \left[\frac{e^{\delta}}{(1 + \delta)^{(1 + \delta)}}\right]^\mu$$

When $\delta \leq 2e - 1$,

$$\Pr[X_i > (1 + \delta)\mu] < e^{-\mu \delta^2 / 4}$$

Given $\mu = C^{*}\ln K$ where $C$ is a constant,

$$\Pr[X_i > (1 + \delta)\mu] < e^{-C^{*}\ln K \cdot \delta^2 / 4} = K^{-C^{*}\delta^2 / 4}$$

According to the union bound,

$$\Pr[X_i < (1 + \delta)\mu \text{ for any } i \leq K] > 1 - K^{1 - C^{*}\delta^2 / 4}$$

Set $\delta$ to 0.1.

This probability increases with the number of active logical-queues. When $K=4$ and there are 41600 ($>4^{*}5200\ln K$) active logical-queues, there is at least $99.999999\%$ chance that no subsystem gets more than $1.1$ times of the average number of logical-queues.

Now, let us consider more complicated cases where subsystems are unevenly loaded. Since any logical-queue can dynamically switch its state between small and large for several times and the large logical-queues can be randomly absorbed by lightly loaded subsystems, this phenomenon is equivalent to randomly dispatch any single
logical-queues for many times which yields more uniform distribution. In other words, the queue distribution in such cases is at least as good as the simple case. Our later simulations confirmed it.

Finally, we present a detailed analysis showing that the transfer delay of cells can be upper-bounded by a constant, thus a delay lasting for $DelayFactor$ timeslots guarantees in-order data operation within each logical-queue. This conclusion is based on two key features. First, each cell is transferred from outer-front-buffer to compact buffer with constant rate. Once a cell is dispatched into a subsystem, the only delay lies on the queuing delay inside both outer-front-buffers and inner-front-buffers. Second, the state machine we defined for active logical-queues allows immediate diverting at any time and the diverted traffics are always dispatched in round-robin fashion. Assuming that the queuing delay inside the inner-front-buffers is a constant delay (i.e. the upper-bound we derived in Section 6), the proposed distributed scheme can be again modeled as $K$ independly and identically distributed finite capacity $M/D/1/n$ queuing systems.

Similar to the analysis we have conducted for the compact buffer, given all the parameters including the maximal buffer depth of $n$, we can derive the loss probability and the expected response time (queue waiting time). Trading off between the loss probability and the response time, when the traffic intensity of the entire packet buffer is $81\%$, we choose $n=101^\ast64$Bytes as a typical value of outer-front-buffer depth for each subsystem. Accordingly, the loss probability in the outer-front-buffers can be neglected as it reaches $10^\ast9$.

Meanwhile, since the delay introduced by a compact buffer is upper-bounded by $2027_B$ with a probability of more than $(1-10^\ast0)$. Given $DelayFactor = 2027_B + (101^\ast64Bytes)/0.9^\ast BW_{overall}$, the probability that out-order happens is no more than $2^\ast10^\ast9$, where $BW_{overall}$ denotes the overall bandwidth of a compact packet buffer. Fig. 14 shows the overall result of our queuing analysis. The entire system requires $(1/0.81)$ speedup factor, that is $(1/0.9)$ for the cut $\alpha$ and additional $(1/0.9)$ for the cut $\beta$.

Notice that the practical inputs is always time slotted, thus the system performance in practice should be at least as good as it is derived above. Meanwhile, recall that our load-balancing algorithm can perform the same as a per-queue round-robin algorithm when the system is heavily loaded. Accordingly, the queuing delay in the outer-front-buffers could be further reduced.

Obviously, the selection of parameter $THRESHOLD$ during this process is extremely important as it determines the time when the load-balancing algorithm starts to behave like a greedy algorithm. A big $THRESHOLD$ helps to stabilize the queue states preventing unnecessary fluctuations. On the other hand, a small $THRESHOLD$ helps to strength load-balancing, increasing the utilization of outer-front-buffers so as to reduce the queuing delay.

It is straightforward to show that $THRESHOLD$ should be no more than half of the maximal queuing delay in an outer-front-buffer. We are also aware that the utilization of outer-front-buffers with load-balancing should be better than that of in a PHSD[16]. So $THRESHOLD$ can be further decreased. In the latter simulations, we found that the average queuing delay for PHSD with $80\%$ traffic intensity is around $40$, thus a $THRESHOLD$ around $(40/4)$ should be a good choice. In this paper, we choose $THRESHOLD$ equals to $10$.

9 SIMULATION RESULTS

Our experimental results are presented in this section. Unless otherwise specified, the default for all the experiments is as specified in TABLE 4.

