Hybrid cache architecture for high-speed packet processing

Z. Liu, K. Zheng and B. Liu

Abstract: The exposed memory hierarchies employed in many network processors (NPs) are expensive in terms of meeting the worst-case processing requirement. Moreover, it is difficult to effectively utilise them because of the explicit data movement between different memory levels. Also, the effectiveness of traditional cache in NPs needs to be improved. A memory hierarchy component, called split control cache, is presented that employs two independent low-latency memory stores to temporarily hold the flow-based and application-relevant information, exploiting the different locality behaviours exhibited by these two types of data. Just like conventional cache, data movement is manipulated by specially designed hardware so as to relieve the programmers from the details of memory management. Software simulation shows that compared with conventional cache, a performance improvement of up to 90% can be achieved by this scheme for OC-3c and OC-12c links.

1 Introduction

To meet the demands of high performance and greater flexibility, simultaneously, network processors (NP) typically employ a bunch of architectural features that are specially adapted to the characteristics of packet processing. For example, multiple RISC-based processing elements (PEs) with instruction sets optimised for protocol handling are often integrated into one single chip, exploiting the parallelism in packet flows. Instead of the data cache that is extensively used in the modern general purpose processor, most NPs expose their memory hierarchies to programmers, expecting explicit allocation of appropriate address regions to data structures. This design is mainly based on the deteriorated worst-case performance of the conventional caching mechanism and the common belief of the lack of locality in network applications [1].

However, most present-day NP-based systems are deployed at metropolitan networks where sophisticated applications like network security are demanded and low cost is one of the major concerns [2]. Providing enough resources for a NP with exposed memory hierarchy is often prohibitively expensive when meeting the worst-case processing requirement of these applications. On the other hand, effective utilisation of this memory organisation adds a lot of software overhead in data management, which potentially increases the cost of NP deployment. For example, critical data should reside in a high-speed on-chip buffer to reduce the access latency. A large data structure that cannot fit into the on-chip buffer has to be divided into several pieces and swapped in and out of the chip, making the program complicated and less efficient.

© The Institution of Engineering and Technology 2007

doi:10.1049/iet-cdt:20060085

Paper first received 8th June and in revised form 5th October 2006

Z. Liu and B. Liu are with the Department of Computer Science and Technology, Tsinghua University, East Main Building 9-416, Beijing 100084, People's Republic of China

K. Zheng is with the System Research Group, IBM China Research Lab, Building 19 Zhongguancun Software Park, No. 8 Dangbeiwang West Road, Haidian District, Beijing 100094, People's Republic of China

E-mail: liuzhen02@mails.tsinghua.edu.cn

Recent studies have revealed that appropriate data caching can effectively speed-up packet processing and consume less off-chip memory bandwidth [3]. Especially, when packets of the same flow are forced to be allocated to the same thread, a such a caching mechanism alleviates the impact of burstiness in traffic on the utilisation of threads [4]. We simulate the packet processing procedure of a four-PE network processor using a traffic trace collected on an OC-12c link. Fig. 1 compares the packet loss rates of a different number of cache entries. Here, it is assumed that each flow has its own control data and these data are organised as entries. The ratio of total memory access delay and register instruction operation time is set as 5:1. The average packet arriving interval of the tested trace is 31 μs. If the queuing delay of a packet is twice that, the packet is discarded. In this figure, the processing time for each packet accounts for only 40% of the theoretical maximum cycle budget. But the burst arrival of packets from the same flow makes adding more threads less attractive. Data caches reduce the suspended time of threads and releases them for other packets as soon as possible. Note the logarithmic scale on the Y axis, the packet loss rate with a cache that holds information for 1024 flows decreases to less than one-tenth compared with non-caching schemes in all of the four cases. Moreover, hardware manipulated data movements between different levels of memory hierarchy in conventional cache also relieve programmers from the details of memory allocation.

Although data cache seems appropriate for mid-end NPs, current cache organisations need to be improved in order to deliver higher performance [3, 5, 6]. We have observed that common programs exhibit a high degree of spatial and temporal locality that can be easily exploited by hierarchical organisations. But in network applications, various types of data have totally different characteristics. When these data are treated in the same cache and with the same strategy, their properties cannot be fully utilised and data with different access patterns may conflict with each other. In this article, we present a novel memory hierarchy component that is specially designed to meet the processing demand in a NP. The proposed architecture, called split control cache, employs two independent memory stores

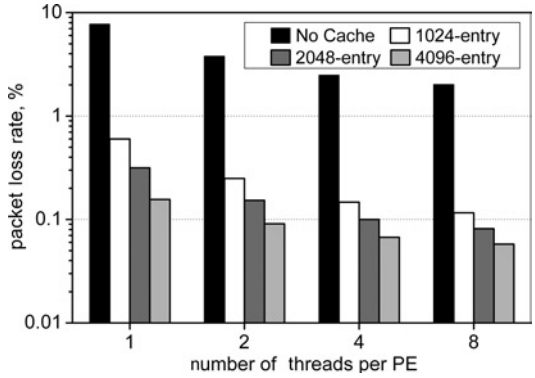

Fig. 1 *Packet loss rates in a four-PE network processor under the workload of 40%*

for flow-based and application-relevant data, respectively. With this component, control data needed by packet processing can be effectively delivered to the execution unit of the PE and only minor modifications are required for the migration of existing tool chains such as the compiler.

