Evaluating and Optimizing IP Lookup on Many core Processors

Peng He, Hongtao Guan, Gaogang Xie, Kavé Salamatian

HAL Id: hal-00737774
https://hal.science/hal-00737774

Submitted on 2 Oct 2012

Evaluating and Optimizing IP Lookup on Many core Processors

Peng He[*][†], Hongtao Guan[*], Gaogang Xie[*], Kavé Salamatian[‡]
[*]Institute of Computing Technology, Chinese Academy of Sciences, China
[†]Graduate University of Chinese Academy of Sciences, China
[‡]LISTIC-PolyTech, Université de Savoie,
France

Abstract—In recent years, there has been a growing interest in multi/many core processors as a target architecture for high performance software router. This is a clear difference from the previous trend to use dedicated network processors and hardware components. Because of its key position in routers, hardware IP lookup implementation has been intensively studied with TCAM and FPGA based architecture. However, increasing interest in software implementation has also been observed. In this paper, we evaluate the performance of software only IP lookup on a many core chip, the TILEPro64 processor. For this purpose we have implemented two widely used IP lookup algorithms, DIR-24-8-BASIC and Tree Bitmap. We evaluate the performance of these two algorithms over the TILEPro64 processor with both synthetic and real-world traces. After a detailed analysis, we propose a hybrid scheme which provides high lookup speed and low worst case update overhead. Our work shows how to exploit the architectural features of TILEPro64 to improve the performance, including many optimization in both single-core and parallelism aspects. Experiment results show by using only 18 cores, we can achieve a lookup throughput of 60Mpps (almost 40Gbps) with low power consumption, which demonstrates great performance potentials in many core processor.

I. INTRODUCTION

Flexibility, scalability and high performance are three mandatory characteristics for modern network routers. In order to achieve these desirable properties, several papers have proposed to use software based router achieving high level of flexibility in place of traditional dedicated hardware architecture and take advantage from parallelism in many-core or multi-processor to achieve the needed performance in packet processing [1], [2], [3]. While there exists a large literature on the usage of dedicated network processors, like the Intel IXP series, however these processors have been heavily criticized for the complexity of their execution model and the difficulty of software development. In contrast, multi/many core chips systems targeted to network applications are becoming widely available at reasonable price and offer a familiar programming environment which allows easy implementation of customized packet processing. While there is a relative large literature about these systems, it mainly looks at the architecture level, and little evaluation of concrete forwarding tasks inside the router are provided.

Among all the forwarding tasks in the data-plane of a router IP lookup is obviously a critical one. Routing tables in nowadays core routers can easily grow to several hundred thousand of prefixes, resulting in large FIB (forwarding information base) sizes and slow lookup speed. The last years have seen, several new IP lookup schemes to overcome the challenge posed by IP lookup with continuously growing routing table and ever increasing line speed. These new lookup approaches include hardware TCAM based solutions [4] as well as algorithmic solutions [5], [6], and even a hybrid solution [7]. TCAM based solutions, while fast, are relatively expensive and have high power consumption, while algorithmic solutions need several memory accesses per lookup that results in slower lookup speed.

In this paper, we will evaluate the performance of algorithmic based IPv4 lookup algorithms on a popular highly multi-core processor, the TILEPro64 processors. TILEPro64 processors contain 64 full programmable processing cores. A full TILEPro64 development board that can support up to 8×1 Gbps plus a 1×10 Gbps Ethernet Interface costs currently several thousands dollars, making this platform affordable for practical usage as a software router. Indeed more powerful processors are available these days meaning that the results presented in this paper are just lower bound on potential IP lookup speeds.

For the software IP lookup we choose two simple and practical algorithms, DIR-24-8-BASIC[8] and Tree Bitmap[9]. DIR-24-8-BASIC is used in many software router prototypes, such as RouteBricks[2], PacketShader[3] *etc.*, while Tree Bitmap is a well-known IP lookup algorithm with low memory footprint and fast lookup speed. Other algorithms are either too complicated or not suitable for the multicore platform. For example, Bloom Filter based IP lookup needs hardware implementation of several hundreds of hash functions, that will need dedicated FPGAs, while our aim in this paper is to study a software only implementation on a many-core platform.

