Field-Based Branch Prediction for Packet Processing Engines

David Bermingham, Zhen Liu, and Xiaojun Wang
School of Electronic Engineering
Dublin City University
Dublin, Ireland
{david.bermingham, liuzhen, wangx}@eeng.dcu.ie

Bin Liu
Department of Computer Science and Technology
Tsinghua University
Beijing, P. R. China
liub@tsinghua.edu.cn

***Abstract*—Network Processors have exploited all aspects of architecture design, such as employing multi-core, multi-threading and hardware accelerator, to support both the ever-increasing line rates and the higher complexity of network applications. Micro-architectural techniques like superscalar, deep pipeline and speculative execution provide an excellent method of improving performance without limiting either the scalability or flexibility, provided that the branch penalty is well controlled. However, traditional branch predictors are not as efficient in network applications as in general purpose processing, due to the fewer variations in branch patterns of packet processing. To improve the prediction accuracy, we propose a flow-based prediction mechanism which caches the branch histories of packets with similar header fields, since they normally undergo the same execution path. For packets that cannot find a matching entry in the history table, a fallback *gshare* predictor is used to provide branch direction. Simulation results show that the our scheme achieves an average hit rate in excess of 97.5% on a selected set of network applications and real-life packet traces, with a similar footprint to the traditional branch predictors used in modern microprocessors.**

Keywords-branch prediction; network processor; network traffic; packet flow

I. INTRODUCTION

As the Internet has evolved, the functions required to be implemented on Network Processor (NP) have grown from simple packet forwarding to complex tasks such as packet classification, intrusion detection/prevention, smart metering and Quality of Service (QoS). This growth in application complexity has been accompanied with the demand to catch up with the ever-increasing line rates. In order to meet the two simultaneous requirements, the design space of NPs has always been exploited from every aspect to extract more performance. Widely adopted architectural features range from integrating multiple processing engines (PEs) to exploit the packet-level parallelism, using hardware-facilitated multi-threading to hide latencies of memory and I/O accesses, to offloading sophisticated functions such as cyclic redundancy check (CRC), encryption/decryption, and pattern matching to dedicated hardware blocks. Commercial examples of these solutions include Intel IXP family of network processors [1], Hifn 5PN4G Network Processor [2], and Cavium OCTEON Processor Family [3].

On one hand, improving NP performance by the above mentioned techniques might potentially suffer from some limitations in scalability as the line rate and application complexity keep evolving. For example, additional threads and PEs complicate both the hardware and software design since it adds complexity to the memory controllers and I/O interfaces, as well as task partitioning, synchronization, and load balancing. Besides, as more and more stateful applications are deployed, the reduced parallelism among packets will limit the number of packets that can be simultaneously processed by the PEs. Solutions such as hardware accelerators provide higher performance for particular functions at the expense of lost flexibility. If these functions are not needed by certain applications, the chip area they consume is wasted.

On the other hand, PE can keep improving its own performance by means of higher clock speed and micro-architectural techniques such as superscalar, deep pipeline, out-of-order execution, and speculative execution. For example, EZchip NP series of network processors incorporate an array of superscalar processors to speed up packet processing [4]. Cavium ECONA CNS-family of processors, targeted for router related applications, utilizes an ARM core [5]. Although the one used by ECONA processors has a 5-stage pipeline, the ARM cores have evolved to a superscalar architecture with a pipeline of up to 13 stages [6]. The desire to catch up with the ever-increasing line rate and application complexity will lead to greater use of these micro-architectural techniques to promote the performance of PEs.

To guarantee the effectiveness of these techniques, the branch prediction should be accurate. Branch prediction for general purpose processing has been thoroughly investigated [8]-[17], but only a small amount of research effort has been dedicated to it in terms of network applications. Compared to technical applications such as the SPEC2000 benchmark suite, or multimedia applications such as the MediaBench benchmark suite, the processing of network applications focuses mainly on packets, under the support of information such as route lookup table, or packet classification ruleset [7]. Many network applications have a similar processing framework. For example, most header processing functions traverse a decision tree to find some corresponded information such as a next hop address or a matching rule, based on the value of header fields. Payload processing functions often check each byte of packet until the last one or

some pre-defined strings are encountered. These characteristics make the branch behavior more deterministic and highly associated with the value of some packet fields.

In our research, we found that although a small table can obtain a good branch prediction rate, it is quite difficult for traditional schemes such as *gshare* to keep increasing the accuracy by using larger tables [8]. This is caused by the more deterministic pattern of branch history of network applications, which leaves a large amount of unused entries in the history table. However, if the predictor can distinguish packets that have similar pattern of processing, which in this case means they have similar value for certain fields, the subsequent packets can simply re-use the branch history of the previous one. This paper presents a field-based branch prediction architecture which utilizes this phenomenon to provide high prediction accuracy.

Instead of keeping either the global or local branch history, the proposed branch predictor maintains the branch history for packets with similar header fields. When a new packet arrives, certain fields are extracted from the header based on the applications to be performed by the processing engines on this packet. Through an effective mapping mechanism, the corresponding branch history is fetched from the table and used to guide the branch direction. Simulation results show that the proposed scheme provides higher prediction accuracy than using a large prediction table in traditional schemes.

