Locality-Aware Predictive Scheduling of Network Processors

Tilman Wolf and Mark A. Franklin
Departments of Computer Science and Electrical Engineering
Washington University in St. Louis, MO, USA
{wolf, jbf}@ccrc.wustl.edu

Abstract

Demands for flexible processing have moved general-purpose processing into the data path of networks. Processor schedulers have a great impact on the performance of these real-time systems. We present measurements that show that the workload of a network processor is highly regular and predictable. Processing time predictions, based on these measurements, can be used in scheduling together with information about locality in the instruction stream to significantly improve throughput performance. We propose two scheduling schemes, Locality-Aware and Locality-Aware Predictive, that try to avoid cold caches when scheduling packets for processors. Simulations of the schedulers using packet processing times obtained from an operational network processor show the tradeoffs between the algorithms and their performance improvements over First-Come-First-Serve scheduling.

1 Introduction

Over the past decade there has been rapid growth in the need for reliable, robust, and high performance communications networks. This has been driven in large part by the demands of the internet and general data communications. To adapt to new protocols, services, standards, and network applications, many modern routers are equipped with general purpose processing capabilities to handle (e.g., route and process) data traffic in software rather than dedicated hardware. This paper addresses how processing tasks can be scheduled efficiently on such network processors making use of locality information and execution time predictions.

In the current router environment, single processor systems generally cannot meet network processing demands. This is due to the high computational requirements of many network applications and the growing gap between link bandwidth and single processor performance. Such application service software or system software, which we call "application," includes routing, QoS, encryption, compression, media transcoding, and other computationally demanding tasks [18], [4]. However, since packet streams only have dependencies among packets of the same flow but none across different flows, processing can be distributed over several parallel processors. From a functional and performance standpoint it is therefore reasonable to consider developing network processors as parallel machines.

Progress in VLSI technology has also advanced integration to the point where it is now possible to design and implement multiple network processors, with cache and DRAM, on a single silicon chip. Benefits of system-on-a-chip designs include reduced memory access latency and higher clock rates. Commercial examples for such network processors are the IBM PowerNP [8], the Intel IPX1200 [9], and the Motorola C-5 [1]. A study of optimal configurations of system-on-a-chip designs [17] has shown that on-chip cache sizes are typically small ($8 - 16kB$) due to die size limitations and can only hold information for the most recently executed program. The information, which is mostly instruction code, can be reused by the processor if subsequent packets require the same program. Data cache information containing packet-dependent data can less easily be reused, since it changes with every packet. Thus, we mainly focus on the reuse of information in the instruction cache of the processors. Changing the program that a network processor executes, causes the caches to become cold, which results in an execution time penalty associated with the initial loading of the cache with new application instructions. This can have a significant negative effect on overall network processor performance. It is possible, however, to use instruction locality information in scheduling tasks in such a way that the assignment of packets, processed by the same program, are placed on the same processor. This reduces the occurrence of cold caches and improves performance.

Section 2 formalizes the scheduling problem that is considered. Section 3 introduces the locality-aware predictive scheduling algorithm and shows the feasibility of processing time prediction. Section 4 shows simulation results and discusses the benefits of the scheduling strategies. Section 5 discusses related work and Section 6 concludes this paper.

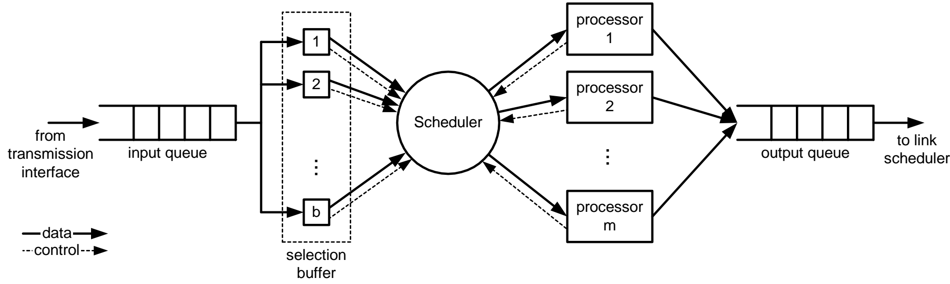


Figure 1: Network Processor Subsystem

2 Scheduling Problem

We consider the parallel network processor shown in Figure 1. The datapath through the system originates at the transmission interface, where packets are received and buffered in the input queue. The scheduler assigns a packet to each of the m processing units whenever a processing unit becomes idle. To do this effectively, the scheduler can pick any of the b packets in the "selection buffer." A packet that is removed from the selection buffer for processing is replaced by the next packet from the input queue. Processed packets are placed into the output queue and sent to the outgoing link or the switch fabric of a router. More details on such a network processor system can be found in [19].

