Cache Memory Design for Network Processors

Tzi-Cker Chiueh Prashant Pradhan
Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
{chiueh, prashant }@cs.sunysb.edu

Abstract

The exponential growth in Internet traffic has motivated the development of a new breed of microprocessors called Network Processors, *which are designed to address the performance problem resulting from exploding Internet traffic. The development efforts of these network processors concentrate almost exclusively on streamlining their data-paths to speed up network packet processing, which mainly constitute routing and data movement. Rather than blindly pushing the performance of packet processing hardware, an alternative approach is to avoid repeated computation by applying the time-tested architectural idea of* caching *to network packet processing. Because the data streams presented to network processors and general-purpose CPUs exhibit different characteristics, detailed cache design tradeoffs for the two also differ considerably. This research focuses on cache memory design specifically for network processors. Using a trace-driven simulation methodology, we evaluate a series of three progressively more aggressive routing-table cache designs. Our simulation results demonstrate that the incorporation of hardware caches into network processors, when combined with efficient caching algorithms, can significantly improve the overall packet forwarding performance due to a sufficiently high degree of temporal locality in the network packet streams. Moreover, different cache designs can result in up to a factor of* 5 *difference in the average routing table lookup time and thus the packet forwarding rate.*

1. Introduction

With the enormous momentum behind Internet-related technologies and applications, demands for data network bandwidth are on the rise at an astounding rate. As a result, a growing number of microchips are being designed and fabricated specifically for networking devices rather than for traditional computing applications. In particular, a new breed of microprocessors called *Network Processors* have emerged that are designed specifically to efficiently execute network protocols on various kinds of network devices, like switches and routers.

A major function that network processors perform is packet routing. At the IP [1] level, the routing table lookup problem is equivalent to *longest prefix matching.* The routing table consists of 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 can logically be thought of as follows. The network mask of an entry selects N 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. Note that the routing table is essentially a set of destination address prefixes, and routing lookup searches for the longest matching prefix of a destination address in this set. In a classical addressing scheme, N could only take a fixed set of values, viz 8, 16 or 24. However, to allow more efficient address space allocation, a technique called *Classless Inter-Domain Routing (CIDR)* is currently in use that allows N to take any value from 1 to 31. This generality complicates the search for the longest matching prefix of a given destination address.

Efficient algorithms to solve this problem have been proposed [5, 22]. However, the architecture-level research question is how to execute them at wire speed. For example, suppose the router's performance target is 10 million packets per second, the per-packet processing, including longest prefix match, should be completed within 100 nsec. While many attempts have been made to build specialized hardware for clever packet routing and filtering algorithms, in this work we chose a time-tested architectural idea, viz caching, to attack this problem, based on the be-

[1] Unless explicitly indicated otherwise, the term "IP" refers to Internet Protocol Version 4.

lief that there is sufficient locality in the packet stream for reusing results of routing computation.

Caching alone is not sufficient due to less locality in packet address streams compared to the instruction/data reference streams in program execution. Given caches of a fixed configuration, the only way to improve the cache performance is to increase their *effective coverage* of the IP address space, i.e., each cache entry covering a larger portion of the IP address space. Towards this end, this work develops a novel *address range merging* technique by exploiting the fact that *there is a limited number of outcomes for routing table lookup (the number of output interfaces in a network device) regardless of the size of the IP address space.* Our simulation results demonstrate that address range merging improves the caching efficiency by a factor of 5 over generic IP host address caching, in terms of average routing table lookup time.

The rest of this paper is organized as follows. In Section 2, we review previous work related to network processors. In Section 3, we describe the network packet traces used and the architectural models assumed in this study. Section 4 presents the results for the baseline cache design, which supports routing table lookup based on individual destination host addresses. Section 5 presents the performance results of caching host address ranges rather than individual destination host addresses. Section 6 presents the simulation results of a further performance optimization technique that exploits the fact that the number of outcomes of routing table lookup is typically small. Section 7 concludes this paper with a summary of the main research results and a brief outline of on-going work in this project.

2. Related Work

State-of-the-art Internet routing devices use general-purpose CPUs or ASICs for routing. BBN's MGR router [14] uses a DEC Alpha 21164 processor as the routing engine and relies on the on-chip L1 and L2 cache for software-based routing table cache. IBM's Integrated Switch Router [1] uses PowerPC 603e for both control engines and forwarding engines on the interface cards. Some IP routers are rooted in massively parallel architectures using general-purpose CPU [18] or ASIC-based routing engines [12] [17] on each node. SUNY at Stony Brook's Suez router project [15] is based on a cluster architecture that uses a 400-MHz Pentium on each node for both packet routing and real-time packet scheduling.

Most high-performance switches and routers implement proprietary routing/filtering algorithms in ASIC chips [20] [9] [11]. Some chipsets [10] can perform lookups based on multiple packet fields in parallel. CAMs are exploited in certain routing co-processors [16] to achieve high performance. Certain protocol processors [4] are based on a distributed pipeline processing architecture and employ a traffic classifier module for packet routing.