The remainder of this paper is organized as follows. After a brief introduction of related work, Section II motivates our research by presenting the background work that implies the inefficiency of traditional large prediction table in network applications. Section III examines the branch behavior of network workloads and provides a description of our field-based branch predictor for packet processing engines. Section IV presents the simulation result achieved with the field-based prediction architecture, while the conclusions are drawn in Section V.

II. BACKGROUND

A. Related Works

Branch prediction can be performed either statically or dynamically. Static prediction utilizes opcode information of instruction at execution time or profiling statistics at compile time. A static prediction scheme optimization for loop intensive program simply assumes forward branches not taken and backward branches (often used by loops) taken [9]. Some microprocessors encode additional information within the instruction at compile time to specify the most possible direction for this branch, which is generated by previous executions on some sample input [10]. The drawback of static prediction schemes is that it is difficult for them to be effectively adjusted to the program and data set currently being executed, which might have different behavior from those that are used to set the tone of the branch predictor.

Dynamic prediction uses run-time execution history and is widely employed in modern microprocessors [11]-[14]. One of the basic schemes is to set up a table of two-bit saturating up-down counters, called Pattern History Table (PHT), indicting if the corresponding branch should be strongly/weakly taken or not. When a branch is encountered, the methods of generating the table index for it include: using part of the instruction address as in *bimodal* [15]; using global branch history (*GAg*), local branch history (*PAg*), or a combination of global and local branch history (*PAp*) as in two-level branch prediction [16]; concatenating part of branch instruction address with global branch history as in *gselect*; or XORing them as in *gshare*, which is illustrated in Fig. 1. A combining scheme uses several different types of predictors and an additional table of counters to decide which predictor should be used to make the final decision for a branch, referred to as *gskew* [17].

Only a small number of papers mentioned the accuracy of branch prediction for network applications, with the majority of them rising from the research work of designing a benchmark suite for network applications. G. Memik and W. H. Mangione-Smith report a prediction accuracy of up to 99.79% with a two-level predictor of 2 KB and 4 KB table sizes using the benchmark of NetBench [18]. However, this result only focuses on small applications such as CRC and ignores the big variations among the tested applications. T. Wolf and M. Franklin give some analysis on the instruction distribution, including conditional branches, for another set of network programs called CommBench, without providing any further simulation results on the accuracy of branch prediction [19]. This is similar with the research work around another network processor benchmark suite called NpBench, represented by B. K. Lee and L. K. John in [20].

B. Packet Traces and Network Applications

As mentioned before, the processing objects of network applications are packets. Throughout this paper, the packet traces used in the experiments are collected from *National Laboratory for Applied Network Research (NLANR)* [21]. The source and destination IP addresses in the packet traces published by NLANR are renumbered to maintain anonymity. This process retains traffic patterns and flow information; but the renumbered IP addresses cannot match the real-life route table or ruleset. To solve this problem, the destination IP addresses used in packet forwarding programs are replaced with addresses derived from the prefixes found in the *AT&T East* routing table. A similar process is applied to the source IP address for packet classification. Another problem with the packet traces is that they only contain packet headers while applications like encryption need to

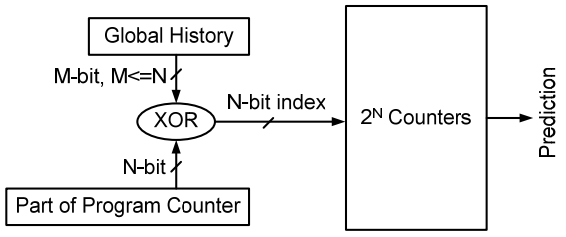

Figure 1. Schematic diagram of *gshare* branch predictor. For a pattern history table of 2^N saturation counters, the N-bit index is generated by XORing a M-bit global branch history where M<=N, and N-bit branch instruction address whose left most and right most bits are usually not used.

scan the payload data. Hence, randomly generated content are attached to packet headers according to the packet length field in the IP header.

Network applications can be briefly divided into Header Processing Applications (HPA) and Payload Processing Applications (PPA). Fig. 2 outlines the typical programming framework for HPA and PPA. Fig. 2(a) is the pseudo code for IP forwarding. In this example, a packet pointer *pkt_ptr* is passed to the function *packet_forward()* for processing. Once the whole IP packet header *iphdr* is fetched and verified, the next hop address is determined based on the destination IP address. If the header or the next hop is not valid, the packet is dropped. Otherwise, it is modified (e.g. decrementing the Time-To-Live field of IP packet header) and forwarded. Fig. 2(b) shows the pseudo code for packet encapsulation and encryption function used for network security. Once the packet is fetched, the header and payload are encrypted, which is typically performed on fixed-length blocks. As shown in Fig. 2(b), function *encrypt_packet()* encrypts the next *block_size* number of bytes starting from *pkt_ptr* and attaches the result to *enc_pkt*. The while-loop continues until every byte of the packet has been fetched and encrypted. After that, the encrypted packet is encapsulated with a new IP packet header for transmission.

In this research, we select 10 applications from the three benchmark suites for network applications, NetBench, CommBench, and NpBench; six of them represent header processing tasks and the remaining four represent payload application.