The application software that is executed by a processing unit on a given packet is implicitly defined by the flow to which packet belongs. This requires a flow classification for each packet, which we assume is performed before packets enter into the selection buffer. An example of an efficient flow classification algorithm is described in [13]. To focus on the scheduling issues, we do not consider the issue of per-flow queuing and the associated resource reservation and enforcement issues. We also do not address issues of how code is dynamically distributed to the processors and executed safely. This is addressed by much of the research in the area of active networks [14].

The scheduler bases the decision of which packet to process next, on control information that is received from the selection buffer and the processors (dashed lines in Figure 1). The selection buffer can provide information on the size of each packet and which application is required. The processors feed back information on when they become idle and which application was executed most recently.

The goal of the scheduler is to assign packets to processors in such a way as to maximize the throughput while bounding delay for the individual packets. In the network processor domain, it is particularly important to make use of locality in instructions caches. Since caches are relatively small in multiprocessor system-on-a-chip designs it is advantageous to reuse cached instructions by executing packets that use the same application back-to-back.

2.1 Definition

The definition of the scheduling problem is as follows:

> Given a sequence of packets $p_1 \dots p_n$, associated processing application requirements $\cup_{i=1 \dots n} a(p_i)$, and a set of identical processors and their associated caches $u_1 \dots u_m$: Find a sequential assignment of processing units $u_i \leftrightarrow p_j$ $(i = 1, \dots, m; j = 1, \dots, n)$ to packets that maximizes a given performance metric (defined later).

We would like to develop a scheduler, $S(u_t, B_t)$, where S is a function of the set of packets in the selection buffer, B_t, at time t, and the processing unit, u_t, which has become idle at time t. That is, anytime a processor u becomes idle, we want to schedule a packet from B on that processor.

The assignment of a packet to a processor may be developed as a function of packet size, application properties, time, and state of the processors (see Table 1). Naturally, a schedule S is prohibited from assigning more than one packet to a processor u at any given time.

2.2 Performance Criteria

The performance of a schedule S can be defined in several (sometimes conflicting) ways. The performance depends in large part on the order of packet execution and the resulting processing time for the packet set. The execution time of a packet depends on the state of the cache of the processor where it is processed. A cache is said to be cold if the application required by a newly assigned packet differs from the application just completed. If the cache is warm, the processing time is $t_a(p)$. If the cache is cold, a penalty of $t_{cc}(p)$ is added to the processing time $t_a(p)$. We define the following performance criteria:

- Throughput $T_S = \frac{\sum_{i=1 \dots n} s(p_i)}{t_S(p_n) - t_S(p_1)}$.
  The throughput is defined as the amount of data (i.e., $\sum s(p_i)$) that is processed in a given amount of time. This is the key performance parameter, since generally network processors are aimed at processing as much

Component	Symbol	Description
packet p	P	the set of all packets ($p \in P$)
	n	number of packets ($\|P\| = n$)
	p_i	the ith packet in the data stream
	$s(p)$	size of packet p
	$a(p)$	application a that is used to process packet p
application a	A	the set of all applications ($a \in A$)
	k	the number of all applications ($\|A\| = k$)
	$t_a(p)$	the actual processing time of packet p
	$t_e(p)$	the estimated processing time of packet p with warm caches
	$t_{cc}(p)$	the cold cache penalty for packet p
processing unit u	U	the set of all processing units ($u \in U$)
	m	number of processors ($\|U\| = m$)
	$W_t(u)$	set of apps for which processor u has a warm cache at time t
selection buffer B_t	B_t	the set of all packets in the selection buffer at time t ($B_t \in P^b$)
	b	number of buffer slots ($\|B_t\| = b$)
schedule S	$S(u, B_t)$	the packet from B_t that is assigned to u under schedule S
	$t_S(p)$	time when packet p is scheduled for a processor by schedule S
	$c(S(u, B_t))$	returns 1 if assigned processor has cold cache, 0 otherwise
	$o_S(p)$	returns the order of packet p under schedule S

Table 1: System Parameters.

data as possible. Note that for simplicity, the execution time remaining after scheduling the last packet is ignored, since it has negligible effect on the results when n is large.

