Designing Packet Buffers in High-Bandwidth Switches and Routers

Dong Lin, Mounir Hamdi and Jogesh Muppala

Department of Computer Science and Engineering
Hong Kong University of Science and Technology, Hong Kong
ldcse@ust.hk, hamdi@cse.ust.hk, muppala@cs.ust.hk

Abstract—High-speed routers rely on well-designed packet buffers that support multiple queuing, large capacity and short response times. Some researchers suggested a combined SRAM/DRAM hierarchical buffer architecture to meet these challenges. However, both the SRAM and DRAM need to maintain a large number of dynamic queues which is a real challenge in practice and limits the scalability of these approaches.

In this paper, we present a scalable, efficient and novel distributed packet buffer architecture. Two fundamental issues need to be addressed to make this feasible: (a) how to design scalable packet buffers using independent buffer subsystems; and (b) how to dynamically balance the workload among multiple buffer subsystems without any blocking. We address these issues by first designing a basic framework that allows flows to dynamically switch from one subsystem to another without any blocking. Based on this framework, we further devise a load-balancing algorithm to meet the overall system requirements. 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 with large number of active connections.

***Index Terms*—Packet Buffer, Memory Hierarchy.**

I. Introduction

High-speed routers are increasingly being called upon to deal with the rapid increase in the communication link bandwidth, requiring them to switch packets extremely fast to keep up with the growing line rate. This demands sophisticated packet switching and buffering techniques. Packet buffers need to be designed to support large capacity, multiple queues and short response times.

The traditional rule of thumb for Internet routers states that the routers should be capable of buffering *RTT*R* [1] data, where *RTT* is a round-trip time for flows passing through the router, and *R* is the line rate. In [6][9][12], 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 40Gbps line rate incorporates a 2 GB packet buffer memory per line card and per side (ingress and egress) [13]. Meanwhile, in order to support fine-grained IP quality of service (QoS) requirements, nowadays, a packet buffer usually maintains thousands of queues. E.g. Juniper E-series routers [7] maintain as many as 64000 queues. Furthermore, a packet buffer should be capable of sustaining continuous data streams for both ingress and egress.

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 [14]-[18], or parallel DRAM [19] sandwiched between SRAMs.

In this paper, we review previous work and present a scalable, efficient and novel distributed packet buffer architecture. Mathematical analysis indicates that the proposed architecture together with its load-balancing algorithm provide guaranteed performance in terms of low time complexity, short access delay, upper-bounded drop rate and uniform allocation of resources. Simulation results further show that the distributed architecture reduces the number of active queues significantly, yielding a more efficient way of building a packet buffer.

The rest of the paper is organized as follows. In Section II, we briefly review the related work from the literature. In Section III, we describe the hardware architecture of the distributed packet buffer and its load-balancing algorithm. In Section IV, we present the analytical evaluation of some properties of our architecture. Then we evaluate the performance of our architecture using simulation in Section V. Finally, we conclude our work in Section VI.

II. Related Work

Current SRAM is fast enough with an access time of around 2.5 *ns* [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 is too large, around 40 *ns* [3]. 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.

This research work is partially supported by the following HKUST grant RPC07/08.EG09.

Iyer *et al.* [1] first introduced the basic Hybrid SRAM/DRAM (HSD) architecture with one large DRAM memory sandwiched between two smaller SRAM memories, where the two SRAMs hold heads and tails of all the flow queues and a 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). They proposed three MMAs, each with different tradeoffs between the size of the SRAM and the pipelined delay. The SRAM size requirement for the HSD architecture is *O(Qb)*, where *Q* is the number of flow queues, and *b* is the DRAM/SRAM access time ratio.

Shrimali *et al.* [14] modified the memory architecture in [1] by using *b* interleaved DRAMs. The memory management between the SRAM and the DRAM was based on a randomized algorithm, thus providing only probabilistic performance guarantees and suffering from out-of-sequence problem seriously.

Some MMAs [15]-[17] based on deterministic algorithms were proposed to avoid the out-of-sequence problem. Each of them can be regarded as a solution to the problem of a bipartite graph for cumulative matching [15]. The major drawback of deterministic algorithms lies on their high time complexity in finding the maximum matching. Given the fact that a single cell could cost $O(Q)$ 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 [16].