For HPA, we analyze two IP packet forwarding applications, one is denoted as *TRIE*, which uses level compressed trie (LC-trie) to perform route lookup, and the other is denoted as *HASH*, which uses a link-list hash structure. Both applications perform header and IP address verification, checksum calculation and header modification before forwarding to the next hop port returned by route lookup. The application denoted as *HYPE* implements packet classification using *HyperCuts* algorithm, which recursively divides the hyperspace represented by the packet fields into smaller hyperboxes until the number of rules contained in them is smaller than a predefined threshold. *STAT* performs statistical analysis, mapping packets to entries maintained in a flow table, where statistics are retrieved and updated. The remaining two applications implement the functions of queuing and metering. *DRR* provides load balance for packets using the algorithm of deficit round robin. *TCM*, short for two-rate three color marker, classifies packets into three categories to smooth out bursty traffic, based on characteristics such as arrival rate.

Among the payload applications, *AES* and *SHA* authenticate and encrypt IP packets, as is demanded by the implementation of Internet Protocol Security (IPsec). *CRC* calculates the checksum of the whole packet using the algorithm of cyclic redundancy check in order to detect errors during packet transmission. *FRAG* implements IP fragmentation specified in RFC 791, in which packets longer than a fragment limit and allowed to be fragmented are divided into smaller ones.

```
packet_forward (char* pkt_ptr)
{
   struct ip iphdr;
   int next_hop;

   iphdr = fetch_header(pkt_ptr);
   if(verify_header(iphdr) == TRUE) {
      next_hop = find_next_hop(iphdr.dst_address);
      if(next_hop == PROBLEM)
         drop(pkt_ptr);
      else {
         modify(iphdr);
         forward(pkt_ptr);
      }
   } else
      drop(pkt_ptr);
}
```

(a) Pseudo code for IP forwarding.

```
packet_encapsulate(char* pkt_ptr)
{
   struct ip iphdr, new_iphdr;
   char* enc_pkt;

   iphdr = fetch_header(pkt_ptr);
   while(iphdr.ip_len) {
      encrypt_packet(pkt_ptr, enc_pkt, block_size);
      iphdr.ip_len -= block_size;
      pkt_ptr += block_size;
   }

   new_iphdr = new_header(iphdr, enc_pkt);
   pkt_ptr = encap_packet(new_iphdr, enc_pkt);
   transmit(pkt_ptr);
}
```

(b) Pseudo code for packet encryption and encapsulation.

Figure 2. Typical programming framework for HPA and PPA.

Fig. 3 shows the number of instructions of the object codes and the percentage of conditional branches. These programs are compiled using *gcc-3.4.3* targeted to ARM microprocessors. As is pointed out in [19], the size of the network applications is much smaller than general purpose programs such as those in SPEC2000. The number of branch instructions ranges from less than 10 to more than 50, which makes the implementation of per-address history table relatively easy.

Note that an unusual feature of ARM instruction set is the employment of branch predication, by which almost every instruction can be conditionally executed. The top four bits of each instruction is the condition code field that represents fifteen conditions, e.g. greater than (GT), always (AL), or not equal (NE). The value of this field causes the instruction to be executed or skipped, depending on the values of the flag bits in status register [6]. The main purpose of this method is to eliminate conditional branches over small code segments, increasing the effectiveness of pipeline and instruction cache. Therefore, the number of conditional branch instructions found in ARM object code is normally less than that of MIPS or Alpha code generated from the same source program.

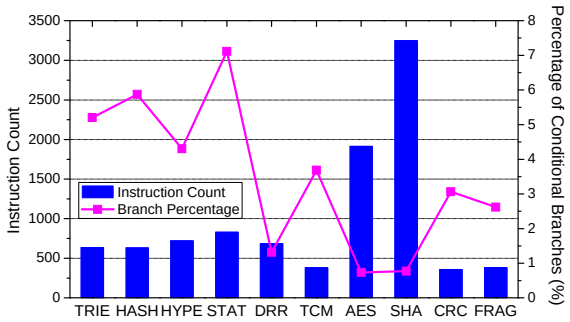

Figure 3. Number of instructions in the compiled code and the percentage of conditional branches.

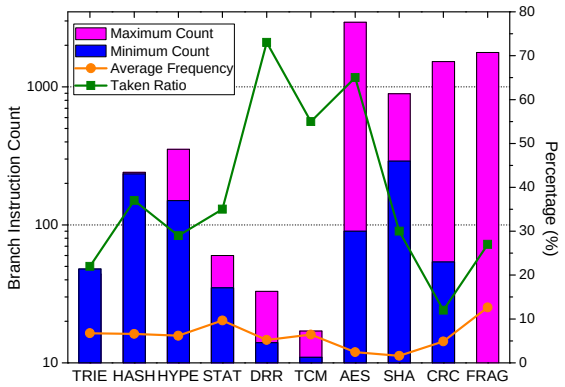

Figure 4. Maximum and minimum number of branches executed for a packet, its average execution frequency and taken rate for an OC-3 trace.