- Fraction of cold caches $C_S = \frac{\sum_{i=1,\dots,n} c(S(u,B_{t_i}))}{n}$, where $t_1 \dots t_n$ is the sequence of times when scheduling decisions occur. The fraction of cold caches is the number of times a packet p is assigned to a processor with a cold cache (i.e., $c(S(u, B_t)) = 1$) divided by the total number of scheduled packets. C_S is an indicator of how much locality awareness a scheduling scheme shows. The lower the fraction, the fewer cold cache penalties are incurred.
- Delay variation $D_S = \sqrt{\sum_{i=1 \dots n}(i - o_S(p_i))^2}$, where $o_S(p_i)$ is the order in which schedule S assigns packets to processors. If packet p_5 is the seventh packet to be processed, then $o(p_5) = 7$. Thus, for in-order processing $D_S = 0$. If packets are processed out of order, D_S is the standard deviation of the variation in the order. The larger D_S, the more variation, which means that certain packets are kept longer in the selection buffer. This increases their overall delay. While it is necessary to change the order of packet processing to make use of locality in reducing the negative cold cache performance effects, the goal is to keep D_S low. This will both reduce delay, and help to avoid large-scale re-ordering of the packet stream.

Using these performance measurements, the different scheduling strategies are evaluated in Section 4.

3 Scheduling Strategies

We consider four scheduling strategies with varying grades of complexity: First-Come-First-Serve (FCFS), Throughput-Optimal (T-Opt), Locality-Aware (LA), and Locality-Aware Predictive (LAP). FCFS and T-Opt are simple strategies that perform optimally for one performance criterion (FCFS causes no delay variation, T-Opt achieves maximum throughput). They are used for comparison with the proposed locality-aware scheduling schemes, which are described below.

FCFS. A simple, basic scheduling scheme is first-come-first-serve (FCFS). In this scheme, packets are assigned to processors in the order of their arrival. If a processor u becomes available at time t, the oldest packet in the selection buffer B is sent to u:
$S_{FCFS}(u, B_t) = p_i, \quad where\ i = \arg\min_j \{p_j \in B_t\}.$
The schedule is independent of the size of the selection buffer and does not take any locality into account. It is optimal in terms of variation in delay for packets since it does not re-order packets and keeps the delay for each packet in a given flow the same.

Throughput-Optimal (T-Opt). We define Throughput-Optimal as the algorithm that achieves maximum locality (and thus maximum throughput) by being allowed to pick any packet out of packet stream P (independent of B_t). T-Opt executes all packets of one application before it switches the processor to another. Thus, the only cold caches are due to compulsory cache misses for the first packet of an application.

$S_{T-Opt}(u, B_t) = p_i,$
$\quad where\ p_i = \arg\min_j \{p_j \in P | a(p_j) \in W_t(u)\}.$
This strategy, though not realistic for actual implementation, gives an upper bound on the possible performance.

Locality-aware (LA). Locality aware scheduling uses information about the recent execution history of processor u to decide on the next packet. Given the set of applications for which the cache of processing unit u is warm, $W_t(u)$, a packet p from B_t is chosen that uses one of these applications ($a(p) \in W_t(u)$). In case there are several such packets, the oldest is chosen.
$S_{LA}(u, B_t) = p_i,$
$\quad where\ p_i = \arg\min_j \{p_j \in B_t | a(p_j) \in W_t(u)\}.$
If there is no packet for which the cache of u is warm, the oldest packet overall is chosen.

The effect of such a scheduling strategy is that a processor executes packets from only one application until there are no more packets from that application available in the selection buffer. Thus, packet execution is clustered together to achieve locality. The drawback of such scheduling is that the packets are re-ordered significantly.

Locality-Aware Predictive (LAP). The locality-aware, predictive scheduling algorithm aims at making use of locality, while keeping a bound on delay of the individual packets. At each scheduling decision, LAP computes the fraction of processing that is necessary for each application based on the packets in the selection buffer. To achieve that, LAP uses an estimation of the processing time, $t_e(p)$, for each packet p. Define $f_{B_t}(a)$ as the fraction of processing required by application a:

$$f_{B_t}(a) = \frac{\sum_{\{p \in B_t | a(p) = a\}} t_e(p)}{\sum_{p \in B_t} t_e(p)}.$$

This fraction is compared to the fraction of processors that are currently executing application a (which means that they have a warm cache for application a). Let $w_t(a)$ be that fraction for a:

$$w_t(a) = \frac{|\{u \in U | a \in W_t(u)\}|}{m}.$$

Given $f_{B_t}(a)$ and $w_t(a)$, LAP attempts to ensure that the fraction of processing power associated with applications (i.e., $w_t(a)$) is close to the that required by the packets in the buffer (i.e., $f_{B_t}(a)$). LAP chooses to continue processing the application a for which u has a warm cache if changing the application would drop its processing fraction, $w_t(a)$, below the required fraction of processing, $f_{B_t}(a)$. Thus, if $w_t(a) - \frac{1}{m} < f_{B_t}(a)$, LAP picks the oldest packet with $a(p) \in W_t(u)$ from B_t. Otherwise it picks the oldest packet overall.