None of the custom-designed processors explicitly mention the use of cache in the chip descriptions, although some of them do mention [19] that the traffic as seen in major Internet backbone routers is not expected to exhibit sufficient locality to justify the use of caches. Feldmeier [7] studied the management policy for the routing-table cache, and showed that the routing-table lookup time can be reduced by up to 65%. Chen [2] investigated the validity and effectiveness of caching for routing-table lookup in multimedia environments. Estrin and Mitzel [6] derived the storage requirements for maintaining state and lookup information on the routers, and showed that locality exists by performing trace-driven simulations. Jain [8] studied the cache replacement algorithms for different types of traffic (interactive vs. non-interactive). More recently, results from Internet traffic studies [13] as well as our own [3] showed that there is significant locality in the packet stream that caching could be a simple and powerful technique to address the per-packet processing overhead in routers. We show that by increasing every cache entry's *coverage* of the IP address space, cache performance can be significantly improved, and the effects of reduced locality can be mitigated.

Pink et al [23] proposed a technique to compress an expanded trie representation of a routing table, so that the result is small enough to fit in the L2 cache of a general-purpose processor, thus reducing lookup time. Our approach is that of designing efficient ways of caching route lookup results by exploiting the structure in a given routing table, rather than that of speeding up route lookup by ensuring that memory accesses involved in the lookup mechanism are cache hits.

3. Performance Evaluation Methodology

3.1. Trace Collection

We use a trace-driven simulation methodology to study the design of network processors' cache memory systems. Although there were several IP packets traces available in the public domain, none of them are suitable for our purposes either because they were out of date, or they were "sanitized" by replacing IP addresses with unique integers, rendering the trace unfit for set-associative cache simulations.

As a result, we decided to collect a packet trace from the periphery link that connects the Brookhaven National Laboratory (BNL) to the Internet via ESnet at T3 rate, i.e., 45 Mbits/sec. This link is the only link that connects the entire BNL community to the outside world. The trace has been collected by setting up a Pentium-II/233 MHz machine to snoop on a mirror port of a Fast Ethernet switch that links

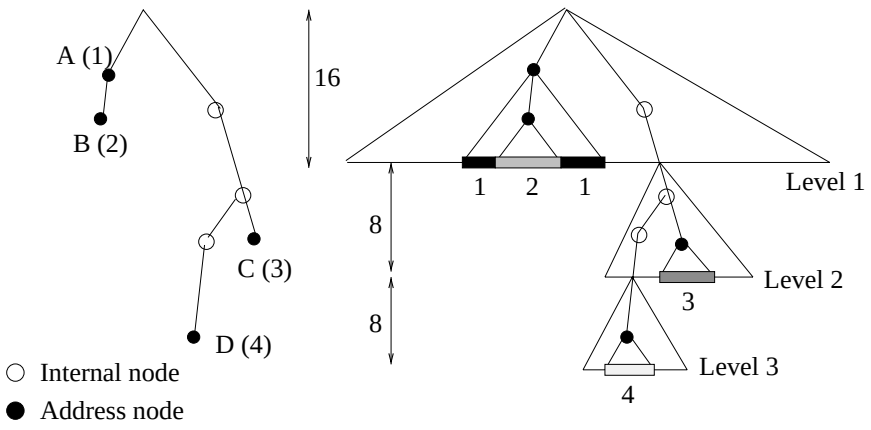

Figure 1. *The network address trie and its corresponding NART data structure. Addresses A and B belong in the level-1 table and have output ports 1 and 2 respectively. Address C falls in one of the level-2 tables and has output port 3. Address D lies in a level-3 table, with output port 4. Note that due to the longest prefix match requirement, part of address A's range in the level-1 table is "culled away" by address B's range.*

BNL's periphery router with ESnet's router. The packet trace was collected from 9AM on 3/3/98 to 5PM on 3/6/98. The total number of packets in the trace is 184,400,259. Because there are only three output interfaces in the BNL router, we used a backbone router's routing table from the IPMA project [21] in the simulations.

Recognizing the fact that the BNL network is at the edge of the Internet and thus the collected trace may not reflect the traffic patterns on backbone routers, we multiplexed portions of the original trace to create an aggregated packet stream that emulates the effect of interleaving the traffic from a large number of unrelated traffic flows. Specifically, we extracted from the collected trace 100 30-minute packet sequences that are temporally spaced as far apart as possible, and interleaved them to form a new amalgamated trace. Essentially we are simulating spatial traffic (un-)correlation using temporal traffic (un-)correlation. To further mitigate the performance skews due to possibly higher traffic locality in the collected trace, we focus our simulation study mostly on caches that are smaller than what is feasible in current processors.

3.2. Architectural Assumptions

In the following sections, we will explore three network processor cache designs and their detailed architectural tradeoffs using trace-driven simulations. The first design is a generic CPU cache for routing-table lookup, where the destination host address is treated as a memory address. The second design is an improvement over the first by exploiting the fact that each routing table entry corresponds to a contiguous range of the IP address space. Therefore, instead of caching individual destination host addresses, the network processor cache can cover a larger portion of the IP address space if each cache entry corresponds to a host address range.

The third design represents a further performance optimization over the second by exploiting the fact that the number of distinct outcomes of routing-table lookup is equal to the number of output interfaces in a router and is thus relatively small. As a result, one could "combine" disjoint host address ranges that share the same routing-table lookup result into larger address ranges, by choosing a different hash function than that used in generic CPU caches. This further increases the cache's effective coverage of the IP address space.