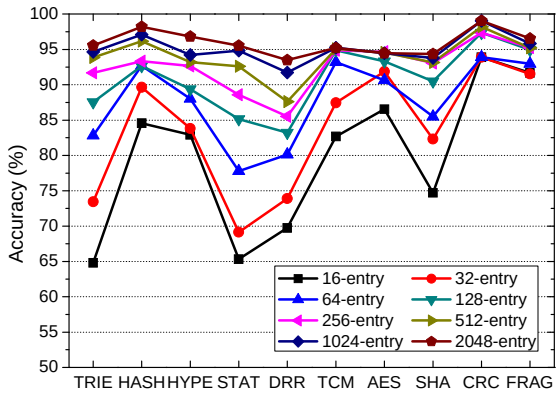

Figure 5. Accuracy of a *gshare* branch predictor for an OC-3 packet trace.

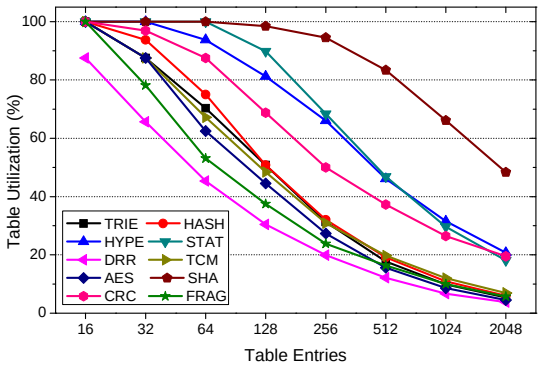

Figure 6. Table utilization of a *gshare* branch predictor for an OC-3 trace.

Fig. 4 shows the maximum and minimum number of conditional branches executed for a packet, its average execution frequency and taken rate for an OC-3 packet trace. Compared with the simulation result in [19], the execution frequency of conditional branches is slightly lower, ranging from 1.59% to 12.67%. For HPA, the variations in the number of executed branches are much smaller than PPA. Taking *TRIE* as an example, the length of path along which packets with different destination IP addresses traverse are almost the same in a well-balanced trie. On the other hand, the number of branches executed for PPA heavily depends on packet length. The taken rate of branches also varies greatly for different applications.

C. *Problems with Branch Prediction for Network Applications*

In this subsection, we briefly analyze the problem faced by traditional branch prediction scheme for network applications, using *gshare* as an example.

Fig. 5 shows the accuracy of a *gshare* branch predictor with different number of entries in PHT for the selected network applications and an OC-3 packet trace. According to Fig. 5, the accuracy for PPA is higher than HPA. For example, a 64-entry *gshare* predictor achieves a hit rate of 91.9% and 91.92% for *AES* and *CRC* but only 85.36% and 75.5% for the *TRIE* and *STAT* applications. This is caused by the relatively complicated pattern of branch history in HPA. On the other hand, greater performance gains are achieved for HPA as the table size is increased. For a small 64-entry table, the average accuracy for HPA is 85.42% while for PPA it is 87.81%. For 1024-entry, the difference in accuracy is negligible, with an average hit rates of 95.1% and 95.2% for HPA and PPA respectively.

Generally speaking, the relationship between prediction rate and PHT size follows a similar pattern to general purpose processing. Since the code size of network application is normally much smaller than general purpose processing applications in SPEC2000, similar accuracy can be achieved with fewer entries in PHT. However, it is hard to keep increasing the accuracy when the number of PHT entries reaches certain level. As shown in Fig. 5, network applications saturate above 512-entry, compared to the 8K-entry table size needed for SPEC benchmark applications [13]. Above 1024-entry it can seen that the performance gain is minimal compared to the additional cost.

Fig. 6 shows the average table utilization of a *gshare* branch predictor for the OC-3 packet trace across all 10 applications. For small table sizes, high miss-prediction rate is the result of large amount of branch interference, with multiple branches mapping to the same location. However, when additional counters are added, the utilization falls significantly compared to the slight increase in prediction accuracy. For example, a 32-entry *gshare* predictor utilizes an average of 83.44% of the available entries for the 10 network applications, while a 512-entry and a 2048-entry table utilizes only 23.34% and 6.89% respectively. This phenomenon demonstrates that although the processing of one packet involves a large number of conditional branches, the branch history does not have a lot of patterns. For most

TABLE II. AVERAGE ACCURACY OF 64-ENTRY *GSHARE* PREDICTOR FOR PPA AND DIFFERENT TRACES (%)

App.	OC-3	OC-12	OC-48
AES	91.99	96.41	96.34
SHA	80.51	86.45	85.57
CRC	91.92	96.61	96.55

TABLE III. ACCURACY OF 64-ENTRY *GSHARE* PREDICTOR FOR PPA AND PACKETS OF DIFFERENT LENGTH (%)