In order to get a full understanding of performance of different algorithms, we do our evaluation experiments on a routing table issued from RouteViews project[10] and containing about 358K prefixes. We've found that, in a single core environment, the DIR-24-8-BASIC algorithms run at least 3 times faster than Tree Bitmap for all IP traces. However, the FIB size generated by Tree Bitmap is almost 20 times lower than DIR-24-8-BASIC. In the parallel experiments, we have observed that the run-to-complete execution model is superior to the pipeline model. Our experiment shows that, by using only 18
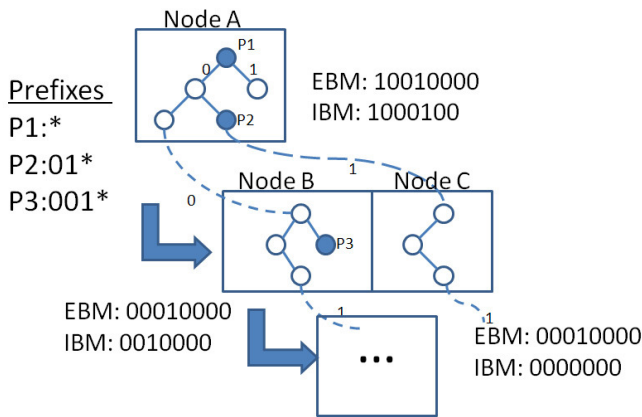
Fig. 1. The Tree Bitmap Algorithm

Fig. 2. The DIR-24-8-BASIC Algorithm

cores out of 64 cores on TILEPro64, we can achieve a lookup throughput of up to 60Mpps (almost 40Gbps for 64 bytes per packet) with a power consumption of less than 20W[11], to be compared with 240W for GPU based PacketShader. Moreover as the packet processing is done directly on the TILEPro64 processor the lookup delay is very small compared to the delay needed for batching in PacketShader.

The contributions of this paper can be summarized as follows: 1) we describe and evaluate how IP lookup algorithms can be implemented in practice on a many-core processor—TILEPro64. We implemented various optimization tricks, including both algorithmic refinements and architecture specific optimizations. 2) We measured the performance of different IP lookup algorithms on many core chips using different traces. 3) Based on our evaluation results, we propose a hybrid scheme to combine the strengths of two algorithms. This hybrid scheme has the similar performance with DIR-24-8-BASIC on single-core but has a much smaller update overhead in the worst case. The remainder of this paper is organized as follows. In Section 2, we will provide some background, including the two algorithms and the TILEPro64. In Section 3, we will present our implementation and the implemented optimizations. In Section 4, we will report our hardware setup and experimental evaluation. In Section 5, we will present a hybrid IP lookup scheme and evaluate its performances. The paper will conclude in section 6.

II. BACKGROUND

A. The Tree bitmap algorithm

The *Tree Bitmap* algorithm is a multi-bit trie IP lookup algorithm using a clever encoding scheme. Fig. 1 shows an example of a 3-bit stride Tree Bitmap trie. In Fig. 1, we can see the whole binary trie is divided into several multi-bit nodes having two bitmaps, the internal bitmap (IBM) and the external bitmap (EBM). The IBM is used to represent the prefixes stored in this multi-bit node, and the EBM is used to represent the position of the child of this multi-bit node.

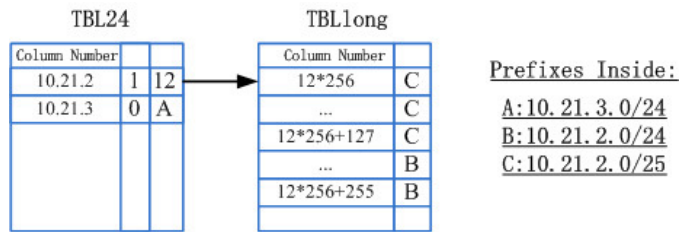We use the Node A as an example to show the encoding scheme of the IBM and the EBM. A 3-bit sub-trie has 8 possible leaves. In Node A, only the first and fourth leaves have the pointers to children. Thus the EBM of this node is 10010000. The IBM has 1 bit for every stored prefixes. It has one bit for prefixes of length 0, two following bits for prefixes of length 1, and four following bits for prefixes of length 2. In Figure 1, we have two stored prefixes, P1= * and P2=01*. P1 is a prefix of length 0, so we set the first bit of IBM to 1, P2 is the second prefix of length 2, so we set the fifth (1+2+2) bit of IBM to 1. Thus the IBM of this node is 1000100. To note, a K stride node has a 2^K bit EBM and $2^K - 1$ bit IBM.

Suppose we have a bit sequence 011111 to be searched. The bit sequence consists of two 3-bit sub bit sequence, 011 and 111. The searching starts from Node A with bits 011. As mentioned, 011 means the fourth "exit point" in Node A. The fourth bit of EBM is 1 and the number of 1s to the left of the fourth bit is 1. So we move to the Node C (the 1st (starting from 0) node in the next level node array) with 111. The eighth bit (111) of EBM in Node C is 0, which means we can't continue to the next node. We begin next, to check if inside the traversed nodes, there is any matching prefix. Since the IBM of Node C is all zero, we turn back to Node A searching bits 011. We successively remove the right-most bits of the bit sequence, and check the corresponding bit position in IBM, until we find a 1 in that position. For example, we get 01* in the first iteration, we check the fifth bit of IBM and find a matched prefix P2.