3.3. Cache Miss Handling

The average routing table lookup time depends on both cache hit ratio as well as cache miss penalty, which is determined by the software algorithm used to perform routing table lookup. A simple and elegant data structure to solve the longest prefix match problem is a binary trie [5] (figure 1). Once network addresses in a routing table are inserted in a trie, every node of the trie corresponds either to a purely internal node or a node corresponding to a network address (called an address node). Given an input address, one can simply walk the trie from the root using the bits in the input, and stop at a node from which further branches are not possible (either because the node is a leaf node, or because the branch corresponding to an input bit does not exist). The last address node encountered along this path is the longest matching prefix of the input address.

Clearly, the worst case number of memory accesses for the trie walking algorithm is the same as the worst case depth of the trie, viz 32 for IP addresses. Instead, we use a data structure that reduces the trie lookup to a small number of table lookups. This is done by *flattening* out the trie to a fixed number of levels. We call this data structure the Network Address Routing Table (NART) [3]. To understand NART construction, refer to figure 1. Suppose we choose to flatten out the trie at three levels, corresponding to the first 16 bits (level 1), the next 8 bits (level 2) and the last 8 bits (level 3) of the address. Given an address, the trie node corresponding to it may lie at or above one of these levels.

We can visualize level 1 as a simple table of $2^{16} = 64K$ entries. We will use the notation $X_{a,b}$ to represent bit numbers a to b of an N-bit number X, with a being the more significant bit position. Also, the most significant and least significant bits will be numbered 0 and $N-1$ respectively. Note that an address A of length L that lies above level 1 corresponds to a range of 2^{16-L} entries in the level 1 table, starting at entry $A_{0,L-1} * 2^{16-L}$. Thus, these entries can be filled with the output port identifier corresponding to address A. Note that due to the longest prefix match requirement, if a level 1 table entry corresponds to more than one address, the output port identifier in that entry is the one corresponding to the longer address.

Whenever an address node A with length L lies between levels 1 and 2, first note that its first 16 bits correspond to entry $A_{0,15}$ of the level 1 table. However, since the address continues beyond level 1, it is kept in a level 2 table which is pointed to by entry $A_{0,15}$ of the level 1 table. Since such level 2 tables correspond to trie levels 16 to 23, the size of such tables is 256 entries. Thus, address A corresponds to a range of 2^{24-L} entries starting from $A_{16,L-1} * 2^{24-L}$ in this level 2 table. The treatment of address nodes lying between levels 2 and 3 is exactly the same as those lying between levels 1 and 2, through the use of level 3 tables of 256 entries. The longest prefix requirement is also taken care of in level 2 and level 3 tables, just as in the case of the level 1 table.

To perform a lookup for address A in the NART, we perform a table lookup using bits $A_{0,15}$ in the level 1 table. If this entry does not contain a pointer to a level 2 table, the lookup is complete and the output port identifier contained therein is the lookup result. Otherwise, the corresponding level 2 table is looked up using bits $A_{16,23}$. If this level 2 table entry does not contain a pointer to a level 3 table, the lookup is complete. Otherwise, a final lookup using bits $A_{24,31}$ is performed on the corresponding level 3 table. We shall call NART table entries that contain output port identifiers as *leaf* NART entries, whereas those containing pointers to other tables shall be called *non-leaf* entries. The NART data structure effectively reduces the worst-case lookup time to three memory accesses, with a reasonable space overhead.

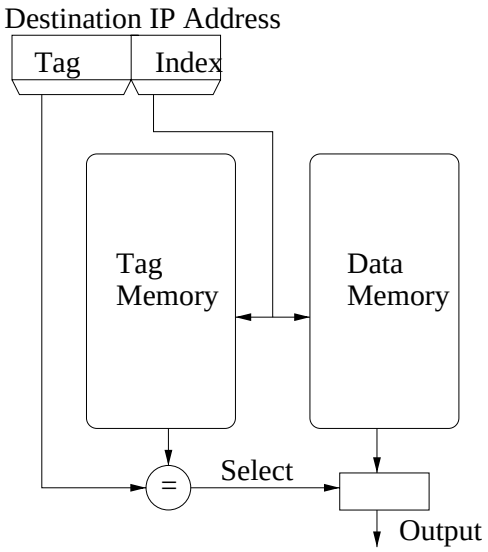

Figure 2. *The baseline network processor cache architecture, which is identical to generic CPU caches. Simulation results show that smaller block sizes are preferred due to lack of spatial locality in network packet stream.*

We have implemented the above NART algorithm on a Pentium-II 233 MHz machine with a 16-KByte L1 data cache. The measured software NART lookup time for the packet trace, using the IPMA routing table, is 120 CPU cycles on the average.

4. Baseline: Host Address Cache (HAC)

Figure 2 shows the baseline network processor cache architecture, which is identical to a conventional CPU cache. In this section, we report the results of generic cache simulations by varying the cache size, the cache block size, and the degree of associativity, for two reasons: to identify possible differences in the locality characteristics of network packet streams and program reference streams, and to establish the baseline model against which subsequent cache design alternatives are compared. The simulated cache miss ratio results in Table 1 show that the cache size and degree of associativity have a similar performance effect on the network processor cache as on the CPU cache. However, a distinct difference between network packet streams and program reference streams is that the former lacks spatial locality, as evidenced by the fact that for a given cache size and degree of associativity, decreasing the block size *monotonically* decreases the cache miss ratio [2]. Intuitively

[2]This result holds for the trace even without interleaving.