App.	Min. Length	Max. Length
AES	91.74	96.57
SHA	81.50	88.58
CRC	90.50	99.00

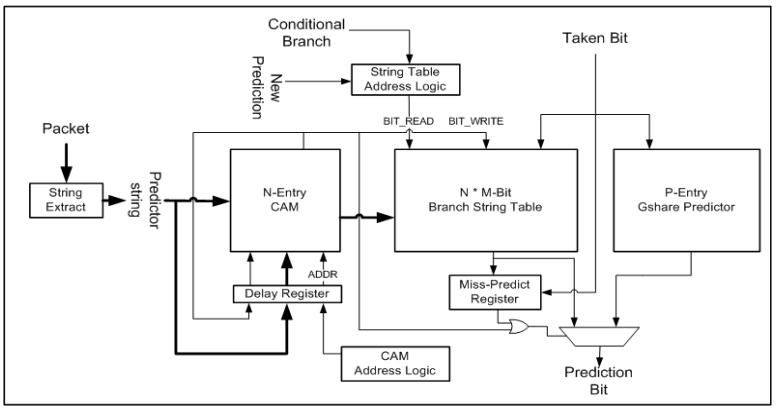

Figure 7. Block diagram of field-based branch predictor for network applications.

network processors, an application will reside on one processing engine long periods of time. A solution in which more than 90% of pattern table is idle represents a significant waste of chip area and energy.

III. FIELD-BASED BRANCH PREDICTION

As mentioned in Section II.B, many network applications follow similar programming patterns. For example, header processing applications might need to verify the correctness of the transmission and whether the values of certain fields are valid. The rest of the processing is often based on the value of one or a combination of fields. Since most packets do not contain errors, the code dealing with exceptions is hardly executed, making the branch history more predictable and having fewer patterns.

Furthermore, packets with similar field value often have the same execution path. For example, packets whose destination IP addresses belonging to the same prefix in the route lookup table follow the same path in the trie. Packets of the same flow fall into the same hyperbox defined by a ruleset and undergo the same subsequent processing according to the associated actions. Similar situations can also be found in PPA. The most obvious examples include encryption and fragmentation, in which the conditional branches are used mainly for testing the length of remaining packets.

On the other hand, the relatively short branch history of HPA makes it easier to cache the entire branch history for the processing of one packet. For example, an average of 44 conditional branches in *TRIE* is encountered for processing one packet, which is a short enough branch history to be stored in PE. This is similar with PPA when small packets are processed.

For payload applications whose branch history for large packet is too long to be completely cached, a fallback *gshare* predictor can be exploited. The reason for doing this is that the prediction performance of schemes such as *gshare* depends on the length of branch history, i.e. the accuracy of *gshare* can be increased when more training is provided. Since the number of branch instructions executed in payload applications scales with packet length, *gshare* exhibits better performance for large packets due to the additional branch history. Table I shows the average accuracy of a 64-entry *gshare* predictor for three payload applications and traces of different link speed. The OC-12 and OC-48 traces contain a higher proportion of large packets, which results in a higher prediction rate than the OC-3 trace. Table II shows the average accuracy of a 64-entry *gshare* predictor for packets of maximum and minimum length in an OC-3 trace, which further demonstrates that schemes like *gshare* are more suitable for PPA with large packets. Since *gshare* is likely to generate incorrect prediction at the beginning of packet processing, the cached branch history of previous packets with similar field value can be used at this stage; once the cached history is consumed, the fallback *gshare* predictor has been well trained for subsequent prediction.

TABLE I. SEARCH KEY EXTRACTED FROM PACKET HEADER

App.	Search Key
TRIE	Source IP + Destination IP
HASH	Source IP + Destination IP
HYPE	Source IP + Destination IP + Protocol
STAT	Source IP + Destination IP + Protocol
DRR	Length
TCM	Length
AES	Length
SHA	Length
CRC	Length
FRAG	Length + Offset

A. Basic Architecture of Field-Based Branch Predictor

Fig. 7 gives the block diagram of a field-based branch predictor for network applications. The major part of the predictor is Branch String Table (BST), which caches the branch history for packets that are categorized by the value of certain header fields. A fallback *gshare* predictor is used either when packet with a new value for certain fields is received, or the cached history has been fully utilized. Unlike a *gskew* predictor where an additional table of counters is maintained for choosing the branch predictor to be used, the current status of BST alone determines whether branch should be taken or not.

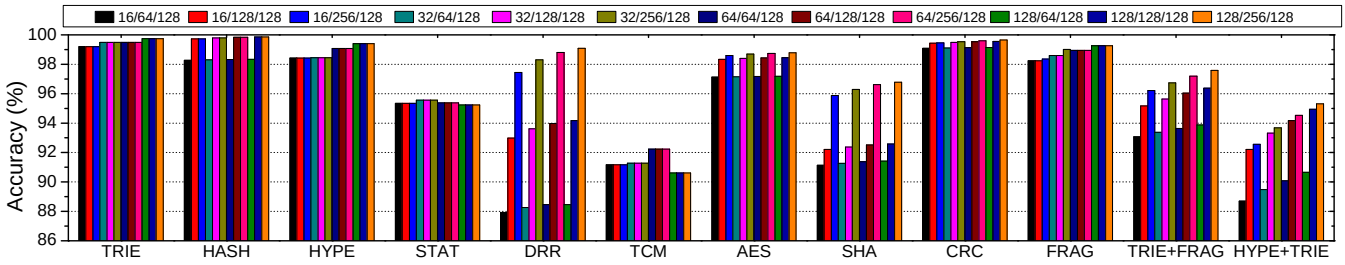

Figure 8. Branch prediction accuracy for field-based predictors with a fallback 128-entry *gshare* predictor.

