Efficient Use of Memory Bandwidth to Improve Network Processor Throughput

Jahangir Hasan Satish Chandra[1] T. N. Vijaykumar

School of Electrical and Computer Engineering
Purdue University
{hasanj,vijay}@ecn.purdue.edu

[1] India Research Lab
IBM Corporation
satishchandra@in.ibm.com

Abstract

We consider the efficiency of packet buffers used in packet switches built using network processors (NPs). Packet buffers are typically implemented using DRAM, which provides plentiful buffering at a reasonable cost. The problem we address is that a typical NP workload may be unable to utilize the peak DRAM bandwidth. Since the bandwidth of the packet buffer is often the bottleneck in the performance of a shared-memory packet switch, inefficient use of available DRAM bandwidth further reduces the packet throughput. Specialized hardware-based schemes that alleviate the DRAM bandwith problem in high-end routers may be less applicable to NP-based systems, in which cost is an important consideration.

In this paper, we propose cost-effective ways to enhance average-case DRAM bandwidth. In modern DRAMs, successive accesses falling within the same DRAM row are significantly faster than those falling across rows. If accesses to DRAM can be generated differently or reordered to take advantage of fast same-row accesses, peak DRAM bandwidth can be approached. The challenge is in exploiting this "row locality" despite the unpredictable nature of memory accesses in NPs. We propose a set of simple techniques to meet this challenge. These include locality-sensitive buffer allocation on packet input, reordering DRAM accesses to increase locality, and prefetching to reduce row miss penalty. We evaluate our techniques on cycle-accurate simulations of Intel's IXP 1200 network processor and find that they boost packet throughput on average by 42.7%, utilizing nearly the peak DRAM bandwidth, for a set of common NP applications processing a real trace.

1 Introduction

Network processors (NPs), such as Intel's IXP [9], IBM's PowerNP [7], and Motorola's C-Port [2], are programmable microprocessors optimized for packet switches. Because NPs implement all packet processing in software, they have cost and flexibility advantages over ASIC-based solutions. An NP-based platform can be used generically to provide a variety of packet processing functions, such as IP forwarding, filtering, network address translation, metering and policing, support for virtual private networks, protocol translation, and others.

Packet switching platforms, including those built from NPs, require significant amount of packet buffer space to prevent congestive losses: typically IP routers have buffer size of *round-trip-time* * *line-rate*, which amounts to several megabytes of storage requirement (e.g., 40 ms * 2 Gbps = 80 M bytes). DRAM technology provides such plentiful buffering at reasonable cost. NPs hide DRAM latency by using multithreading and multiple engines to process several packets in parallel. Because handling of one packet is largely independent of another, this multithreading successfully hides the packet buffer latency. However, because each packet must be written to and read out of the packet buffer, the buffer must also support high bandwidth—at least twice the line rate to sustain peak operation. Therefore, DRAM bandwidth is a key consideration in the design of packet switching platforms [1].

In this paper, we consider packet buffers for NP-based packet switches that target cost-effective performance rather than the highest-end performance. A typical system may use DRAM that can deliver 64 bits every cycle at 100 MHz for a peak bandwidth of 6.4 Gbps, implying a peak packet throughput of 3.2 Gbps. While the DRAM can be scaled to deliver higher peak bandwidth, brute-force scaling incurs not only substantially higher cost but also lower utilization. The problem is that a typical NP workload may be unable to utilize the peak DRAM bandwidth, resulting in poorer packet throughput in the average case. The gap between peak performance and average-case performance may be 25-50%. Our goal is to improve average packet throughput via efficient use of DRAM bandwidth in a cost-effective manner.

Our key observation is that average packet throughput can be improved by enhancing *row locality* in the packet buffer accesses. Modern DRAMs exploit their wide, internal organization and latch an entire *row*, e.g. 4K-bytes, of data even if an access requires only a few, e.g., 8 bytes. DRAM can supply data at substantially higher rates if subsequent accesses fall within the latched row, than if the accesses go to different rows. For example, a DRAM may deliver the first 8 bytes of a row in 5 cycles, followed by a maximum of 8 bytes from that row every cycle. Assuming a cycle time of 10 ns, the bandwidth reaches the peak of 6.4 Gbps only if all 8-byte accesses go to the same row (i.e., 0% row miss rate); if each access goes to a different row (i.e., 100% row miss rate), the effective bandwidth is only 1.28 Gbps. An NP with such a packet buffer can exploit row locality to a modest extent by employing larger memory accesses. With 64-byte accesses, the system still incurs a 12.5% row miss rate, delivering a DRAM bandwidth of 4.2 Gbps, and hence packet throughput of 2.1 Gbps, 33% less than the theoretical maximum.

It is difficult to exploit row locality to the fullest extent while maintaining cost effectiveness. First, DRAM chips typically have only a few, e.g. 4, internal row latches, requiring most buffer accesses to fall within the handful of rows. Adding more DRAM chips to increase the number of row latches increases system cost. Second, row locality in the sequence of DRAM accesses depends on the mapping of packets to buffers and the order in which packets make buffer accesses. Any scheme to increase row locality must take into account (1) the interleaving of accesses due to the NP's multithreaded nature, and (2) the inherent variability in the amount of processing, and time interval between input and output, of a packet.

While graphics and vector processors exploit row locality via streaming, these processors usually are not multithreaded, and typically handle regular, uniformly-sized data. Because of interleaving of accesses, and variability in the processing time per unit of input data, NPs do not lend themselves to streaming. Routers commonly use an SRAM cache in front of the packet buffer in order to aggregate multiple small accesses to a wide access [4, 11]. (This cache is distinct from a general-purpose processor's cache, whose primary function instead is to exploit reuse of data and *reduce* traffic to DRAM.) The technique in [11] not only aggregates small accesses into a wide access using an SRAM cache, but also uses a wide array of DRAM banks, a wide bus and a sophisticated lookahead mechanism to scale aggregate DRAM bandwidth to match higher line rates. Because its goal is to *guarantee* a certain throughput for high-end routers, [11] uses the extra hardware despite the costs, and without concern for DRAM utilization. While networking DRAMs [22] could provide high bandwidth, the cost of these special-purpose DRAMs may be an issue for NP systems. Moreover, networking DRAMs also exploit row locality, and can benefit from our techniques.

We present new, cost-effective techniques to improve the utilization of existing DRAM bandwidth, without depending on an SRAM cache, without explicitly widening memory, and without deploying special-purpose memories. Our techniques can be considered *opportunistic* rather than deterministic, as they improve row locality where opportunity arises, but without giving any worst-case guarantees. Our techniques include the following: (1) *Allocation:* While the common practice of allocating buffer space from a pool of 64-byte units minimizes fragmentation, the buffer pool loses row locality over time. We propose locality-sensitive allocation that attempts to allocate buffer space for contemporaneously arriving packets in the same row. (2) *Access Reordering:* To enhance row locality, we reorder DRAM references such that accesses that are likely to go to the same row are made consecutively in small groups (e.g., 4 accesses), without being intervened by other accesses. We also generate output-side accesses in an order that is more conducive to exploiting row locality. (3) *Prefetching:* For the row misses that occur despite our allocation and reordering, we perform prefetching to overlap a row miss access with the preceding access, if the two accesses go to different internal DRAM banks.