Cache Size	Block Size	Associativity	Miss Ratio
4K	32	1	57.09%
		2	53.25%
		4	50.92%
	8	1	36.51%
		2	31.29%
		4	29.00%
	1	1	12.71%
		2	8.42%
		4	6.86%
8K	32	1	43.70%
		2	40.78%
		4	38.05%
	8	1	26.35%
		2	21.72%
		4	19.33%
	1	1	7.57%
		2	4.59%
		4	3.29%
32K	32	1	18.65%
		2	16.52%
		4	15.58%
	8	1	9.59%
		2	6.66%
		4	5.49%
	1	1	2.39%
		2	1.07%
		4	0.75%

Table 1. *Miss ratios for the baseline host address cache under varying cache sizes, cache block sizes and degrees of associativity. Cache sizes are reported in numbers of entries rather than numbers of bytes.*

this behavior is expected as there is no direct temporal correlation among network activities of the hosts residing in the same subnet. Poorer performance for caches with larger block size results because larger block size leads to inefficient cache space utilization when references to addresses within the same block are not correlated temporally, i.e., when there is low spatial locality. The performance difference between cache configurations that are identical except the block size could be dramatic. For example, the miss ratios of a 4-way set associative 8K-entry cache with a 32-entry block size and one with 1-entry block size is nearly an order of magnitude apart, 38.05% vs. 3.29%. As cache size increases, the performance impact of block size decreases, (although still significant) because the space utilization efficiency is less of an issue with larger caches. We conclude that the block size of network processor caches should always be small, preferably one entry wide.

Whenever the base data structure from which a cache is built changes, there is a cache consistency problem. For the host address cache, modification of the routing table due to the routing protocol's message exchanges gives rise to the consistency problem. However, unlike CPU cache, temporal inconsistency in the host address cache is tolerable, because the routing protocol itself takes time to converge to the new routes. Therefore, there is much more latitude in the timing of consistency maintenance actions. To simulate the effects of routing table changes, we flush the contents of the host address cache and measure the impacts of routing table update frequency on the effectiveness of the host address cache. The results are shown in Table 2, which assume the cache is direct-mapped and its block size is one entry wide. As the flush interval increases, the miss ratio decreases as expected. But the performance difference due to flushing, as shown by the ratio of the miss rates corresponding to the 100K and ∞ flush intervals, increases with the cache size. The reason for this behavior is that larger caches require a longer cold-start time, and therefore tend to suffer more than smaller caches when the flush interval is small. Consequently, the relative performance difference between different flush frequencies is more significant for larger caches. To put the flush intervals used in these simulations in perspective, 100K packets is equivalent to 100 msec for a router that can process one million packets per second. In reality, the interval between consecutive routing table changes is of the order of seconds.

Cache Size	Flush Interval	Miss Ratio
4K	100K	14.23%
	400K	13.16%
	∞	12.71%
8K	100K	9.69%
	400K	8.23%
	∞	7.57%
32K	100K	5.40%
	400K	3.41%
	∞	2.39%

Table 2. *The impacts of the frequency of routing table updates, which translate into cache flushes, on the miss ratios. The flush interval is the number of packets the host address cache processes between consecutive flushes. An ∞ flush interval corresponds to the "no flush" case.*

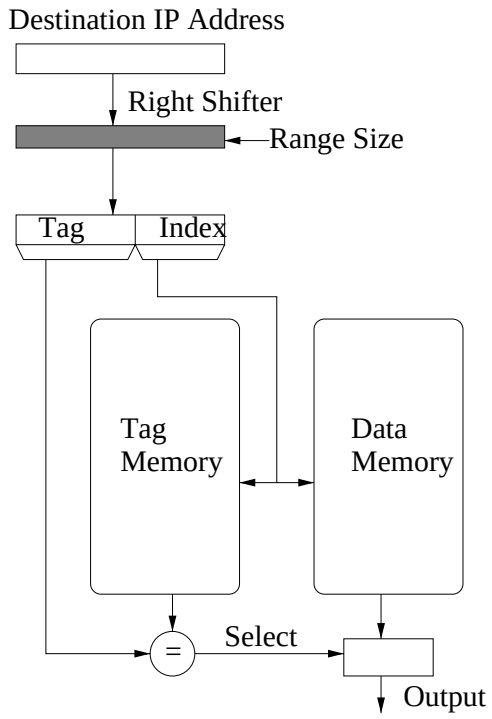

Figure 3. *The network processor cache architecture that caches host address ranges rather than individual host addresses.* Range size *is a global parameter applied across the entire address space, and is determined by maximally concatenating address ranges in the IP address space.*

5. Host Address Range Cache (HARC)

Each routing table entry corresponds to a contiguous range of the IP address space. For example, a routing table entry with a network address field of $0x82f50000$ and a network mask field of $0xffff0000$ corresponds to a contiguous range $< 0x82f50000 \ldots 0x82f5ffff >$ in the $< 0 \ldots 2^{32} - 1 >$ IP address space. If a network packet's destination address falls within a routing table entry's range, it should be routed to that entry's output interface. One could exploit the above fact to increase the effective coverage of a host address cache, by caching host address ranges instead of individual addresses. Network addresses need to go through two additional processing steps before *host address range cache* (HARC) could be put to practical use.