$$S_{LAP}(u, B_t) = \begin{cases} \arg\min_j \{p_j \in B_t | a(p_j) \in W_t(u)\}, \\ \quad if \quad w_t(a) - \frac{1}{m} < f_{B_t}(a) \\ \arg\min_j \{p_j \in B_t\}, \\ \quad else \end{cases}$$

LAP differs from LA in that it tries to group processors such that each group processes one application and thus keeps a warm cache for this application. The size of each group is determined by the amount of processing pending for packets in the selection buffer. The effectiveness of LAP is based on the assumption that the processing time for packets is predictable from their size and the application they execute.

3.1 Predictability of Processing

Based on a study of a network processor benchmark [18], there are two key characteristics in NP workloads that differ from traditional workstation workloads. These are:

- Packet size dependent processing time due to the streaming nature of data.
- Small processing kernels (few kB) and thus good instruction cache performance on small caches.

The streaming nature of data causes the applications to repeatedly execute the same code over the data that is passed through the processor. This leads to good predictability in processing times. The small program sizes and the good performance on caches reduce the variation in processing time due to jumps into instruction code that is rarely used and therefore not cached. The good performance on small caches indicates that even after processing only a single packet, the instruction cache can be considered warm.

To show the processing properties of network traffic on a network processor, we have measured the processing times of packets on a programmable router. Three payload processing applications were selected: encryption, compression, and forward error correction. The applications are similar to payload processing applications presented in CommBench [18]. For the measurements, the Washington University Gigabit Switch [2] enhanced with a fully programmable, single processor linecard [6] was used. The software environment for the processing utilized the Active Network Node operating system [5].

Figure 2 shows the processing time for packets of different sizes using the three applications. As can be seen, processing time is linearly dependent on the packet size. The error bars indicate the 95% percentile of processing time. For encryption and FEC, the processing times are very close to the average. For compression, however, which is a data dependent computation, the variations are slightly larger. This linear dependency can be used to develop an expression for the estimated processing time for an application as a function of packet size.

$$t_e(p) = c_p(a(p)) + c_b(a(p)) \cdot s(p),$$

c_p is the per-packet cost of processing for application $a(p)$ and represents the constant processing overhead associated

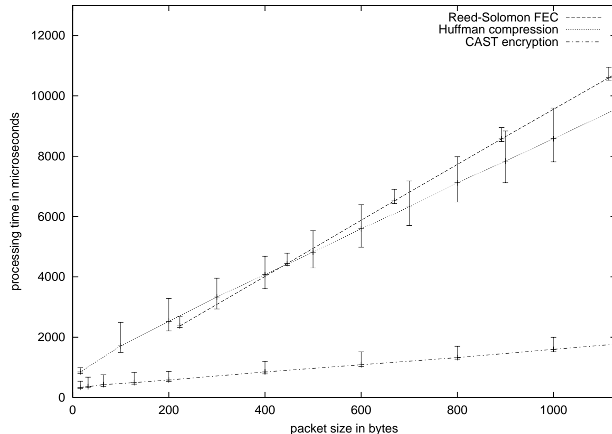


Figure 2: Packet Processing Times for Network Processor Applications.

application a	per-packet cost $c_p(a)$ (μs / pkt.)	per-byte cost $c_b(a)$ (μs / byte)	cold cache penalty t_{cc} (μs / pkt.)
encryption	320	1.3	170
compression	970	7.6	950
FEC coding	320	9.2	175

Table 2: Packet Processing Parameters and Cold Cache Penalties.

with a packet. c_b is the per-byte cost of processing and is multiplied by the size of the packet, $s(p)$. Using the results from Figure 2, we can derive the costs shown in Table 2. Thus, the processing time of a packet can be predicted with good accuracy given the packet size and the application characteristics.

3.2 Cold Cache Penalties

With the same measurement setup, the effect cold caches can also be shown and quantified. When sending a stream of packets, which require the same application, the first packet encounters a cold cache. For subsequent packets, the processing time is reduced due to locality in the instruction code and the resulting warm cache. These measurements are not shown due to space limitations, but Table 2 shows the average cold cache penalty, t_{cc}, for all applications. It can also be shown that this cold cache penalty is independent of the packet size.

