Memory Hierarchy Design for a Multiprocessor Look-up Engine

Jean-Loup Baer, Douglas Low, Patrick Crowley, Neal Sidhwaney
Department of Computer Science and Engineering
University of Washington
{baer,douglas,pcrowley}@cs.washington.edu

Abstract

We investigate the implementation of IP look-up for core routers using multiple microengines and a tailored memory hierarchy. The main architectural concerns are limiting the number of and contention for memory accesses.

Using a level compressed trie as an index, we show the impact of the main parameter, the root branching factor, on the memory capacity and number of memory accesses. Despite the lack of locality, we show how a cache can reduce the required memory capacity and limit the amount of expensive multibanking. Results of simulation experiments using contemporary routing tables show that the architecture scales well, at least up to 16 processors, and that the presence of a small on-chip cache increases throughput significantly, up to 65% over an architecture with the same number of processors but without a cache, all while reducing the amount of required off-chip memory.

1 Introduction

To reduce cost and improve flexibility, modern networking equipment is built around network processors (NPs) – a new class of commodity, software-based microprocessor. The design requirements for NPs are demanding: they must support diverse functionality in a wide range of network environments. In this paper, we focus on the problem of supporting the longest prefix match (LPM) algorithm at high speeds with a network processor. One important application of this task – and the one used to motivate this paper – can be found in core Internet routers, where LPM is used in IP packet look-up to match destination addresses with a large number of forwarding rules. IP look-ups have often been studied, but a number of novel considerations arise when they are implemented on NPs, namely their multiprocessor nature and the design of their memory hierarchy.

Conceptually, IP packet look-up is the process of searching a *forwarding table* for a *rule* (i.e., an entry in the table) for which there is a match between a packet's destination address and a destination address entered in the table. If it were not for the large sizes of the tables (several tens of thousand entries), this matching problem would be simple. For example, one could use hashing if all possible Internet addresses (32-bit strings for IPv4, 128-bit strings for IPv6) were entered in the table. In practice this is not possible since such tables would have billions of entries, the great majority of which would not be relevant. Furthermore, IP networks are addressed hierarchically, so one bit-string prefix, and hence one routing rule, can often be used for all hosts within an organization or subnet. Thus, what is stored in forwarding tables is a set of prefixes and associated output ports. The look-up process requires a *longest prefix match* (LPM).

Forwarding imposes stringent requirements on core routers, where packet arrival rates are high. Assume, as an example, that we would like to process 1 Million packets per second, i.e., one packet must be forwarded in 1 microsecond. Any software-based solution on a general purpose processor can barely meet this speed constraint if the forwarding table is large since a single access to the memory where the table is stored will take over 50 ns and several memory accesses, to the table and/or some large indexing structure, will be necessary. Conventional cache-based solutions are of limited use because of the total lack of spatial locality and short-lived temporal locality of incoming packet addresses. Therefore, network processors used for forwarding must include engines and a memory system tailored to the search process while retaining some programmability for performing other functions.

One of the saving factors is that packets can be forwarded independently of each other [3]; in fact, network processors are organized as on-chip multiprocessors of *microengines* to exploit this situation. Therefore several microengines can be dedicated to forwarding tasks in order to increase forwarding throughput. However, the drawback is that the index, in our case a variant of a compressed trie called a level-compressed trie (LC-trie), will be accessed by several processes concurrently and the resulting contention must be reduced as much as possible.

We can summarize both the novelty and the challenges of implementing IP look-ups on a commercial NP: to provide a memory system that allows multiple processors, or microengines, to concurrently access a globally shared trie-based index structure. In this paper, the reduction in the number of contention for memory accesses will be achieved in two ways: caching since the levels of the trie closest to the root are accessed most frequently and thus exhibit some limited form of locality and multibanking to allow concurrent accesses to the memory holding the LC-trie.

The rest of this paper is organized as follows. In Section 2 we give a short introduction to IP look-ups in the context of large routing tables and review how the longest prefix matching problem can be solved using an LC-trie [10]. In particular we show the impact of a key parameter of the data structure, namely the root branching factor. As the root branching factor *bf* grows, so does the size of the LC-trie. On the other hand, the average number of memory accesses decreases with a larger *bf*. We show how a cache, even with a low hit-rate, can reduce the need for a large capacity LC-trie without adversely affecting the number of memory accesses.

In Section 3 we describe a multiprocessor architecture and memory hierarchy for the mapping of the LC-trie, the forwarding table, and ancillary data structures. We provide an initial analysis of the impact of multibanking the memory holding the LC-trie.

In Section 4 we present experimental results obtained via trace-driven simulation. We vary the factors that can influence throughput, namely those related to the data structure (branching factor), to the parallelism in the architecture (number of microengines), and to the memory hierarchy (latency, caching, and multibanking). In the absence of a cache, the largest branching factor yields the best throughput. This throughput is limited by the number of memory banks when the memory latency is large but this effect is not as visible for low memory latencies. In general, throughput increases almost linearly with the number of processors until contention to memory becomes important. When a cache is present, the branching factor does not need to be as large since hits in the cache produce two complementary effects: faster access to the part of the index that is currently cached and and reduced contention for the part that is not. With large memory latencies, the throughput is practically independent of the branching factor and superior to the best throughput obtained without a cache. When memory and compute times are balanced, the cache impact is less dramatic but still yields improvements in throughput. Moreover, with caching both the size of the LC-trie and the number of memory banks can be reduced without affecting adversely the throughput.

In Section 5 we review previous work in this area and in Section 6 we summarize the results and suggests areas of further study.

2 IP Look-up using LC-trie indexing

The longest prefix match problem within the context of general-purpose processors acting as Internet routers has been thoroughly researched. The problem is characterized by:

- The forwarding tables mapping destination addresses and output ports are large, with say a number of entries $n \approx 50,000$, but they are small compared to the number of possible entries (2^{32} for IPv4 and 2^{128} for IPv6).
- There is no spatial locality in the accesses to these tables and temporal locality (access to same addresses) is short-lived.
- The distribution of the length of "longest prefixes" is heavily skewed with most of the prefixes being between 14 and 24 bits [9] with peaks at 16 and 24 bits for historical reasons, namely the IP scheme of classification into A, B, and C subnets [12].
- Search for a match occurs at least two orders of magnitude more frequently than insertions/deletions of prefixes.
- IP look-up must be done at wire speed thus any solution employing a programmable device must have a limited number of off-chip memory accesses.
- IP look-up is the bottleneck in the pipelined process of forwarding a packet to its next destination.