First, with the longest prefix match requirement, it is possible that some routing table entry's address range covers another's address range. The former is called an *encompassing* routing table entry while the latter is an *encompassed* entry. An encompassing entry's network address is a prefix of those entries it encompasses. The address range associated with each encompassed routing table entry needs to be "culled" away from the address ranges of all the entries that encompass it, so that every address range in the IP address space is covered by *exactly one* routing table entry. This culling step is essential because it ensures that an IP destination address lying in a particular address range has a unique lookup result.

Second, *adjacent* address ranges that share the same output interface should be merged into larger ranges as much as possible. Once this merging is done, these ranges are "aligned", that is, ranges are potentially split to make all range sizes powers of 2 and to make starting addresses of all ranges aligned with a multiple of their size. Then the minimum of all resulting address range sizes is calculated. This minimum size becomes the the *minimum_range_granularity* parameter of the HARC. *Range size*, which is defined as $\log$(*minimum_range_granularity*), thus represents the number of least significant bits of an IP address that could be ignored during routing-table lookup, since destination addresses falling within a minimum address range size are guaranteed to have the same lookup result. Figure 3 shows the hardware architecture of the HARC, which is the baseline cache augmented with a logical shifter. The destination address of an incoming packet is logically right-shifted by *range size* before being fed to the baseline cache. Because each address range corresponds to a cacheable entity, HARC's effective coverage of of the IP address space is increased by a factor of *minimum_range_granularity*.

Cache Size	Assoc	Miss Ratio HARC	Miss Ratio HAC/HARC	Average Lookup Time HAC/HARC
4K	1	7.54%	1.69	1.62
	2	4.58%	1.84	1.71
	4	3.64%	1.88	1.72
8K	1	4.48%	1.69	1.58
	2	2.20%	2.09	1.78
	4	1.56%	2.11	1.72

Table 3. *Cache miss ratio comparisons between host address range cache (HARC) and host address cache (HAC), assuming the block size is one entry wide and the* range size *is* 5*. The last column is the ratio between HAC's and HARC's average routing-table lookup times, assuming the hit access time is one cycle and the miss penalty is 120 cycles.*

We processed the IPMA routing table according to the steps described above, and calculated the *range size* parameter, which turned out to be 5. This means that each HARC entry now corresponds to a continuous range of

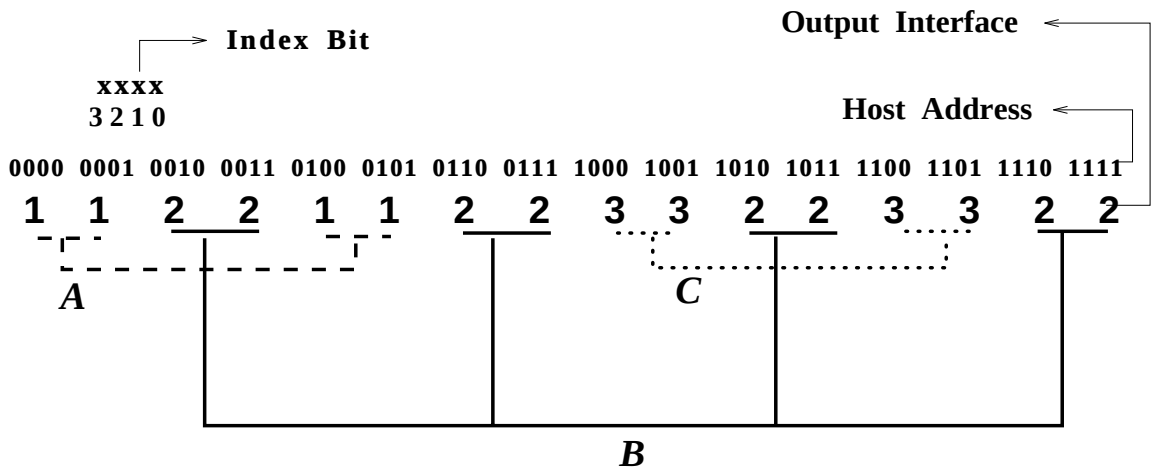

Figure 4. *A routing table example that illustrates the usefulness of carefully choosing the index bits. In this case, the* 1*-th bit is chosen to be the index bit. The total number of distinct address ranges is reduced from 8, if the basic merging operation used in the formation of the host address range cache is used, to 3. The three address ranges are labeled as A, B, and C.*

32 addresses, a factor of 32 increase in the cache's effective coverage. Table 3 shows the comparison between the cache miss ratios of the HARC and the host address cache (HAC), assuming the block size is one entry wide. HAC's miss ratio is between 1.68 to 2.10 times higher than that of HARC. In terms of average routing-table lookup time, HARC is between 58% and 78% faster than HAC, assuming that the hit access time is one cycle and the miss penalty is 120 cycles. Because the logical right shifting step in the HARC lookup lowers the "degree of variation" in the address stream as seen by HARC, the probability of conflict miss increases. As a result, the miss ratio gap between HAC and HARC widens with the degree of associativity, because HARC benefits more from higher degrees of associativity by eliminating more conflict misses than HAC.