An experimental evaluation of our techniques on a set of common NP applications processing a real traffic trace shows that they can increase row locality substantially, enabling nearly peak utilization of existing DRAM bandwidth. Compared to a reference design based on the IXP 1200, which assumes row misses are inevitable and optimizes for reducing the *cost* of row misses, our techniques provide on average about 42.7% higher packet throughput by reducing the *number* of row misses. This comparison is representative in that many commercial NPs including the IBM PowerNP [8] and the Motorola C-Port [3] similarly optimize for row misses. The only hardware addition we assume is enlargement of an existing buffer by 3K-bytes. Although our experiments are based on the IXP 1200 for the sake of concreteness, our techniques and results are not specific to this NP (see Section 5.4).

We also compare our techniques to the cache-based technique from [11], adapted for row locality. Because we are aiming for high utilization at a low cost rather than guaranteed performance, our adaptation retains the SRAM cache from [11] as the only additional hardware. Our results show that opportunistic techniques perform comparably to the adaptation, without incurring the expense of an SRAM cache.

The paper makes the following contributions:

- We identify the root causes of the loss of row locality in a typical NP workload.
- We perform a thorough quantitative analysis to show that our techniques achieve significant gains by increasing row hits in comparison to the alternative strategy of optimizing for row misses.
- Our set of opportunistic techniques is the first to address the bandwidth problem in packet buffers without the use of an SRAM cache and without widening memory.

The rest of the paper is organized as follows. Section 2 provides the necessary background material. Section 3 discusses in detail the challenges in exploiting row locality and Section 4 describes the techniques we propose. Section 5 describes our experimental infrastructure and methodology. It includes a description of the IXP 1200 and the software we used. Section 6 presents the experimental results, and Section 7 discusses related work. In Section 8 we draw some conclusions.

2 Background on network processors

NPs are microprocessors designed specifically to build packet switches [1, 15]. NPs present a close coupling of link-layer interfaces with the processing engine, minimizing overhead; unlike general microprocessors, no device driver is needed. NPs use multiple execution engines–each of which is a multithreaded processor core to hide DRAM latency–to increase their overall computing power. NPs may also contain hardware support for hashing, CRC calculation, etc., not found in typical microprocessors.

Figure 1 shows a schematic of an NP. For our purposes, an NP consists of a set of multithreaded processing engines connected to link-layer interfaces and to the packet buffer. Additional storage is also present in the form of SRAM and DRAM to store program data. In general, processing engines are intended to carry out data-plane functions. Control-plane functions could be implemented in a co-processor, or a host processor. Multiple NPs may be combined to form a distributed packet switch. At least three major NP offerings fall in this broad architecture: IBM's Power NP [7], Intel's IXP [9] and Motorola's C-Port [2].

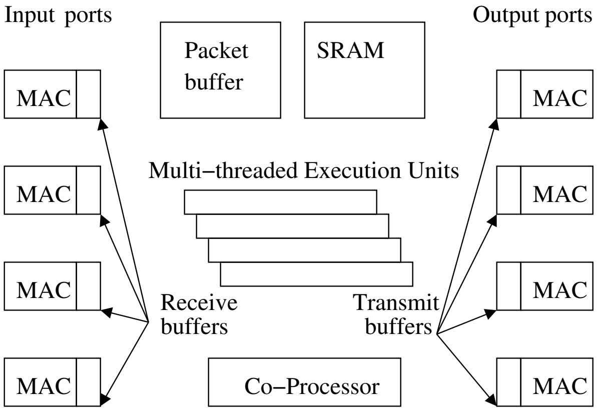

Figure 1: Schematic diagram of a typical NP.

An NPs operation can be explained in terms of a representative application: IP forwarding. (1) A thread on one of the processing engines finds that a new packet has arrived in the receive buffer of one of the input ports. (2) It reads the packet's header into its registers. (3) Based on the header fields, it looks up a forwarding table to determine which output queue the packet needs to go. Forwarding tables are organized carefully for fast lookups [24], and are typically stored in the high-speed SRAM. (4) The thread moves the rest of the packet from the input interface to packet buffer. It also writes a modified packet header in the buffer. (5) A descriptor to the packet is placed in the target output queue, which is another data structure stored in SRAM. (6) One or more threads monitor the output ports and examine the output queues. When a packet is scheduled to be sent out, a thread transfers it from the packet buffer to the port's transmit buffer.

3 Problems in exploiting row locality

First, there are only a small number of row latches available, as there typically are only 2 to 8 internal banks in a DRAM chip. As a result, row locality is *extremely fragile* to maintain, because the probability of a random access falling on a row that is not latched is high. Second, row locality in a stream of DRAM accesses depends both on the assignment of buffer addresses to packets in transit through the NP, and on the order in which these packets make buffer accesses. While one can exercise control over the former, the latter is much more difficult to manage.

Consider the pattern of accesses to packet buffers generated by the IP forwarding application. Each incoming packet p requires its own buffer space and is assigned one upon arrival. During input processing, p initiates a write request W_p to the address of its buffer. During output processing, p initiates a read request R_p to the same buffer address when it is ready to be copied to the transmit buffer.

In the interval of time between W_p and R_p, some of the packets arriving after p, denoted p_{+1}, p_{+2}, ..., will be written to newly-allocated buffers, and some of the packets already residing in buffers when p arrived, denoted q, q_{+1}, ..., will be read for output. At the time p is being input, some n-th earlier packet p_{-n} (denoted q for clarity) is being output; n depends on the number of packets in transit in the NP and is determined by the distribution of packet service times. Together, the input and output sides initiate memory accesses as an unpredictable *interleaving* of the two streams

$$\ldots W_q \ldots W_{p_{-2}}, W_{p_{-1}}, W_p, W_{p_{+1}}, W_{p_{+2}} \ldots$$

and

$$\ldots R_{q_{-2}}, R_{q_{-1}}, R_q, R_{q_{+1}}, R_{q_{+2}} \ldots R_p \ldots$$

where W_p and R_q occur consecutively in the interleaving. The upshot here is that it is difficult to assign buffers to packets in such a way that the interleaved stream of memory accesses generated by the input and output sides will predictably have row locality.

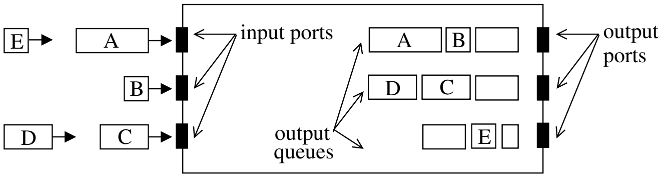

Figure 2: Re-ordering of packets from input to output. The size of boxes show relative packet sizes.

Even if we wish away the interleaving problem, there is another major difficulty in achieving row locality. We implicitly assumed in the previous paragraph that the order in which packets depart the NP is the same as the order in which they arrived. However, in reality there could be a substantial *shuffling*. See Figure 2. Suppose packets marked A through E arrive as shown in the figure and allocate buffers consecutively in the order A, B, C, D, E. We sequence incoming packets by the order in which they allocate their buffers—which is the same order in which they initiate their write requests. Let A be p, B be p_{+1}, etc. Then, these packets generate the following stream of write requests:

$$W_p, W_{p_{+1}}, W_{p_{+2}}, W_{p_{+3}}, W_{p_{+4}}$$

Due to the variability in input-processing times—which depends mostly on packet size—the order of events that each of these packets get placed on an output queue is different from the arrival order. Thus, the *relative* order of the events of enqueuing of these packets could be B, C, A, D, E; we emphasize *relative* because it is possible other packets before A or after E in sequence might also get enqueued in between.

Because some of these packets may depart from the same output port, and because different output queues may have different occupancy at any instant, these five packets may reach the heads of output queues in yet another relative order: E, B, C, A, D. The stream of read requests they initiate for departure is the following:

$$R_{p_{+4}} \ldots R_{p_{+1}} \ldots R_{p_{+2}} \ldots R_p \ldots R_{p_{+3}}$$

which is substantially different from their write request stream; ellipses denote possible intervening reads from other head-of-queues in the general case.