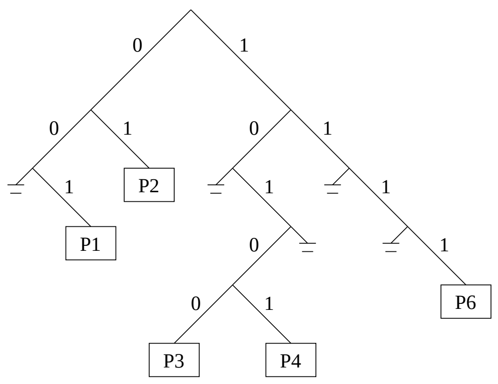
The size of the tables and the performance constraints preclude direct searches in the forwarding tables. Instead, search in an indexing structure such as a trie is commonly used. The trie is searched for a given string and upon a successful match yields a pointer to an entry in the forwarding table. All the techniques (see [15] for an excellent survey) share some common features such as prefix expansion and level compression that are reviewed next. They differ, among other criteria, in the internal representation of the data structures and the choice of the selection of levels in the compressed tries. We choose to concentrate on one particular data structure, namely LC-tries (LC stands for Level Compressed) [10], because it is rather simple to construct and can be easily modified from IPv4 to IPv6 addresses. (In this paper, our experiments will only be conducted with IPv4 addresses.) The conclusions that we reach for the efficient utilization of LC-tries should also be correct for other compressed tree implementations because the latter are based on the same basic structures.

2.1 Level Compressed Tries (LC-tries)

As mentioned above, a basic data structure to represent strings for efficient storage and retrieval is the trie. In the case of binary strings, like Internet addresses, the trie becomes a binary tree. Strings are stored at the leaves and the value of the string is the value of the path used to reach it, with 0 (1) being the value of a traversal of the left (right) pointer of a node. While binary tries are attractive for their simplicity, they yield search times involving a number of comparisons, and hence of memory accesses, equal in the worst case to the length of the string (32 for IPv4). This is unacceptable performance-wise and the two techniques presented below have as their goal to reduce the number of these comparisons. We will illustrate them with the forwarding table shown in Figure 1 (a) along with its binary trie. Notice that we have removed the entry corresponding to P5 which is a prefix of P6. This is to facilitate the LPM process and avoid backtracking in the trie. The justification is that updates to the forwarding tables are relatively infrequent, and therefore the tables can be preprocessed and strings that are prefixes of other addresses can be removed and stored in a *prefix table*. In our example, the forwarding table entry for P6 will have a link to the prefix table entry containing P5. Measurements on existing tables [8] show that prefix tables contain fewer than 10% of all entries.

The first technique to reduce the number of LC-trie comparisons is *path compression*, a technique derived from Patricia tries [6]. We remove from the trie any internal node, say A, that has a single child. The *skip* value [10], i.e., the number of internal links that have disappeared, is stored in either the first node with 2 children or the leaf (whichever comes first) on the path below node A. The path compressed trie of the trie of Figure 1 (b) is shown in Figure 1(c)

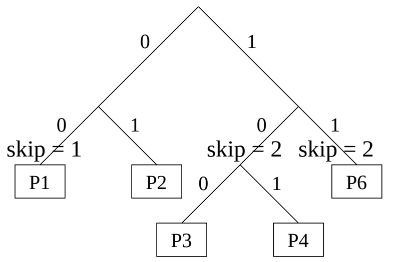
The second technique is *level compression*. Instead of a node having 2 children, we let it have 2^k children, where k is called the *branching factor*. For example, with $k = 2$ the 4 children of the root will be those reached by strings "00*", "01*", "10*" and "11*" respectively as shown in Figure 1 (d). In most cases though, some of these children might not exist in the original trie. However we can have several nodes in the trie pointing to the same entry in the forwarding table if they correspond to strings with the same longest prefix. We can therefore perform a *prefix expansion* replacing internal nodes with one child by internal nodes with 2 children and so on depending on the branching factor. We expand each node to cover a different number 2^n of children, ensuring the expansion will yield less than $\lceil 2^n(1-x) \rceil$ empty leaves, where x is called the *fill factor*.



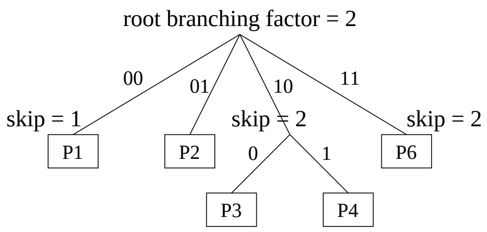
(b) Trie corresponding to the Forwarding Table

Rule name	Prefix
P1	001
P2	01
P3	10100
P4	10101
P5	11
P6	1111

(a) Forwarding Table



(c) Path compressed trie



(d) Level Compressed trie

Figure 1: Forwarding table and corresponding binary trie

2.2 LC-tries for forwarding tables

The indexing structure that we will consider, a variant of LC-tries, will use:

- Prefix expansion at the root. This expansion is justified by the fact that very few prefixes are of length less than 14. How to choose the branching factor of the root is part of the tuning process.
- Path compression and level compression. We will use a fill factor of 0.5 as suggested in [10] (we experimented with values of x between 0.25 and 0.75 and found little difference in the metrics discussed below).

In addition to the LC-trie, the data structures involved in the forwarding process are:

- The *forwarding table*. Each entry in the table consists of a prefix, its length (to check the correctness of the match), an output port number, and a pointer (possibly null) to the prefix table. For IPv4, a possible implementation would devote: 4 bytes (32 bits) for the prefix, 1 byte for the prefix length, 1 byte for the port number (allowing 256 output ports), and 2 bytes for the pointer to the prefix table, i.e., 8 bytes per entry. Even for large forwarding tables, this represents less than 1 MByte.