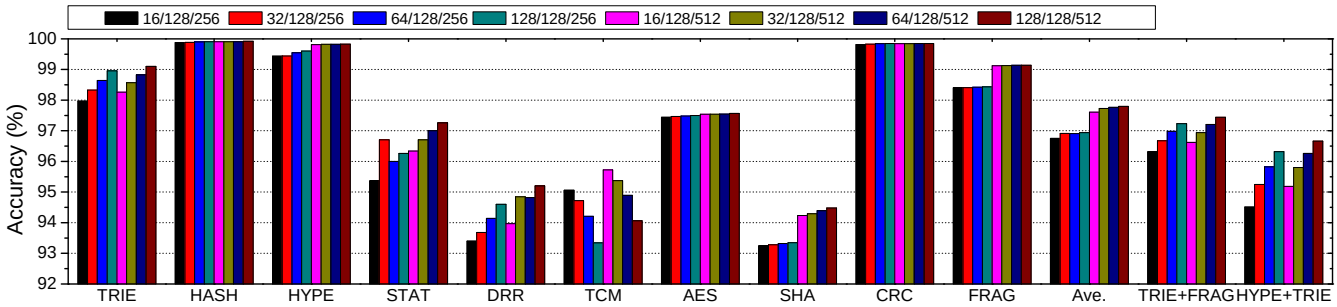

Figure 9. Branch prediction accuracy for field-based predictors with fallback *gshare* predictor of changing number of entries.

When a packet is received by the network processor, the string extract block extracts the relevant header fields and combines them into a search key defined by the network application. The search key can be part of one particular header field or a concatenation of several parts from a number of header fields. Table III gives the list of header fields used to generate search keys in our experiments for the 10 applications. In practice, network workloads consist of multiple types of applications which have different branch patterns. In this case, the search key can be constructed as a combination of the header fields corresponding to each application.

The generated search key, or predictor string as labeled in Fig. 7, is used to identify the entry to be used for branch prediction in the BST. This mapping relationship can be implemented a number of methods. In this experiment, a full-associative BST is achieved using Content-Addressable Memory (CAM) [22]. CAM should have the same number of entries as BST. In our experiment, the data width of CAM is set to 32-bit. As shown in Table III, if the length of the search key is less than 32-bit, part of the data bits in CAM entry will not be used. Otherwise, part of bits in the selected header fields will be ignored in order to fit the search key into 32-bits.

Using the search key, the CAM logic returns either the address of corresponding branch history if previous packets also have the same key, or allocates a new entry in both the CAM and BST if the key fails to match anything. If all of the CAM entries have been used, the replaced entry is selected in a round-robin way for the simplicity of hardware implementation. For a matched key, the next bit in the branch history is fetched from the BST every time a conditional branch is encountered. In the case of the number of conditional branches exceeding the history length, the fallback *gshare* predictor is used, which has been updated for all of the previous branches. For packets without matching entry in CAM, prediction is also provided via the *gshare* logic, with the BST being updated when the branch direction is resolved.

A field-based branch predictor can be expanded on a number of dimensions, which can be denoted as *N/M/P*. The number of CAM/BST entries *N* can be increased to allow more patterns of branch history to be cached simultaneously and reduce the chances that the fallback *gshare* predictor should be resorted to. Additional branch history for each packet can be retained by extending the branch string width *M*, while the table size *P* of fallback predictor can be expanded to give a better base prediction for packets not previously seen.

B. Variations of Field-Based Branch Predictor

Unlike traditional branch predictors, the generation of BST index does not depends on branch address in our scheme, which means it is not necessary to access BST after the address of the branch is known. In fact, when all the fields needed for predictor string are received, the CAM and BST search can be issued even before the actual packet processing begins. Therefore an *M*-bit shift register can be used to hold the branch history of BST indexed by the matching entry in CAM. Whenever a branch is encountered, the next bit in the register can be used, without accessing the CAM and BST again. Similarly, if no matching entry is found in CAM, the branch history of this packet can be first stored in the register and written back from this register to BST at the end of packet processing or when the register is full.

Other possible variations include using set-associative cache instead of CAM, or calculating the accuracy of BST and fallback *gshare* to determine which one should be used for the next branch. Note that even if a cache is used, our scheme is still quite different from YAGS branch predictor,

TABLE V. ACCURACY OF *GSHARE* PREDICTOR FOR COMBINATION OF APPLICATIONS (%)

App.	32	64	128	256	512	1024	2048
TRIE+FRAG	81.61	88.68	89.71	91.92	92.59	92.44	92.78
HYPE+TRIE	73.67	78.26	81.90	87.67	89.55	91.10	93.03

where the branch addresses play an important role in table index generation and cache content comparison [23].

IV. PERFORMANCE EVALUATION

A. Accuracy of Field-Based Branch Predictor

Fig. 8 shows the accuracy of various field-based branch predictors with a 128-entry fallback *gshare* predictor for an OC-3 packet trace. It can be seen that a 16/128/128 scheme increases the hit rate of a single 128-entry *gshare* predictor from 90.76% to 96.60% across all 10 applications, which is higher than the average value of 95.83% for a 2048-entry *gshare* predictor. For applications that are highly relevant to certain header fields, such as *TRIE* or *HYPE*, the prediction accuracy gain is greater, with an improvement of 13.80% and 9.44% respectively. The only application which shows no significant gain is the metering algorithm *TCM*, whose branch behavior is difficult to predict since it is related to the packet inter-arrival time rather than header fields.