B. The DIR-24-8-BASIC algorithm

Compared to the Tree Bitmap algorithm, *DIR-24-8-BASIC* is much simpler. It uses two tables to store all the prefixes. The first table, *TBL24* which uses the first 24 bits of an IP address as an index, stores all the prefixes with length shorter than 25 bits. If more than one prefixes share the same first 24 bits, the corresponding entry of these prefixes in *TBL24* is filled with a pointer pointing to a 256 entries block in the second table, *TBLlong*, storing all the possible suffix of the left 8-bits. When there is only a single prefix with matching first 24 bits, *TBL24* contains the next hop information. However *TBLlong* always contains the next hop information.

An example is shown in Fig. 2, where no prefixes share the first 24 bits of Prefix 10.21.3.0/24, thus the egress A is directly stored in *TBL24*; Prefix 10.21.2/24 and 10.21.2.0/25 has the same first 24 bits, thus the corresponding entry in *TBL24* stores a pointer which points to the 12th block in *TBLlong*. When searching an IP address, we first use the

type	clock frequency	cache size
TILEPro64	700MHz	L2 64KB / L3 4MB
E5506	2133MHz	L2 1MB / L3 4MB

type	cache system	penalty(cycles)
TILEPro64	distribute	L1 8 / L2 $30 \sim 80$
E5506	share	L1 $14 \sim 15$ / L2 ~ 100

first 24-bits of IP address as an index to read one entry of Table *TBL24*. Depending on the content of *TBL24*, the lookup is terminated or we proceed to table *TBLlong* following the pointer in *TBL24*.The leftmost 8-bit of the IP address are used to obtain the index of the prefix in Table *TBLlong* and access it with one more memory access. Since currently, most of prefixes have length less than 25 bits in the core routing table, it only takes one memory access to do any IP lookup.

C. The TILEPro64 architecture

TILEPro64 is a many-core processor based on Tile Architecture that consists of a 2D grid of homogeneous computing elements, called tiles or cores. Each tile is a full-featured CPU that can independently run an entire operating system. As the name implies, TILEPro64 consists of 8×8 cores, that is much larger compared to mainstream multi-core processors, which usually have only $4 \sim 8$ cores. However, TILEPro64 cores have differences with for example an Intel Xeon cores. Table I lists the differences between a TILEPro64 core and an Intel Xeon E5506 one.

As can be seen from Table I and Table II, a TILEPro64 core is relatively weaker than one in an Intel Xeon E5506. Therefore, while we can assign heavy processing to a single Intel Xeon core, *e.g.* all the processing of a software router's dataplane, including the decoding, IP lookup, checksums, *etc.* in a single thread on a single core of a Xeon E5506, on TILEPro64 we split different dataplane activities between several cores. Following this, we have assigned entire cores of the TILEPro64 to only do IP lookup. As the programmable on-chip network on TILEPro64 can be used to eliminate the communication overhead of adding to other cores other dataplane activities, and the distributed cache system ensures that the cache isolation, we can evaluate the IP lookup load independently of the other activities of the control plane that will be assigned to other cores.

III. IP LOOKUP ON TILEPRO64

In this section, we present our implementation and detail the optimization tricks we used.

A. Implementation

Tree Bitmap: We implemented two versions of this algorithm, *bitmap* and *bitmap_5*. The *bitmap* implementation uses the built-in type *uint16_t* in TILE64 core to store the bitmaps inside the multi-bit node. This implementation is specially suitable for Tree Bitmap with 4 bits stride as it eliminates the overhead of querying the stride information during the lookup process. The *bitmap_5* implementation is more general and uses array *uint32_t* type to store the bitmaps. This implementation can be tuned to any trie with 5 bits or more strides. In both implementations, a single 32-bit pointer is used to point to both the child and result arrays. Therefore each node of *bitmap* needs $2 \times 2 + 4 = 8$ Bytes, and each node of *bitmap_5* costs $2 \times 4 \times 2^{(stride-5)} + 4$ Bytes.

DIR-24-8-BASIC: We implemented *DIR-24-8-BASIC* using 32-bits integer for each entry of both *TBL24* and *TBLlong*. For each entry of Table *TBL24*, one bit is used as a flag to signal if this entry point to *TBLlong* or if it is a definitive prefix, 5 bits are used to store the prefix length, and 26 bits are used to store an index or a pointer to the next-hop information. The 26 bits index is necessary for lookup and update. In each entry of table *TBLlong*, 5-bits are used to store the prefix length, and the leftover bits are used as a pointer to the next hop information. As table *TBL24* needs 2^{24} entries, our implementation of DIR-24-8-BASIC needs at least 64MB DRAM memory (4 bytes per entry in table *TBL24*).

B. Optimization tricks

Large page Rather than using the by default memory page size of 4 KBytes we have used in our development large page, which contains 16MB memory space. This optimization reduces the TLB misses during the IP lookup, than can become very important for DIR-24-8-BASIC that uses a large amount of memory. This is even beneficial for the Tree Bitmap that uses much less memory, since it reduces the TLB misses. Our experiment shows that this implementation detail improves highly the performance the lookup by decreasing lookup time by almost 20%.

