Exploiting a Computation Reuse Cache to Reduce Energy in Network Processors

Bengu Li[1], Ganesh Venkatesh[2], Brad Calder[2], and Rajiv Gupta[1]

University of Arizona, CS Dept., Tucson, AZ 85737[1]
University of California, San Diego, CSE Dept., La Jolla, CA 92093[2]

Abstract. High end routers are targeted at providing worst case throughput guarantees over latency. Caches on the other hand are meant to help latency not throughput in a traditional processor, and provide no additional throughput for a balanced network processor design. This is why most high end routers do not use caches for their data plane algorithms. In this paper we examine how to use a cache for a balanced high bandwidth network processor. We focus on using a cache not as a latency saving mechanism, but as an energy saving device. We propose using a *Computation Reuse Cache* that caches the answer to a query for data-plane algorithms, where the tags are the inputs to the query and the block the result of the query. This allows the data-plane algorithm to perform a complete query in one cache access if there is a hit. This creates slack by reducing the number of instructions executed. We then exploit this slack by fetch-gating the data-plane algorithm while matching the worst case throughput guarantees of the rest of the network processor. We evaluate the computation reuse cache for network data-plane algorithms IP-lookup, Packet Classification and NAT protocol.

1 Introduction

Network processors are designed to achieve high throughput. With the tremendous increase in line-speed, the amount of throughput required by network processors is increasing significantly. For example, OC-192 (10 Gig/Sec) requires a packet to be completed every 52 nanoseconds and OC-768 (40 Gig/Sec) has a packet completed every 13 nanoseconds.

In these network processors most of the packet processing algorithms are based on some utility data structures. The existence of these utility data structures is ubiquitous. These data structures involved in routing tasks are stored on-chip in large SRAMs for high end routers with latencies in the range of 30 cycles. Thus, much of the time spent during packet processing tasks is taken up by reading (and sometimes writing) of utility data structures in this SRAM. The three examples we focus on in this paper are IP-lookup, Packet Classification, and NAT protocol. In IP-lookup, a routing table is used to do longest prefix matching lookup [10, 24]. In packet classification, a classification table is used to record the flow status [13, 12]. In NAT protocol, maps are maintained so that

source IP addresses, TCP ports, destination IP addresses and TCP ports are used to retrieve a translated source port [11].

In this paper we propose using a *Computation Reuse Cache* that caches the answer to a query for these data-plane algorithms, where the tags are the inputs to the query and the cache block the result of the query. This allows the data-plane algorithm to perform a complete query in one cache access if there is a hit. The computation reuse cache cannot increase the throughput of a balanced network processor. It is instead used to save energy. A hit in the computation reuse cache creates slack by reducing the number of instructions executed for the processing of a packet. This allows the processor to exploit this slack through fetch-gating for the data-plane algorithm while still matching the worst case throughput guarantees of the rest of the network processor. The use of the computation reuse cache is controlled by the data plane algorithm through a programmable interface consisting of specific instructions to lookup and use the cache. Therefore, how it is used (what the tag and data represent) can be different from one data plane algorithm to the next.

In Section 3 we describe why a traditional cache will not help a balanced network processor design. Section 4 examines how much value locality and temporal data locality exists in the fields of packet headers across packet streams in the network traffic for the three data-plane algorithms examined. Section 5 describes the design and use of our computation reuse cache, and Section 6 describes how to use it in combination with fetch gating to save energy. Section 7 provides performance results for our approach and we conclude in Section 8.

2 Related Work

We first briefly summarize related work on temporal locality and caching in network processors, reuse caching, and fetch gating.

2.1 Locality and Caching in Network Processors

Memik et al. [20] examine the use of a traditional cache for a set of networking applications on a StrongARM 110 processor. They found that most of the cache misses came from a small number of instructions. To address this, they use a filter for their data cache to remove the memory accesses with low locality. Li et al. [16] investigate a range of memory architectures that can be used for a wide range of packet classification caches. They study the impact of factors like cache associativity, cache size, replacement policy and the complexity of hash functions on the overall system performance. Both of these studies use a traditional cache, which cannot be used to increase the throughput of data plane algorithms for a balanced network processor. This is why our focus is on saving energy by using a computation reuse cache to create slack in the data-plane algorithm's schedule.