6. Intelligent Host Address Range Cache (IHARC)

A traditional CPU cache of size 2^K and block size 1 directly takes the least significant K bits of a given address to index into the data and tag arrays. In this section, we show that by choosing a more appropriate hash function for cache lookup, it is possible to further increase every cache entry's coverage of the IP address space.

Consider the example routing table in Figure 4, where there are 16 4-bit host addresses with three distinct output interfaces, 1, 2 and 3. The merging algorithm used in calculating the *range size* of the HARC will stop after all adjacent address ranges with identical output interfaces are combined. In this case, the total number of address ranges is 8, because the minimum range granularity is 2. To further grow the address range that a cache entry can cover, one could choose the index bits carefully such that when the index bits are ignored, some of the identically labeled address ranges are now "adjacent" and thus could be combined. For example, if bit 1 (with bit 0 being least significant) is chosen as the index bit into the data/tag array, then the host addresses 0000, 0001, 0100, and 0101 can be merged into an address range because they have the same output interface, 1, and when bit 1 is ignored, they form a continuous sequence, 000, 001, 010, and 011. Similarly, 1000, 1001, 1100, and 1101 also can be merged into an address range, as well as all the host addresses whose output interface is 2. With this choice of the index bit, the total number of address ranges to be distinguished during cache lookup is reduced from 8 to 3. Note that index bit 1 induced a partitioning of the address space such that in each partition, some address ranges that were not adjacent in the original address space became adjacent.

Intuitively, IHARC provides more opportunities of merging identically-labeled address ranges by decomposing the address space into partitions, based upon a set of index bits, and merging identically-labeled ranges that are adjacent within a partition. Note that HARC insists on merging ranges that are adjacent in the *original* IP address space, and thus is a special case of IHARC.

IHARC selects a set of K index bits in the destination address that correspond to 2^K cache sets. Each cache set corresponds to a partition of the IP address space. In a partition, some address ranges that were not originally adja-

```
S = ∅;
for (i=1; i ≤ K; i++) {
                score = ∞;
                candidate = 0;
                for (j=range_size+1; j ≤ N; j++) {
                    if (!(j ∈ S)) {
                        currentscore = Score(S,j);
                        if (currentscore < score) {
                            score = currentscore ;
                            candidate = j;
                        } } }
                S = S ∪ {candidate};
}
```

Figure 5. *A greedy index bit selection algorithm used to pick the bits in the input addresses for cache lookup*

cent in the IP address space, will become adjacent. Any adjacent ranges that are identically labelled are then merged into larger ranges. Thus, we get a set of distinct address ranges for every partition (or cache set). Since distinct address ranges in a cache set need unique tags, the number of distinct address ranges in a cache set represents the degree of contention in the cache set. Thus, the index bits are selected in such a way that after the merging operation, the total number of address ranges and the difference between the number of address ranges across cache sets is minimized.

We first describe our index bit selection algorithm. Assume N and K are the number of bits in the input address and the index key, respectively. In general, any subset of K bits in the input addresses could be used as the index bits, except the least significant *range size* bits as determined by the basic merging step in constructing the HARC. We use a greedy algorithm to select the K index bits, as shown in Figure 5. S represents the set of index bits chosen by the algorithm so far. $Score(S, j)$ is a heuristic function that calculates the desirability of including the j-th bit, given that the bits in S have already been chosen to be included in the index bit set. For each partition of the IP address space induced by the bits in $S \cup \{j\}$, the algorithm first merges adjacent identically-labelled ranges in that partition. This step gives us, for every partition, the number of distinct address ranges that need to be uniquely tagged.

As mentioned earlier, the number of distinct address ranges in a partition represents the extent of contention in the corresponding cache set. Thus, for a candidate bit set $S \cup \{j\}$, we define the i-th partition's metric $M_{i,(S,j)}, \forall i$, as the number of distinct address ranges in partition i. Then, the algorithm minimizes $Score(S, j)$, given by

$$Score(S, j) = \sum_i M_{i,(S,j)} + W \sum_i |\overline{M_{S,j}} - M_{i,(S,j)}|$$

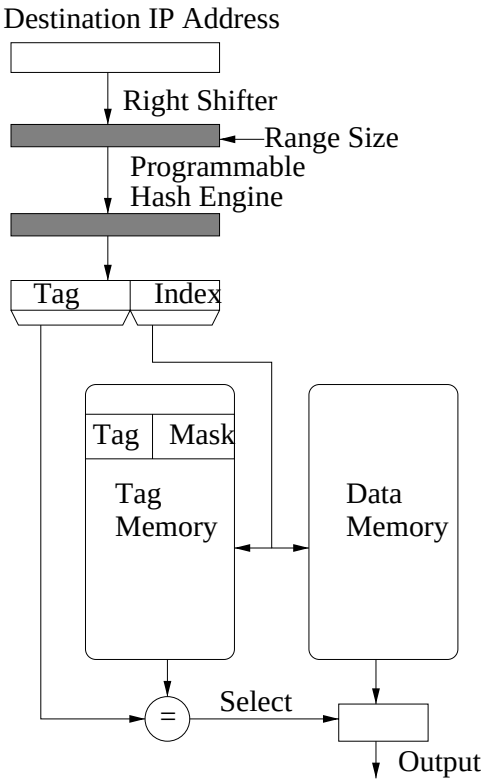

Figure 6. *The intelligent host address range cache architecture that uses a hash function specific to a routing table in order to combine disjoint address ranges into a logical host address set which is then mapped to a cache entry. The programmable hash engine provides the flexibility needed to tailor the hash function to the network routing table, which is changing dynamically.*