Initializing an array for trie A way of improving the lookup speed is to implement a lookup table for the first consecutive bits in the trie-based IP lookup. For example, for the first 13 bits of an IP address, we build an initial array with 8K entries that enables fast access to the node storing these prefixes. The array speeds up the lookup, however it increases the update overhead. In our implementation of Tree Bitmap, we have used such an array both in *bitmap* and *bitmap_5*.

Counting the number of 1's in a bitmap The Tree Bitmap algorithm needs to count how many 1's in one bitmap. This task can easily be done in hardware. However, in software, thing is more complex and one have to use a lookup table to get the number of 1's in one bitmap. This adds more memory accesses during the lookup and degrades the performance. Fortunately, the TILEPro64 processor implements a set of bit manipulation instructions that can count the number of 1's in hardware. We use these instructions to count the number of 1's in one bitmap.

Lazy checking As mentioned above, one single multi-bit node can have two operations: checking the EBM to find the "exit point" and checking the IBM for the prefixes inside the node. Since the IBM checking is time consuming, we perform

a lazy checking, *i.e.* we only check the EBM of traversed node and we use an extra stack to store them; when the searching cannot proceed to the next node, we pop the nodes in the stack to perform IBM checking. As long as there is a single prefix match, the lookup process terminates. Our experiment shows this trick can save up 30 to 50 cycles per lookup.

IV. Performance Evaluation

In this section, we evaluate the performance of the two IP lookup algorithms with both synthetic and real world traces. We discuss our evaluation traces and the performance results in detail.

A. Evaluation traces

The nature of the traces used to evaluate IP lookup is very important. An IP trace with high locality can lead to a very high performance result because many memory accesses cached in the L2 cache of CPU can be reused. Random IP traces, while having limited locality, contain too many "invalid" IP addresses (IP address does not match any prefix in one routing table), that usually have very short searching path in the trie-based IP lookup algorithm. Using these trace can also result into "illusion" high performance. In order to get a full understanding of the performance of IP lookup in software, we use both synthetic and real world traces. We use three types of traces that are listed below:

Random Match: Let S be the set of all the prefixes in one routing table. We use the prefixes in S to construct a binary trie and we leaf pushing this trie, *i.e.* all the prefixes are stored only at the leaf nodes. We are representing as $L(p)$ the leaf nodes that stores the prefix $p \in S$. For any prefix $p \in S$, let's define a set, $P(p)$, containing all paths starting from the root node and ending at the leaf nodes that belong to $L(p)$. These paths can be viewed as the "leaf pushing" prefixes for the original prefix p. For any prefix $p \in S$, we collect the longest path in $P(p)$, and use this paths to form a new prefix set. We call this set the Random Match Set. Random Match traces are generated by repeating the following steps:

1) Choose randomly one prefix in the Random Match Set.
2) If the prefix is not 32-bit long, we use a random number to complement this prefix into a 32-bit IP address.

The Random Match trace has three characteristics: 1) it is unbiased for all prefixes in S; 2) it has low locality; 3) IP addresses in Random Match trace have the longest searching path. Thus, the performance result gathered on this trace can be considered as the worst case for all implementation. Fig. 3 shows how to construct a Random Match trace:

Realistic Random Match: We now add some locality to our evaluation trace thanks to realistic traces. We have used traces provided by CAIDA [12], and we have extracted all the destination IP addresses. Unfortunately, these IP addresses can not be used directly, because they are anonymized and many of them can't match any prefix in a real routing table. However the anonymization maintains the prefix structure. Therefore, in order to generate a trace with realistic locality,
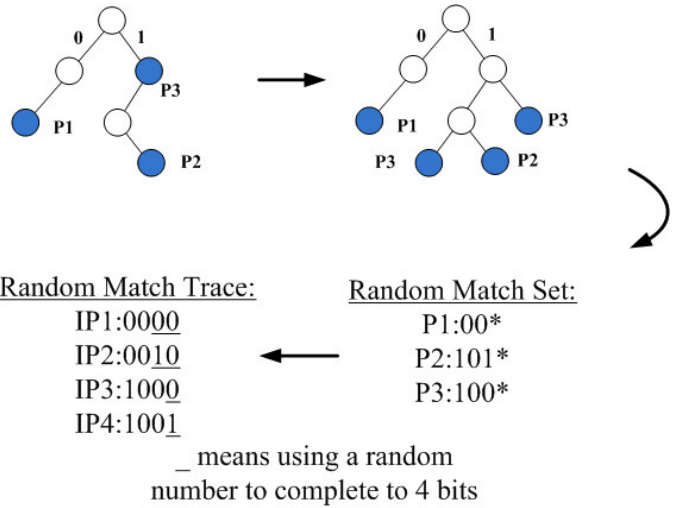