If the output ports implement some QoS policy other than FCFS, the packets' QoS characteristics could cause even more shuffling. The only constraint routers must follow is that packets within each flow, e.g. C and D, must depart in the order in which they arrived. The upshot here is that even if we allocate buffers such that the input-side accesses have row locality, there is little reason to expect row locality on the output side, due to significant shuffling in the departure order of packets.

4 An opportunistic approach

Given these problems, and given the limited degree of control we have—mapping packets to buffer addresses upon their arrival—we propose the following *opportunistic approach* to exploiting row locality.

1. We allocate buffer space on arrival of packets in a way that achieves good row locality at least on the input-side processing of packets.
2. To counter interference, we reorder some DRAM accesses in a small window to take advantage of row locality when the opportunity arises.
3. To counter the effect of shuffling in the NP, we generate DRAM accesses on the output side in a slightly modified order when the opportunity arises.
4. For situations in which we cannot improve row locality, we also propose a prefetching scheme at the DRAM controller to reduce row miss penalty.

In the following discussion, we assume that all DRAM references are accesses to the packet buffer. Otherwise, however carefully we manage row locality for accesses to the packet buffer, there may be other data structures in DRAM and accesses to those may interfere with packet buffer accesses. This assumption is reasonable because most NPs today are equipped with either separate DRAM banks or SRAM for such auxilliary data structures.

4.1 Contiguous allocation on input

Row locality from the perspective of input side can be maximized if packets p, p_{+1}, p_{+2}, ... are allocated buffers close together in space and therefore are likely to write to the same row. This means that we must allocate only as much space to each packet as necessary, and allocate packets contiguously in address space.

Many packet buffer management schemes use a *fine-grain* approach and rely on a pool of available 64-byte "cells" to avoid fragmentation in variable-size-packet traffic. An incoming packet procures just enough cells to store the entire packet; these cells are freed upon transmit. With this scheme, after a few allocations and de-allocations have taken place, cells in the pool are likely to be randomized in terms of their addresses, even if the pool was initially populated with locality in mind. There is no guarantee of row locality in cells allocated to packets arriving together.

In our *linear* allocation approach, we view buffer space simply as one large array. We maintain a global allocation *frontier* and allocate only the required amount of memory, in granularity of 64-byte cells, simply by advancing the frontier. By design, fragmentation is not a problem. When the allocation frontier approaches the end of the buffer space, it wraps around to the beginning. To deallocate a packet, we partition the buffer space into 4K-byte pages, and maintain counters that track the number of free 64-byte cells in each page. When a packet departs, we increment the counter corresponding to the packet's page. Returning discontiguous pages to the free pool may incur significant processing overhead. To avoid this overhead we do not reclaim an empty page as soon as it becomes empty. Rather, we allow the allocation frontier to wrap around and if the allocation frontier finds the contiguously-first page to be empty, the page is reclaimed and reused. If the page is not empty, allocation does not skip over the page but instead waits for the page to become empty.

It is possible that the allocation frontier stalls even though some non-contiguous pages are empty. Packets belonging to a slow-draining port can stall the allocation frontier, causing severe underutilization of the packet buffer memory. To address this problem, we propose *piece-wise* linear allocation. This scheme is a middle-ground between the fine-grain scheme, which has no underutilization problems but little locality, and linear allocation, which has high locality but severe underutilization problems. In piece-wise linear allocation, we maintain a pool of moderate-size (e.g., 2K-byte) pages, avoiding the fine-grain scheme's locality problems. At the same time, we avoid linear allocation's underutilization problems by returning a page to the free pool as soon as it becomes empty. We retain the global allocation frontier, except that it points to the start of free space in the most-recently-allocated (MRA) page. For an incoming packet, we allocate only the required amount of memory, similar to linear allocation. When a new packet cannot be fit in the space remaining in the MRA page, a new page is allocated and the allocation frontier is set to the first byte in the new page. For deallocation, we maintain counters for each page, similar to linear allocation.

Piece-wise linear allocation still has internal (i.e., within-page) fragmentation problems, albeit less severe than the underutilization problems of linear allocation. Fragmentation and row-locality impose fundamentally conflicting requirements on the allocation scheme and it is hard to simultaneously fulfill both requirements.

One limitation of our techniques is that the software must be able to extract a packet's size from the packet header before allocation, a requirement not shared by fine-grain allocation.

4.2 Batching to prevent interference

If the input/output interference is such that W_p and $R_{p_{-n}}$ happen consecutively, it is likely that they access different rows, because the allocation frontier from p_{-n} to p can advance over several rows in general. We propose to serve the stream of memory requests arriving at the DRAM's controller in a different order: service the stream of read requests and the stream of write requests in small *batches*, rather than in the order in which they arrive.

It is convenient to assume two physical request queues at the DRAM controller, one for read requests and one for write requests. In fact, high-performance DRAM controllers often have multiple request queues, e.g., a separate one for each bank. The decision to switch from servicing reads to servicing writes, or vice versa, depends on one of three events, whichever occurs first: (1) The next element on the current queue would definitely cause a row miss; (2) Some k requests from that queue have been processed; (3) The current queue becomes empty before k items have been handled. Batching depends upon some amount of internal queuing at various places inside the NP. We have found in our experiments that a small value of k, e.g. 4, works well. If a large value of k is chosen, it may either cause congestion on input links or starve output links, degrading overall performance; however with small batch sizes such issues do not arise. Batching does not interfere with QoS, as it does not alter the sequence of output events as dictated by the output scheduler.

4.3 Blocked outputs for output locality

The order of read requests generated from the output side depends on the output scheduler, which examines the output ports, and then schedules packets from output queues to be copied from the packet buffer over to a transmit buffer. In order to serve output ports evenly, the output scheduler must not permit a long packet to monopolize the read request queue. Therefore, it schedules only a small, fixed-size "cell" from a packet to be read at a time, which causes packets from multiple queues to be read out from the packet buffer in an interleaved fashion. By choosing a small cell size, only a small amount of transmit buffer is needed in front of the output ports.

The trouble with output scheduling as just described is that there is no locality to be found *across* packets at the heads of output queues, owing to shuffling (Section 3). Our experiments (Section 6.5) show that packets at heads of output queues can access so many rows that no permutation of a "frontier" of read requests will have better row locality. The only locality to be found on the output side is *within* the cells that comprise a large packet.

In order to exploit this intra-packet locality, we propose to modify the output scheduler so that it transfers up to a block of t cells belonging to the current packet from the packet buffer to transmit buffer. We also assume an increase in the space available at the transmit buffer by a factor of t, so that for each port, t cells can be read out from DRAM at a time. With this change, we reduce the spread in the row addresses generated by the output side in any given window of time, which enhances locality. Setting t to a small value, say 4, works well in practice ($t = 1$ in the original scheme), and is also a reasonable increase in capacity (1K to 4K bytes). A large value of t may have a detrimental effect on performance, as it could lead to one of the output queues monopolizing DRAM bandwidth at the expense of others; for small t, this issue is not a concern. Blocked output does not interfere with QoS: in essence, it creates a larger cell size and any QoS policy should be oblivious to the cell size.

4.4 Precharging and prefetching

When a memory access is a row hit, only a column access strobe (CAS) needs to be performed. However, upon a row miss, a new row must be fetched into the row latch of the accessed DRAM bank. First, the bank is *precharged*, then a row access strobe (RAS) for the required row is performed. DRAM controllers takes advantage of internal banking by overlapping precharges of one bank with data transfers of another. We present a DRAM controller policy that optimizes performance assuming most accesses are row hits.