These results confirm our assumption that the processing time as well as the cold cache penalty are well predictable.

3.3 Complexity

Finally, the usefulness of these scheduling algorithms depend on how efficiently they can be implemented in hardware. Both LA and LAP have constant processing cost per packet, making them well suited for high performance systems. The following briefly discusses a possible data structure for LAP that can be implemented in hardware and has $O(1)$ update complexity. Since LA is similar and somewhat simpler to LAP, LA can be also be implemented with $O(1)$ update complexity.

There are three components necessary for LAP scheduling: the current values of $f_{B_t}(a)$ and $w_t(a)$, a list of packets pending processing for each application in order of packet age, and a list of all packets in order of packet age. Each of these structures can be updated in constant time when a packet is received or scheduled. The update of $f_{B_t}(a)$ can be done every time a packet is entered into selection buffer by adding its expected processing time. When a packet is removed, the processing time is subtracted. Similarly, $w_t(a)$ can be adjusted by incrementing and decrementing as processors change the applications that they process. An update occurs only when a packet enters or leaves the selection buffer. Thus, the complexity is $O(1)$ per packet. Maintaining lists of packets for different applications that are sorted by the age of the packets can also be done in constant time. Since the age of packets corresponds to the arrival order, a simple queue can be used. Updates to queues can be done in $O(1)$ time per update. Regarding efficient hardware implementations, there has been much work done in implementing efficient queueing systems of this sort [3].

4 Evaluation

The evaluation of the scheduling algorithms was done using a trace-driven simulation. Packet traces that were obtained from a network processor were used as input to a discrete event simulator that emulated the behavior of the scheduler and the processors. After a packet trace was processed, the performance metrics were recorded.

The packet traces were obtained from the Washington University Gigabit Switch [2] that has a Pentium class processor on each input port [6]. Using the Berkeley Packet Filter (BPF), arrival and departure times of packets were recorded and used to compute the actual processing times for all packets of a given application (encryption, compression, or FEC coding). Given the packet size, application, and actual processing time, traces of 100,000 packets were generated having an equal share of bandwidth for each application. To simulate more than three applications, the original traces were replicated with different application identifiers. We assumes that a processor could only have one application in its instruction cache at any time, which is reasonable for the small cache sizes considered.

Measurements were taken over a variety of configurations. The number of processors ranged from 1 to 64, the size of the selection buffer from 1 to 512 packets, and the number of applications in the packet trace from 3 to 300.

4.1 Basic Operation and Adaption to Workload Changes

To illustrate the basic operation of each of the algorithms, we look at the case where we have three applications, 16 processors, and a selection buffer size of 64 packets. The application workload is such that the first 10,000 packets require equal processing. Thus, each application on average should be processed on one third of the processors. After 10,000 packets, the workload changes, such that application 1 requires 80% of the processing and applications 2 and 3 require 10% each (see Figure 3(a) and 3(b)). This is used to illustrate the adaptability of the various algorithms to changes in the workload. Figures 3(c)-3(h) show the different scheduling algorithms. The lines show how many processors have warm caches for each application (i.e., how many processors process each application at that moment) for packets 8,000 through 12,000.

FCFS scheduling shows the expected "random" behavior. Since packets are scheduled in the order of arrival, no locality is exploited and the number of processors executing a given application changes quickly. This behavior leads to a large number of cold caches and low performance. LA scheduling show much less variation in the number of processors assigned to an application. This comes from making use of warm caches until all packets of a given application are processed. The smoothest scheduling behavior is produced by LAP scheduling, which tries to partition the processors according to the processing requirements. Figure 3(g) and 3(h) shows that the partitioning follows very closely to the offered load as shown in Figure 3(a) and 3(b).

All scheduling algorithms adapt quickly to changes in the workload. LA and LAP reach a processor assignment that corresponds to the offered load within a few hundred packets of the change in workload (3 to 4 times the size of the selection buffer). During this period, packets from before the change remain in the selection buffer and influence the scheduling decisions.

4.2 Throughput

Figure 4 shows a throughput comparison for the four scheduling algorithms over a range of selection buffer sizes. The number of processors considered is 16 and the number of applications is 30. FCFS has the lowest throughput of just a bit over $1.5MB/s$. This can be expected, since FCFS does not take locality into account. On the other hand, T-Opt achieves the highest throughput of about $1.8MB/s$. For a very small buffer, LA and LAP are close to FCFS, since the number of packets from which the algorithm can select is small and locality can only be maintained for short times. At a selection buffer size of about 16 to 64 packets, LA and LAP perform significantly better than FCFS. For large selection buffers, both algorithms converge towards the throughput of T-Opt.