Feng *et al.* [19] proposed a parallel hybrid SRAM/DRAM architecture named PHSD. Compared with the interleaved architectures [14]-[17], the PHSD reduces the time complexity of MMA to $O(k)$, but still inherits most of the drawbacks. Specifically, it still requires $O(Qb)$-size of SRAM in the worst case.

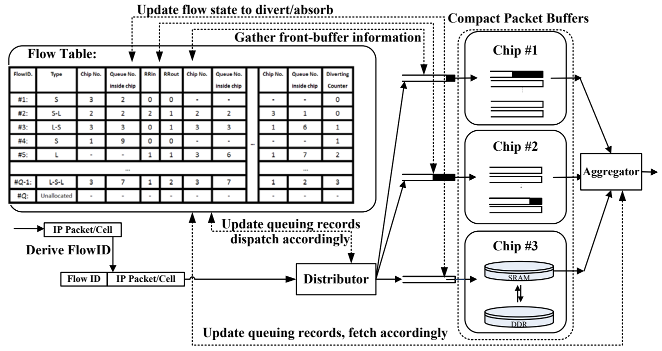
Designing a packet buffer for general purpose is always a difficult task. In contrast, for specific applications where buffer behavior is predictable, the task can be greatly simplified. Kabra et al. [18] introduced a parallel DRAM approach that benefits by the foregone departure time. Lin *et al.* [20] found the short-term stability of 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.

III. Distributed Memory Hierarchy

When we carefully examine the hierarchical packet buffer architectures, whether the HSD architecture [1], interleaved DRAMs [14]-[18], or parallel DRAM [19], they all rely on two parameters, Q and b. The required size of SRAM is always $O(Qb)$. According to the N^2-hypothesis, the number of flow connections is proportional to the square root of line rate [8], thus Q has to be increased continuously at a speed of $k^{1/2}$, where k is the increasing speed of the line rate. On the other hand, due to the sluggish advance in the access delay of DRAM, continuously decreasing the access time of SRAM will also make b increases at a speed of k. Hence, the size of SRAM has to be increased exponentially at a speed of $k^{3/2}$.

In our view, all packet buffering techniques so far have adopted a flow-agnostic approach while designing the packet buffering algorithms. We must clarify that even though the existing approaches do use Q flow queues, each flow is treated the same by the buffer management algorithms. No effort is made to exploit the inherent characteristics of the flows like the arrival rate, burst sizes, transit time requirements through the router etc. For small flows, this incurs significant overhead, even though there is no practical reason to maintain queues among multiple DRAM chips to support this flow. However a flow-aware approach to the problem, we believe, will yield new possibilities for conquering the scalability problem.

In this paper, we will investigate a new dimension to the problem, viz. how to extend the packet buffer architectures by using independent packet buffer sub-systems. Such distributed system where k independent packet buffers work together providing increasing performance but at a linearly increasing scale needs to be designed.


Update flow state to divert/absorb
Compact Packet Buffers
Gather front-buffer information
Flow Table:
Chip #1
Chip #2
Chip #3
Aggregator
IP Packet/Cell
Derive FlowID
Flow ID
IP Packet/Cell
Update queuing records dispatch accordingly
Distributor
SRAM
DDR
Update queuing records, fetch accordingly

FlowID.	Type	Chip No.	Queue No. inside chip	RRin	RRout	Chip No.	Queue No. inside chip		Chip No.	Queue No. inside chip	Diverting Counter
#1:	S	3	2	0	0	-	-		-	-	0
#2:	S-L	2	2	2	1	2	2		3	1	0
#3:	L-S	3	3	0	1	3	3		1	6	1
#4:	S	1	9	0	0	-	-	...	-	-	1
#5:	L	-	-	1	1	3	6		1	7	2
...									...		
#Q-1:	L-S-L	3	7	1	2	3	7		1	2	3
#Q:	Unallocated	-	-	-	-	-	-		-	-	-

Fig. 1. Distributed scalable packet buffer architecture

Deciding on a suitable distributed system architecture that achieves the best overall performance while incurring minimal overhead is not straight-forward. As a starting point, we consider a simple architecture. Fig. 1 illustrates an example of a distributed packet buffer system consisting of three compact packet buffers. Each compact packet buffer acts as an independent unit implementing its own packet buffer architecture and memory management algorithm. Flows are mapped to the compact packet buffers and this information is tracked in the flow table. A dispatching module located between the flow table and packet buffers delivers packets 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 *front-buffer*.