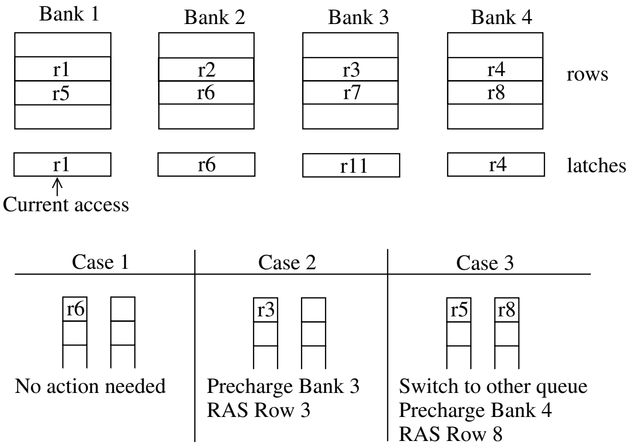

Figure 3: Precharging and Prefetching. The cases shown correspond to the three cases in the text, with left queue current in each.

First, we precharge a bank lazily, when we know an impending access is going to access another row in the bank and a RAS is inevitable. By contrast, a DRAM controller that optimizes for row misses might prefer eagerly precharging an idle bank, in anticipation that a future reference to a row in that bank can have a smaller row miss servicing time. That policy can be counter-productive if row hits are expected—as we intend to be the case—because eager precharge will have discarded the contents of the row latch even if a subsequent access were to the latched row.

Second, we also issue the RAS to hide the RAS latency, essentially *prefetching* the row into the row latch. To determine impending accesses, we exploit the fact that the DRAM controller can examine the heads of read and write request queues. As in batching, we maintain a single read and a single write queue at equal priorities, but our strategy works with or without batching.

When our DRAM controller processes the head item of one of the queues, it dequeues the item and processes its request. During the processing of that request, it also examines the new head of the same queue and performs one of the following actions (See Figure 3):

1. If the address is to another bank, and the addressed row is already latched, it does nothing further.
2. If the address is to the other bank, and the addressed row is not latched, issue a precharge to that bank followed by a RAS for that row.
3. If the address is a different row of the current bank, or if the current request was the last in a batch of k requests, then it peeks at the address to be requested by the head item of the other queue. It performs steps 1 and 2 above for that address.

In the third case, even after switching to another queue, we might incur a row miss. Since we assign rows to b banks

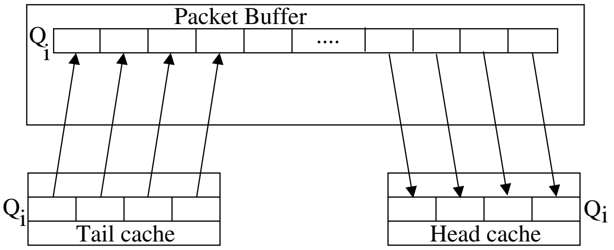

Figure 4: Caching the head and tail of each output queue. The arrows denote a wide transfer of data. Only the i'th queue is shown.

round robin, there is a $1/b$ chance the next address would map to the current bank. Clearly, having more banks decreases the chance of a bank conflict preventing a prefetch.

For our DRAM configuration, the precharge and prefetching actions outlined above can be completed by the controller in the "delay slot" of an 8-CAS access, but may be only partially completed under narrower accesses. When they are not completed, the latency of the RAS is exposed to the subsequent access.

4.5 Wide accesses via caching

Although a traditional SRAM cache in front of the packet buffer is not helpful because of the streaming nature of NP workloads, it is still possible to use a cache-like SRAM structure for widening accesses to the packet buffer. The scheme presented in this section is derived from [11]. We have adapted it as follows: (1) We exploit row locality in internal banks rather than the original scheme's parallel transfers from multiple external banks, which incurs additional costs in memory and bus. (2) Unlike the original scheme, which intends to satisfy deterministic bandwidth guarantees, we do not use a lookahead buffer and we do not assume a hardware implemented algorithm to make use of a lookahead buffer. These changes are made to produce a relatively less expensive memory system, one that only improves the average case rather than guarantee worst-case performance.

The idea behind this adaptation is to cache the prefix (or tail end) and suffix (or head end) of each output queue separately in SRAM. When a newly arrived packet destined to some output queue makes its write request to packet buffer, the request is satisfied by storing packet data into that queue's prefix cache. When the space in the prefix cache dedicated to a given queue is full, the set of consecutive cells from the cache are written to the DRAM in a wide accesses. For this approach to work, the packet buffer space for packets in each output queue is allocated linearly. The output side works in the same way. The read requests coming from the output scheduler are serviced by the suffix cache, which is refilled periodically from DRAM in a wide access. See Figure 4. If the size of the prefix (or suffix) that is cached per queue is 4 cells of 64-byte each, then this technique essentially widens DRAM accesses to 256 bytes, bringing down the miss rate by a factor of 4.

The cost of the adaptation is the cost of the additional SRAM cache. If the cache accomodates m cells per queue, and there are q output queues, then storage of $2 \times m \times q$ cells is required [11]. For 64-byte cells, $m = 4$ and $q = 16$, this is 8K-bytes. If a system is designed for running multiple output queues per port for QoS, the value of q could be much larger; a realistic 8 queues per port implies $q = 128$ and 64K-bytes of SRAM cache. Adding a large SRAM cache is significantly more expensive, either off-chip or on-chip, than merely increasing the existing on-chip transmit buffer by 3K-bytes. Because we rely only on intra-packet locality, the transmit buffer increase is agnostic to the number of output queues per port.

For DRAMs with internal banking, the prefetching optimization of Section 4.4 can be used in combination with the cache-based scheme presented here. Section 6 presents a quantitative comparison of our approach with the cache-based scheme with and without prefetching.

5 Methodology

We use three common NP applications processing a real traffic trace on an Intel IXP 1200. Although we base our evaluation on the IXP 1200, we believe that the IXP is representative of commercial NPs in key architectural aspects, as discussed in Section 5.4.

5.1 Hardware

The IXP 1200 consists of six 4-way multithreaded "microengines" for traffic processing, a StrongARM core for control-plane functions, a controller for off-chip SRAM, a controller for off-chip DRAM, on-chip scratchpad memory, and receive and transmit "fifos", which are receive and transmit buffers that interface both with I/O ports and microengines. Transfer of data between the fifos and DRAM, or between the fifos and registers, is carried out explicitly by memory instructions. For DRAM accesses, a single instruction can transfer up to 64 bytes of data between fifo and DRAM, or 32 bytes between registers and DRAM; the smallest DRAM access is an 8-byte quantity, which is the bus width on this system. Typically, a memory instruction also switches context to another thread to tolerate latency. We defer a description of IXP's DRAM controller until Section 6.2.

5.2 Software

We use three representative NP applications for our experiments: IP forwarding, network address translation (NAT), and firewall. We briefly describe these applications pointing out their similarities and dissimilarities.

We use *L3fwd16*, a sample IP forwarding software that comes with an SDK provided by Intel [10]. This application performs Layer 3 (IP) forwarding for 16 100-Mbps Ethernet ports. Input and output processing in *L3fwd16* closely follows the description in Section 2. In terms of packet buffer accesses, for input of the first 64 bytes of a packet, it makes a 32-byte write for the modified header and a 32-byte write for the remaining cell. The rest of the packet is stored in 64-byte writes (except possibly for the end-of-packet). On the output side, reads from the packet buffer are all 64-byte wide (except possibly for the end-of-packet). *L3fwd16* maintains a single FIFO queue per output port.