Chiueh et al. [6] use a combined hardware/software approach to construct a cache for high performance IP routing in general-purpose CPU based routers. The destination host address is mapped to a virtual address space and used to lookup a destination route in the Host Address Cache (HAC). A part of the

normal L1 data cache is reserved for use as the HAC. We would classify this approach as being similar to our computation reuse cache in that a hit in the cache skips the full IP lookup. In case of a lookup miss, a 3-level routing table lookup algorithm is consulted for the final routing decision. Their focus is on using the HAC to provide throughput for their network processor design, and not for energy savings. The contribution of our work is to define the general notion of using a programmable computation reuse cache for data plane algorithms and to use it to save energy in a balanced network processor while still providing worst-case throughput guarantees.

2.2 Instruction Reuse

Sodini and Sohi observe that many instructions or groups of instructions have the same input and generate the same output when dynamically executed. They exploit this phenomenon by buffering the computation result of the previous execution of instruction dependency chains. They use the same result for future dynamic instances of the same dependency chains if the input to these chains are the same. In this way, the execution of many groups of instructions can be avoided and the early outcome can allow dependent instructions to proceed sooner. They use a hardware mechanism called the Reuse Buffer (RB) to store the previous computation results. The program counter is used as an index to search the RB for cached chains. Our computation reuse cache is motivated by the reuse buffer, but it differs in that it is programmable and used to cache computations at much larger levels (methods) than just dependency chains.

Ding and Li [9] present a pure compiler memoization technique to exploit value locality. They detect code segments that are executed repeatedly, which generate a small number of different values. The code segment is replaced by a table recording the previous computation results for later lookup if the same values are seen. Performance improvement and energy consumption reduction are achieved. Their work is related to ours because we are also using a computation reuse mechanism to reduce energy consumption. Their approach is a purely software technique. In contrast, we provide a programmable hardware technique specific for data-plane algorithms in order to save energy while providing worst-case throughput guarantees.

2.3 Clock and Fetch Gating

Luo et al. [18] use a clock gating technique to reduce power consumption in multi-core network processors. They observe that when the incoming traffic rate is low, most processing elements on the network processor are nearly idle and yet still consume dynamic power. When the number of idle threads increase, they start to gate off the clock network of a processing unit. When the pressure from the incoming buffer rises, they stop clock gating. Since the activation takes time, they need extra buffer space to avoid packet loss. They also developed strategies to terminate and reschedule threads during activation and deactivation. This work is related to ours because we both aim at gating some part of the network

processor to reduce energy. We both use the queue occupancy information in gating decisions. Their approach is complementary to ours and can be used in tandem. Their focus is on applying fetch gating when the traffic rate is low, whereas we focus on applying fetch gating when we can find and exploit value locality, and this includes when the traffic rate is high.

Manne et al. [19] observe that due to branch mispredictions wrong path instructions cause a large amount of unnecessary work in wide-issue super-scalar processors. They develop a hardware mechanism called pipeline gating to control rampant speculation in the pipeline. They use a confidence estimator to assess the quality of each branch prediction. In case of low confidence, they gate the pipeline by stalling instruction fetch. Baniasadi and Moshovos [2] extend this approach to throttle the execution of instruction flow in wide-issue super-scalar processors to achieve energy reduction. They use instruction flow information such as rate of instructions passing through stages to determine whether to stall stages. When the rate is sufficiently high and there is enough instruction level parallelism, they may stall fetch because introducing extra instructions would not significantly improve performance. Karkhanis et al. [15] also propose a mechanism called Just-In-Time instruction delivery to save energy. They observe that performance-driven design philosophy causes useful instructions to be fetched earlier than needed and stall in the pipeline for many cycles or they wait in the issue queue. Also when a branch misprediction occurs, all those early-issued instruction along mispredicted branch are flushed. This wastes energy. Their suggested mechanism monitors and dynamically adjusts the maximum number of in-flight instructions in the processor according to processor performance. When a maximum number is reached, the instruction fetching is gated. Buyuktosunoglu et al. [5] collect issue queue statistics to resize the issue queue dynamically to improve issue queue energy and performance on a super-scalar processor. The statistics are derived from counters that keep track of the active state of each queue entry on a cycle-by-cycle basis. They divide the issue queue into separate chunks and may turn off/on certain block based on statistics. The above prior works are related to our work since they all gate/throttle the execution of instruction flow to achieve the goal of energy reduction. For our approach, we build upon these techniques to gate fetch for our data-plane micro-engines.