When a single compact buffer cannot satisfy the buffering needs of a flow, then the packet distributor will map the same flow to multiple buffers. Similarly, mapping multiple active flows to a single compact buffer may overwhelm it. 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 queues. Using this information, we need to devise a load-balancing algorithm that can figure out the best configuration of the flow table and flow mapping.

In order to achieve the above, we first build a basic framework which allows flows to dynamically switch from one subsystem to another without any blocking. Unlike the simple linked-list based scheme in [16], in our distributed system, any flow can be mapped to:

a) A single subsystem: we refer to such flows as "small" flows.

b) All subsystems: we refer to such flows as "large" flows.

This distinction for a flow can be applied both at the ingress (distributor) and egress (aggregator), thus leading to combinations of states. e.g. a flow 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. 2, where each color represents a physical queue in each subsystem.

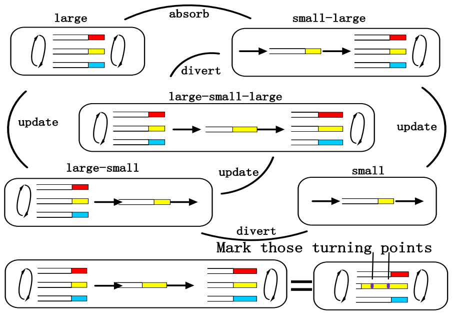


Fig. 2. Flow Allocation among Multiple Subsystems

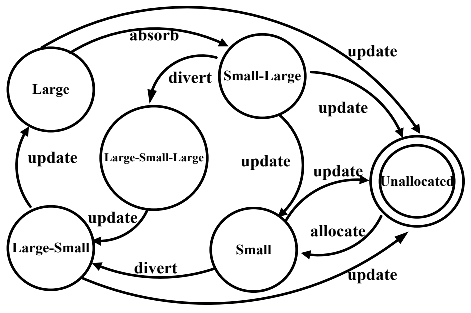


Fig. 3. Flow States

Fig. 3 shows the state machine we have defined. Although there could be thousands of combinations, we only reserve six critical states. They are "unallocated", "large", "small" and three intermediate states "large-small", "small-large" and "large-small-large". Any flow can switch its state between "small" and "large" smoothly with certain constrains. Meanwhile, it is strictly controlled that any flow 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. 4. 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 flow table.

```
Distributor:
For each cell, derive its flow ID, get access to the flow table accordingly.
If the state of this flow is "unallocated", then
1) Find a subsystem currently the lightest loaded by comparing the length
of waiting lists in each front-buffer. In case all are equal, select randomly.
2) Allocate this new flow to an empty queue of the selected subsystem.
3) Record this information and update the flow state to "small".
4) Dispatch this cell accordingly.
Else if the state of this flow is "small" or "small-large", then
Dispatch the cell to the corresponding subsystem accordingly.
Else then
Dispatch the cell to the corresponding subsystem in per-flow round-robin*.
Each Compact Packet Buffer Subsystem:
Fetch one cell from the front-buffer, save it to the corresponding queue.
If the number of backlogged cells is no more than THRESHOLD &_
  current cell belongs to a flow which is originally severed as large &_
  it has been diverted for less than MaxDivertTimes, then
Update its flow state to small-large, bit-mark this cell as turning point.
Else if the number of backlogged cells is more than 2* THRESHOLD &_
  current cell belongs to a flow which is originally severed as small or
small-large, then
Update its flow state to small-large, bit-mark this cell as turning point,
increase the MaxDivertTimes by one.
Else then do nothing.
Aggregator:
Given a flow ID to fetch,
1) Delay this request for DelayFactor timeslots
2) Check the flow table.
3) Fetch one cell in per-flow round-robin* or from one subsystem only.
4) If it is a turning point, then update the flow state accordingly.
```

* per-flow round-robin: if a packet is the i-th cell in a flow, then it should be dispatched into DRAM j, where $j = i \bmod k$.