- The *prefix table* is a set of linked-lists, each linked-list corresponding to prefixes of one particular forwarding table entry. For each element of a linked-list, an entry consists of: the length of the string (1 byte), the output port number (1 byte), and a pointer, possibly null, to the next entry in the list (2 bytes). Thus each entry is 4 bytes. If we assume that at most 10% of the forwarding table has to be stored in the prefix table, the latter should be less than 50 KBytes.

The *LC-trie* itself will be built starting from the forwarding table that will need to be sorted beforehand. The LC-trie is represented as an array corresponding to a breadth-first traversal of the trie. Each element of the array has the following fields that fit within 4 bytes:

- The branching factor bf of the node; $bf = 0$ indicates a leaf (5 bits)
- The skip value (5 bits which is sufficient for IPv4)
- A pointer. If $bf \neq 0$, the pointer is the index in the array of the leftmost child of the node. This pointer is restricted to 22 bits thus limiting the branching factor to be 21. If $bf = 0$, the pointer points to an entry in the forwarding table and 21 bits allows for 2 million entries, an order of magnitude more than what is needed.

The number of nodes in the LC-trie depends principally on the branching factor at the root and to a much lesser degree on the fill factor and the number of skip values.

A root branching factor of k compares the first k bits of the IP address with the prefixes of length k in a single access to the trie. Thus the number of comparisons to find the LPM will decrease with a larger k. However, this comes with a significant increase in the number of nodes at the first level of the trie. For example, with the the above implementation constraints of a 4 byte LC-trie node, the largest branching factor is 21. If we were to choose this root branching factor, the first level would have about 2 million nodes, two orders of magnitude more than the number of entries in the forwarding table. Clearly, a large proportion of these nodes will never be accessed.

In Table 1, we show how the 2^k nodes of the first level are divided into the 4 categories: exact matches (i.e., number of rules of length k), prefixes (i.e., there are rules of length greater than k having prefixes of size k), nodes that are prefix expansions, and nodes that do not correspond to any rule at all. The data is for a Mae-West table from January 1st, 2002 [8] with 26664 entries (1862 entries that are prefixes of one or more of these entries have been removed as explained earlier) with the root branching factor k varying from 8 to 16. We also show the total number of nodes in the LC-trie. As can be seen, there is no rule of prefix less than 8 (no expansion at $k = 8$) and the number of unused nodes grows exponentially (in fact more than doubles with every increment of k), reaching 88% of the first level and more than 50% of the total trie size at $k = 16$. In order to keep a reasonable size for the LC-trie, we will limit our experiments with k between 8 and 16.

Root bf	Matches	Prefixes	Expansions	Unused	Total at 1st level	Total # nodes
8	4	95	0	157	256	55537
9	0	171	8	333	512	55289
10	1	308	16	699	1024	55761
11	1	565	34	1448	2048	56189
12	8	1016	70	3002	4096	57807
13	15	1766	156	6255	8191	60559
14	38	2444	342	13160	16384	66871
15	78	4239	760	27691	32768	80461
16	1816	4216	1676	57828	65536	109705

Table 1: Influence of root branching factor on LC-trie first level and total sizes

In Figures 2 and 3 we show the influence of the branching factor on the average number of memory accesses per packet. We use the same table and two synthetic traces of 1 Million packets, RandIP (Figure 2) and RandNet (Figure 3), that will be described in Section 4. As expected the number of memory accesses decreases with the branching factor, with savings in the average number of memory accesses a_{bf} of one memory access when the root branching factor grows from 8 to 16.

Although there is no locality of IP addresses in the synthetic traces, the trie levels closest to the root are accessed more frequently. It is therefore interesting to see what would be the result of introducing a cache for the index. Figures 2 and 3 show the miss rate m and the average number of memory accesses $a_{bf} \times m$ for the same table and traces when we introduce a 16 KByte, 2-way set associative cache of line size 4 bytes, i.e., a cache that can store 4 K trie nodes. The best miss rate is 0.2 for RandIP and $k = 8$ and the worst is 0.7 for RandNet

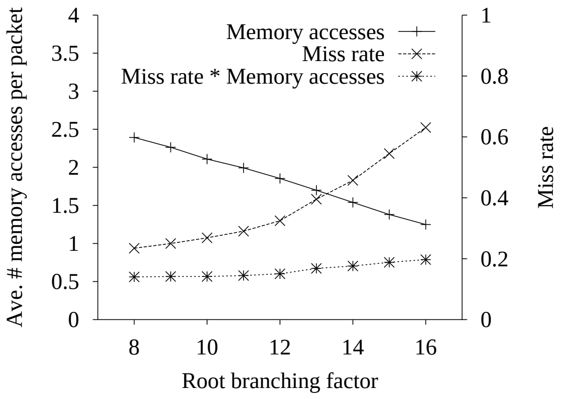


Figure 2: RandIP average number of memory accesses per packet versus root branching factor.

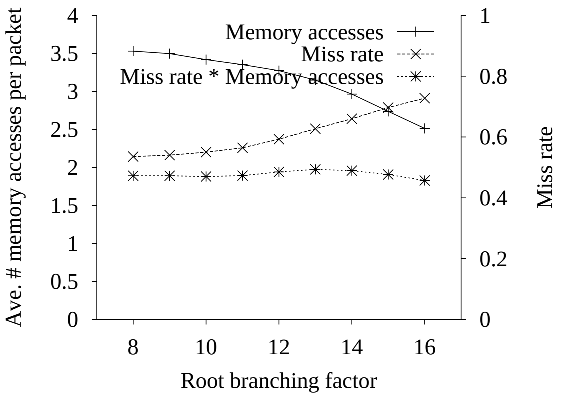


Figure 3: RandNet average number of memory accesses per packet versus root branching factor.

and $k = 16$. We also experimented with an 8 KByte and a 32 KByte cache. In the case of the smaller cache, the miss rates were noticeably worse, sometimes by more than 50%. With the larger cache, the miss rates were all the same except for RandIP and $bf = 16$ where it was 3% lower. In the remainder of the paper, we will always use a 16 KByte cache.

