# AM-Trie: A High-Speed Parallel Packet Classification Algorithm for Network Processor

Bo Zheng and Chuang Lin

Dept. of Computer Science and Technology, Tsinghua University,
Beijing 100084, P.R. China
{bzheng, clin}@csnet1.cs.tsinghua.edu.cn

**Abstract.** Nowadays, many high-speed Internet services and applications require high-speed multidimensional packet classification, but current high-speed classification often use expensive and power-slurping hardware (such as TCAM and FPGA). In this paper, we present a novel algorithm, called AM-Trie (Asymmetrical Multi-bit Trie). Our algorithm creatively use redundant expression to shorten the height of Trie; use compression to reduce the storage cost and eliminate the trace back to enhance the search speed further. Moreover, AM-Trie is a parallel algorithm and very fit for the "multi-thread and multi-core" features of Network Processor; it has good scalability, the increase of policy number influences little to its performance. Finally, a prototype is implemented based on Intel IXP2400 Network Processor. The performance testing result proves that AM-Trie is high-speed and scalable, the throughput of the whole system achieves 2.5 Gbps wire-speed in all situations.

## 1 Introduction

In recent years, the Internet traffic increases dramatically and the users of Internet increase exponentially. At the same time, the services of Internet turn from traditional best-effort service to quality of service (QoS). Hence, network devices should evolve from the traditional packets forwarding to content awareness, and packet classification is one of the most important basic functions. Many network technology such as VPN, NAT, Firewall, IDS, QoS, and MPLS require packet classification. So the speed of packet classification algorithm will affect the performance of these network devices directly, and it also takes great effect on the next generation Internet. The high-speed packet classification is a hot topic in today's network research, there are many papers about classification [1, 2, 3, 4] published in the top conference, such as Sigcomm and Infocom.

By now, hardware (such as TCAM, FPGA, etc.) is the most common way to achieve high-speed classification, but the hardware is expensive and power-slurping, so the industry hopes to find some high-speed software methods. Network Processor Unit (NPU) is a kind of programable chip which is optimized for network applications processing. Most NPUs are "Multi Thread and Multi-core" architecture [5], they have multiple hardware threads in one microengine and multiple microengines in one NPU chip; NPUs usually have optimized memory

© Springer-Verlag Berlin Heidelberg 2006

management unit for packet forwarding; and some NPUs provide special-purpose coprocessors for some specific network operations. With the well designed architecture, NPU combines the flexibility of software and the high-speed of hardware together, it can also greatly shorten the product development cycle. So it is considered as the core technology which impels the development of next generation Internet. Network Processor has a good prospect and becomes a new direction and hot topic [6, 7, 8] in network research after ASIC.

But the traditional software classification algorithms are not fit for NPU, they usually based on the data structure of binary tree or hash table. The complexity of these algorithms is $O(dW)$, because they are not parallel algorithm, where $d$ is the classification dimension (field number of the policy table) and $W$ is the bits of the field. They cannot satisfy current high-speed network, not to mention the high-speed IPv6 applications in the near future. Hence, researches on high-speed packet classification algorithm based on NPU platform are very important.

In this paper, we present a parallel high-speed multidimensional classification algorithm, called AM-Trie (Asymmetrical Multi-bit Trie), which has the following features:

– Use redundant expression to express prefix with arbitrary length and build AM-Trie, which is much lower than normal Trie.
– Compress AM-Trie to eliminate trace back. This increases the lookup speed and reduces the storage cost. The time complexity of AM-Trie is $O(d+h)$ and the space complexity is $O(N^2)$, where $d$ is the dimension of the classification, $h$ is the height of AM-Trie and $N$ is the number of classification policy.
– AM-Trie is a parallel algorithm and very fit for the "multi-thread and multi-core" feature of Network Processor.
– Our algorithm has good scalability, there is almost no influence to the performance when the number of policy increase much.

The remainder of this paper is structured as follows. We describe the detail of AM-Trie algorithm and analyze the algorithm complexity in Section 2. Performance evaluation can be found in Section 3. Finally, Section 4 gives a further discussion and summarizes the paper.

## 2 AM-Trie Classification Algorithm

The performance metrics of packet classification include time complexity, space complexity, scalability, flexibility, worst case performance, update complexity, preprocessing complexity, etc., and for different applications the importance priority of these metrics are different. At present, the speed of Internet increases dramatically, many new kinds of services appear. Hence, in order to adapt packet classification algorithm to the evolution of next generation Internet, we should consider time complexity, scalability and flexibility first.

However, the the search complexity of traditional tree/hash based classification algorithm is direct proportion of the classified field length. For example, to classify a IPv4 address we have to build a 32-level Binary Trie or establish

32 Hash tables; and 128-level Binary Trie needed for IPv6 address, the search efficiency is too low. Therefore, we hope to construct $m$bits-Trie to reduce the height of Trie, and speed up the search. But because of the widely used Classless Inter-Domain Routing (CIDR) technology, the prefix length may be any length between 0 to field width. Such policy table cannot transform to $m$-bits-Trie conveniently, because the prefix length may not right suit with the multi-bit Trie (for example, 2bits-Trie, which takes 2-bit as one unit, can not express prefix length for odd number situation).