Fig. 4. Load-balancing algorithm

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

If a subsystem is temporarily overloaded, (i.e. the backlogged cells residing at the 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 flow which is originally served by this subsystem only (i.e. its state is "small"), the subsystem will mark the cell, and update the flow table by changing its corresponding flow state from "small" to "large-small". At the ingress, if the serving state of a flow is changed to "large", the cells of this flow 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" flow can immediately be diverted to other subsystems which helps to relieve the burden of the overloaded subsystem. As the output continues, the "large-small" flow will be updated to "large" flow when the previously marked cell is fetched.

Now this large flow can be absorbed by lightly loaded subsystems. Generally speaking, it is a reverse process of the flow diversion described above except that one state called "large-small-large" is introduced. With this state, it provides us with an alternative choice that any flow can be diverted immediately at any time.

IV. Analytical Evaluation

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 flow in real life. Our load-balancing algorithm introduces a probabilistic method which distinguishes the type of flow dynamically. Assume a flow contributes P ($0<P\leqslant 1$) unit of traffic and a subsystem is capable of serving 1 unit of traffic at most, this flow has a probability of P to be diverted when it is served as small flow. As the flow state changes dynamically, any long-lasting active flow is finally served as a large flow. Because the flow 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 flow contributing more traffic has higher probability to be diverted. Hence, the expected time that a flow achieves its final state could vary greatly. It can be formulated as follows, where m equals to the *MaxDivertTimes*.

$$ExpectedTime = \sum_{i=0}^{\infty} C_{m+i-1}^{m-1} P^m (1-P)^i \ (m+i) \qquad (1)$$

It increases exponentially with P while linearly with m. Thus, given average lifetime of flows, we can achieve approximation of distinguished services for different flows when m is appropriately chosen. e.g. the state of small flow varies between large and small rarely and remains small for the most of the time. In contrast, a large flow quickly stays at "large" after a series of frequent state alternations.

The proposed load-balancing algorithm only takes the bandwidth into consideration. One straightforward concern is that the number of active queues in each subsystem could vary greatly and further result in unbalanced allocation of physical queues. Here, we prove that such kind of unbalanced allocation can be neglected when the number of active flows is large enough. First, considering a simple condition where no subsystem exceeds its maximal capacity. Despite the variable size of flows, our load-balancing can be seen as a randomized scheduling as long as no cell is backlogged cells in the front-buffers. According to the *chernoff* bound [22], in a distributed system with K subsystems and c*ln$K$ flows, the probability that the i-th subsystem received 1.1 times the average number of flows can be formulated as follows.

$$\Pr[X_i > (1 + 0.1 * c * \ln K)] < \mathrm{e}^{-\frac{c * \ln K}{400}} = K^{-\frac{c}{400}} \qquad (2)$$

According to the union bound,

$$\Pr[X_i < (1 + 0.1 * c * \ln K) \text{ for any } i \leq K] > 1 - K^{1-\frac{c}{400}} \qquad (3)$$

When K=4 and there are 10400 (>5200ln$K$) flows, there is at least 99.99999% chance that no subsystem gets more than 1.1 times of the average number of flows.

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

Next, 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 flow. This conclusion is based on two key features. First, each cell is transferred from front-buffer to DRAM with constant rate. Once a cell is dispatched into a subsystem, the only delay lies on the queuing delay inside front-buffers. Second, the state machine we defined for active flows allows immediate diverting at any time and the diverted traffics are always dispatched in round-robin fashion. As a result, the proposed scheme can be modeled as K independently and identically distributed finite capacity M/D/1/n queuing systems.