TABLE 4

<table>
<thead>
<tr>
<th>Default Parameters</th>
<th>64Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell Size</td>
<td>64Bytes</td>
</tr>
<tr>
<td>Chunk Size</td>
<td>5Cells (320Bytes)</td>
</tr>
<tr>
<td>Number of DRAMs in each subsystem</td>
<td>10</td>
</tr>
<tr>
<td>The maximal depth of inner-front-buffers</td>
<td>101^\ast320Bytes</td>
</tr>
<tr>
<td>The maximal depth of request lists</td>
<td>equivalent to 101chunks</td>
</tr>
<tr>
<td>Number of subsystems</td>
<td>4</td>
</tr>
<tr>
<td>The maximal depth of outer-front-buffers</td>
<td>101^\ast64Bytes</td>
</tr>
<tr>
<td>(\text{MaxDivertTimes})</td>
<td>10</td>
</tr>
<tr>
<td>(\text{DelayFactor})</td>
<td>$2027_B + (101^\ast64Bytes)/0.9^\ast BW_{overall} = 9191$ timeslots*</td>
</tr>
</tbody>
</table>

*We define a timeslot as the minimal working span where each compact buffer is capable of processing exactly one cell. In order to fulfill the speedup requirement, inside a compact buffer, a DRAM needs only 9 timeslots to...
process a single cell while requesting 45 timeslots to process a single chunk.

Since there are four subsystems, for each timeslot, 3.6 cells on average can be generated depending on the traffic intensity. To be more specific, for each timeslot, there are at most 4 cells can be generated where each of them is generated with a maximal probability of 90%. In this way, we satisfy the extra \(1/0.9\) speedup of the cut \(\alpha\). We must clarify that “3.6 cells per timeslot” is the maximal traffic intensity that system allows, which can be regarded as 81% traffic intensity, when the speedup inside a compact buffer is taken into consideration.

Meanwhile, in order to observe the dynamic behavior of the entire system, the simulations are always separated into three phases. Assume the simulation lasts for \(X\) timeslots. For the first \(0.2X\) timeslots, there is only input without output where cells are backlogged. In this way, we can create an initial backlog and also simulate the situation when the congestion happens. After \(0.2X+1\) timeslots, a full-speed output (i.e. 90% when the internal speedup of a compact buffer is considered) based on random scheduling (i.e. every logical-queue has the same probability to be fetched out) begins while input maintains, which represents the normal conditions. With backlogged cells in the first phase, we can monitor the system performance in detail, especially how the load-balancing algorithm behaves. After \(0.5X\) timeslots, the input stops while only the output maintains fetching any backlogged cells. In this way, we can simulate the situation when the system is lightly loaded. Moreover, we choose the parallel system (i.e. PHSD in [16]) as the basic reference standard of our distributed system. Because it is the best parallel architecture we known so far which represents the previous “traffic-agnostic” approaches.

To make it a fair comparison, the PHSD architecture also consists of 40 DRAMs. Accordingly, a logical-queue is always mapped to 40 physical-queues which require \(2b\) size of SRAM for each. Thus, a logical-queue inside the PHSD requires \(80b\) size of SRAM in total. In contrast, a physical-queue inside the distributed buffer requires \(2b\) size of SRAM. It can be mapped to at most 4 physical-queues simultaneously.

In the first part of our simulations, we inject the distributed packet buffer with only 100 active data streams to have a clear observation on the load-balancing algorithm. The input traffic pattern is unbalanced (hotspot) traffic that top 10% active data streams account for 90% of the cells in this simulation. In other words, a large data stream typically contributes 81 times as many cells as that of a small data stream. We arbitrarily selected two data streams (one large, one small) to have a close-up view of their states. As illustrated in Fig. 15, the queue states of these two data streams change frequently. We observe that the large data stream is mapped to multiple physical-
queues most of the time (i.e. the queue state finally remains as “large” after MaxDivertTimes changes of queue state), while the small data stream is served by single physical-queue mostly. The probability method works quite well.

Further increasing the number of active data streams to $4*10^9$, we simulate the system performance under both uniform traffic and unbalanced (hotspot) traffic (i.e. top 10% data streams account for 90% of the overall traffic). Fig. 16 presents the results of total number of active queues under different situations. The simulation lasts for $10^7$ timeslots. As shown in the figures, the parallel system (i.e. PHSD in [16]) always dispatches the logical-queues in per-queue round-robin introducing a lot of overheads. The total number of active physical-queues always achieves $1.6*10^9$ during the initial periods no matter the intensity of injected traffic. In contrast, our distributed system which introduces the “traffic-aware” approach always results in much less active physical-queues. Taking unbalanced traffic pattern and 78% traffic intensity as an example, in the first phase, the distributed scheme only maintain around $4.8*10^8$ active physical-queues which is about 3% that of PHSD. As the second phase starts, the number of active physical-queues for both architectures drops greatly. However, the distributed one still outperforms the PHSD where $1.2*10^9$ and $1.7*10^9$ physical-queues are maintained respectively. As the further decreasing of traffic intensities, our distributed system shows more obvious advantages.