where $\overline{M_{S,j}}$ is the mean of $M_{i,(S,j)}$ over all partitions i and W is a parameter that determines the relative weight of the two terms in the minimization. Note that the second term of the weighted sum minimizes the standard deviation and is included to prevent the occurrence of hot-spot partitions, and thus excessive conflict misses, in the IHARC cache sets.

Figure 6 shows the hardware architecture of a host address range cache with a programmable hash function engine that allows tailoring the choice of the index bit set to individual routing tables.

While HAC and HARC use a fixed three-level-table NART structure (16, 8 and 8 bits) that is independent of the hardware cache configurations, the NART associated with IHARC depends on the hardware configuration. In particular, the number of entries in the level 1 table is equal to 2^K, where K is the number of index bits in IHARC,

Number of Index Bits	Bits Chosen	**No. of Ranges** (without constraint)	**No. of Ranges** (with constraint)
9	13 14 16 17 18 19 21 22 25	76,867	116,789
10	13 14 16 17 18 19 21 22 25 30	77,379	116,789
11	13 14 16 17 18 19 20 21 22 25 30	88,465	134,093
12	13 14 16 17 18 19 20 21 22 24 25 30	92,579	138,756
13	13 14 16 17 18 19 20 21 22 24 25 26 30	98,563	145,538

Table 4. *The number of address ranges that need to be distinguished and the bits chosen as index bits after applying the index bit selection algorithm to a routing table from the IPMA project. The last two columns correspond to the number of address ranges with and without the constraint that each address range's size has to be a power of 2, respectively.*

and the set of K selected index bits is used to index into the level 1 table. As a result, the cache miss penalty for IHARC may be different for different cache configurations. However, since the miss penalty is dominated by the number of memory accesses made in software NART lookup (3 lookups in the worst case), which is comparable in all configurations, measurements from our prototype implementation show that the average miss penalty is almost the same for all IHARC cache configurations we experimented with, and moreover is close to that of HAC and HARC, i.e., 120 cycles.

Another important difference is that in addition to the output interface, a *leaf* NART entry must contain the address range it corresponds to, so that after an NART lookup following a cache miss, the cache set can be populated with the appropriate address range as the cache tag. Given an N-bit address, the K index bits select a particular cache set, say C. The remaining $N - K$ bits of the address form a value, say T, which lies in one of the address ranges in this partition. Initially, when a cache set is not populated, T is looked up in software using the NART and the address range A in which T falls becomes the tag of the cache set C. If the cache entry was already populated with an address range A, a range check is required to figure out whether the lookup is a hit (which corresponds to checking that T lies in the range A). However, a general range check is still too expensive to be incorporated into caching hardware. By guaranteeing that each address range size is a power of two and that the starting address of each range is aligned with a multiple of its size during the merge step, one can perform the range check simply by a mask-and-compare operation. Therefore, each tag memory entry in the IHARC includes a tag field as well as a mask field, which specifies the bits in the address to be used in the 'tag match'. The price of simplifying cache lookup hardware is an increase in the number of resulting address ranges, as compared to the case when no such alignment requirement is imposed. If the range check results in a miss, the NART data structure is looked up and the cache is populated with the appropriate address range.

Compared to the generic host address range cache (HARC), the intelligent host address range cache (IHARC) reduces the number of distinct address ranges that need to be distinguished, by a careful choice of the index bits. Table 4 shows the number of distinct address ranges that result after applying the index bit set selection algorithm to the IPMA routing table, for different numbers of index bits. To put these numbers in perspective, the number of entries in the original routing table is 39,681, and the number of address ranges from HARC is 2^{27} or 134,217,728. In other words, the index bit set selection algorithm effectively reduces the number of distinct address ranges from HARC to IHARC by three orders of magnitude. In addition, this number is only 3 to 4 times the number of entries in the original routing table, even though the resultant address ranges can now be efficiently looked up with conventional cache lookup hardware. As mentioned before, for address ranges to be tag-matched based on masks, their size has to be a power of two. Table 4 shows the difference in the number of distinct address ranges with and without this constraint.