4.3 Cold Cache Fraction

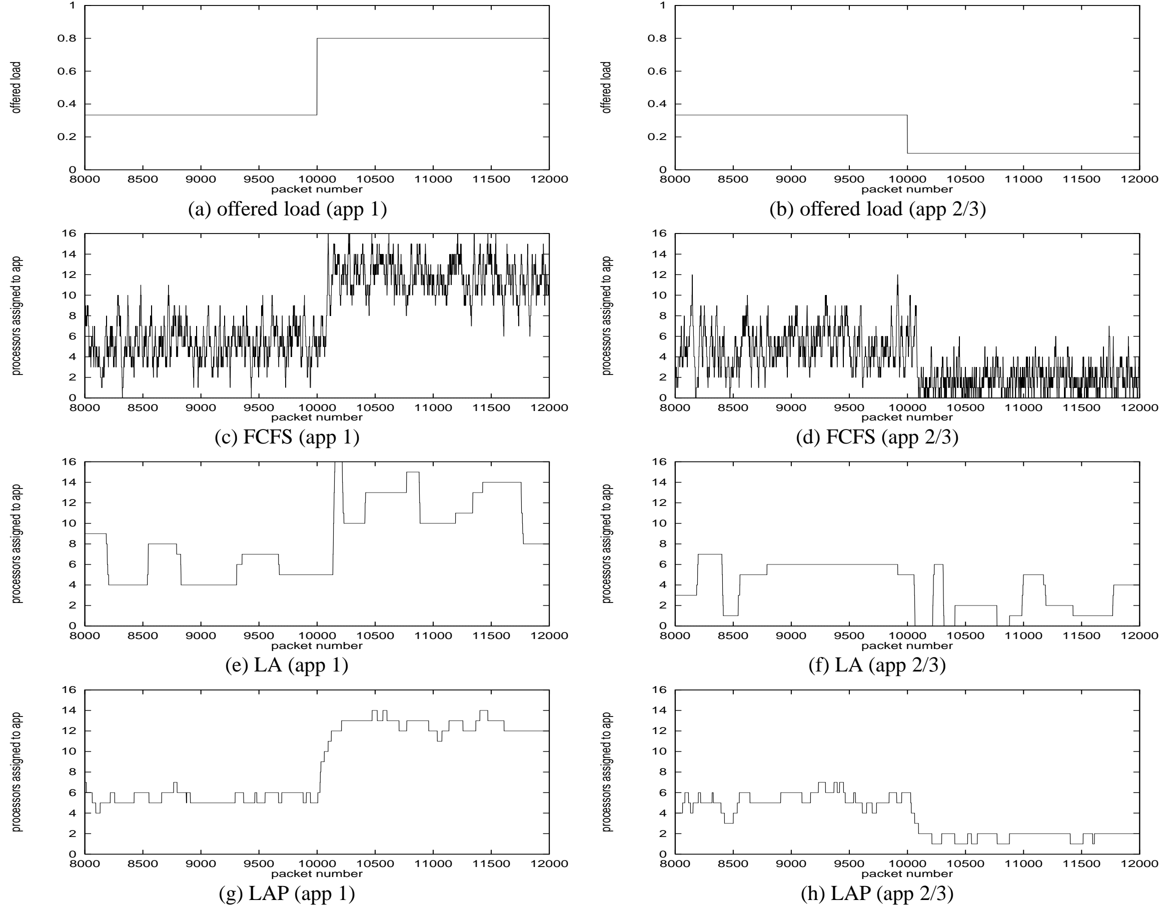
To show the correlation between the use of locality information and throughput, Figure 5 shows the cold cache fraction of packets for the same parameters as used in Figure 4. The cold cache fraction gives the percentage of packets that are executed with a cold cache (i.e., do not make use of locality). FCFS has the highest rate of cold caches with about 96%. This is due to the random assignment of packets to processors in FCFS, which causes only 1 in 30 assignments to be to a processor with warm caches (assuming $a = 30$ applications).

The cold cache fraction for LA and LAP are close to that of FCFS for small selection buffer sizes. As one would expect, with larger buffer sizes, more packets are available and hence scheduling for warm caches is more effective. Thus, the cold cache fraction drops for selection buffer sizes of 16-64 packets before it gets close to T-Opt for very large selection buffers. Note, that the throughput in Figure 4 is directly related to the drop in cold cache fraction around $b = 16$.