NAT translates network addresses for 2 1-Gbps ports. For each packet, it computes an index using the source and destination IP addresses, and the source and destination port numbers. Using this index, it looks up a hash table to retrieve a replacement address and port. Unlike *L3fwd16*, *NAT* needs to look up and modify the TCP header. Therefore, the first 64 bytes are read into microengine registers, modified, and then written to the packet buffer, all in 2 32-byte transfers. The rest of the packet is processed similarly to *L3fwd16*. *NAT* differs from *L3fwd16* in that (1) *NAT* dynamically updates the hash table when TCP SYN or FIN packets are encountered. Because the hash table resides in SRAM, *NAT* generates many SRAM accesses. (2) Due to the NP's multithreading, the hash table updates have to be atomic, requiring lock and unlock operations.

Firewall is also implemented for 2 1-Gbps ports. The application first extracts values for various fields from the packet headers and then walks through a list of templates against which the values are matched. Based on the matches, it decides whether to forward or drop the packet. For instance, a firewall may be configured to drop packets with a directed broadcast, or packets sent from a particular source. A key feature distinct from the previous applications is that the templates are stored in the NP's SRAM as a linked list. *Firewall* traverses this list for each packet, generating more SRAM accesses than the other applications. In general, *Firewall* performs more computation per packet than the other applications considered.

All the applications dedicate the first four microengines (16 threads) to input processing, mapping threads statically to input ports, and the last two (8 threads) handle output processing. In all the applications, buffer allocation is done by popping from a shared stack of fixed-size, 2K-byte buffers; IXP 1200 has hardware support for operations on a shared stack that resides in SRAM.

5.3 Simulation infrastructure

We build on the IXP 1200 simulator provided with Intel's SDK. This execution-driven simulator models the details of the IXP 1200 and provides cycle-accurate execution time statistics. We disable the simulator's DRAM module, and plug in our own DRAM module to respond to the simulator's DRAM requests. Our DRAM module's outputs are fed back to the simulator in a timing-accurate manner. We have validated our DRAM module against the simulator's.

The NP applications executing on stock IXP 1200 are compute bound—even if we improved DRAM bandwidth via row locality, we would not see any improvements in overall throughput. In the following table, we show the utilization of the microengines, for *L3fwd16*, to justify this claim.

Configuration		*Packet Size in bytes*		
(uEng/DRAM Mhz)		64	256	1K
200/100	uEng idle	8.0%	8.1%	8.1%
	DRAM idle	13.4%	12.3%	11.0%
400/100	uEng idle	31.5%	31.0%	32.0%
	DRAM idle	1.1%	1.2%	1.3%

We see that for various fixed packets sizes (based on a synthetic trace), the system using processor clock of 200 MHz and DRAM clock of 100 MHz is completely compute-bound. However, when the processor speed is scaled up to 400MHz and DRAM speed stays at 100 MHz, the system is no longer compute-bound. Because processor speeds improve at a much faster rate than memory speeds, we believe that future NPs will be DRAM-bandwidth bound rather than compute bound.

In all our experiments, we simulate processor at 400MHz and DRAM at 100 MHz. To ensure that any potential increase in throughput we measure is not limited by finite speed ports, but rather by the system's processing speed, we scale port speeds in the simulator so input threads always have a packet available to process, similar to [23].

We use a real, edge-router traffic trace, *IND-1027393425-1.tsh* from [18]. The average packet size for the is trace is 540 bytes. Our experiments forward tens of thousands of packets, running the simulations for 24-hour periods.[1] We also did these experiments with a synthetic trace generated by the Packmime tool [5] and found the results obtained to be similar to those presented here.

5.4 Generality of our results

We argue that neither the problems in exploiting row locality, nor our proposed solutions are specific to the IXP. The basic premise of this paper is that NPs use DRAM for packet buffering. This premise is true for all commercial NPs including the Intel IXP 1200, IBM PowerNP [8], and Motorola C-port [3]. Our claim is that exploiting row locality to improve throughput is hard because NPs are multithreaded and introduce shuffling and interleaving. This claim is also true for all the above NPs. Our solutions involve allocation in software, modifying the DRAM controller for batching and prefetching, and lengthening the transmit fifo for blocked output. All the above NPs do allocation in software, use a DRAM controller to interface to the DRAM, and use the equivalent of a transmit fifo to buffer parts of packets in between DRAM and the output ports.

We use a design based on the IXP 1200 as the reference point for our results. The salient features of this design: distributing free buffer pools across odd and even banks of the DRAM, and alternating between the odd and even banks for DRAM requests to hide row misses, are common across the IXP, the PowerNP and the C-Port. All these NPs assume that row misses are inevitable and attempt to reduce the cost of row misses, unlike our approach to reduce the number of row misses.

6 Experimental Results

In all the experiments, we report packet throughput (and not DRAM bandwidth) in Gbps (gigabits per second). Because most inexpensive DRAMs have 2 to 4 internal banks, we vary the number of banks as 2 and 4, with rows in our tables marked “2 banks”, and “4 banks”. Because the impact of each of our techniques in isolation is similar across our applications, we present a detailed analysis of our techniques using *L3fwd16*. Then, we present overall results for *NAT* and *Firewall*.

6.1 Opportunity

In the first experiment, we show the maximum achievable packet throughput assuming perfect row locality. We configured the memory simulator to return row hit timings for

[1] We ran some of these experiments for several days each, processing hundreds of thousands of packet, and did not find any change in the relevant statistics.

all accesses, irrespective of reality. In Table 1, we compare the IXP 1200 reference design, REF_BASE, against an idealized REF_IDEAL endowed with all row hits. For the ideal case with all row hits, the number of internal DRAM banks is irrelevant, but we vary the number of banks for REF_BASE.

Comparing REF_BASE and REF_IDEAL throughput improves by 46.2% with 2 banks and by 37.8% with 4 banks, clearly showing the large scope for improvement enabled by row locality. The improvements reduce from 2 to 4 banks as may be expected, because 4 banks allow the system to hide some of the row misses via eager precharges (Section 4.4). The peak DRAM throughput allowed by the 100-MHz DRAM with a 64-bit bus is 6.4 Gbps, corresponding to 3.2 Gbps packet throughput. Thus, REF_IDEAL uses 90%, and not 100%, of the DRAM's peak bandwidth. To understand this disparity, we ran the same experiments with the processor scaled to 600 Mhz while keeping DRAM at 100 Mhz, and obtained only 93.7% DRAM utilization. The scaled run establishes that the utilization is less than 100%, due not to compute-boundedness but to imperfect computation-memory overlap inherent to *L3fwd16*.

Table 1: Packet Throughput (Gbps) of REF_BASE vs. ideal memory for *L3fwd16*

banks	REF_BASE	REF_IDEAL
2	1.97	2.88
4	2.09	2.88

Table 2: Packet Throughput (Gbps) of REF_BASE vs. OUR_BASE for *L3fwd16*

banks	REF_BASE	OUR_BASE
2	1.97	1.93
4	2.09	2.05

6.2 Baseline

Because REF_BASE assumes row misses are inevitable and optimizes for row misses, and we want to optimize for row hits, we make three preparatory changes before applying our techniques. OUR_BASE is our starting design which includes the preparatory changes, but not any of our techniques. In the following sections, we evaluate our techniques on top of OUR_BASE, and not REF_BASE. In this section, we show that these preparatory changes do not give us any performance advantage even before applying our techniques. Note that REF_BASE is not unique in optimizing for row misses, the same optimizations are also advocated by the IBM PowerNP [8] and the Motorola C-Port [3].