It can be seen in Fig. 8 that for most applications, increasing the number of CAM entries provides only marginal performance gain. This is because the amount of packets collected by the OC-3 link is small, which limits the number of branch patterns. On the other hand, increasing the branch history length is much more effective, especially for applications such as *DDR* or *SHA*, in which doubling the 128-bit history to 256-bit increases the hit rate by 4.95% and 4.35% respectively.

Fig. 8 also includes the accuracy for the combinations of several applications in order to simulate the situation of real-life network software. The first combination is *TRIE* and *FRAG*, which fragments the packet after route lookup. The second is *HYPE* and *TRIE*, which classifies packets into flows before forwarding them. Table IV lists the accuracy of *gshare* predictors for these application combinations. Compared with single application, both *gshare* and field-based predictor exhibit performance degradation but in most configurations, our scheme continues to outperform the 2048-entry *gshare* predictor.

Fig. 9 shows the accuracy of field-based predictor as the size of fallback *gshare* predictor is increased. Since for most applications the utilization of predictor above this size is less than 50%, we limit our analysis to *ghsare* predictors with no more than 512 entries. It can seen that a field-based 16/128/256 predictor achieves an average hit rate of 96.77% across all 10 applications, with a minimum accuracy of 93.24% for *SHA*. Except for *SHA* and *FRAG*, increasing the number of CAM entries has a better performance gain than using a larger fallback *gshare*. This can be explained by Table V, which shows the percentage of packets whose field value has already existed in CAM. Even when the CAM size is small, less than 15% packets have to resort the fallback

TABLE IV. UTILIZATION OF FIELD-BASED PREDICTOR FOR NEW PACKET (%)

App.	*N*=16	*N*=32	*N*=64	*N*=128
TRIE	92.526	91.047	96.607	97.473
HASH	86.233	89.380	96.607	97.473
HYPE	86.233	89.380	93.773	95.327
DRR	86.067	91.047	92.433	95.040
TCM	86.067	95.220	92.433	95.040
STAT	86.233	95.220	93.773	95.327
AES	86.067	89.380	92.433	95.040
SHA	86.067	89.380	92.433	95.040
CRC	86.067	89.380	92.433	95.040

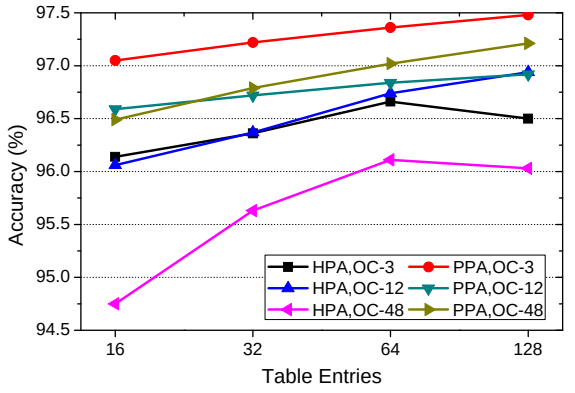

Figure 10. Average accuracy of field-based predictor for different traces.

gshare when they are allocated to this PE. The exception of *SHA* and *FRAG* in Fig. 9 is caused by their large branch history. In this case, a larger *gshare* provides better prediction when the conditional branches fall out of the range that can be cached by field-based predictor.

Fig. 10 shows the average accuracy of field-based predictor for different packet traces. The branch history length is 128-bit and fallback *gshare* also has 128 entries. For OC-12 trace, a 64-entry and 128-entry CAM provides an average prediction rate of 96.66% and 96.93% respectively, higher than 96.26% of a 2048-entry *gshare*. For higher speed OC-48 trace, a 64-entry and 128-entry CAM provides an average prediction rate of 96.56%, and 96.62% respectively, still higher than the 96.24% of 2048-entry *gshare*. Link speed has more impact on header applications because higher speed links accumulate more packets with a larger number of different field values. Since this OC-12 trace contains a higher proportion of large IP packets, the performance gain across the payload applications is even lower than that of OC-48.

B. Latency of Field-Based Branch Predictor

For the basic architecture of field-based branch predictor, two memory accesses are needed to get the result from BST (note that *gshare* can be accessed simultaneously with CAM and BST). In the first variation described in Section III.B, as long as the entry of BST is returned before the first branch is encountered, the latency is that of the register, which is negligible compared to the speed of PE. The reduced number

of memory accesses in this variation consumes less energy as well as making the scheduling of predictor updates easier.

C. Chip Area of Field-Based Branch Predictor

Since most network processors have multiple processing engines integrated into one chip, which makes them sensitive to silicon area. The transistor cost of a field-based predictor includes the CAM, BST and the fallback predictor. The latter two can be implemented using 6-transistor SRAM while each CAM cell normally needs 9 transistors. Ignoring the decoder and other control logic, the transistor cost of an *N/M/P* field-based predictor can be approximated as requiring $(9\times N\times 32)+(6\times N\times M)+(6\times 2\times P)$ transistors. For a 16/128/128 scheme, the silicon area is roughly equivalent to a 1536-entry *gshare* table, much smaller than the prediction tables used in modern general purpose processors.