2 NP architecture

Fig. 2 shows a generic NP architecture, which resembles that of a AMCC nP3700 [7]. Its processor hierarchy includes a set of fully programmable PEs, a number of coprocessors (e.g. packet classifier and hash unit), and some hardware assists (like the traffic manager). On-chip memories (such as scratch pad memory) are incorporated to provide access for a small amount of critical data. Packets are managed by dedicated hardware (i.e. packet buffer controller) and preprocessed before passing to the PE. PEs and coprocessors can cooperate in several ways based on the configurations of the interconnection network. In one possible approach, the PE may trigger the coprocessor using special instructions and get the results through memory mapping. Coprocessors can also work as stages of a pipeline and send the result to PEs directly using the interconnection network.

Note that Fig. 2 has almost all the prevailing architectural characteristics of commercial network processors. For example, the Intel IXP network processors also have a lot of processing cores, flexible memory hierarchy and some hardware functional units; but their packet receive/transmit

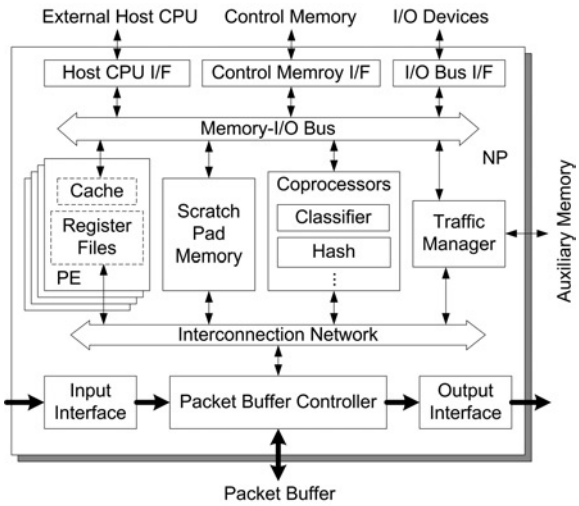

Fig. 2 *Architecture of a NP*

unit does not have much hardware support and no traffic manager is provided [8]. As for Hifn' 5NP4G, the major difference is that it employs a lot more types of coprocessors and hardware accelerators [9].

3 Split control cache

NPs deal with three kinds of information: instruction code, packets and control data. This article only focuses on cache design for control data, since most NPs have separate instruction stores that can meet the processing demand and the storage of packets has already attracted much research effort [10]. The proposed scheme comes from the observation that different types of control data usually exhibit different locality properties. In particular:

- Packet-relevant data structures are generated during the processing of individual packets. They are typically temporary variables that will not be used by other packets. Their memory consumption is relatively small (approximately several hundreds of bytes) and references to them constitute a large percentage of the total memory accesses [3]. Therefore, on-chip local memory such as scratch pad is most appropriate for holding them.
- Flow-relevant data structures maintain information for packet flows in applications such as quality of service (QoS), sophisticated billing and monitoring. They are shared among packets from the same flow. An example of this is a Transfer Control Protocol (TCP) connection table recording the connection flags, sequence numbers, window sizes and acknowledge numbers for each flow transmitted by a router. Memory references to these data for individual packet exhibits primarily spatial locality. Temporal reuses of them occur only when packets belonging to the same flow are received.
- Application-relevant data structures are used for implementing some specific functions and are typically accessed by each incoming packet. Tables containing policy, routing, and QoS information fall into this category. The reference pattern of them relies on distinct algorithms. For example, packets with similar destination addresses will access the same nodes of a prefix trie. Therefore, the temporal locality of a trie-based route lookup will be higher than a hash table-based mapping function, which transfers the destination address to an index number with only one memory reference. However, the spatial locality of the former function is much lower than that of a pattern table lookup if linear-search is employed to find the matching entry.

Information related to each flow occupies up to a few hundred bytes and simultaneously existing flows on a particular link can be hundreds of thousand [1]. In addition to some global variables that only account for several tens of bytes, most application-relevant data are space consuming. For example, a medium-sized route table may contain more than 50 000 of prefixes. Therefore, both of them have to be stored in external large-capacity memories with long access latency and PEs can get the demanded data from these devices through cache. The proposed scheme, split-control cache, consists of two independent memories (referred as subcaches). One is called Flow-Cache, which is designed to accommodate flow-relevant data and the other is App-Cache for application-relevant data. The separation of this cache architecture comes from the following considerations:

- Application data typically requires a much larger memory space than information maintained for one flow.

If too much flow data are kept in the same cache with application data, information of those less frequently encountered flows will reduce the available space for application data. Therefore, the chance of conflicts (called cache pollution) is increased, which is totally avoided in separated caches.