Because of space limitation, details are omitted here. Trading off between the loss probability and the response time, when $\rho = 0.9$ (90% traffic intensity), we choose n=100 as a typical value of buffer depth for each subsystem. Accordingly, the loss probability can be neglected as it reaches 10^{-9}. Thus, the response time is upper-bounded by 100 with a probability of $1\text{-}10^{-9}$. Given *DelayFactor* = 100, the probability that out-order happens is no more than 10^{-9}. 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. Recall that our load-balancing algorithm can perform the same as a greedy algorithm when the system is heavily loaded (i.e.100% traffic intensity). Thus, the utilization of subsystems is always 100% as long as no front-buffer goes empty. In other words, the queuing delay has to be finally stabilized with or without the speedup. The upper-bound of queuing delay does not rely on the speedup. A speedup just helps to reduce it.

Obviously, the selection of *THRESHOLD* during this

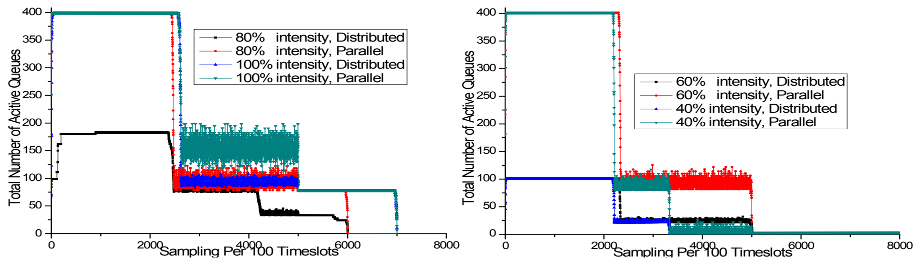


Fig. 5. The overall active queues for both architectures

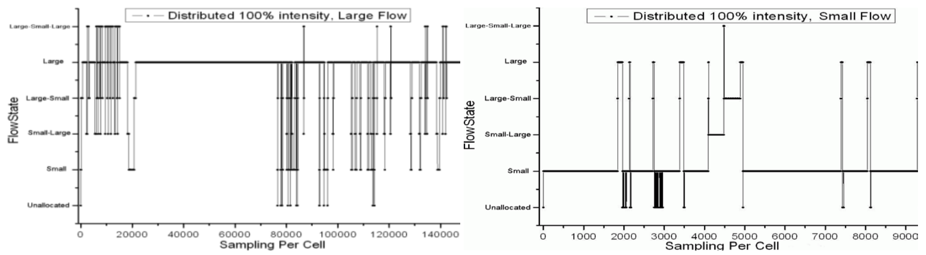


Fig. 6. Flow-aware services based on probability method

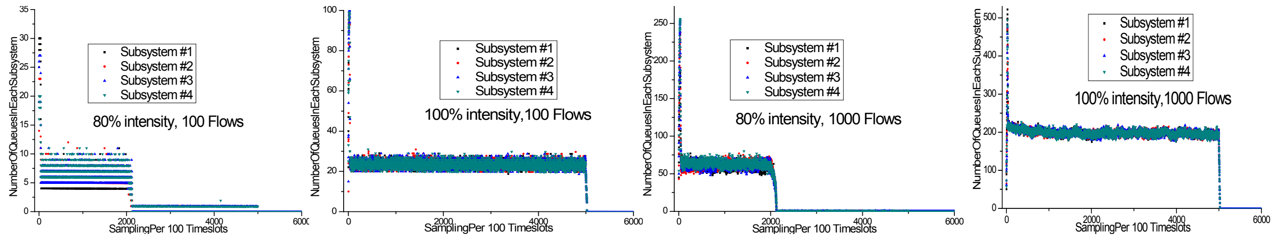


Fig. 7. Active queues allocations among four subsystems

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 flow states preventing unnecessary fluctuations. In contrast, a small *THRESHOLD* helps to strength the load-balancing increasing the utilization of front-buffers so as to reduce the upper-bound of queuing delay. It is straightforward that *THRESHOLD* should be less than 0.5**DelayFactor*. We are also aware that the utilization of front-buffers with load-balancing should be better than that of in a PHSD[19]. So *THRESHOLD* can be further decreased. In the latter simulations, we found that the average queuing delay for PHSD with 100% traffic intensity is around 50, thus a *THRESHOLD* around 10 should be a good choice.