V. Conclusions

In this paper we have examined branch prediction for network processors. Although network processors have utilized techniques such as multiple PEs, multi-threading and hardware accelerator to meet the requirements of high speed and programming flexibility, improving the performance of each PE is always important. Micro-architectural techniques such as superscalar and deep pipeline remain effective once the branch penalty is mitigated.

However, traditional branch schemes are not as efficient in network applications as in general purpose processing. Increasing the PHT size only slightly improves the accuracy with a large percentage of table entries remaining idle. On the other hand, the similarity found in the programming framework associated with network applications allows for novel methods of exploiting runtime execution history.

One of the key observations that inspire our scheme is that packets with similar header fields normally follow the same execution path. To improve prediction accuracy, we propose a field-based predictor which maps the incoming packet to a table of cached branch history of previous executions, based on selected header fields according to the characteristics of the application. Simulation on a set of 10 network applications and real-life packet traces shows that such a prediction scheme can achieve an average prediction rate of over 97.5% with reasonable chip area consumption.

Acknowledgment

This work is funded by the Irish Research Council for Science, Engineering and Technology (IRCSET) and the School of Electronic Engineering, Dublin City University.

References

[1] Intel Corporation, Intel IXP1200 Hardware Reference Manual. Santa Clara, CA, USA, 2001.

[2] Hifn Inc., Wire Speed Performance for Demanding Network Applications: Hifn 5NP4G Network Processor (Product Brief). Los Gatos, CA, USA, 2008.

[3] M. Raghib Hussain, "Octeon Multi-Core Processor," presented at the 2006 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2006), Dec. 2006.

[4] EZchip Technologies Ltd., Network Processor Designs for Next-Generation Networking Equipment (White Paper), Yokneam, Israel, 1999.

[5] Cavium Networks Inc., ECONA CNS21XX Connected Home and Office Processors (Product Brief), Mountain View, CA, USA, 2008.

[6] S. Furber, ARM System-on-Chip Architecture. London, Great Britain: Addion-Wesley, 2000.

[7] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," Proc. International Symposium on Microarchitecture (MICRO-30), pp. 330-335, Dec. 1997.

[8] S. McFarling, "Combining branch predictors," technical report TN-36, June 1993, ftp://gatekeeper.dec.com/pub/DEC/WRL/research-reports/WRL-TN-36.pdf.

[9] J. E. Smith, "A study of branch prediction strategies," Proc. International Symposium on Computer Architecture (ISCA 1981), May 1981, pp. 135–148.

[10] C. Young and M. D. Smith, "Static correlated branch prediction," ACM Trans. Program Languages and Systems, vol. 21, no. 5, pp. 1028–1075, Sep. 1999.

[11] R. E. Kessler, "The Alpha 21264 microprocessor," IEEE Micro, vol. 19, no. 2, pp. 24–36, Mar. 1999.

[12] T. Ball and J. R. Larus, "Branch prediction for free," Proc. ACM Conference on Programming Language Design and Implementation (SIGPLAN 1993), June 1993, pp. 300–313.

[13] P. Chang, M. Evers, and Y. N. Patt, "Improving branch prediction accuracy by reducing pattern history table interference," International Journal of Parallel Programming, vol. 25, issue 5, pp. 339–362, Oct. 1997.

[14] J. A. Fisher and S. M. Freudenberger, "Predicting conditional branch directions from previous runs of a program," ACM SIGPLAN Notices, vol. 27, issue 9, Sep. 1992, pp. 85–95.

[15] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge, "The bi-mode branch predictor," Proc. International Symposium on Microarchitecture (MICRO-30), Dec. 1997, pp. 4–13.

[16] T. Yeh and Y. N. Patt, "Alternative implementation of two-level adaptive branch prediction," ACM SIGARCH Computer Architecture News, vol. 20, issue 2, pp. 124–134, May 1992.

[17] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides, "Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor," Proc. Annual International Symposium on Computer Architecture (ISCA 2002), May 2002, pp. 295–306.

[18] G. Memik and W. H. Mangione-Smith, "Evaluating network processors using NetBench," ACM Trans. Embedded Computing Systems, vol. 5, issue 2, pp. 453–471, May 2006.

[19] T. Wolf and M. Franklin, "CommBench-a telecommunications benchmark for network processors," Proc. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2000), Apr. 2000, pp. 154–162.

[20] B. Lee and L. K. John, "NpBench: a benchmark suite for control plane and data plane applications for network processors," Proc. International Conference on Computer Design (ICCD 2003), Oct. 2003, pp. 226–233.

[21] National Laboratory for Applied Network Research (NLANR), Passive Measurement Analysis (PMA). [Online]. Available: http://pma.nlnar.org/.

[22] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," IEEE Journal of Solid-State Circuits, vol. 41, no. 3, pp. 712–727, Mar. 2006.

[23] A. N. Eden and T. Mudge, "The YAGS branch prediction scheme," Proc. International Symposium on Microarchitecture (MICRO-31), Dec. 1998, pp. 69–77.