Cache Size	Assoc	Miss Ratio IHARC	Miss Ratio HARC/ IHARC	Lookup Time HARC/ IHARC	Lookup Time HAC/ IHARC
4K	1	2.30%	3.28	2.67	4.31
	2	1.12%	4.09	2.77	4.72
	4	0.57%	6.39	3.18	5.46
8K	1	1.54%	2.91	2.24	3.53
	2	0.48%	4.58	2.30	4.11
	4	0.22%	7.09	2.26	3.90

Table 5. *Miss ratios for the intelligent host address range cache (IHARC), assuming that the block size is one entry wide. The last two columns are the ratio between HARC's and IHARC's average routing-table lookup times, and that between HAC's and IHARC's, respectively, with the HARC's $range\_size$ as 5.*

Table 5 shows the miss ratios for IHARC, assuming that the block size is one entry wide. In terms of average routing-table lookup time, HARC is between 2.24 and 3.18 times slower than IHARC. This is because HARC's miss ratios are 2.91 to 7.09 times larger than IHARC's. In addition, the miss ratio gap between HARC and IHARC increases with the degree of associativity. This result conclusively demonstrates that there is significant performance improvement to be gained from IHARC over HARC. Compared to HAC, IHARC reduces the average routing table lookup time by up to a factor of 5.

7. Conclusion

This paper reports the results of one of the first research efforts on cache memory designs for emerging network processors. Based on a real packet trace collected from the main router of a national laboratory, we studied a series of routing-table cache designs. Major results from this research are summarized as follows:

- Based upon the interleaved trace used in the study, there seems to be sufficient temporal locality in the packet stream to justify the use of a routing-table cache in network processors. However, spatial locality is weak, and therefore the block size should be small, preferably one entry wide.
- Caching address ranges rather than individual addresses greatly improves the effective coverage of caches of a given size and therefore their hit ratios.
- A careful choice of the index bits during cache lookup is crucial and can dramatically reduce the number of address ranges that need to be distinguished, and thus the cache miss ratio.

We are currently investigating the performance impacts of routing table updates on HARC and IHARC, both of which exploit the current contents of the routing table to dynamically reconfigure the cache hardware, and therefore need to be changed on the fly. Specifically, developing an incremental version of the index bit selection algorithm will considerably enhance IHARC's practical usability.

References

[1] E. Basturk et al. Design and implementation of a qos capable switch-router. *Sixth International Conference on Computer Communications and Networks*, pages 276–84, September 1997.

[2] X. Chen. Effect of caching on routing-table lookup in multimedia environments. *IEEE INFOCOM*, pages 1228–36, April 1991.

[3] T. Chiueh and P. Pradhan. High performance ip routing table lookup using cpu caching. *IEEE INFOCOM*, April 1999.

[4] Xaqti Corporation. Gigapower protocol processor. *(http://www.xaqti.com/gp_01.htm)*.

[5] W. Doeringer, G. Karjoth, and M. Nassehi. Routing on longest matching prefixes. *IEEE/ACM Transactions on Networking*, 4(1):86–97, February 1996.

[6] D. Estrin and D. Mitzel. An assessment of state and lookup overhead in routers. *IEEE INFOCOM*, pages 2332–42, May 1992.

[7] D. Feldmeier. Improving gateway performance with a routing-table cache. *IEEE INFOCOM*, pages 298–307, March 1988.

[8] R. Jain. Characteristics of destination address locality in computer networks: a comparison of caching schemes. *Computer Networks and ISDN Systems*, 18(4):243–54, May 1990.

[9] Kawasaki LSI. Longest match engine. *(http://www.klsi.-com/products/lme.html)*.

[10] Berkeley Networks. The integrated network services switch: A new architecture for emerging applications. *(http://www.berkeleynet.com/html/tech2.html)*.

[11] MMC Networks. 20 mpps network processor with wire-speed layer 3 processing for building switches and routers. *(http://www.mmcnet.com/91097.html)*.

[12] Neo Networks. Streamprocessor 2400 backbone switch router. *(http://www.neonetworks.com/product%-20literature/sp2400.htm#sp2400_top)*.

[13] C. Partridge. Locality and route caches. *NSF Workshop on Internet Statistics Measurement and Analysis (http://www.caida.org/ISMA/Positions/partridge.html)*, 1996.

[14] C. Partridge, P. Carvey et al. A fifty gigabit per second ip router. *IEEE/ACM Transactions on Networking*, 6(3):237–48, June 1998.

[15] P. Pradhan and T. Chiueh. Operating systems support for programmable cluster-based internet routers. *IEEE Workshop on Hot Topics in Operating Systems*, pages 76–81, March 1999.

[16] Music Semiconductor. Muac routing coprocessor (rcp) family. *(http://www.music-ic.com/muac.pdf)*.

[17] Avici Systems. The world of terabit switch/router technology. *(http://www.avici.com/html/a_new_world_1.html)*.

[18] Pluris Terabit Network Systems. Next generation internet router. *(http://www.pluris.com/data.htm)*.

[19] Torrent Network Technologies. High-speed routing table search algorithms. *(http://www.torrentnet.com/-general/download/highspeed.pdf)*.

[20] Torrent Network Technologies. The ip9000 gigabit router architecture. *(http://www.torrentnet.com/general/download/-ip9000_Arch.pdf)*.

[21] Michigan University and Merit Network. Internet performance management and analysis (ipma) project. *(http://nic.merit.edu/ ipma)*.

[22] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed ip routing lookups. *ACM SIGCOMM*, September 1997.

[23] A. Brodnik, S. Carlsson, M. Degermark and S. Pink. Small Forwarding Tables for Fast Routing Lookups. *ACM SIGCOMM*, September 1997.