V. Simulations

Our experimental results are presented in this section. Unless otherwise specified, the default for all the experiments is as specified in Table 1.

TABLE. 1. DEFAULT PARAMETERS

DelayFactor	100
THRESHOLD	10
The maximal depth of front-buffers	100
Number of subsystems	4

We define a timeslot as the minimal working span where each subsystem is capable of processing exactly one cell. Since there are four subsystems, for each timeslot, at most four cells are generated depending on the traffic intensity. Analyses on real-life traces indicated that the top 10% of flows account for over 90% of the packets and the bytes transmitted [23]. To be modest, in the following simulations, top 20% of flows accounts for 80% overall cells.

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 $0.2*X$ timeslots, there is only input without output where cells are backlogged. In this way, we can create an initial backlog and simulate the situation when the congestion happens. After $0.2*X+1$ timeslots, a full-speed output begins while input maintains. With backlogged cells in the first phase, we can monitor the system performance in detail, especially how the load-balancing algorithm behaves. After $0.5*X$ 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 [19]) as the basic reference standard of our distributed system. Because it is the best parallel architecture we known so far which represents the previous "flow-agnostic" approaches.

Fig. 5 presents the results of total number of active queues under different situations. There are 100 distinct flows in total and the simulation lasts for 10^7 timeslots. As shown in the figures, the parallel system (i.e. PHSD in [19]) always dispatches the flows in per-flow round-robin introducing a lot of overheads. The total number of active queues always achieves 400 during the initial periods no matter the intensity of injected traffic. In contrast, our distributed system which introduces the "flow-aware" approach always results in much less active queues. Taking 80% traffic intensity as an example, in the first phase, the distributed scheme only maintain around 165 active queues which is about 42% that of PHSD. As the second phase starts, the number of active queues for both architectures drops greatly. However, the distributed one still outperforms the PHSD where 35 and 100 queues are maintained respectively. As the further decreasing of traffic intensities, our distributed system shows more obvious advantages.

By further prolonging the simulation to 10^8 timeslots, we arbitrarily selected two flows (one large, one small) to have a close-up view of their states. As illustrated in Fig. 6, when the intensity of traffic is 100%, the flow states of these two flows change frequently. We observe that the large flow is mapped to multiple subsystems most of the time, while the small flow is served by single subsystem mostly. The probability method works quite well.

Since our algorithm does not refer the flow assignment status in balancing, we are curious about the actual distribution of active queues among subsystems. Fig. 7 shows the distribution of active queues among multiple subsystems. By increasing flow number from 100 to 1000, we observe that the unbalanced distribution is greatly improved which matches pervious mathematical analyses.

We also monitor the length of front-buffers. As shown in Fig. 8, the average FIFO lengths for both algorithms are much less than 100 and the distributed system always outperform the parallel one. We clearly observe that the average FIFO length of distributed system stays at a constant around 16, which matches the mathematic analyses as well. Besides smaller average value of FIFO length, the distributed system also performs more smoothly.

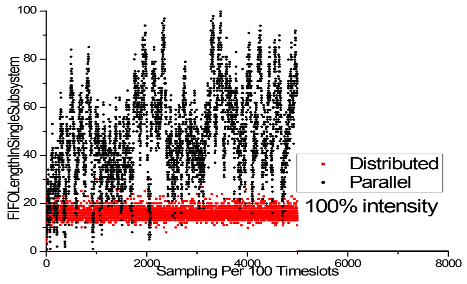


Fig. 8. The number of backlogged cells in a front-buffer

VI. Conclusion

Unlike the previous approaches [1][14]-[19], our design dispenses with both the head and tail caches, keeping only tiny distributed front-buffers inside each subsystem. Each flow is only mapped to approximately one queue in a compact buffer. Thus, it maintains much less physical queues compared to other approaches and reduces the size of SRAM significantly.