Fig. 3. Random Match Trace Generation

TABLE III
EVALUATION TRACES

Name	unique IP addresses	Generated from
Random Match	353398	routing tables from [10]
Realistic Random Match A	24424	[12]
Realistic and Filtered A	17020	[12]
Realistic Random Match B	81811	[12]
Realistic and Filtered B	41654	[12]

we have replaced the anonymized IP addresses with "valid" IP addresses:

1) We define an association array H mapping anonymized addresses to the "valid" one.
2) For any anonymized IP p, if there exists $H[p]$, we replace it with $H[p]$
3) If not, we generate a "valid" IP address q using the method described in random match and we replace p with q, and let $H[p] = q$.

We can expect to have higher performance on this trace as realistic locality is enforced.

Realistic and Filtered: For this trace we directly used the anonymized realistic trace coming from CAIDA. We filtered out all "valid" IP addresses. This trace will have the highest locality among the three kinds of traces. However the trace will only match a small fraction prefixes in a routing table.

We have generated five traces: one Random Matched, two for Realistic Random Matched and two Realistic and filtered ones. Each trace was containing 1 million IP addresses. The detailed information for these traces is listed in Table III.

B. Single-core Performance evaluations

In each experiment, we have used two cores: one core only for loading the traces and extracting the IP addresses, the other core receiving the IP addresses on the on-chip network and doing the IP lookup. We have used two configurations for Tree Bitmap, one using an initial array of 2^{13} entries, and 4 bit stride; the second using an initial array of 2^{11} entries, and 7 bits of stride. We name them respectively TBP 13-4-4-4-3
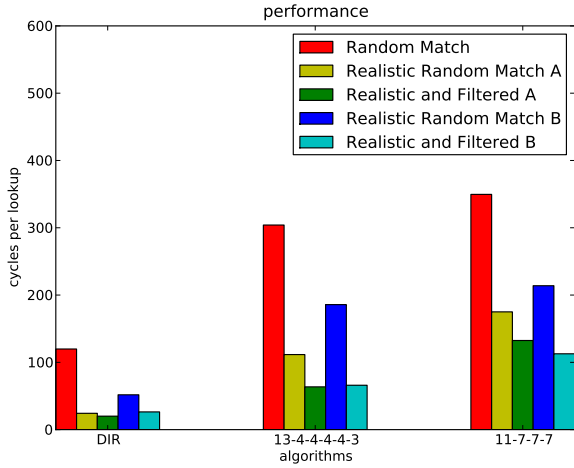
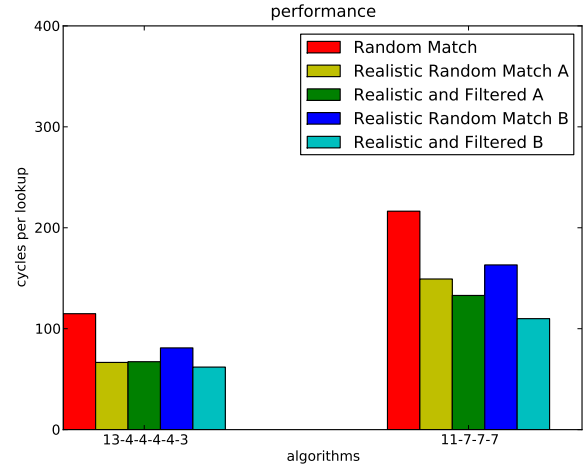Fig. 4.   Single core Performance Results

Fig. 5.   Pipeline Parallel Performance of Tree Bitmap

and TBP 11-7-7-7. The memory footprint of FIB generated by DIR-24-8-BASIC, TBP 13-4-4-4-4-3 and TBP 11-7-7-7 is respectively 69.1MB, 2.9MB and 4MB. Fig. 4 shows the performance results for a single core of TILEPro64.

From the Fig. 4, the following observations can be made. First, the lookup speed is highly related to the number of memory accesses. Although the FIB size of Tree Bitmap is almost 20 times less than DIR-24-8-BASIC, DIR-24-8-BASIC is still 3 times faster. Second, the time spent on processing instructions can not be ignored. Small stride leads to a faster IBM checking, which makes TBP 13-4-4-4-4-3 faster. Third, the locality of the trace determines the final performance. To note, we measure the lookup speed in cycles. The clock frequency of TILEPro64 is 700MHz, which means the each cycle is 1.4 ns. So in the single-core environment, the fastest algorithms can only achieve around 168 ns per lookup.

C. Parallel Performance Evaluations

We have used for the experiment in this section two parallel execution models: pipeline and run-to-complete model.