4.4 Delay Variation

With respect to the variation in delay, as defined in Section 2, there is a significant difference between LA and LAP. Figure 6 shows the standard deviation of the variation in packet order for FCFS, LA, and LAP. The delay variation for T-Opt is arbitrarily large and thus not plotted here. For FCFS, there is no variation, because packets maintain their order. One can see that LAP shows only very little delay variation for small selection buffer sizes. This is expected since the reordering is roughly limited to the size of the selection buffer. Even for a selection buffer size of 32 packets, the delay variation is only 20 packets for LA and 8 packets for LAP. Differences increase greatly however for large buffer sizes. This is due to the fact that LA tries to execute packets of the same application regardless of their "age" in the selection buffer. LAP on the other hand attempts to group processors with respect to the estimated required processing and with large buffers the quality of that estimate improves. Thus, the variation in delay is kept smaller, because packets from all applications are processed according to the required processing.

Finally, the processor utilization for all scheduling algorithms is $\rho = 1$, because no processor is left idle by the scheduler. Overall, we can see that the throughput performance for LA and LAP is significantly better than FCFS, even for selection buffer size of only 16 packets. With increasing buffer size, LA and LAP approximate the optimal throughput of T-Opt. With respect to delay variation, LAP performs better than LA when the number of processors is large.



(a) offered load (app 1) (b) offered load (app 2/3)

(c) FCFS (app 1) (d) FCFS (app 2/3)

(e) LA (app 1) (f) LA (app 2/3)

(g) LAP (app 1) (h) LAP (app 2/3)

Figure 3: Processor assignment by different scheduling schemes. Since applications 2 and 3 behave similarly, only one set of figures is shown.

5 Related Work

Cache-affinity scheduling, which uses of locality information for the scheduling decision has been used mostly in shared memory multiprocessors [16], [7], [12], [15]. The focus in this domain is to schedule the same process/thread on processors that can reuse previously established cache state. While this is similar to the network processor environment, it does not consider the reuse of instruction cache state for different threads that use the same instruction code (as it is done with packets that use the same application).

An example for scheduling that uses hints about the processing requirement is [10]. In this work, the compiler provides information about thread requirements that are used by the scheduler to determine a thread execution schedule with high cache locality.

Salehi $et.al.$ show the effect of affinity-based scheduling on network processing in [11]. While this also considers the processing of network traffic, the focus is on the operating system level, where packet processing is disrupted by a background workload. This switching between packet processing and the background workload reduces locality in execution and can be avoided by appropriate scheduling.

6 Summary and Conclusions

In this paper, we have discussed processor scheduling issues associated with programmable network multiprocessors. We have shown that locality in instruction data can be exploited with two scheduling algorithms, Locality-Aware and Locality-Aware Predictive. We have evaluated and quantified their throughput improvements over First-Come-First-Serve. The results show that for modest selection buffer sizes (16 packets), the throughput can be improved significantly over FCFS, while keeping the delay variations very limited. For large selection buffer sizes, near optimal throughput can be achieved while the delay variation in LAP stay relatively small. Therefore, the contributions of this work can improve network processor throughput while increasing the complexity of the scheduler only slightly.

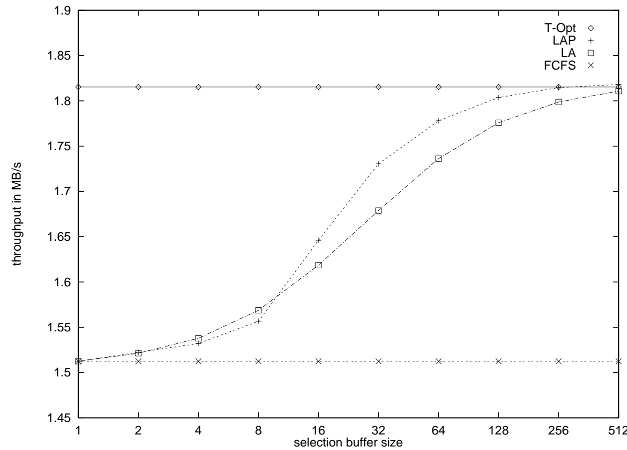


Figure 4: Throughput for different selection buffer size (30 applications, 16 processors).