The main metric of interest in IP look-up is throughput (number of packets processed per cycle). Throughput is inversely proportional to the time it takes to forward a packet. In turn, the time to forward a packet depends on the number of accesses to the LC-trie. As shown in Figures 2 and 3 this number depends on the branching factor bf.

Consider first an architecture with a single processor and no cache. The time to process a packet T_{bf} can be expressed as:

$$T_{bf} = a_{bf} \times (c + L) + b \times L$$

where a_{bf} is average number of memory accesses as defined above, L is the memory latency, c the time to compute the next node index in the LC-trie (for this data structure c is the same for all levels of the trie), and the term $b \times L$ represents access to the base vector and possibly the prefix table (b is in general slightly above 1).

With a cache for the LC-trie memory of miss rate m and access time w then the time to process a packet TC_{bf} becomes:

$$TC_{bf} = a_{bf} \times (c + w + m \times L) + b \times L$$

In a memory bound environment, the significant factor is the average number of accesses to the LC-trie memory, i.e., a_{bf} for the no-cache case and $a_{bf} \times m$ if a cache is present. Despite the miss rates shown in Figures 2 and 3 that would be considered dismal for general-purpose processors, on the average between 1 and 2 “expensive” memory accesses are saved and the average number of accesses to the non-cached memory is almost the same for all branching factors between 8 and 16.

When the memory access time and the compute time are balanced, i.e., c and L are of the same order of magnitude, and w is always small with respect to L, the savings introduced by a cache will not be as important since in addition to the average number of memory accesses $a_{bf} \times m$ (almost constant over the range of bf) we must now also take into account the compute cycles $a_{bf} \times c$ (smaller for large bf).

The analysis in the previous paragraphs holds only for a single processor. Our main interest though is in a multiprocessor environment. The miss rate m will remain essentially the same with extremely minor differences that might arise due to the sequencing of accesses to memory. However, contention for access to the LC-trie memory will be reduced in the presence of a cache. We will return to this aspect of the overall performance in the next section.

We also have to be aware that the gains in throughput in the presence of a cache will be tempered by the fact that for each packet we have to spend $b \times L$ cycles accessing the memory that holds the base vector and the prefix table. There is no point in caching elements of these data structures since they are accessed randomly. This is reflected in the architecture presented in the next section.

3 A multiprocessor architecture for IP look-up

3.1 Architecture

The basic architecture of our multiprocessor for IP forwarding is shown in Figure 4. Its features include:

- A control processor - for building the look-up structures
- Microengines - for performing the look-ups
- Two memory channels - one holds the LC-trie index, and might have a cache, and the other holds the other structures, including the route and prefix tables.

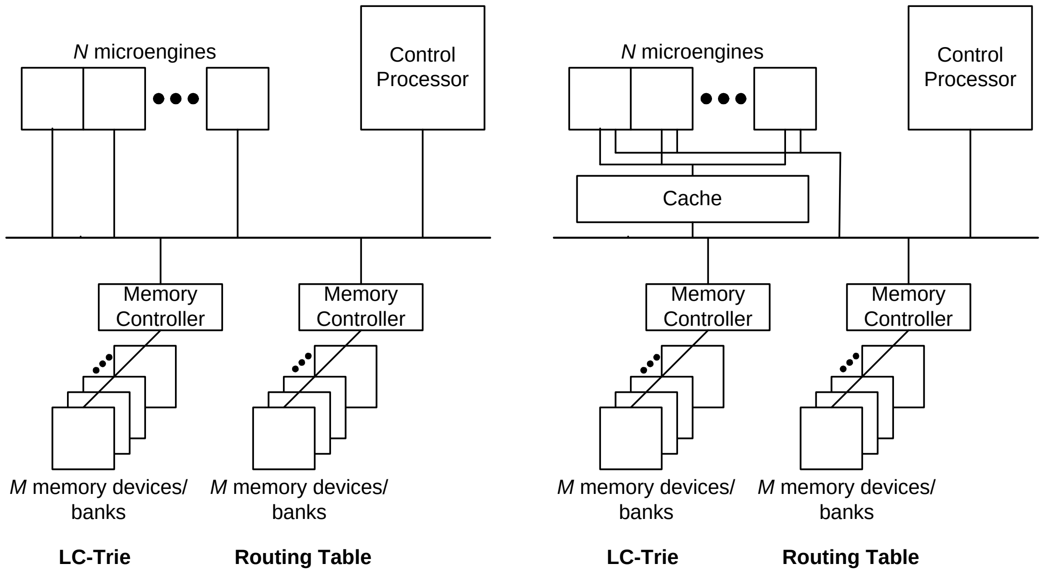


Figure 4: Multi-engine architecture

When there is a cache, only nodes from the LC-trie will be cached. While there is some locality in accessing the index as shown in Figures 2 and 3, there is none at all in the access to the base vector. Sharing the cache between these two structures would certainly be detrimental.

As indicated in the figure, we vary a number of architectural parameters in our experiments. Specifically, we consider:

- Whether a cache for the LC-trie index is present or not. If it is, it will be a 16 KByte, 2-way set-associative, 4 byte line size, LRU replacement, 1 cycle access time cache
- The number of processors (1, 2, 4, 8, 16)
- The latency of memory (12, 42, 100 cycles)
- The number of memory banks (1, 2, 4, 8)

We set the latency of buses to be 2 cycles from the processors to the memory system, allowing 1 cycle to model bus arbitration and 1 cycle to transfer data. The latency of buses from the memory or cache is set to 1 cycle since we assume bus arbitration is not required in this case. The cache is shared amongst all processors and hence needs to be lock-up free. Our experiments show that contention at the cache level does not affect performance.