The pipeline model is only applied to the Tree Bitmap algorithm. In the pipeline model, for each IP lookup, each core only need to do the processing of one multi-bit node (including both the IBM and EBM checking) in one level of the Tree Bitmap trie, then transfer the intermediate result to the next core. There are many proposed algorithms [13][14] to balance the memory utilization of each pipeline stage. However these works assume that the IP lookup engine has multiple single port memories. For example, [13] splits the whole IP lookup into 24 stages, requiring 24 banks of single port memory. TILE64Pro only has 4 DRAM memory interface which does not conform this assumption. So we do not adopt any of these algorithms and simply divide the Tree Bitmap trie by its levels.

We use 5 cores for TBP 13-4-4-4-4-3 and 3 for TBP 11-7-7-7. It is noteworthy that in the pipeline model, we can't perform the "lazy checking" optimization trick. We show in Fig. 5 the performance achieved by the pipeline. Compared to

the Fig. 4, we can observe that the performance gain is about 3 fold. This can be explained as most of the IP addresses in the evaluation traces match prefixes that have length less than 25. Looking up these IP addresses only need 3 to 4 memory accesses. So in average, the speed up rate is around 3 times. And once again, TBP 13-4-4-4-4-3 is faster.

In Fig. 6-7 we show the performance achieved by the run-to-complete model. In this approach, one core is used as a dispatcher that splits the workload by forwarding the IP addresses to the other cores in a round-robin fashion. Whenever a core finishes its lookup, a new IP address is forwarded to it and looked up. This model parallels all algorithmic components. In both figures, *Limit* represents the average transfer time of the on-chip network, it also provides an upper bound on the performance we can achieve.
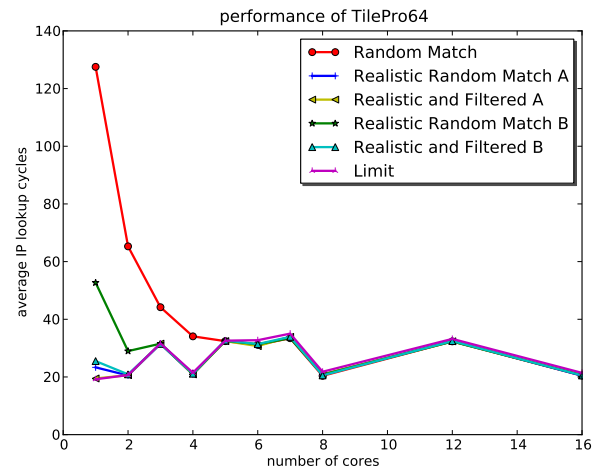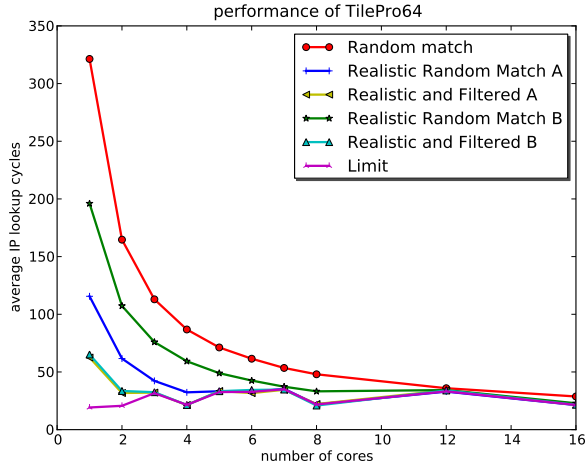

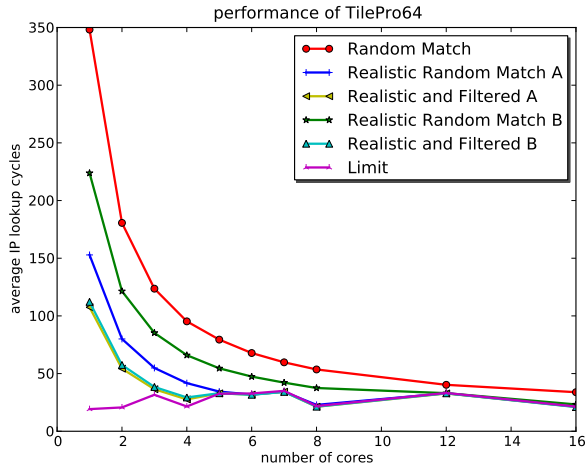Fig. 6.   Run-to-complete Parallel Performance of DIR-24-8-BASIC

The Fig. 6 shows that the highest performance, about 20 cycles per lookup, is achieved when the number of parallel

(a) TBP 13-4-4-4-4-3

(b) TBP 11-7-7-7

Fig. 7.   Run-to-complete Parallel Performance of Tree Bitmap

TABLE IV
UPDATE OVERHEAD OF TWO ALGORITHMS

Algorithms	Add/Del	Entry set	Node copy	Node alloc
DIR-24-8-BASIC	Add	586.67	null	null
	Del	553.95	null	null
TBP 13-4-4-4-3	Add	1.08	1.15	0.97
	Del	0.39	1.43	0.97

Suppose we want to delete a /8 prefix, we need $2^{24-8} = 65536$ memory accesses. This worst case update overhead may become a performance bottleneck in practice. In contrast, the update overhead of Tree Bitmap is much less. In this section, we propose a hybrid IP lookup scheme to combine the strength of both.