- Since flow-relevant data exhibits a high spatial locality, we can use a much larger cache-line in a separate cache to fully exploit it without disturbing the contents of application data.
- If we further limit the flow information into a continuous memory space (sacrificing a little programming flexibility), the starting address of these areas can be easily acquired through packet classification. If packet classification takes place as the very first processing step (in fact, this is the case for most network systems) by the coprocessor (e.g. the classifier shown in Fig. 2), flow information can be prefetched into cache before the other processing begins.

The method we use to distinguish flow data from application data for PEs is that of assigning non-overlapping address spaces. In another words, the memory space allocated to the control data is partitioned into two continuous areas, one for flow-based data and the other for application-based data. Thus, the type of data can be identified by the significant bits of their memory addresses.

Fig. 3 shows the architecture of split control cache. The upper and lower boundaries of each address space are guarded by a pair of Range Registers in the Address Recogniser. When PE issues a Control Memory reference, both pairs of registers are simultaneously checked to determine into which respective range it falls. Depending on the result, at most one subcache is selected and used by the PE. The contents of Range Registers are specified during initialisation and can be reprogrammed by the host processor through interruption. This arrangement makes the programming model for memory reference no different from the commonly used high-level language and only minor modifications of the compiler are needed.

3.1 Flow-Cache

The App-Cache has the same organisation with conventional caches whereas the Flow-Cache employs a quite different architecture. In order to simplify the memory management strategy and the hardware design, memory space allocated to flow-based information is divided into blocks of equal size. A block will be assigned to a flow and the mapping relationships are recorded as part of the packet classification results. When a new packet is received, it is classified by the coprocessor and the starting address (labelled as prefetch address in Fig. 3) of the block containing its flow information is returned. Then this address is looked-up in the Flow-Cache to determine whether it has already resided in it. In the case of a cache miss, a burst read of the demanded block is initiated. Once the new block has been placed into Flow-Cache, the PE is allowed to start the operation on that packet. Throughout the course of protocol handling, PE is not permitted to access the information of other flows. Therefore, cache miss will only be incurred by App-Cache during the processing procedure for that packet. In this way, the uncertainty of processing time is reduced, which is favourable for the achievement of wire-speed packet forwarding.

To ensure that flow-based data prefetching does not disturb the processing of other packets, we employ two independent banks. At any time, the bank holding flow information of the packet being processed is used by the PE for program execution. The other bank can prefetch demanded block for the next packet in the case of a Flow-Cache miss. Fig. 4 shows a sample execution procedure of a NP for a series of packets with equal inter-arrival time. It can be seen that packet classification, flow information prefetching and packet processing can be executed in a pipelined fashion. If no prefetch is needed for a certain packet, PE goes on with the bank containing its flow information. Otherwise, a bank switching occurs at the end of the processing for the previous packet.

The architecture of Flow-Cache is shown in Fig. 5. Each bank contains a Data RAM and the associated Tag RAM. The cache line size is the same as that of the block in Control Memory so that each cache miss incurs only one cache line refilling. Note that Tag RAM is only used to verify whether a block has already been prefetched into one of the two banks. When a prefetch address arrives, its lower part is indexed into the two Tag RAMs simultaneously. If the corresponding entry is valid (i.e. the 'V' bit in Fig. 5 is set), the higher part of the prefetch address will be compared with the tag. Then the bank number of the matching entry is recorded in the registers of the bank switch controller for that packet.

Flow-Cache follows a write-back policy to reduce off-chip memory accesses. Several dirty-bits (labeled as