This architecture has a number of features in common with commercial network processors, such as the Intel IXP2800. Most notably, this architecture employs multiple simple processors, called microengines here, to exploit packet-level parallelism [3]. The experiments in this paper investigate how to use, and modify, such an architecture for fast IP look-ups. These conclusions are of interest to IXP2800 programmers as well, since the programmer must decide how many microengines and memory channels to devote to a given task.

3.2 Impact of Multibanking

In a single processor environment, the goal is to reduce the number of accesses to the memory holding the LC-trie. In the case of multiple engines we want also to reduce the contention in the concurrent memory accesses. This can be partially achieved by using memory banking where the various banks can retrieve data in parallel.

A simplistic analysis in the case where compute time between memory accesses is small gives nonetheless an idea of the impact that banking can have. Consider the following analogy. Let a memory access be represented as choosing a ball from an urn. The urn contains balls of b colors where there are as many colors as there are banks. There is a very large number of balls compared to b and to n, the number of processors that are going to pick up balls, and balls are equally distributed among the b colors. That is, the probability of choosing a ball of a given color is $1/b$. At each step of the computation, each of the n processors chooses a ball; only one ball of each color can be chosen at each step. So, if 2 processors choose a ball of the same color, one of them has to be returned in the urn.

If there is only one processor, only one ball can be picked up at a given time. Similarly, if there is only one bank, all balls but one will be returned, regardless of the number of processors. Now if there are n processors and b banks, the expected number of balls picked up will be: $e = \sum_{i=1}^{b} i \times P_n(i)$ where $P_n(i)$ is the probability that i balls of different colors will be picked by the n processors. That is, the expected number of balls not returned is e. Couched in terms of n processors and b memory banks, e is the speed-up in execution time or increase in throughput a single processor.

For example, with 2 banks and 2 processors, $P_2(1) = 0.5$ and $P_2(2) = 0.5$ yielding $e = 1.5$. Working up the combinatorics show that $e = 1.87$ for 4 processors and 2 banks and $e = 2.85$ for 4 processors and 4 banks.

In reality, the contention is not as bad as indicated in the previous paragraphs since after a memory accesses, on average, the processor will have to access the forwarding table, thus freeing an LC-trie memory time slot for the other processors. For example, in the case of RandIP and a branching factor of 16, the processors split almost evenly the accesses to the LC-trie and those to the base vector. Thus for n processors, there is a diminishing return in having more than $n/2$ banks. This is even more true for the LC-trie memory if we introduce a cache since now the contention is almost halved (miss factor is about 0.5). Note however, that as the number of processors increase, so does the contention for base vector access and for that data structure access is completely random and no caching can help.

In the next section, we present simulation results that give a more accurate view of the impact on throughput due to the data structure parameter *bf*, the number of processors, and the memory hierarchy parameters (latency, caching and multibanking).

4 Experimental results

4.1 Methodology

We use trace-driven simulation to assess the performance of variations of the multiprocessor architecture of the previous section. A trace represents a sequence of LC-tries searches. A search consists of a sequence of tuples (compute time, memory access) where the compute time corresponds to the determination of (1) whether the longest prefix match has been obtained, and (2) in case it is not of the index of the LC-trie node to be searched subsequently. When a match is obtained (a leaf of the trie has been reached), the forwarding table is accessed and the prefix table is also searched if so required. The compute time of the access to the first level is 13 cycles (the root node is kept in a register) and subsequent compute times, all the same, are 15 cycles (this was determined by instrumenting the look-up function in [10]).

In order to determine the memory accesses, we use two synthetic traces, RandIP and RandNet [9]. Both traces, 1 million packets each, only contain IP addresses for which there exists a matching rule in the routing table. This is consistent with real traces where very few addresses will not match any rules and the default route has to be used.

These synthetic traces represent two different approaches to generating random traffic for a given route table. RandIP generates a random IP address and checks to see if it matches a rule in the routing table. If the address does match a rule, it is added to the trace. The process is repeated until the trace contains the required number of addresses. RandNet randomly chooses a rule in the routing table, then extends this prefix to 32 bits (for IPv4). RandIP tends to produce traces in which short prefix rules predominate because a randomly generated number is more likely to match eight bits rather than 24. As shown in [9], RandNet has similar characteristics to a core router packet trace, whereas RandIP is more like an edge router packet trace. Note, however, that neither trace includes the small amount of temporal locality found in real traces, thus biasing even more against the use of caches. The synthetic traces are pessimistic with respect to real traces, adopting a more even distribution of packets amongst all rule lengths.

4.2 Simulation Experiments and Results

4.2.1 Impact of the branching factor

In Section 2 we presented data on the influence of the branching factor bf on the average number of memory accesses a_{bf}. We also presented an initial analysis of the impact of the presence of a cache on the time to process a packet, and henceforth on throughput. Figures 5 and 6, showing the results of simulating the traces in both a cache and a no-cache single engine environment, confirm the analysis.

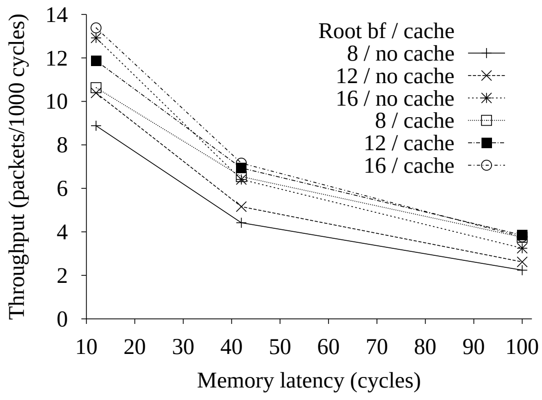


Figure 5: RandIP 1 processor throughput versus memory latency.

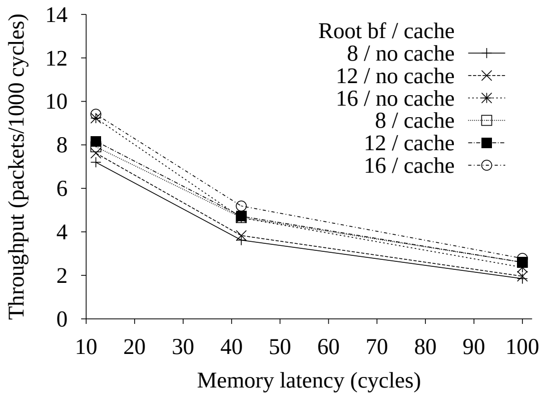


Figure 6: RandNet 1 processor throughput versus memory latency.