3 Why a Traditional Cache Does Not Help the Throughput of a Balanced Network Processor

High end routers are targeted at providing worst-case throughput guarantees over latency. A network processor that is a balanced design cannot have the throughput of its data-plane algorithms (e.g., ip-lookup, classification, etc) increased by adding a traditional cache. By traditional cache, we mean a cache that uses a standard memory address as its index and tag, and then a N-bytes of consecutive memory is stored in the cache block starting at the block address. A traditional cache being used for a data-plane algorithm would be indexed by a standard load memory address to access an arbitrary level of the data-plane's algorithm's data structure. By *balanced design* we mean a network processor

design where the memory latency is already completely hidden for the desired worst case throughput [22] and each of the components of the network processor are designed for the same worst case throughput guarantees. Balanced network processors are designed with enough overlapped execution (threads) that the latency to perform each memory lookup in the data-plane algorithm is completely hidden. In such a design, traditional caches cannot help increase throughput, since they just attack latency.

4 Value Locality in Network Processing

By value locality we refer to the phenomenon that when a stream of packets is examined, certain fields of the packet headers have the same values occurring in them repeatedly. The occurrence of value locality in the fields of packet headers are a direct result of the behavior of high level network protocols and network applications. For example, in a file transfer protocol, bursts of packets are generated between a specific pair of hosts within a short amount of time. Thus, the source and destination address fields exhibit value locality. The values in packet headers often directly determine which portions of the data-plane algorithm's data structure a network processing application will access. Therefore value locality in packet header fields gives rise to temporal locality in accesses to data in these data structures. In this section we examine how value locality in packet header fields result in temporal locality in accesses to data structures used in the three network processing tasks examined in this paper.

IP Routing Table Lookup. We first examine the longest prefix matching routing lookup application. The routing table is a set of entries each containing a destination network address, a network mask, and an output port identifier. Given a destination IP address, routing lookup uses the network mask of an entry to select the most significant bits from the destination address. If the result matches the destination network address of the entry, the output port identifier in the entry is a potential lookup result. Among such matches, the entry with the longest mask is the final lookup result [10]. In some implementations, the routing table is often organized as a trie, either multi-bit Patricia trie [24] or reduced radix tree [10]. Each lookup generates a sequence of accesses to the trie entries from the top of the trie down to the external nodes of the trie. Most important of all, the entries of the trie that are accessed are actually determined by the value of the destination IP address field of the packets. As shown by our study as well as work of other researchers [7, 21], the destination IP address field shows high value locality. The routing lookup generates many sequences of memory accesses to the same trie entries over a fairly long interval of time and thus exhibiting temporal locality. We can exploit this temporal locality by putting the recently accessed trie entries with the routing lookup result in a computation reuse cache. Therefore other accesses in the near future, with the same destination, can then be avoided by checking this cache first.

Packet Classification Table Lookup. Packet classification is an essential part of network processing applications. Many applications require packet clas-

sification to classify packets into different flows based upon multiple fields from packet headers [13, 12] (e.g., DiffServ uses a 5-tuple lookup consisting of IP source and destination addresses, the TCP source and destination ports, and the protocol field for classification). A classification table is used to record different flows which must be kept up to date. The entries accessed are actually determined by the above mentioned 5-tuple. Since once a flow is established, large number of packets in the flow are typically sent in a burst, the combination of this 5-tuple of fields demonstrates high value locality. The same classification table entry is accessed many times. We can exploit this behavior by caching the flow identification found so that closely following accesses to the same classification table entry can be avoided. Factors that affect packet classification are studied in [4, 16].

NAT Portmapping Table Lookup. The Network Address Translation (NAT) protocol [11] can multiplex traffic from an internal network and present it to the Internet as if it was coming from a single computer having only one IP address in order to overcome the shortage of IP addresses, address security needs, or to ease and increase the flexibility of network administration. The NAT gateway uses a port mapping table that relates the client's real local IP address and its source port plus the translated source port number to a destination address and port. When a packet from an internal network is being sent out, the NAT gateway looks up the port mapping table and modifies the source address of the packet to the unique local network IP address and replaces the source port with translated source port. When a reply from the remote machine arrives, the port mapping table is used to reverse the procedure. The four fields in the internal network packet header, including the source address, source port, destination address and destination port, which defines a TCP/IP connection, determine which entry in the port mapping table is accessed. Since once a TCP/IP connection is established, bursts of packets in the connection will be sent out, these four fields have high value locality. As a result, the same entry in the port mapping table will be accessed very frequently and demonstrate temporal locality. We can exploit this behavior by caching the recently accessed entries in the port mapping table to avoid the lookups.