The root of high update overhead lies in the short prefixes ($< /17$) stored in *TBL24*. These short prefixes overlap a large range in *TBL24*. Updating these prefixes need to modify all the entries in this range. In order to prevent the high overhead, one can put all these short prefixes in a Tree Bitmap trie. This will result as a side effect, reducing the number of memory access. Because these prefix are all near the root node in the trie. In TBP 13-4-4-4-4-3, only one memory access is needed for such short prefixes. As mentioned above, the lookup speed is highly related to the number of memory access. So this hybrid scheme can also achieve high performance for these short prefixes.

The basic idea of our hybrid lookup scheme is as follows:

1) Store the short prefixes of length 1 to 16 in a Tree Bitmap trie.
2) Store the prefixes of length 17 to 24 in the Table *TBL24*.
3) For the prefixes of length 25 to 32, we use only one entry in Table *TBL24* to store a pointer and put the remaining 8-bit in a sub-trie.

A simple comparison of update overhead is listed in Table IV. We measure the average memory operation times in both DIR-24-8-BASIC and TBP 13-4-4-4-3 when adding and deleting all the /8 ∼ /16 prefixes in our routing table. There are 12894 prefixes in total. As mentioned, one entry of DIR-24-8-BASIC is 4 bytes; one node of TBP 13-4-4-4-3 is 8 bytes. From the table, we can estimate the update overhead of TBP-13-4-4-4-3 is about several hundreds times less than DIR-24-8-BASIC. Because our hybrid algorithm uses the TBP 13-4-4-4-3 to store these prefixes, we can conclude the update overhead of this hybrid algorithm is much less.

The lookup process on the hybrid scheme is similar to the lookup process in DIR-24-8-BASIC. We first perform the long prefix lookup ($> /16$) using Table *TBL24* and the attached sub-tries. If there are not any prefixes matching this IP address, we perform the lookup process on the independent tree Bitmap trie which store the short prefixes. The data structure of the hybrid algorithm is shown in Fig 8.

Fig. 9 shows the performance achieved by our proposed hybrid scheme on a single-core. We used the TBP 13-4-4-4-4-3 as the independent trie, and stride of 4 as the sub multi-bit tree attached to Table *TBL24*.

From the Fig. 9, we see that, as we expecting from the

cores reaches 8. This is equivalent to about 20Gbps of throughput when packets are 64 Bytes. TILEPro64 has four memory controllers and we only use one of them in our experiment. This means that, if necessary, the two memory controllers can be used to provide enough memory bandwidth to support 18 cores (2 for dispatching and 16 for lookup) reaching a 40Gbps lookup throughput. In Fig. 7, as the number of lookup cores increased, the performance increased almost linearly (or the lookup time decreases). However, we achieve at best 28 cycles per lookup by using 16 cores, which is still slower than DIR-24-8-BASIC. This confirms that DIR-24-8-BASIC is superior in speed to tree bitmap (as it lookup time is 8ns less) by at the cost of a memory footprint that is 20 times larger.

V. A HYBRID IP LOOKUP SCHEME

From the evaluation above, we can conclude that the DIR-24-8-BASIC runs faster than Tree Bitmap on TILEPro64. However, this algorithm suffers from a high update overhead.
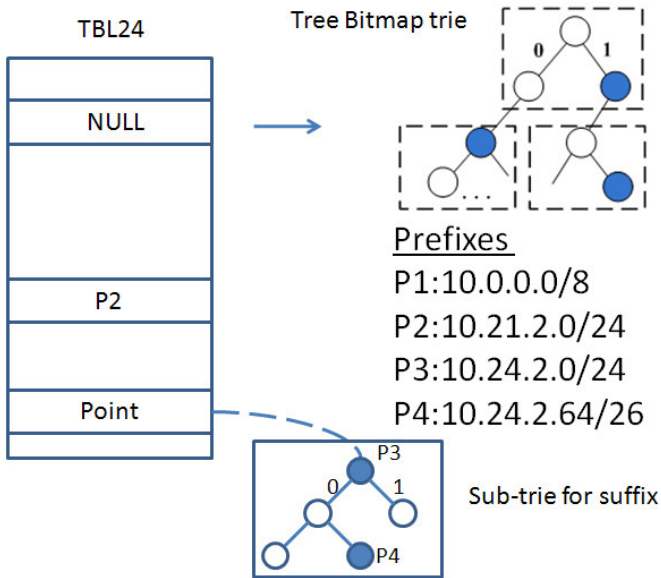
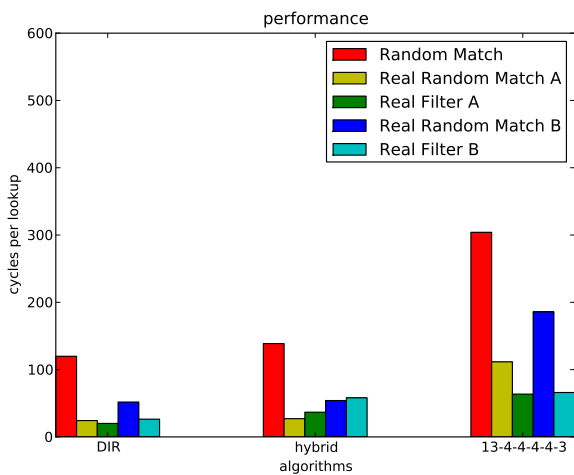Fig. 8.   Data Structure of the hybrid algorithm