Fig. 17 presents the comparison of SRAM occupancy between both architectures. Given a modest traffic intensity (i.e. <76%), the distributed architecture reduces the SRAM occupancy by more than 95% no matter the traffic pattern. Even with a maximal 81% of the traffic intensity, still, the distributed architecture requires <9.5% of the SRAM than the PHSD during its peak time. With 63% traffic intensity, the distributed architecture requires only 2.6% the overall size of SRAM.

Since our algorithm does not refer the physical-queues assignment status in balancing, we are curious about the actual distribution of active queues among subsystems. Fig. 18 shows the actual distribution of active physical-queues among four subsystems where $4*10^9$ data-streams are generated. We observe that the unbalanced distribution is always trivial for both traffic patterns. It matches pervious mathematical analyses.

We also monitor the length of outer-front-buffers. As shown in Fig. 19, the FIFO lengths for both architectures are much less than 101. Meanwhile, the distributed system performs almost the same as the PHSD. In Fig. 19 (a), the distributed architecture even outperforms the PHSD after $2*10^6$ timeslots, where only a few sampling points located beyond ten.

**10 IMPLEMENTATION ISSUES**

In this section, we discuss some implementation concerns and their possible solutions.

First, high-speed dispatchers and aggregators are very difficult to extend to large-scale. As demonstrated in Fig.
20, the distributed architecture enables us to build them in a hierarchical way that each level only handles limited number of physical ports, which is more efficient.

Second, a centralized queue table has been widely used in a packet buffer to track the status of individual logical-queues, including the memory address of current head/tail of a physical-queue, the per-queue round robin counters for both ingress and egress, and etc. With the increasing number of logical-queues and the overall throughput, a centralized queue table could become the bottleneck of entire system. The proposed distributed packet buffer architecture adopts a hierarchical structure which reduces the number of physical-queues significantly leading to a much smaller queue table. Taking the top level queue table as an example, it records the mapping information of only four compact buffers instead of 40 DRAMs.

To further optimize the system, we propose a pipelined query scheme which allows the queue table to be implemented by using off-chip memories. Fig. 21 is a simple demonstration. Instead of keeping only one FIFO, the system maintains two FIFOs which are used to store the data and control information of cells respectively. Whenever a cell arrives, it will be directly dispatched to the data FIFO while its corresponding query is issued to the queue table immediately. As soon as the queue table returns the result, the control information will be stored in the control FIFO. Finally, a cell tagged with the destination information can be derived by fetching data from both FIFOs, before being forwarding to a dispatcher. On the other hand, being aware of the data consistency problem introduced by the pipelined query, the system must provide a small fully index table to store the most recent updates. Fig. 22 demonstrates how the value of counters can be increased without disturbing the pipelined query. The depth of these two FIFOs and the entries number of fully index table rely on the round trip time of a query, typically less than 30. Therefore, they cost just a little and can be implemented inside a chip.

Besides, we also notice that majority of the queue table updates rarely. For example, the mapping information of a single logical-queue can be updated for at most MaxDivertTimes times in its entire lifetime. Accordingly, multiple read-only copies of the queue table can be maintained simultaneously to maximize the throughput. Meanwhile, a queue table can be decoupled into multiple sub-tables. For example, the RRIn and RRout can be maintained by ingress and egress respectively. These schemes could further increase the throughput of a queue table.

### 11 Conclusions

Building packet buffers based on a hybrid SRAM/DRAM architecture while introducing minimum overhead is the major issue discussed in this paper. To distinctly increase the throughput and storage capacity of a packet buffer, parallel mechanism using multiple DRAM chips should be deployed. Our analysis shows that previous algorithms make very little effects in exploring the advantage of parallel DRAMs leading to the requirement of large size SRAM and high time-complexity in memory management. In this paper, we present novel packet buffer architecture by using both fast batch load scheme and a hierarchical distributed structure. It reduces the requirement of SRAM size greatly. Both mathematical analysis and simulation results indicate that the proposed archi-
tecture provides guaranteed performance in terms of low time complexity, short access delay and upper-bounded drop rate, when a small speedup is provided.

ACKNOWLEDGMENT

The authors wish to thank A, B, C. This work was supported in part by a grant from XYZ.

REFERENCES


Dong Lin received the BSc degree in Computer Science from the Beijing University of Aeronautics & Astronautics in 2005 and MS degree in Computer Science from the Tsinghua University in 2008. He is currently working toward the PhD degree in computer science and engineering at the Hong Kong University of Science and Technology. His research interests are generally in computer networks. In particular, he is focusing on the design and analysis of high-performance switches/routers. He is also interested in wireless networks. He is a student member of the IEEE and the IEEE Communications Society.

Second B. Author Jr. biography appears here. Degrees achieved followed by current employment are listed, plus any major academic achievements.

Third C. Author is a member of the IEEE and the IEEE Computer Society.