Table 1 summarizes the behavior of the applications discussed. Next we present results measuring the degree of value locality in these applications. For this study we use traces of IP packets taken from Auckland-II Trace Archive [1]. Characteristics of these traces, including the total number of packets and distinct destination addresses, are given in Table 2. We selected packet traces from the trace archive that had a large number of distinct destinations with respect to the number of packets in the trace.

We measured the value locality in terms of the percentage of the packets which have a frequently occurring combination of values for the relevant n-tuple of header fields where n is the number of relevant header fields for a given application. According to Table 1, *IP Routing* examines a 1-tuple, *Packet Classification* examines a 5-tuple, and *NAT Protocol* examines a 4-tuple of fields. To measure this locality we divided the packet stream into intervals of 16K of

Application	Data Structure	Packet Header Field	Description
IP Routing	Routing Table	IP Destination Address	Dest. address field is used to lookup next hop info. in the routing table.
Packet Classification	Classification Table	IP Source Address TCP Source Port IP Destination Address TCP Destination Port Protocol Number	The 5-tuple is used to lookup the classification table and identify the flow the packet belongs to.
NAT Protocol (In)	Port Mapping Table	IP Destination Address TCP Destination Port	The destination address and port fields are used to lookup for the internal destination address and port inside the subnet for incoming traffic.
NAT Protocol (Out)	Port Mapping Table	IP Source Address TCP Source Port	The source address and port fields are used to lookup for a translated source port for outgoing traffic.

Table 1. Application Properties

Packet Stream	Source	Num. of Packets	Num. of Distinct Destinations
1	19991129-134258-0	17045569	8920
2	20000112-111915-0	17934563	14033
3	20000117-095016-0	18433128	11746
4	20000126-205741-0	18823943	9012

Table 2. Packet Stream Characteristics

consecutive packets. We determined the top 16, 32, 64, and 128 values for the above n-tuples for each interval by examining the packets in each interval.

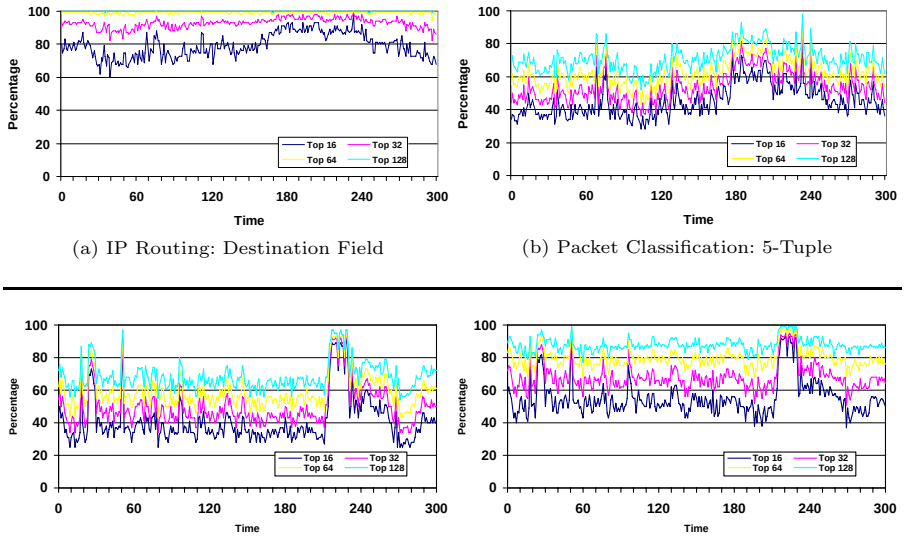
Figure 1 plots the percentage of packets that match the top 16, 32, 64 and 128 n-tuples over time for the packet stream trace 1 in Table 2. We omit the graphs for the other packet stream traces as they are similar. For this trace, even though there was 8920 unique destination addresses, in the 16K intervals shown, capturing the top 128 n-tuples accounted for 60% to 100% of the packets. This shows that within 16K intervals a significant amount of value reuse occurs for the applications examined. In addition, we also examined varying the interval size from 1K to 1024K packets, and the results were roughly the same. Therefore, the degree of value locality in these packet streams is quite significant.

We now consider the *unique recurrence distance* between a pair of packets with the same n-tuple value. This is calculated as the number of packets between two distinct occurrences of two packets with the same top n-tuple value. This notion is similar to the working set size of a cache and more accurately reflects the value locality. Table 3 shows the average number of packets between unique recurrences for the n-tuple values that are in the top 16, 32, 64 and 128 reoccurring tuples over 16K packet intervals. Results are shown for all four packet stream traces from Table 2. The results show that the most frequently occurring n-tuples, Top 16, have a smaller reuse distance than the Top 128 n-tuples.