More precisely, the following conclusions can be reached:

- In the absence of a cache, bf is the most important factor. Throughputs for $bf = 16$ are significantly better than those for $bf = 12$ or $bf = 8$ for all 3 memory latencies considered. For RandIP at all latencies, the improvement in throughput is 17% when bf increases from 8 to 12 and 46% from 12 to 16. For RandNet, the improvement is 6% and 21% respectively.
- In the presence of a cache, the value of bf has much less importance. At long latencies, all bf's yield the same throughput and at low latencies, a larger bf is still better but relatively less so.
- In all cases, a cache improves throughput. For example when the root branching factor is 12 and the memory latency is 100 cycles, a cache improves throughput by 47% for RandIP and 33% for RandNet.

We observe that root branching factors of 8 and 12 give approximately the same throughput with a slight advantage to root branching factor 12 (this observation remains true for all the experiments that we run). Since from Table 1 we can see that these two root branching factors require the same amount of LC-trie memory, we will from now on consider only $bf = 12$ and $bf = 16$.

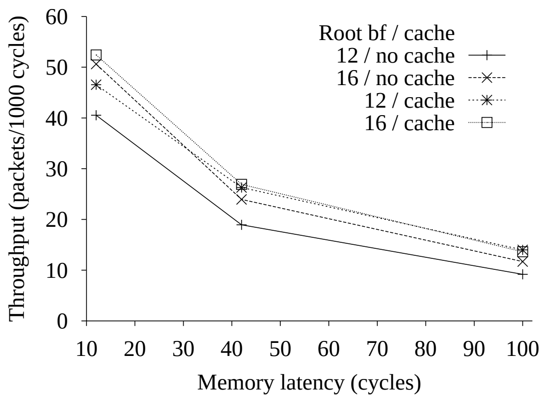


Figure 7: RandIP 4 processors and 4 memory banks throughput versus memory latency.

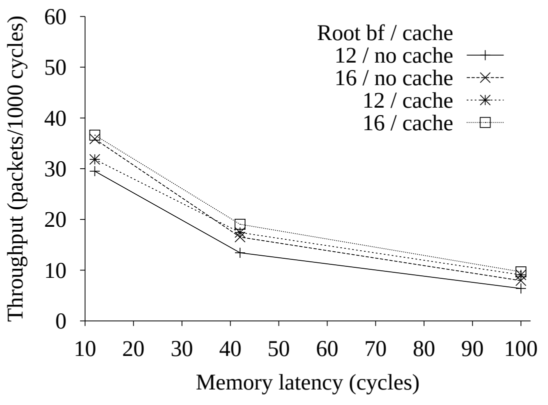


Figure 8: RandNet 4 processors and 4 memory banks throughput versus memory latency.

It is important to see if the conclusions we reached regarding the branching factor for a single processor hold in the case of multiple engines. To that effect we simulated several "balanced" systems where the number of engines and of memory banks were the same. As a representative of these experiments we show in Figures 6 and 7 the throughput for a system including 4 engines and 4 memory banks. As can be seen, the three points set previously hold also in a multiprocessor environment.

Looking at the performance for 1 and 4 processors in Figures 4 and 6 for RandIP and Figures 5 and 7 for RandNet, we see an approximate 4-fold increase in throughput when scaling from 1 to 4 processors. In the next subsection, we investigate more thoroughly the impact of scaling the architecture.

4.2.2 Impact of scaling the architecture: processors and memory banks

The current trend in network processors is to increase the number of microengines. For example, in the Intel IXP we see an increase from 6 to 16 engines from one generation to the next. It is therefore of interest to see if forwarding throughput can scale with the number of microengines devoted to it. To this effect, we simulated the same workload, increasing the number of processors up to 16 but keeping the number of memory banks set to 4. The results for RandIP are shown in Figures 8 and 9 (similar curves are obtained for RandNet). In these figures, the throughput is normalized to the case of 1 processor, no cache and $bf = 12$.

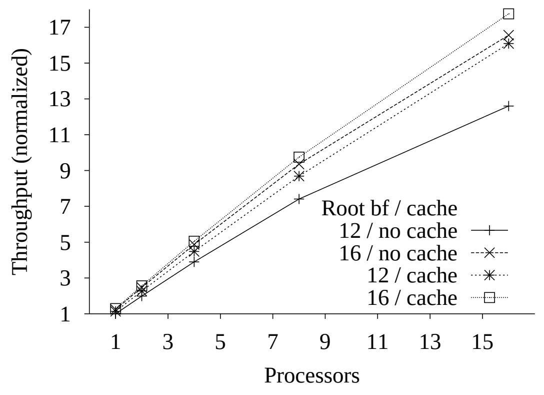


Figure 9: RandIP 12 cycle memory latency, 4 memory banks throughput versus processors. Throughput normalized to 1 processor, $bf = 12$, no cache.

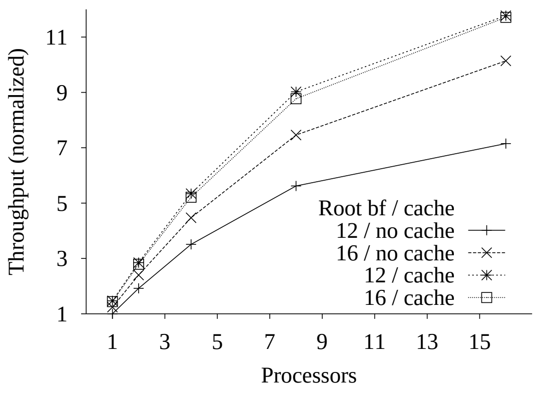


Figure 10: RandIP 100 cycle memory latency, 4 memory banks throughput versus processors. Throughput normalized to 1 processor, $bf = 12$, no cache.