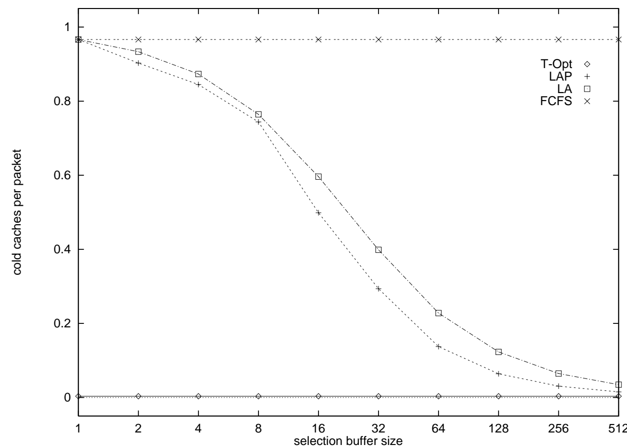


Figure 5: Cold cache fraction for different selection buffer size (30 applications, 16 processors).

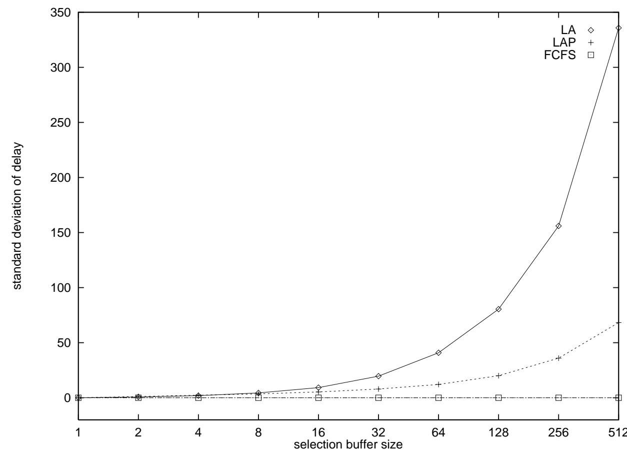


Figure 6: Delay variation for different selection buffer size (3 applications, 64 processors).

References

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

[2] T. Chaney, A. Fingerhut, M. Flucke, and J. Turner. Design of a gigabit ATM switch. In *Proc. of IEEE INFOCOM 97*, Kobe, Japan, Apr. 1997.

[3] Y. Chen and J. S. Turner. Design of a weighted fair queueing cell scheduler for ATM networks. In *Proc. of IEEE GLOBECOM 98*, Sydney, Australia, Nov. 1998.

[4] P. Crowley, M. E. Fiuczynski, J.-L. Baer, and B. N. Bershad. Workloads for programmable network interfaces. In *IEEE Second Annual Workshop on Workload Characterization*, Austin, TX, Oct. 1999.

[5] D. Decasper, G. Parulkar, S. Choi, J. DeHart, T. Wolf, and B. Plattner. A scalable, high performance active network node. *IEEE Network*, 31(1):8–19, Jan. 1999.

[6] J. DeHart, W. Richard, E. Spitznagel, and D. Taylor. The smart port card: An embedded UNIX processor architecure for network management and active networking. unpublished.

[7] M. Devarakonda and A. Mukherjee. Issues in implementation of cache-affinity scheduling. In *Proc. of Winter USENIX Conference*, pages 345–357, Jan. 1992.

[8] IBM Corp. *IBM Power Network Processors*, 2000. http://www.chips.ibm.com/products/wired/communications/network_processors.html.

[9] Intel Corp. *Intel IXP1200 Network Processor*, 2000. http://developer.intel.com/design/network/ixp1200.htm.

[10] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li. Thread scheduling for cache locality. In *Proc. of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, Oct. 1996.

[11] J. D. Salehi, J. F. Kurose, and D. Towsley. The effectiveness of affinity-based scheduling in multiprocessor networking. In *Proc. of IEEE Infocom 96*, San Francisco, CA, Mar. 1996.

[12] M. S. Squillante and E. D. Lazowska. Using processor cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, Feb. 1993.

[13] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast scalable algorithms for level four switching. In *Proc. of ACM SIGCOMM 98*, Vancouver, BC, Sept. 1998.

[14] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, Jan. 1997.

[15] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 24:139–151, Feb. 1995.

[16] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proc. of Thirteenth Symposium on Operating Systems Principles*, pages 26–40, Pacific Grove, CA, Oct. 1991.

[17] T. Wolf and M. Franklin. Design tradeoffs for embedded network processors. unpublished, 2001.

[18] T. Wolf and M. A. Franklin. CommBench - a telecommunications benchmark for network processors. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 154–162, Austin, TX, Apr. 2000.

[19] T. Wolf and J. S. Turner. Design issues for high performance active routers. *IEEE Journal on Selected Areas of Communication*, 19(3):404–409, Mar. 2001.