The reuse distance and frequency of value locality results tell us that a cache of a small size is enough to catch a reasonable amount of value locality.

5 Computation Reuse Cache

In this section we present our computation reuse cache design for network processors. The cache design we propose has two distinct features. First, the cache can be used across several applications. This goal is achieved by making the cache programmable (i.e. the composition of the tag and data parts can be changed from one application to next). Second, this cache is designed to eliminate redundant computation associated with the network processing data-plane algorithm.



(a) IP Routing: Destination Field

(b) Packet Classification: 5-Tuple

(c) NAT Protocol (in) : 2-Tuple

(d) NAT Protocol (Out) : 2-Tuple

Fig. 1. Average Percentage of Packets with n-tuple Values from the Top 16/32/64/128 Frequently Observed n-tuple Values in 16K Packet Intervals.

Because of the repeated occurrence of the same packet header fields, the data-plane algorithm often performs redundant computation. The computation reuse cache remembers previously performed computations so that later redundant data-plane algorithm queries can be avoided.

It is important to note that the caching we perform corresponds to a coarse-grained computation made up of many instructions. Our approach is similar in idea to the dynamic instruction reuse techniques by Sodani and Sohi [23]. They focused on identifying arbitrary dependency changes of instructions in high performance processors that are performing redundant calculations. If these can be discovered, then the whole computation can be avoided. In our paper we exploit this concept with a programmable computation reuse cache. Our level of reuse focuses on reusing complete set of function calls (the data-plane algorithm query), instead of arbitrary dependency chains as examined by Sodani and Sohi. Our computation reuse cache is set up specifically for each data-plane algorithm to exploit reuse in its calculations. We focus on using this to streamline the processing of packets in order to save energy.

For redundancy elimination, a cache line is designed to contain the input (tag) and output (data) of the computation. The input is the relevant fields (n-tuple) in packet headers, working as the tags of the cache line, and the cache line data is the computation result. The cache is configurable by the application. Each data-plane algorithm is broken into three stages – preprocessing, data-plane processing, and post-processing, and a packet goes through these three stages when being processed by the data-plane algorithm. In the preprocessing stage of a packet, when portions of the packet are read in, we form a tag from the relevant

Stream	Top 16	Top 32	Top 64	Top 128
1	6.65	8.23	9.49	10.03
2	12.43	14.57	17.08	19.97
3	5.14	6.09	7.01	7.56
4	6.34	7.28	8.11	8.49

(a) Routing Lookup: 1-tuple

Stream	Top 16	Top 32	Top 64	Top 128
1	9.12	11.69	14.56	17.02
2	16.39	20.72	26.92	31.43
3	8.41	10.36	12.53	14.71
4	9.31	11.07	12.88	14.70

(b) NAT (out) : 2-Tuple

Stream	Top 16	Top 32	Top 64	Top 128
1	17.68	21.11	24.41	27.29
2	10.22	12.71	15.70	18.39
3	12.80	15.64	18.37	20.90
4	10.08	12.03	13.91	15.82

(b) NAT (in) : 2-Tuple

Stream	Top 16	Top 32	Top 64	Top 128
1	20.19	24.02	28.09	31.90
2	20.35	25.88	32.32	36.70
3	15.24	18. 50	21.84	24.69
4	12.57	14.78	16.84	19.20

(c) Packet Classification: 5-Tuple

Table 3. Average Unique Reuse Distance.

fields. Before the preprocessing stage ends, a lookup in the cache with the n-tuple for the packet is triggered. If a cache hit occurs, the hardware automatically changes the control flow to the post-processing stage for that packet's processing thread and avoids the whole execution of data-plane processing phase. During this post-processing, the computation result is copied from the cache block to registers. In case of a cache miss, the processing continues normally and when the starting point of post-processing phase is reached, the hardware updates the cache with the computation result. The cache is developed in such a way that the worst case throughput of the processing phase is not increased. This is implemented with software assistance and dedicated hardware.

A special register called the *jump target register* (JTR) is added in the micro-engine controller for each thread (hardware context). JTR remembers the starting point of the postprocessing phase of the algorithm so that in case of a cache hit, the control can be directly transfered to this point. In case of cache miss, at this instruction address the computation and its result are sent to the cache for updating the cache line. This register is set during the initialization part of the data-plane algorithm.

The cache is configurable in both the input and output of the processing phase. When initializing the data-plane algorithm at boot time, the configuration of the cache is specified. Masks in the cache are set up so that appropriate header fields can be extracted from the packet for use as the index into the cache. The starting point of postprocessing stage is put into JTR.