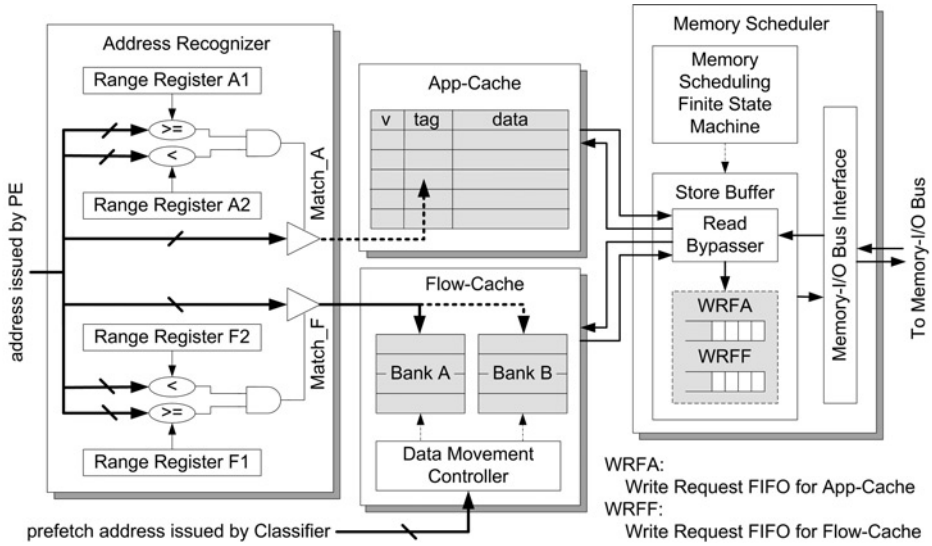

Fig. 3 *Block diagram of split control cache*

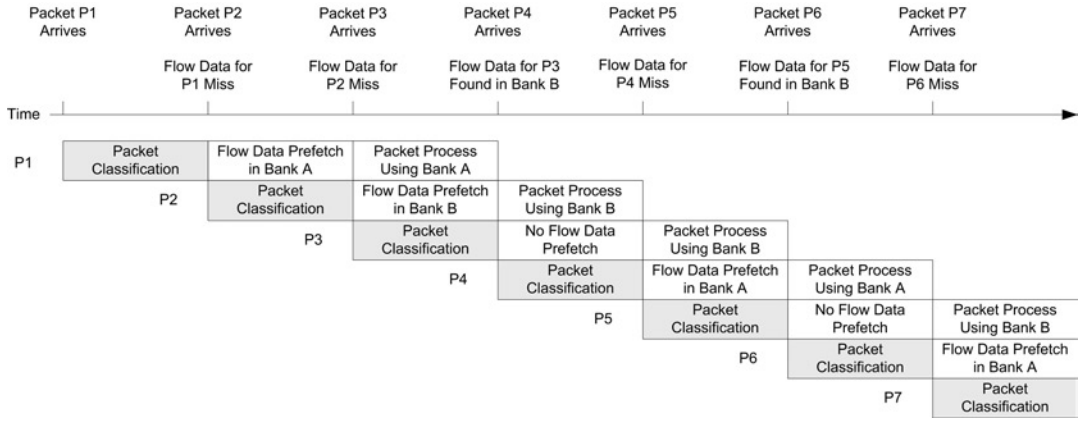

Fig. 4 *Pipelined execution in NP and the bank utilisation in Flow-Cache*

"D" in Fig. 5) are kept for each cache-line within the two banks, indicating whether the corresponding part of a cache-line has been modified. The actual number of dirty-bits is determined by memory size limitation and some implementation considerations. For example, one dirty-bit can be set for a number of bytes equal to the length of a burst write. By the time a miss happens, the cache-line selected by the index part of the prefetch address in the bank not used by PE is chosen as the replaced one.

The architecture of split control cache only establishes the access fashion of different memory space. For each packet, the whole memory space designated to App-Cache and a certain block within the space designated to Flow-Cache can be accessed by the PE. However, for a data structure, which area to be located into is determined by the programmer, based on whether this information will be shared among all the incoming packets or not.

In practice, some optimisation can be made to take full advantage of the proposed scheme. For example, operations such as route lookup and QoS enforcement are identical for all packets of the same flow. Thus, in addition to flow state information, execution results of these operations can also be stored in Flow-Cache. Only when a new flow is encountered or the result becomes invalid (e.g., the route table is updated), will they be performed [11]. Operations returning different results for packets of the same flow such as a TCP connection maintenance procedure cannot be treated in this way.

3.2 Memory Scheduler

Memory Scheduler coordinates memory access requests from the two subcaches. As shown in Fig. 3, read requests have a higher priority and are sent directly to the Memory-I/O bus. Write requests, however, are queued in the two subcaches' respective FIFOs (first-in/first out) and issued when all the read transactions are completed. Among the same type of requests, missing data of App-Cache are needed immediately for program execution, while the Control Memory accesses issued by Flow-Cache are making preparations for the next packet. On the other hand, application data are seldom modified by packet processing but flow information may be changed by each related packet. Therefore, App-Cache read misses take the highest priority but its write misses have the lowest. The Memory Scheduling Finite State Machine rearranges the issuing order of memory access requests according to these rules.

4 Performance evaluation

4.1 Software simulation

We use a dedicated network processor simulator NePSim 1.0 to study the effect of our scheme [12]. NePSim employs an event-driven framework to keep track of the