Fig. 9.   Performance of our hybrid IP lookup scheme

design, our hybrid scheme achieves a performance similar with the DIR-24-8-BASIC. We now give a brief analysis of the worst case update overhead. As mentioned above, the worst case update overhead of our hybrid scheme is bound by the update overhead of Tree Bitmap. When updating happens in Tree Bitmap algorithm, the worst case is to reconstruct a full child array. In our case, we use a stride of 4, which means the largest child array has up to 16 multi-bit nodes. So the update overhead is bounded by $16 \times 8 = 256$ bytes memory copy. Compared to DIR-24-8-BASIC, which needs larger than 65536 memory accesses in the worst case, our scheme has a much less update overhead. However the memory footprint of the hybrid scheme and the DIR-24-8-BASIC are comparable as the *TBL24* is reused.

VI. CONCLUSION

To summarize, in this work, we implemented two widely used IP lookup algorithms on TILEPro64 and evaluated the performance of them with both synthetic and real world traces. We have been able to achieve a throughput of 40 Gbps that is in the same area the one attained by the PacketShader on a GPU[3] with much lower power consumption. We also found that, on our platform, the IP lookup speed is highly related to the number of memory accesses. Although the small sized FIB can be easily cached, IP lookup with less memory accesses is always faster. We also evaluated the performance of different parallel model. Our experiments show the run-to-complete model is more suitable on many core chips. With 18 cores, we can achieve almost 60 Mpps lookup throughput. In the end of this paper, we propose a new hybrid IP lookup scheme which provides a low bound to the worst case update overhead for DIR-24-8-BASIC. Our work demonstrates the performance power of many core chips, and also gains some insight into the IP lookup on many-core processors.

REFERENCES

[1] B. Chen and R. Morris, "Flexible control of parallelism in a multiprocessor pc router," in *Proceedings of the 2001 USENIX Annual Technical Conference*, 2001, pp. 333–346.
[2] M. Dobrescu, N. Egi, K. Argyraki, B. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: Exploiting parallelism to scale software routers," in *ACM SOSP*, vol. 9.   Citeseer, 2009.
[3] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," in *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4.   ACM, 2010, pp. 195–206.
[4] D. Shah and P. Gupta, "Fast updating algorithms for tcam," *Micro, IEEE*, vol. 21, no. 1, pp. 36–47, 2001.
[5] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion," *ACM Transactions on Computer Systems (TOCS)*, vol. 17, no. 1, pp. 1–40, 1999.
[6] B. Lampson, V. Srinivasan, and G. Varghese, "Ip lookups using multiway and multicolumn search," in *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3.   IEEE, 1998, pp. 1248–1256.
[7] L. Luo, G. Xie, Y. Xie, L. Mathy, and K. Salamatian, "A hybrid ip lookup architecture with fast updates," in *to appear in INFOCOMM'12*. IEEE, 2012.
[8] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3.   IEEE, 1998, pp. 1240–1247.
[9] W. Eatherton, G. Varghese, and Z. Dittia, "Tree bitmap: hardware/software ip lookups with incremental updates," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 2, pp. 97–122, 2004.
[10] "Routeviews," http://www.routeviews.org.
[11] "Tilepro64 power consumption," http://tilera.com/products/processors/TILEPRO64.
[12] P. H. kc claffy, Dan Andersen, "The caida anonymized 2011 internet traces 20110217," http://www.caida.org/data/passive/passive_2011_dataset.xml.
[13] W. Jiang and V. Prasanna, "A memory-balanced linear pipeline architecture for trie-based ip lookup," in *High-Performance Interconnects, 2007. HOTI 2007. 15th Annual IEEE Symposium on*.   IEEE, 2007, pp. 83–90.
[14] F. Baboescu, D. Tullsen, G. Rosu, and S. Singh, "A tree based router search engine architecture with single port memories," in *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*.   IEEE, 2005, pp. 123–133.