When a packet is preprocessed, the data to perform data-plane algorithm comes from the packet header. This same data is used to form the cache index and tag. Therefore, the memory read instructions in the preprocessing phase are marked so that as they read the relevant packet header fields, those values are also sent to the computation reuse cache. Multiple memory read instructions may need to be marked depending upon the number of fields in the n-tuple which acts as the input to the data-plane algorithm. The cache is setup to receive for each thread the input n-tuple, and the arrival of the last value triggers the computation reuse cache lookup.

When performing the computation reuse cache lookup, the n-tuple is hashed into a cache set index, and then the n-tuple is compared to all of the tags in the set. Note, the tag is itself a n-tuple. If there is a hit for the n-tuple, the result data is copied into registers for the postprocessing phase and control is transfered to the instruction in the JTR register. In case of a cache miss, the cache tag is updated with the n-tuple, and the cache block is updated with the result data that becomes available after the processing phase of the data-plane algorithm.

The results in this paper assume that there is a separate computation reuse cache for each data-plane algorithm examined.

6 Using Fetch Gating

In this section, we describe our approach for performing fetch gating while using the computation reuse cache. Fetch gating is a form of pipeline gating proposed by Manne et. al. [19]. Pipeline gating was proposed to stop fetching and executing instructions down wrong (branch mispredicted) paths of execution in order to save energy. We use the same concept here to stall fetch, resulting in energy savings, for the data-plane algorithm when the calculations can be reused due to the computation reuse cache hits. This is possible since the overall network processor is balanced, and if the data-plane algorithm can reuse and jump ahead in its computations this creates slack in the data-plane algorithm's schedule. It can therefore gate execution while the rest of the network processor continues to process the packets for the overall designed throughput.

Figure 2 gives a simplified high level view of a network processor using the computation reuse cache for a data-plane algorithm. In our design we assume that the data-plane algorithm has two queues connecting it to the other parts of the processor so that it can be scaled independently of other stages. The input queue to the data plane algorithm initiates the fetch gating logic. Each time there is a change in the queue length, the hardware decides whether to perform fetch gating or keep the processor in normal state. The fetch gating algorithm currently uses two levels. One corresponds to the standard operation when no fetch gating is performed, and the other corresponds to the fetch gated power saving mode. During the latter mode, no fetching or execution will be performed by the fetch gated data-plane algorithm micro-engine.

The underlying principle of our approach is to perform fetch gating based upon the occupancy of the input queue shown in Figure 2. In a balanced network processor design, the occupancy of the input queue should be low. If this is the case, and we are getting a reasonable number of computation reuse cache hits, we can save energy by applying fetch gating to the micro-engine. This can continue up to a point. Once the input queue becomes occupied enough, fetch gating is turned off in order to still provide worst-case throughput guarantees and to prevent packets from being dropped.

In Figure 2, we assume that the number of packets in the input queue is n_i. The fetch gating algorithm checks whether the input queue size, n_i, is smaller than an worst case throughput input queue size threshold. If the input queue

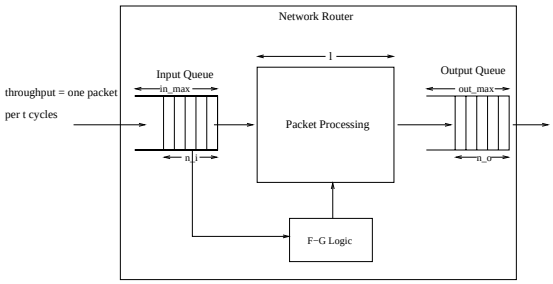


Fig. 2. Network Router with Fetch Gating (F-G) logic

to the data-plane algorithm micro-engine has enough empty slots to guarantee worst case throughput for its implementation, then the micro-engine is allowed to be in fetch-gated, else the algorithm performs normal fetch.

7 Experimental Evaluation

We use the Nepsim [17] simulator in our experiments. Nepsim is a cycle-accurate simulator of the Intel IXP1200 network processor. We modified Nepsim to represent a balanced network processor with higher throughputs and extended it with our computation reuse cache and our fetch gating algorithm. For our results we model a 30 cycle on-chip SRAM latency to store the data structure for the data-plane algorithm being examined.