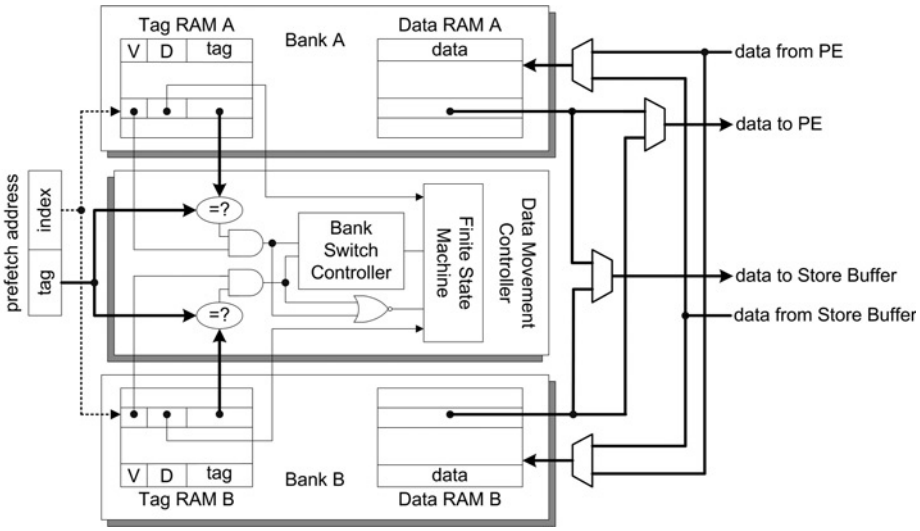

Fig. 5 *Block diagram of Flow-Cache and its major data path*

Table 1: Simulator parameters

PE frequency	696 MHz
Pipeline depth	5
Thread number per PE	4
Data cache	direct-mapped, 1-cycle latency, write through, no refilling, 8-byte cache line
Control memory device	SDR SDRAM
Memory bus bandwidth	116 MHz × 64 bit
SDRAM access latency	8.62 ns row access 8.62 ns column access 17.24 ns precharge

commands issued by threads, as well as the states of memory devices. In its original design, packet receiving and transmission are controlled by the user's program. Since these operations are nontrivial and consume a large amount of instructions, we modify the network interface to provide hardware support for packet loading. We also extend the simulator to include caching mechanisms. Each PE has its own data cache. The coherence between different caches is maintained by software. Table 1 presents the major simulator parameters.

The application we implemented includes two kernels, an IP packet forwarder and a simple flow meter. The former program is ported from the benchmarks provided by NePSim and is the basic function that should be implemented by routers. It validates the IP header and performs a longest prefix match (LPM) algorithm using multi-bit trie lookup. The latter maintains a record for each incoming flow, which contains a packet count field and a byte count field. In this simulation, flows are specified by the source and destination IP addresses. Supposing each field of the flow record consumes 4 bytes, then only 16 bytes are needed in Flow-Cache (two 8-byte banks), which is negligible compared with the size of App-Cache. To eliminate the duplicated information in Flow-Cache and simplify the synchronisation among PEs, we allocate packets of the same flow to the same thread.

We utilised real-life traffic traces collected by the National Laboratory for Applied Network Research (NLANR) from two Internet exchange points [13]. Details of the two traces, BWY (BroadWaY) and MRA (MeRit Abilene), are listed in Table 2. The source and destination IP addresses in the packet traces are renumbered to maintain anonymity by NLANR. This process retains traffic patterns and flow information; but the renumbered IP addresses cannot be found in route table. To solve this problem, for each unique destination IP address, we randomly generate a new IP address according to the prefixes of the route table.

Table 3 lists the average miss rate comparisons between the App-Cache in a split control cache and a conventional cache with a different number of PEs. It shows that App-Cache outperforms conventional cache under every configuration. In both BWY and MRA, the miss rate of App-Cache is about 5% less than that of a conventional cache before the miss rate decreases to nearly zero. The average miss rate increases with the number of PEs where the cache is divided into several small local caches. This is caused by duplicated data in the local caches when packets sharing the same table information are allocated to different PEs.

Fig. 6 shows the comparisons of packet throughputs (con. and app. represent conventional cache and App-Cache,

Table 2: Packet trace characteristics

Trace	Site location & description	Rate	Date	Packets	Unique IPs
BWY	Columbia University (BroadWaY)	OC-3c	2004-3-2	2, 598, 848	21 351
MRA	MichNet Backbone, Southeast Michigan (MeRit Abilene)	OC-12c	2004-3-9	8, 727, 924	189 303

Table 3: Average miss rate comparisons between conventional cache and App-Cache (%)

			Total cache size (KB)						
			4	8	16	32	64	128	256
1-PE	BWY	conventional cache	30.76	24.57	10.02	8.07	6.03	1.39	0.50
		App-Cache	27.30	20.92	5.99	3.97	1.84	1.25	0.33
	MRA	conventional cache	40.81	32.40	16.35	12.03	9.02	3.01	1.91
		App-Cache	35.63	27.10	10.81	6.73	3.79	2.03	0.85
2-PE	BWY	conventional cache	35.69	28.25	21.75	8.40	6.60	5.15	0.98
		App-Cache	31.39	23.95	17.56	4.83	3.00	1.51	0.80
	MRA	conventional cache	47.05	37.55	29.49	14.63	10.99	8.20	3.02
		App-Cache	42.40	32.81	24.63	9.34	6.06	3.41	2.06
4-PE	BWY	conventional cache	43.88	33.88	26.31	19.93	7.07	5.50	4.43
		App-Cache	39.16	29.42	22.25	16.21	3.70	2.11	1.07
	MRA	conventional cache	56.01	45.12	36.48	28.65	13.20	9.82	7.73
		App-Cache	51.28	40.36	31.89	24.21	8.17	5.24	3.34
8-PE	BWY	conventional cache	60.35	44.08	33.07	25.82	19.92	6.74	5.08
		App-Cache	56.67	39.56	28.66	21.22	16.08	3.83	2.08
	MRA	conventional cache	67.48	50.96	40.84	32.46	25.74	11.08	8.28
		App-Cache	63.64	45.57	35.62	27.26	20.99	7.04	5.07