As can be seen, there is an almost linear increase in throughput until we reach 8 processors. After that, the contention for memory access starts to take its toll, mostly in the case of an architecture without a cache. This is consistent with the analysis of Section 3.1. With 8 processors and no cache there is limited contention with 4 banks and much more when

16 processors vie for memory access. When there is a cache, the contention is halved and the combination of 16 processors and 4 banks is still sufficiently balanced. The important point, though, is that adding processors to the forwarding task will improve performance. Moreover, at long latencies, if we have a cache we can reduce the number of processors and achieve approximately the same throughput. For example, a configuration with 8 processors and a cache has sometimes slightly better (with $bf = 12$) and sometimes slightly worse (with $bf = 16$) throughput than a configuration with 16 processors and no cache.

Since too small a number of memory banks might limit performance, we performed experiments where we fixed the number of processors and varied the number of banks.

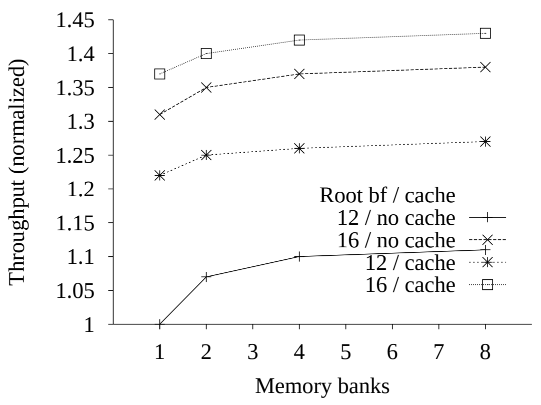


Figure 11: RandIP 12 cycle memory latency, 4 processors throughput versus banks. Throughput normalized to $bf = 12$, no cache.

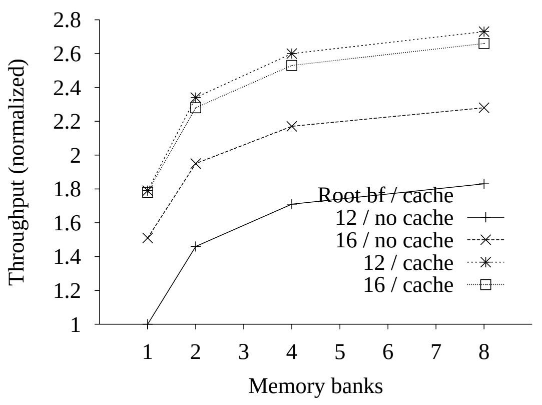


Figure 12: RandIP 100 cycle memory latency, 4 processors throughput versus banks. Throughput normalized to $bf = 12$, no cache.

In Figures 11 and 12 we show the throughput for 4 processors for RandIP with a varying number of banks (experiments with RandNet yield the same overall picture). The throughput is normalized to the case of 4 processors and a single bank with $bf = 12$. For large memory latencies and no cache, passing from 1 bank to 2 yields a 46% improvement and from 2 to 4 another 17% (the percentages are 62% and 29% for RandNet). With a cache the improvement is 30% and 11% respectively (49% and 18% for RandNet). As can be expected, the improvements are relatively smaller for low latencies.

Scaling the architecture brings forth three important points:

- All other parameters being equal, caching improves throughput in all cases. For example, sometimes by over 60% as for $bf = 12$ for 8 and 16 processors.

- The architecture scales well: the number of microengines devoted to forwarding can be increased with an almost linear increase in throughput.
- Although of secondary importance compared to caching, the impact of multibanking is not negligible. An interesting observation is that the performance achieved with an architecture with caching and $b/2$ banks is better than that with an architecture with no cache and b banks.

4.3 Discussion

From a performance viewpoint, the results of our experiments indicate that IP look-up throughput will be best when using an LC-trie index if (1) the branching factor is large and (2) we have a multiengine architecture with as many engines as is practical, a cache, a memory with low latency, and a number of banks being about one fourth the number of processors.

However, some of these desired features interfere with each other. For example, having a low memory latency is synonymous with having on-chip memory. Unfortunately, scaling the architecture (more micro engines) leaves less on-chip real estate for the memory and using a large branching factor requires more memory (recall Table 1). Note also that we have not addressed the problem of updates which, in general, will be performed in batches and might involve two copies of each data structure: one for active use and one for updating. Accommodating updates in this way will double the required amount of memory.

If we assume that there is a limited amount of memory that we can put on-chip, as is the case for the current generation of network processors, then it is best to organize it as a cache. In the results that we report, we have considered a single-ported, shared cache. We have experimented with various cache structures – multiported, multibanked, private, and shared – and found no significant differences in throughput so the cheapest implementation is adequate. Furthermore, the hit rate in the cache is independent of the number of engines and therefore the cache capacity does not have to scale with the number of engines. Since in current systems the index memory is off-chip (long latency) we don't have to choose a large branching factor because, as we saw, its value does not matter much, performance-wise. A branching value of, say 12, will save some off-chip memory needs.

Therefore, a possible configuration would be: 16 microengines, a 16K on-chip cache, and an off-chip memory with 4 banks holding an LC-trie of branching factor 12 (less than 1 MByte for tables of up to 100,000 entries), in addition of course to the off-chip memory holding the base vector. We can give a *rough estimate* of the absolute throughput of this IP look-up engine as follows. We assume an implementation consistent with current NPs, i.e., 1 GHz microengines and an off-chip memory latency of 100 cycles. According to Figures 4 and 5 a single microengine without a cache could sustain a throughput of about 2 Million packets/sec or 1 Gbit/sec (less than what is required by OC-48). A 16 processor configuration without a cache would improve throughput by a factor of 7 (Figure 10). Adding a cache would almost improve by another factor of 1.65 (Figure 10), resulting in a 12x improvement overall. At 24 Million packets/second (12 Gbit/sec), we would fulfill OC-192 requirements for the IP look-up, the most time consuming portion of forwarding.

5 Previous work

Previous work in the area of IP look-up can be divided into hardware-oriented solutions and software approaches. Since our architecture is programmable, we emphasize the latter.