In our evaluation, we use four benchmarks which are modified from Intel's Workbench suite or developed by ourselves and migrated to run on the Nepsim simulator. They are IP-Lookup, Packet Classification, and NAT Protocol (in and out). The properties of the application are summarized in Table 4. These applications have the code size of 200 to 300 instructions, shown in the first column. We also show the worst-case packet processing time (latency) that was observed across the four traces in the second column. For IP-Lookup this is 646 cycles and for packet classification it is 1160 cycles. This processing time is considered as the time period between when a packet enters into the processing stage and when it leaves the processing stage (i.e., not counting the time that the data-plane algorithm is spinning and waiting for the packet to arrive). It also does not include the cycles spent receiving and transmitting the packet. We also show the average-case packet processing time in the third column – the average is computed over the four traces. The last column shows the number of SRAM references needed in the worst case for the algorithm.

7.1 Cache Behavior

Table 5 shows the hit rate when using a 64 entry direct mapped computation reuse cache when the four different packet streams from Table 2 are used as input. The cache hits vary between 52% and 89%. These results are also consistent with the results of our packet value locality study.

Applications	Code Size	Proc. Time (Worst)	Proc. Time (Average)	#SRAM Refs
ip-lookup	291	646	435	5
classification	254	1160	1010	13
nat-in	205	754	603	6
nat-out	205	757	597	6

Table 4. Application Properties

Applications	Packet Stream Trace			
	1	2	3	4
ip-lookup	68.43%	70.18%	88.89%	85.79%
classification	60.03%	84.97%	77.42%	77.42%
nat-in	74.70%	67.87%	84.45%	82.57%
nat-out	52.52%	82.78%	71.68%	71.68%

Table 5. Computation Reuse Cache Hit Rate

Applications	ME % Total Energy	Time No Cache	Time Reduction	Cycles Gated	ME Energy Reduction
ip-lookup	33.47%	435	18.62%	22.45%	16.61%
classification	29.66%	1010	47.23%	30.78%	18.89%
nat-in	27.71%	603	51.58%	45.82%	37.69%
nat-out	29.38%	597	41.37%	42.64%	28.43%

Table 6. Program Behavior with Computation Reuse Cache

7.2 Fetch Gating and Energy Savings

Table 6 shows the effect of cache hits on the data-plane algorithm. These results were obtained by running trace 1 from Table 2 through each of the algorithms. The first column, ME Energy, gives the energy used by the micro-engine on which the data-plane algorithm is being run as a percentage of total energy of the network processor. The second and third columns show the average packet processing latency (cycles) in the absence of cache and the percentage reduction in this time when the cache is used. The primary benefit of a computation reuse cache hit is the reduction in instructions executed in order to perform the packet processing. For IP-Lookup, we observe close to 19% reduction while for classification we achieve roughly 47% reduction in processing time. The results tell us that the computation reuse cache can provide significant reductions in instructions executed and this creates significant slack.

The last two columns in Table 6 show the results of applying our fetch gating algorithm in the prior section. The energy is recorded using the models provided in the Nepsim [17] simulator augmented to take into consideration the cache and our network processor changes. The fourth column shows the percentage of cycles the algorithm was fetch gated. The last column shows the percentage of energy savings when using the fetch gating when compared to the energy used for just that micro-engine shown in column one. For these results, we only examine one data-plane algorithm at a time, and the computation reuse cache is 64-entry and directed mapped. The results show that for IP-Lookup we spend 22% of the cycles fetch gated for the data-plane algorithm micro-engine and achieve an energy savings of nearly 17%. This results in an overall network processor energy savings of 6%. Across all applications, the computation reuse cache allows 22%

to 46% of packet processing to be performed in fetch gated mode which produces 17% to 38% in energy savings for the micro-engines. Note, that slack is generated by reducing the amount of computation needed for a packet when there is a hit in the computation reuse cache, and then energy savings comes from using fetch gating to exploit this slack.

8 Conclusions

High end network processors are built with a balanced processor design, where using a traditional cache for the data-plane algorithm will not increase throughput. These processors are built such that there is sufficient threading (packet parallelism) to hide each data-plane algorithm SRAM lookup latency, so a traditional cache cannot increase throughput.

In this paper we presented a computation reuse cache, where a hit hides the full data-plane algorithm processing of the packet, not just one SRAM lookup as in a traditional cache. This is accomplished by having a cache block contain the input as the tag and output as the data of the data-plane algorithm computation. Therefore a complete query performed by the data-plane algorithm takes one cache access if there is a hit. Slack is therefore generated by reducing the number of instructions executed when there is a hit. This reduction in number of instructions allows us to exploit the slack to save energy through fetch gating for the data-plane algorithm micro-engine while still matching the worst case throughput guarantees of the rest of the processor. Overall, the computation reuse cache allowed 22% to 46% of the execution time to be performed in fetch gated mode with 17% to 38% reduction in data-plane algorithm energy across the different algorithms examined.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments on this paper, and early feedback from Nathan Tuck on this topic. This work was funded in part by NSF grant CNS-0509546, and grants from Microsoft and Intel Corporation to the University of California, San Diego and NSF grant CCF-0208756, and grants from Intel Corp., IBM Corp., and Microsoft to the University of Arizona.