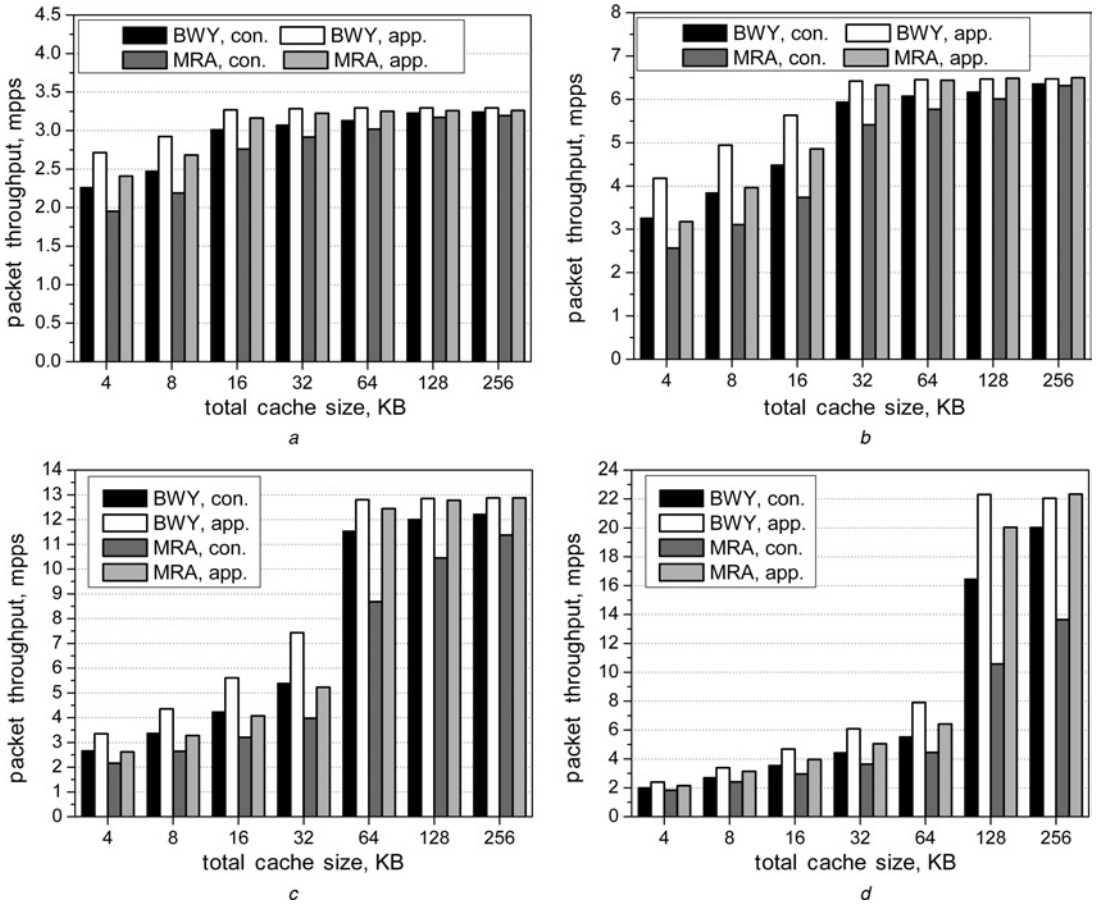

Fig. 6 *Packet throughput comparisons for network processor with different number of PEs*

a 1-PE
b 2-PE
c 4-PE
d 8-PE

respectively). For the 1-PE or 2-PE configuration, when the cache size is relatively small, our scheme is much more effective than conventional cache. For instance, a 4 kb split control cache with one PE achieves a performance improvement of about 20% in BWY and 23% in MRA. As the miss rate keeps falling with increased cache size, PEs begin to reach nearly full utilisation rate. Compared with conventional cache, split control cache only slightly improves the performance since almost no computation power can be exploited by hiding the access latency (note that we have four threads in each PE that can be switched upon cache misses). Take one PE with a 128 kb cache as an example: the active state accounts for 98.95 and 99.87% in conventional cache and split control cache, respectively. Hence, the packet throughput changes from 3.1712 mpps (million packets per second) to 3.2578 mpps, which is less than a 3% improvement.

If a large number of PEs is used, more contentions on memory bus are introduced and the access latency becomes much longer because of the queuing delay. Together with the increased cache miss rate (as analysed in Table 3), adding more PEs may not achieve a higher packet throughput under the same total cache size. This can be observed in the comparisons of Fig. 6*b*, 6*c* and 6*d* when the cache size is small. However, NP with split control cache still outperforms conventional cache. In this case, the decrease of cache miss rate becomes especially important. When the cache size is sufficiently large, spit control cache is much more effective in terms of increasing the PE's utilisation rate and NP's performance. For example, in MRA, the packet throughput of an 8-PE network processor with a total cache size of 128 kb increases from 10.58 mpps using a conventional cache to 20.03 mpps using our scheme, which is an improvement of nearly 90%.