An attractive hardware solution is to use ternary CAMs (ternary because of the importance of don't cares in the LPM setting). However, CAMs are expensive, require high-power, and updating them is difficult[4]. With base vectors of tens of thousands of entries, this approach is not feasible [14]. Many current routers use ASICs. Two examples with widely different designs are Cisco's Toaster and the Iflow processor. Toaster 2 [7] uses 16 microcoded processors arranged in a matrix of 4 rows by 4 columns and working in a pipeline fashion. Routing tables are off-chip. The Iflow processor [11] uses large embedded DRAMs as well as three rows of SRAMs to hold the first 3 levels of a B-tree representing the index. The LPM is pipelined over the SRAMs and the DRAM.

Previous work on software approaches has been focused on designing data structures and algorithms to bound the worst-case latency by minimizing levels in an index structure for a given amount of memory and reduce the size of the index. The intent is to have general-purpose processors perform the forwarding function. The two main techniques use respectively tries [15] and binary searches on hash tables [16]. While the LC-trie method [10] does not result in the most compact trie representation, it is competitive and easier to build and update. The main differences between these studies and ours is that we consider a multiprocessor environment and consider the impact of caching on throughput. This is consistent with the trend in current network processors, e.g., Intel IXP and IBM NP, where multiple programmable engines are devoted to specific tasks [5].

The synthetic traces RandIP and RandNet are introduced and their characteristics are compared to those of real traces in [9]. This paper also uses average throughput as a metric and models and validates cache performance when varying the branching factor of the root. The main differences with our study is that in [9] the emphasis is on validating an L2 cache model when the root branching factor can be very large, thus yielding extensive on-chip and off-chip memory requirements for the index, while we try and limit the on-chip memory to stay in the spirit of current network processors. The study is also only for a single processor.

A totally different approach to caching is introduced in [1, 2]. In these papers, the cache hierarchy holds the most recent IP addresses translations [1] or ranges thereof [2]. The specialized cache design takes advantage of the predominance of some prefix lengths by selecting wisely the bits that will index the cache. Because of this last constraint, the method is more beneficial when tuned to local environments as for example in edge routers.

A recent study [13] investigates the use of a wide word pipelined memory that allows concurrent accesses. This is an interesting alternative to the multibanked shared memory that we have been assessing but performance comparisons are yet to be done.

6 Conclusion

In this paper we have investigated the throughput performance of a multiprocessor architecture dedicated to fast IP look-ups. Our results show that when a trie-based data structure with prefix expansion (e.g., LC-trie) is used on a network processor with multiple microengines, a cache can be used to effectively increase throughput while decreasing the need for more memory banks.

Specifically, the cache reduces the number of references to external memory by taking advantage of the locality among references to the nodes in the upper levels of the trie; even completely random accesses to index entries create some amount of locality among nodes nearest to the root. Although hit rates are low, reducing the number of external memory references has two positive effects: reduced memory latency for those references that hit in the cache, and reduced contention for external memory for those references that do not hit in the cache.

We found that throughput scales almost linearly with the number of processors as long as memory contention is not serious. Throughput is greatly enhanced, up to 65% in some cases, by introducing a cache for the index structure.

In future work, we plan to extend this investigation to other applications of the longest prefix match algorithm. One challenging example can be found in firewalls or distributed denial of service detection systems where extremely large legitimacy lists (often hundreds of thousands) of valid or invalid hosts must be maintained. This application occurs near the edge of the network, where packet arrival rates are lower, but other challenges emerge: updates are generally much more frequent, and matching must often be done in multiple dimensions (e.g., both source and destination addresses).

Acknowledgments : This work was supported by NSF Grant CCR-0072948 and a gift from Intel Corporation.

References

[1] T. Chiueh and P. Pradhan. High-performance IP routing table lookup using CPU caching. In *Proc. IEEE Infocom*, pages 1421–1428, 1999.

[2] T. Chiueh and P. Pradhan. Cache memory design for network processors. In *Proc. 6th Int. Symp. on High-Performance Computer Architecture*, pages 409–418, 2000.

[3] P. Crowley, M. Fiuczynski, J.-L. Baer, and B. Bershad. Characterizing processor architectures for programmable network interfaces. In *Proc. Int. Conf. on Supercomputing*, pages 54–65, 2000.

[4] D.Shah and P.Gupta. Fast updating algoritms for TCAMs. *IEEE Micro*, 21(1), Jan. 2001.

[5] The role of network processors in next generation networks. http://developer.intel.com/design/network/papers/279048.htm, 2001.

[6] D. Knuth. *The Art of Computer Programming. Vol 3, Searching and Sorting.* Addison-Wesley, Reading, Ma, 1973.

[7] J. Marshall. Cisco systems - Toaster 2. In P.Crowley et al., editor, *Network Processor Design*, pages 235–248. Morgan-Kaufmann, 2003.

[8] IPMA reports. http://www.merit.edu/ipma/reports, 2002.

[9] G. Narlikar and F. Zane. Performance modeling for fast IP lookups. In *Proc. of ACM SIGMETRICS*, 2001.

[10] S. Nilsson and G. Karlson. IP-address lookup using LC-Tries. *IEEE Journal on Selected Areas in Communications*, 17(6):1083–1092, June 1999.

[11] M. O'Connor and C. Gomez. The iFlow address processor. *IEEE Micro*, 21(2):16–23, Mar. 2001.

[12] L. Peterson and B. Davie. *Computer Networks: A systems approach - 2nd Ed.* Morgan Kaufman, San Francisco, Ca, 2000.

[13] T. Sherwood, G. Varghese, and B. Calder. Suporting network algorithms with a pipelined architecture. In *Proc. of ISCA 2003*, 2003.

[14] S.Keshav and R. Sharma. Issues and trends in router design. *IEEE Communications magazine*, pages 144–151, May 1998.

[15] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM TOCS*, 17(1):1–40, Feb. 1999.

[16] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high-speed prefix matching. *ACM TOCS*, 19(4):440–482, Nov. 2001.