References

1. Auckland-II Trace Archive, http://pma.nlanr.net/Traces/long/auck2.html
2. A. Baniasadi and A. Moshovos, "Instruction Flow-based Front-end Throttling for Power-Aware High-Performance Processors," *International Symposium on Low Power Electronics and Design*, August 2001.
3. P. Brink, M. Casterlino, D. Meng, C. Rawal, and H. Tadepalli, "Network Processing Performance Metrics for IA- and IXP-based Systems," *Intel Technology Journal*, Vol. 7, No 4, Nov. 2003.
4. N. Brownlee and M. Murray, "Streams, Flows and Torrents," *Passive and Active Measurement Workshop*, April 2001.

5. A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi, "An Adaptive Issue Queue for Reduced Power at High Performance," *International Workshop on Power-Aware Computer Systems*, November 2000.
6. T. Chiueh and P. Pradhan, "High Performance IP Routing Table Lookup Using CPU Caching," *IEEE Conference on Computer Communications*, April 1999.
7. K. Claffy, "Internet Traffic Characterization," Ph.D. thesis, Univ. of California, San Diego, 1994.
8. M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small Forwarding Tables for Fast Routing Lookups," *ACM Conference of the Special Interest Group on Data Communication*, September 1997.
9. Y. Ding and Z. Li, "A Compiler Scheme for Reusing Intermediate Computation Results," *International Symposium on Code Generation and Optimization*, March 2004.
10. W. Doeringer, G. Karjoth, and M. Nassehi, "Routing on Longest Matching Prefixes," *IEEE/ACM Transactions on Networking*, Vol. 4, No. 1, pages 86-97, Feb. 1996.
11. K. Egevang and P. Francis, "The IP Network Address Translator (NAT)," RFC 1631, May 1994.
12. P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," *ACM Conference of the Special Interest Group on Data Communication*, September 1999.
13. P. Gupta and N. McKeown, "Algorithms for Packet Classification," *IEEE Network*, Vol. 15, No. 2, pages 24-32, Sept. 2001.
14. E. Johnson and A. Kunze, *IXP2400/2800 Programming*, Intel Press, 2003.
15. T. Karkhanis, J.E. Smith, and P. Bose, "Saving Energy with Just In Time Instruction Delivery," *International Symposium on Low Power Electronics and Design*, August 2002.
16. K. Li, F. Chang, D. Berger, and W. Feng, "Architectures for Packet Classification Caching," *The 11th IEEE International Conference on Networks*, Sept./Oct. 2003.
17. Y. Luo, J. Yang, L. Bhuyan, and L. Zhao, "NePSim: A Network Processor Simulator with Power Evaluation Framework," *IEEE Micro Special Issue on Network Processors for Future High-End Systems and Applications*, Sept./Oct. 2004.
18. Y. Luo, J. Yu, J. Yang, and L. Bhuyan, "Low Power Network Processor Design Using Clock Gating," *42nd Annual Design Automation Conference*, June 2005.
19. S. Manne, A. Klauser, and D. Grunwald, "Pipeline Gating: Speculation Control for Energy Reduction," *International Symposium on Computer Architecture*, June 1998.
20. G. Memik and W.H. Mangione-Smith, "Improving Power Efficiency of Multi-Core Network Processors Through Data Filtering," *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, Oct. 2002.
21. C. Partridge, "Locality and Route Caches", *NSF Workshop on Internet Statistics Measurement and Analysis*, Feb. 1996. (http://www.caida.org/ISMA/Positions/partridge.html).
22. T. Sherwood, G. Varghese, and B. Calder, "A Pipelined Memory Architecture for High Throughput Network Processors," *International Symposium on Computer Architecture*, June 2003.
23. A. Sodani and G.S. Sohi, "Dynamic Instruction Reuse," *International Symposium on Computer Architecture*, pages 194-205, June 1997.
24. W. Szpankowski, "Patricia Tries Again Revisited", *Journal of the ACM*, Vol. 37, No. 4, pages 691-711, October 1990.