4.2 Efficiency of Flow-Cache

Although only one cache line is enough for each bank in Flow-Cache, adding more blocks will reduce the frequency of the flow information prefetching and save memory bandwidth. Fig. 7 shows the prefetch percentage for the two traces. Since flow records are not shared among different flows, the prefetch rate varies slightly with a different

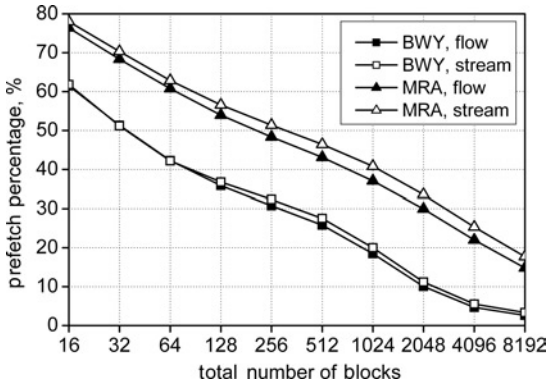

Fig. 7 *Prefetch rate in Flow-Cache with one PE*

Table 4: Stratix EP1S40F1508C5 FPGA device utilisation

Resources	Address recogniser	App-Cache	Flow-Cache	Memory scheduler
LE	50	159	315	543
register	0	43	83	332
RAM bit	0	17 792	17 856	17 408

number of PEs. Hence, we only consider the situation of one PE. The definition of flow is based on two granularities: (1) flows are sets of packets with the same source and destination IP addresses; (2) streams are IP sessions specified by the 5-tuple of <source IP address, source port, destination IP address, destination port, protocol identifier>. Since the majority of flows contain only one stream, the difference between their prefetch rates is minor. Although the total number of flows in the two traces reaches 41 487 and 329 882, a Flow-Cache with 1024 blocks can reduce the number of flow information prefetching to less than 20% for BWY and 40% for MRA. This is because most of the flows sustain just a few seconds and at any time, only a small subset of flows are being processed by the NP.

Unlike conventional cache, the tag comparison and cache data access in Flow-Cache do not take place at the same time. Therefore, the data output delay for the bank currently being used by the PE is not affected by the comparison delay of the tag. In fact, the cycle time of Flow-Cache is equivalent to that of a SRAM instead of a conventional cache. Since the cache line of Flow-Cache is typically longer than App-Cache, the access time is further reduced because of the drop in the address decoder delay (when the total cache size is the same) [14].

In conventional cache, tag comparisons are performed for each flow information access. But in Flow-Cache, at most one tag comparison is required for each incoming packet, which dissipates less power when more than one memory access for flow data is issued by each packet. However, tag and data RAM share the same address decoder in a conventional cache [14]. Thus, the area consumption of Flow-Cache is slightly larger than conventional cache with the same total and block size due to the additional address decoder for tag.

5 Functional prototype

We implemented this system based on a simplified OpenRISC processor core in an Altera Stratix EP1S40F1508C5 field-programmable gate array (FPGA) [15]. EP1S40 contains 41 250 logic elements (LEs) and a total number of 3 423 744 RAM bits [16]. The App-Cache is configured with sixty-four, 32-byte direct-mapped cache lines and Flow-Cache with eight 128-byte ones. We synthesised the design using an Altera Quartus II 5.0. Table 4 lists the resource utilisation of different parts of our scheme. The control logic of Flow-Cache is more complex than App-Cache with the result that it consumes more LEs and registers. But similar to the analysis in Section 4, the access latency of Flow-Cache is much lower. In particular, Flow-Cache can work on a frequency of 197 MHz while App-Cache only achieves 128 MHz after optimisation.

6 Related works

The method of caching recently used route lookup results, or route cache, has been used for a long time [17]. Tzi-cker Chiueh and Prashant Pradhan have proposed several strategies to reduce miss rate in a specially designed route cache [18]. In a recent work of Kaushik Rajan and R. Govindarajan, the content of caching has been changed from route results to the nodes of LC-trie (level compressed trie) that are used for lookup [19].

Sophisticated caching mechanisms have been designed for applications other than the IP route lookup. Xu *et al.* [20] have extended their work to hardware design and policy selection for the caching of packet classification results. Li *et al.* [21] used caching as an energy saving mechanism instead of latency hiding. But their applications still concentrate on holding the results for algorithms like routing, packet classification and NAT (network address translation). Patrick Crowley has proposed a segmented instruction cache for network processors to meet its real-time requirement [22].

7 Conclusions

Current solutions of NP memory architecture are often expensive and hard to program. In this article, we have presented a novel cache design for high-speed packet processing based on some key observations of the characteristics of control data locality. According to their access patterns, dedicated storage elements are employed for flow information and application data, respectively. In particular, flow data is constrained to blocks and prefetched into a subcache with a large cache line size and a specially designed data movement controller. This scheme achieves a lower miss rate than conventional cache and can greatly improve the packet throughput of network processors. To reach the same forwarding rate, the resources needed and the hardware design complexity of the proposed scheme are significantly reduced compared with previous NP architectures.