The three changes are: (1) REF_BASE's DRAM controller partitions internal DRAM banks into *odd* and *even*. Input-processing engines allocate buffer space alternately from odd and even banks. The controller maintains separate queues for requests to odd and even banks, and services the two queues in strict alternation. While one bank is transferring data in CAS cycles, an idle bank is precharged eagerly to hide the precharge latency when it is accessed next. The only exception to this eager precharge is if the DRAM controller notices in time that the next access is in fact to the bank's latched row. This eager precharge policy is chosen to work well in conjunction with the odd/even queues when row misses are expected rather than row hits. Instead, OUR_BASE combines the two pools into one and disables eager precharging to maximize row hits by increasing the chances of buffers falling in the same row. (2) Apart from the odd/even request queues, there is a third, higher-priority queue that is always serviced in preference to the odd/even queues. The output-side requests are handled at a higher priority via the third queue. In contrast, OUR_BASE uses two non-prioritized queues–one for reads and the other for writes–to facilitate batching. (3) OUR_BASE maps rows round-robin across banks (i.e., row x maps to bank i and row $x + 1$ maps to bank $i + 1$). If contemporaneously arriving packets occupy multiple consecutive rows, the rows are latched in their respective banks, allowing row hits to all the rows without any contention. In contrast, REF_BASE assigns rows 1 through N/2 to odd banks, and rows N/2 through N to even banks, to hide precharges by alternating between odd and even banks.

In Table 2, we show the packet throughput achieved by OUR_BASE and REF_BASE. We see that the two systems perform similarly in each case; our preparatory changes make no difference.

6.3 Allocation

We show the impact of both our allocation schemes in this experiment. REF_BASE uses a fixed-size allocation of 2K-byte buffers, irrespective of the incoming packet size. While this fixed-size scheme enables fast allocation and deallocation, and keeps buffer pool management simple, it fragments memory, especially because small packets can contribute 40% or more of real traffic. In view of this problem, many routers use fine-grain allocation of 64-byte cells (see Section 4.1).

In Table 3, we compare REF_BASE using fixed-size allocation against L_ALLOC and P_ALLOC which are OUR_BASE augmented with linear and piece-wise linear allocation, respectively. We also show F_ALLOC which uses fine-grain allocation keeping everything else the same as REF_BASE. L_ALLOC and P_ALLOC should exploit row locality in at least the input side; the output side is likely to be disrupted by shuffling due to variability in packet sizes. There is also likely to be some interference between the input and output sides.

Comparing REF_BASE against L_ALLOC and P_ALLOC, we see that L _ALLOC achieves packet throughput improvements of 0.5% using 2 banks, and 8.1% using 4 banks, while P_ALLOC achieves improvements of 3% using 2 banks, and 7.6% using 4 banks. We do not see much improvements using 2 banks because the input side accesses touch almost 4 rows for L_ALLOC and 5.6 rows for P_ALLOC (see Table 5), which causes thrashing in the 2 row-latches. With 4 banks, this problem is alleviated yielding higher improvements. We see that P_ALLOC is comparable to L_ALLOC in performance, but because piece-wise linear allocation has better memory utilization behavior we use P_ALLOC as the allocation scheme for the following sections.

Comparing REF_BASE and F_ALLOC, we see that there is little difference between the two. The fine-grain alloca-

Table 3: Packet Throughput (Gbps) of piece-wise linear allocation for *L3fwd16*

banks	REF_BASE	F_ALLOC	L_ALLOC	P_ALLOC
2	1.97	1.89	1.98	2.03
4	2.09	2.04	2.26	2.25

Table 4: Packet Throughput (Gbps) of batching for *L3fwd16*

banks	P_ALLOC	P_ALLOC+BATCH
2	2.03	2.08
4	2.25	2.34

tor, over time, draws the chunks scattered throughout memory, as explained in Section 4.1. Hence, F_ALLOC has poor row locality like REF_BASE. Therefore we continue to use REF_BASE, instead of F_ALLOC, as our comparison point in the following sections. Note that P_ALLOC has neither REF_BASE's fragmentation problem nor F_ALLOC's poor row locality problem.

6.4 Batching

In Table 4, we compare P_ALLOC against P_ALLOC+BATCH which augments P_ALLOC with batching. Because batching tries to gather accesses together to increase row hits, batching is meaningful only in the presence of our allocation scheme. Therefore, we do not evaluate batching without our allocation. We set the maximum batch size to be 4 (i.e., the number *k* in Section 4.2 is 4). We expect batching to improve upon our allocation by preventing the interference between the input and output sides.

Comparing P_ALLOC and P_ALLOC+BATCH we see that the 2-bank case does not gain as much from batching because of the input-side not fitting in 2 row latches, as mentioned in the previous section. The 4-bank case does not have this problem, and P_ALLOC+BATCH achieves a 4% improvement over P_ALLOC.

In Figure 5, we show the effect of varying the maximum batch size on the packet throughput (top line graph) and on the observed batch size (bottom bar graph). Because the 2-bank case does not appreciably improve by batching, we do not plot this case in the figure, we plot only the 4-bank case. The maximum batch size is defined as before (e.g., maximum batch size of 2 means upto 2 requests are batched).

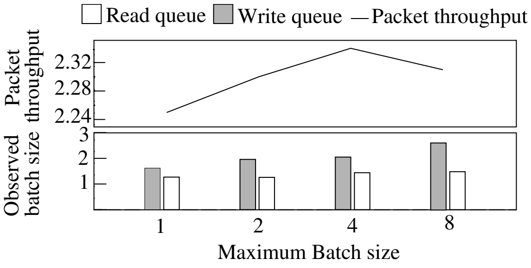

Figure 5: Observed batch size and packet throughput vs. maximum batch size for 4-banks

Table 5: Rows touched in a window of 16 references.

Rows touched	INPUT	OUTPUT
L_ALLOC	3.9	11.4
P_ALLOC	5.6	12.9

However, computing the observed batch size is a little more involved: Because the transfers between the NP and DRAM occur in a mix of different sizes (see Section 5.2) we use the average transfer size as the unit of the observed batch size. For our input trace, this average is 58.4 bytes.

From the top line graph, we see that the packet throughput increases with maximum batch size as expected, peaks at the maximum batch size of 4, and then drops. The bottom bar graph explains this drop: The bar graph shows that while the observed batch size for writes (input side) increases quickly, the observed batch size for reads (output side) increases slowly. The output side locality is poorer and frequent row misses break up batches. At the maximum batch size of 8, the disparity between input- and output-side batches causes the input side to starve the output side of the DRAM bandwidth, resulting in a drop in packet throughput.

6.5 Blocked output

Before we show the effectiveness of blocked output in alleviating shuffling at the output side, we show the extent of the problem. In Table 5, we show the average number of unique rows accessed by the input and output side requests, at any given instant of time. We see that the output side touches at least 11 rows, showing that the variability in the service times of the packets takes its toll on the output-side locality. Because DRAM accesses touch more than 11 rows at the output side, our throughput in the last experiment (see Table 4) remains far less than ideal, despite using batching. (see Table 1).

To address the shuffling problem, we perform blocked output of up to 4 64-byte cells from the DRAM to the transmit buffer, and we assume the extra cells are buffered in a transmit buffer 4-times deeper than that in REF_BASE, as explained in Section 4.3. Using the deeper transmit buffer allows blocked output to be done via 4 overlapped 64-byte transfers, *without* any intervening handshake between the transmit buffer and NP. However, in REF_BASE because these 4 transfers occupy the same slot in the transmit buffer, these transfers need to be serialized via explicit handshakes between the transmit buffer and the NP. Consequently, this overlapping enables ideal throughput (i.e., all row hits) using the deeper transmit buffer to be higher than that of REF_IDEAL, which uses a 1-cell transmit buffer in Section 6.2. Accordingly, we simulate a new idealized case IDEAL++, in which we use the deeper transmit buffer and all DRAM accesses are row hits.