The significance of this research lies in the bold new direction towards distributed scalable buffer design that we plan to pursue. In particular, our approach yields a scalable, independent subsystem based packet buffer architecture that can easily be tailored to meet the specific line rate and traffic requirements for different switches, and match the flow-level requirements including bandwidth and delay guarantees within a switch. Our approach will also yield a power efficient buffer design, where parts of the buffer may be switched on and off based on the real-time traffic arrival rates and buffering requirements.

References

[1] S. Iyer, R. Kompella and N. McKeown, "Designing packet buffers for router linecards", in *IEEE Transactions on Networking*, vol.16, Jun. 2008, Issue 3.
[2] Samsung SRAM Chips, http://www.samsung.com/global/business/semiconductor/products/sram/Products_HighSpeedSRAM.html
[3] Samsung DRAM Chips, http://www.samsung.com/global/business/semiconductor/products/dram/Products_DRAM.html
[4] J.Corbal, R.Espasa, and M.Valero, "Command vector memory systems: High performance at low cost," *In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp.68-77, October 1998.
[5] K.G. Coffman and A.M.Odlyzko, "Is there a "Moore's Law" for data traffic?" *Handbook of Massive Data Sets*, eds., Kluwer, 2002,pp.47-93.
[6] G. Appenzeler, I.Keslassy, and N.McKeown, "Sizing Router Buffers," *ACM SIGCOMM Computer Comm. Rev.*, Vol. 34, No. 4, Oct. 2004, pp. 281-292.
[7] Juniper E Series Router, http://juniper.net/products/eseries/
[8] B.St.Arnaud, "Scaling issues on Internet networks," 2001, http://www.canet3.net/library/papers/scaling.pdf
[9] D. Wischik and N. McKeown, "Part I: Buffer Sizes for Core Routers," *Computer Comm. Rev.*, vol. 35, no. 3, pp. 75-78, 2005.
[10] G. Iannaccone, M. May, and C. Diot, "Aggregate Traffic Performance with Active Queue Management and Drop from Tail," *SIGCOMM*, vol. 37, pp. 277-306, 2001.
[11] C. Crisp, "Provisioning Internet Backbone Networks to Support Latency Sensitive Applications," PhD dissertation, Stanford Univ., June 2002.
[12] A. Dhamdhere, H. Jiang, and C. Dovrolis, "Buffer Sizing for Congested Internet Links," *Proc. IEEE Infocomm*, Mar. 2005.
[13] Cisco, "Cisco Carrier Router System," http://www.cisco.com/en/US/products/ps5763/index.html
[14] G. Shrimali and N. McKeown, "Building Packet Buffers with Interleaved Memories", *HPSR* 2005.
[15] F. Wang and M. Hamdi, "Scalable Router Memory Architecture Based on Interleaved DRAM", *HPSR* 2006.
[16] J. Garcia, J. Corbal, L. Cerda`, and M. Valero, "Design and Implementation of High-Performance Memory Systems for Future Packet Buffers," *Proc. MICRO* '03, Dec. 2003.
[17] J. Garcia-Vidal, M. March, L. Cerda, J. Corbal and M. Valero, "A DRAM/SRAM Memory Scheme for Fast Packet Buffers," *IEEE Trans. On Computers*, Vol. 55, No. 5, May 2006, pp. 588-602.
[18] M. Kabra, S. Saha and B. Lin, "Fast Buffer Memory with Deterministic Packet Departures," *Proc. 14th IEEE Symp. High Performance Interconnects (HOTI 06)*, IEEE CS Press, 2006, pp. 67-72.
[19] Feng Wang,Hamdi, M.,Muppala, J.K,"Using Parallel DRAM to Scale Router Buffers," *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, VOL. 20, NO. 5, MAY 2009.
[20] Dong LIN and Mounir Hamdi, "Two-stage Fair Queuing Using Budget Round-Robin", accepted by *ICC* 2010.
[21] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, Vol. 38, No. 2, 2008, pp. 69–74.
[22] Jon Kleinberg and Eva Tardos, *Algorithm Design*, Prentice Hall, pp. 758-760,2006.
[23] Fang W, and Peterson L, "Inter-as traffic patterns and their implications," *Proc. IEEE GLOBECOM* 1999.