8 Acknowledgment

This work if supported by National Natural Science Foundation of China (No. 60373007, 60573121 and 60625201), the Cultivation Fund of the Key Scientific and Technical Innovation Project, Ministry of Education of China (No. 705003), the Specialised Research Fund for the Doctoral Program of Higher Education of China (No. 20040003048 and 20060003058), China/Ireland Science and Technology Collaboration Research Fund (CI-2003-02), and the Tsinghua Basic Research Foundation (JCpy2005054).

9 References

1 Peyravian, M., and Calvignac, J.: 'Fundamental architectural considerations for network processors', *Comput. netw.*, 2003, **41**, (5), pp. 587–600
2 Lawton, G.: 'Will network processor units live up to their promise?', *IEEE Computer*, 2004, **37**, (4), pp. 13–15
3 Mudigonda, J., Vin, H.M., and Yavatkar, R.: 'Managing memory access latency in packet processing'. Proc. Int. Conf. on Measurement and Modeling of Computer Systems, (SIGMETRICS 2005), Banff, Canada, June 2005, pp. 396–397
4 Liu, Z., Che, H., Zheng, K., Chen, S., Hu, C., and Liu, B.: 'A trace driven comparison of latency hiding techniques for network processors'. Proc. Int. Conf. on Communications, (ICC 2006), Istanbul, Turkey, June 2006
5 Venkatachalam, M., Chandra, P., and Yavatkar, R.: 'A highly flexible, distributed multiprocessor architecture for network processing', *Comput Netw.*, 2003, **41**, (5), pp. 563–586
6 Mudigonda, J., Vin, H.M., and Yavatkar, R.: 'Overcoming the memory wall in packet processing: hammers or ladders?'. Proc.

Symp. on Architecture for Networking and Communications Systems, Princeton, NJ, USA, October 2005, pp. 1–10
7 AMCC nP3700 product brief: http://www.amcc.com/MyAMCC/retrieveDocument/SNP/nP3700_060428.pdf, accessed September 2006
8 Intel IXP2800 Network Processor Hardware Reference Manual Intel Inc., May 2003
9 http://www.hifn.com/products/5np4g.html, accessed September 2006
10 Pnevmatikatos, D.N., Sourdis, I., and Vlachos, K.: 'An efficient, low-cost I/O subsystem for network processors', *IEEE Design & Test of Computers*, 2003, **20**, (4), pp. 56–63
11 Decasper, D., Ditta, Z., Parulkar, G., and Plattner, B.: 'Router Plugin: a software architecture for next-generation routers', *IEEE/ACM Trans. Networking*, 2000, **8**, (1), pp. 2–15
12 Luo, Y., Yang, J., Bhuyan, L.N., and Zhao, L.: 'NePSim: a network processor simulator with a power evaluation framework', *IEEE Micro.*, 2004, **24**, (5), pp. 34–44
13 National Laboratory for Applied Network Research: http://pma.nlanr.net/PMA/, accessed October 2004
14 Shivakumar, P., and Jouppi, N.P.: 'CACTI 3.0'. http://research.compaq.com/wrl/people/jouppi/CACTI.html, accessed November 2005
15 http://www.opencores.org/, accessed September 2004
16 Altera Corporation: 'Stratix Device Handbook', vol. 1, http://www.altera.com/literature/hb/stx/stratix_vol_1.pdf, accessed March 2005
17 http://www.cisco.com, accessed September 2006
18 Chiueh, T., and Pradhan, P.: 'Cache memory design for internet processors', *IEEE Micro.*, 2000, **20**, (1), pp. 28–33
19 Rajan, K., and Govindarajan, R.: 'A heterogeneously segmented cache architecture for a packet forwarding engine'. Proc. Conf. on Supercomputing, (ICS'05), Boston, MA, USA, June 2005, pp. 71–80
20 Xu, J., Singhal, M., and Degroat, J.: 'A novel cache architecture to support layer-four packet classification at memory access speeds'. Proc. IEEE INFOCOM'00, Tel-Aviv, Israel, March 2000, vol. 3, pp. 1445–1454
21 Li, B., Venkatesh, G., Calder, B., and Gupta, R.: 'Exploiting a computation reuse cache to reduce energy in network processors'. Proc. Int. Conf. on High Performance Embedded Architectures and Compilers, Barcelona, Spain, November 2005, pp. 251–265
22 Crowley, P.: 'Supporting mixed real-time workloads in multithreaded processors with segmented instruction caches in' in Crowley, P., Franklin, M.A. Hadimioglu, H., and Onufryk, P.Z. (Ed.): 'Network processor design: issues and practices' (Morgan-Kaufmann Publishers, 2005, vol. 3, 1st edn.), pp. 9–31