In Table 6, we compare PREV+BLOCK, which adds blocked output to P_ALLOC+BATCH, against P_ALLOC+BATCH and IDEAL++. We see that PREV+BLOCK improves throughput over P_ALLOC+BATCH by as much as 25.9% and 18.8% in the 2- and 4-bank cases, respectively. The disparity between the two cases is due to the input-side spread of 5.6

Table 6: Packet Throughput (Gbps) of blocked output for *L3fwd16*

banks	P_ALLOC+BATCH	PREV+BLOCK	IDEAL++
2	2.08	2.62	3.19
4	2.34	2.78	3.19

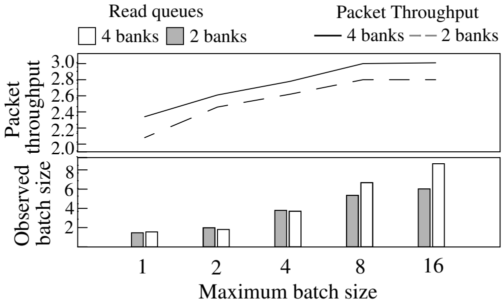

Figure 6: Observed block size and packet throughput vs. maximum block size for 2 and 4 banks

rows, as shown in Table 5. This spread implies that the probability of an output-side access contending for a bank with an input-side latched row is higher in the case of 2 banks than 4 banks. Blocked output reduces the frequency of output side row misses, reducing the contention. Therefore, the 2-bank case shows a larger improvement than the 4-bank case.

In Figure 6, we show the effect of varying the maximum output block size on the packet throughput (top line graph) and on the observed output batch size (bottom bar graph). The maximum output block size (mob-size) is defined as before (e.g., mob-size of 2 means upto 2 cells are blocked). Mob-sizes of 8 and 16 use batch sizes of 8 and 16, respectively, as using mob-size larger than the batch size is meaningless. Similar to the previous section, we use the average output-side transfer size as the unit of the observed output batch size. For our input trace, this average is 60 bytes.

From the top line graphs, we see that the packet throughput increases with mob-size as expected, and levels off at 8. Further increasing the mob size results in diminishing returns because fewer and fewer packets in the traffic can benefit from a larger mob size. The bottom bar graph shows that the 4-bank case has larger observed output batch size than the 2-bank case; the higher bank contention in the 2-bank case breaks up output batches more often than in the 4-bank case. Although mob-size of 8 performs better than 4, we use 4 because of its lower cost. Mob-size of 8 requires an 8KB transmit buffer, twice as large as that required by mob-size of 4. We make up for the performance difference via prefetching.

6.6 Prefetching

In this experiment, we evaluate the impact of prefetching. In Table 7, we compare PREV+BLOCK against ALL+PF (shown highlighted), which is PREV+BLOCK augmented with prefetching.

Table 7: Packet Throughput (Gbps) of prefetching for *L3fwd16*

banks	PREV+BLOCK	ALL+PF	PREV+PF
2	2.62	**2.80**	2.25
4	2.78	**3.08**	2.62

Table 8: Packet Throughput (Gbps) of our adaptation for *L3fwd16*

banks	ADAPT	ADAPT+PF
2	2.76	**2.89**
4	2.84	**3.05**

We see that ALL+PF achieves 6.8% and 10.8% improvements over PREV+BLOCK for 2 and 4 banks, respectively. The improvements are higher for 4 banks than for 2 banks because the probability that the next access will go to different bank than the current access is higher for 4 banks than for 2 banks. By employing prefetching, the throughput approaches close to IDEAL++'s throughput for the 4-bank case in Table 6. Thus, all our techniques put together approach close to ideal throughput, showing that optimizing for row hits works better than optimizing for misses.

We also show the improvements achieved without requiring *any* extra hardware–specifically the deeper transmit buffer of the blocked output optimization. The PREV+PF column in Table 7 corresponds to P_ALLOC+BATCH augmented with prefetching. Compared to P_ALLOC+BATCH (in Table 4), PREV+PF achieves 8.17% and 11.9% improvement using 2 and 4 banks, respectively.

6.7 Adaptation

Finally, we evaluate our adaptation of [11]. In Table 8, we show ADAPT which is the adaptation using 16 queues in each of the two SRAM caches at the input and output sides of the NP. To match ALL+PF's maximum batch size of 4, ADAPT moves data between the SRAM caches and DRAM in units of 4, 64-byte transfers. We also show ADAPT+PF (highlighted) which is ADAPT augmented with our prefetching. From the table, we see that ADAPT performs well, exploiting the wide transfers enabled by the caches. While ADAPT lags behind IDEAL++ in Table 6, ADAPT+PF closes the gap, achieving throughputs similar to those of ALL+PF in Table 7. Thus, we have shown that without using any extra SRAM caches, our opportunistic techniques achieve the same performance as the adaptation.

6.8 NAT and firewall

Having presented a detailed analysis of our techniques using *L3fwd16*, we present overall results for the *NAT* and *Firewall* applications.

In Table 9 we show the throughput achieved by REF_BASE, ALL+PF, and ADAPT+PF for *NAT*. Compared to REF_BASE, ALL+PF applies all our techniques together and achieves overall 39.3% and 41.3% performance gains for 2 and 4 banks, respectively. ADAPT+PF performs comparably to our techniques with 39.8% and 40.8% improvements in throughput over REF_BASE for 2 and 4 banks, respectively.

Table 9: Packet Throughput (Gbps) for *NAT*

banks	REF_BASE	ALL+PF	ADAPT+PF
2	2.11	2.94	2.95
4	2.13	3.01	3.00

Table 10: Packet Throughput (Gbps) for *Firewall*

banks	REF_BASE	ALL+PF	ADAPT+PF
2	2.01	2.77	2.77
4	2.05	2.86	2.89

Table 10 shows the throughput achieved by REF_BASE, ALL+PF, and ADAPT+PF for *Firewall*. We see that the performance gains of ALL+PF are 37.8% and 39.5% for 2 and 4 banks, respectively. ADAPT+PF performs comparably by achieving 37.8% and 40.9% for 2 banks and 4 banks, respectively.

6.9 Summary of results

We see that all our techniques put together achieve on average 42.7% improvement in packet throughput over REF_BASE for the three applications presented. In Table 11, we compare the overall DRAM bandwidth utilized by REF_BASE and by ALL+PF. ALL+PF achieves 96%, 94% and 89% utilization of available peak DRAM bandwidth, for the three applications. In contrast, REF_BASE achieves 65%, 66% and 64% utilization of the available DRAM bandwidth.

7 Related work

7.1 Software

Bux et al. [1] point out that NP memory bandwidth is the key bottleneck in achieving high throughput. In a recent paper [23], the authors examine the challenges in programming NPs. They show that an IXP 1200-based, inexpensive router can forward minimum-sized packets at a rate of 3.47 million packets per second. This rate is nearly an order of magnitude higher than those of existing pure PC-based routers. At less aggregate line speeds, they exploit the excess resources available on the IXP 1200 to be used robustly for extra packet processing.

Using batching to avoid context switching overhead has parallels in locality driven co-scheduling policies that have been proposed for operating systems [16].

Table 11: DRAM bandwidth utilization for the three applications

	L3fwd16	*NAT*	*Firewall*
REF_BASE	65%	66%	64%
ALL+PF	96%	94%	89%

7.2 Hardware

We have already made extensive comparisons to [11], which presents a technique suited to high-end routers. The work in [4] also uses an SRAM buffer to improve DRAM latency in a shared-memory ATM switch.

Several papers have proposed DRAM row locality optimizations in general-purpose computer systems [6, 17], and media processors [21]. [17] and [21] optimize for bandwidth by leveraging the fact that L2-cache blocks are large and contiguous in general-purpose computers, and media applications access regular streams of data; NPs do not have L2 caches, and have inherently irregular access patterns due to interference and shuffling. Other DRAMs, such as the Direct Rambus DRAM (DRDRAM), also provide significantly higher bandwidth for row hits than row misses, implying that our optimizations work for these DRAMs as well. In [25], the authors propose locality-aware hardware scheduling schemes to reduce instruction cache misses in NP packet processing. Katevenis [19] proposes an ASIC-based router which uses DRAM to maintain per-flow queues for QoS purposes. The paper employs out-of-order execution techniques to hide DRAM latencies, and uses optimizations to avoid DRAM bank conflicts.

7.3 Other packet processing architectures

Alternatives to a NP-based solution include a traditional PC-based architecture [14] on the low end and a hardware-only router on the high end. There are higher-end routers based on commodity microprocessors [20]. A high-end hardware router—albeit one with fixed functionality—can have a substantially higher forwarding capacity than an NP (see a description in [13]). In comparison to such routers—which may be more appropriate for the backbone—growing demand for functionality on the edge of the network might favor an NP-based solution.

8 Conclusion

DRAM bandwidth is a key bottleneck to higher throughput in network processors. Existing NP systems do not necessarily make use of the peak DRAM bandwidth, which is available only when a majority of references to DRAM are row hits.

We proposed a number of opportunistic techniques to increase row locality and to also reduce the cost of row misses. These techniques were: (1) Locality-sensitive, space-efficient buffer allocation on packet input; (2) reducing input-output interference at the DRAM controller by batching; (3) blocked transfers on packet output; and (4) reducing row miss penalty by prefetching in multi-bank DRAMs. Unlike previous schemes which use SRAM caches to achieve high DRAM bandwidth, our techniques better utilize the existing DRAM bandwidth without incurring the expense of an SRAM cache, wide memory, or special-purpose memory.

Our techniques, evaluated on a cycle accurate simulator of the IXP 1200, improved throughput on average by 42.7%, utilizing nearly peak DRAM bandwidth, for a set of common NP applications processing a real traffic trace. This improvement is representative in that many commercial NPs including the IBM PowerNP [8] and the Motorola C-Port [3] share key DRAM-related features with the IXP 1200. The higher throughput made available by our optimizations may be used to support more connections on an otherwise saturated system, lowering the amortized cost per connection.

Our results showed that reducing the number of row misses achieves higher throughput than reducing the cost of row misses, as is done by many commercial NPs including the IXP 1200, PowerNP, and C-Port; and that without using any extra SRAM caches, our techniques performed comparably to an adaptation of a previously-proposed, cache-based technique. As NP-based systems scale, efficient use of DRAM bandwidth will pose an even greater challenge. We claim that by showing improvements in a bandwidth-bound scenario, we alleviate a real problem in NPs now and in the future.

Acknowledgements

We would like to thank Dimitrios Stiliadis and the anonymous reviewers for their useful comments and suggestions. An earlier version of this work appeared as [12]. This work was supported in part by NSF grants CCR-9875960 and CCR-9986020.

References

[1] Werner Bux, et al. Technologies and building blocks for fast packet forwarding. *IEEE Communications Magazine*, pages 70–77, January 2001.

[2] C-Port Corporation. *C-5 Digital Communications Processor*. http://www.cportcorp.com/solutions/docs/c5brief.pdf, 1999.

[3] C-Port Corporation. *C-5 Network Processor D0 Architecture Guide*. http://e-www.motorola.com/collateral/C5NPD0-AG.pdf, 2001.

[4] Tzi cker Chiueh and Srinidhi Varadarajan. Design and evaluation of a DRAM-based shared memory ATM switch. In *Proceedings of ACM Sigmetrics '97 Conference*, pages 248–259, 1997.

[5] W. S. Cleveland, D. Lin, and D. X. Sun. IP packet generation: Statistical models for TCP start times based on connection-rate superposition. In *Performance Evaluation Review: Proc. ACM Sigmetrics*, pages 166–177, 2000.

[6] S. I. Hong et al. Access order and effective bandwidth for streams on a direct rambus memory. In *Proceedings of Fifth International Symposium on High Performance Computer Architecture*, pages 80–89, January 1999.

[7] IBM. *The Network Processor: Enabling Technology for High-Performance Networking*. IBM Microelectronics, 1999.

[8] IBM. *IBM PowerNP NP2G Datasheet*. http://www-3.ibm.com/chips/techlib/techlib.nsf/products/PowerNP_NP2G, 2002.

[9] Intel Corporation. *Intel IXP1200 Network Processor Family Hardware Reference Manual*. http://developer.intel.com/ design/network/ixa.htm, 2001.

[10] Intel Corporation. *IXP1200 Software Development Kit*. http://developer.intel.com/design/network/ixa.htm, 2001.

[11] S. Iyer, R.R. Kompella, and N. McKeown. Analysis of a memory architecture for fast packet buffers. In *Proc. IEEE Workshop High Performance Switching and Routing (HPSR)*, 2001.

[12] J. Hasan, S. Chandra and T. N. Vijaykumar. Enhancing row locality to improve network processor throughput. Technical report, 10009638-020318-11TM, Bell Labs, Lucent Technologies, Mar 2002.

[13] S. Keshav and R. Sharma. Issues and trends in router design. *IEEE Communications MAgazine*, pages 144–151, May 1998.

[14] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *Computer Systems*, 18(3):263–297, 2000.

[15] Mark Kohler. NP complete. *Embedded Systems Programming*, page 45, November 2000.

[16] James Larus and Michael Parkes. Using cohort scheduling to enhance server performance. In *Proceedings of the Usenix Technical Conference*, June 2002.

[17] W. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *Proceedings of Seventh International Symposium on High-Performance Computer Architecture*, pages 301–312, January 2001.

[18] National Laboratory for Applied Network Research. *Daily Traces*. http://pma.nlanr.net/PMA/, 2002.

[19] A. Nikologiannis and M. Katevenis. Efficient per-flow queueing in DRAM at OC-192 line rate using out-of-order execution techniques. In *Proceedings of the IEEE International Conference on Communications*, pages 2048–2052, June 2001.

[20] C. Partridge, et al. A fifty gigabit per second IP router. *IEEE/ACM Transactions on Networking*, 6(3):237–248, June 1998.

[21] Scott Rixner et al. Memory access scheduling. In *Proceedings of 27th Annual International Symposium Computer Architecture*, pages 128–138, June 2000.

[22] SAMSUNG Corporation. *SAMSUNG Network DRAM*. http://www.samsungelectronics.com/semiconductors/dram/ technical_data/application_notes/network-dram_app_note_2.pdf, 2002.

[23] Tammo Spalink, Scott Karlin, Larry Peterson, and Yitzchak Gottlieb. Building a robust software-based router using network processors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 216–229. Association for Computing Machinery, October 2001.

[24] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP routing lookup. In *Proceedings of the ACM Communication Architectures, Protocols, and Applications (SIGCOMM'97)*, September 1997.

[25] T. Wolf and M. Franklin. Locality-aware predictive scheduling of network processors. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 152–159, November 2001.