The first idea of AM-Trie is: express arbitrary length of prefix using the redundancy. Considering a $k$-bits field, all possible prefixes are $*, 0*, 1*, 00*$, $01*, \ldots$, $\overbrace{11\ldots1}^{k-1}*$, $\overbrace{00\ldots0}^{k}$, $\ldots$, $\overbrace{11\ldots1}^{k}$, totally $2^{k+1}-1$ prefixes. We can number them as $1, 2, \ldots, 2^{k+1}-1$. They may be expressed using one $(k+1)$bits number which we call *Serial Number*. If the prefix is $P*$, and $P$ is a $p$-bits number, then

$$SerialNumber(P*) = 2^p + P \qquad (1)$$

For example, * is the 1st prefix and $1111*$ is the 31st prefix. Using this redundant expression, we can use one $(k+1)$-bit number to express all $k$-bit prefix. Then we can use such expression to build appropriate multi-bit Trie.

## 2.1 AM-Trie Creation

The redundant expression can express arbitrary length of prefix. Therefore, we can cut a classification field in to several variable length sections, and then construct multi-bit Trie – AM-Trie. Without loss of generality, assume a $W$-bit-wide field is divided into $l$ parts (and constructed as a $l$-level AM-Trie), the width of each part is $h_1, h_2, \ldots, h_l$ bits, and $\sum_{i=1}^{l} h_i = W$, i.e. the $i$th level of constructed AM-Trie has $2^{h_i}$ nodes (How to optimize the $l$ and $h_i, i = 1 \ldots l$ will be mentioned in section 2.3).

Each node of AM-Trie have two pointer, *pNext* points to its children level and *pRule* points to the related policy rule. When creating the AM-Trie of a policy table, we add the policy one by one. Firstly, divide the policy into predefined parts $h_1, h_2, \ldots, h_l$, and calculate the serial number of each section use the redundant expression. Then calculate the offset value of each level according to the serial number. If the related leaf node does not exist, add the policy rule node to the right place, or compare the existing node and the new one and add the higher priority rule. After that, add the next policy. The pseudocode for add a node into AM-Trie is:

```
int AddNode(struct AMTrieNode *root, Rule newrule)
1   current=root;
2   divide newrule into predefined parts;
3   offset=Serial Number(first part);
4   while(not the last part)
5       current<--current[offset];
6       if(current->pNext == NULL) create children under current node;
7       offset=Serial Number(next part);
```

```
current<--current[offset];    //handle the last part
if(current->pRule==NULL) current->pRule=this rule; return 1;
else current->pRule=the higher priority rule; return 0;
```

We use an example to give the direct-viewing of AM-Trie creation. Figure 1 shows the AM-Trie derived from Table 1, here we chose $l$=2, $h_1 = h_2 = 2$bits. Firstly, we add the the first policy R0 which has only one part (*), it's serial number is 1. So R0 is add to the first node under root. Similarly, R1, R2 is the 2nd and the 5th child of root. As for R3, its first part (01) has serial number 5, under the 5th child node of root we continually process its second part (1*), so R3 is joined to the 3rd child node in the second level. The situation of R4 is similar to R3. If there is a rule R5 (011*), when it joins to the AM-Trie, its position is already occupied by R3, then we needed to compare the priority of R3 and the R5 to judges whom takes this position.

From the generated AM-Trie, we can find that only the nodes in the right part of the Trie may have children nodes, because the first $2^{k_i} - 1$ nodes map to the prefix with a wildcard '*', which is the end of a prefix, so they cannot have child. Therefore this Trie looks asymmetrical, we name this Trie Asymmetrical Multi-bit Trie.

**Table 1.** A simple policy table

| Rule | Field |
|---|---|
| R0(Default) | * |
| R1 | 0* |
| R2 | 01* |
| R3 | 011* |
| R4 | 1111 |

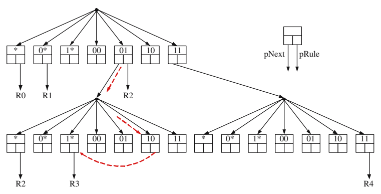

**Fig. 1.** AM-Trie Creation and Search

## 2.2 AM-Trie Search, Optimization and Update

Finding a policy in the AM-Trie is similar to search a common Trie except that if a leaf node doesn't contain a policy we should go on searching its "prefix node"[1]. For instance, the red broken line in Figure 1 shows the search progress in the same AM-Trie. If the classification field of the incoming packet is 0110, firstly we calculate the serial number of its first part '01' and go to the 5th child of the root, and then calculate the serial number of the second part '10' and go to the 6th node in the second level. This leaf node has no policy, so we lookup the prefix node of '10' (i.e. '1*'), and go to the 3rd node in this level and find the result is R3.

This example shows that when a leaf node has no rule, we have to trace back to its prefix nodes. We can use a compression algorithm to avoid the trace back and optimize the AM-Trie. We find that when the policy table determined, the

[1] We call a prefix with wildcard '*' *covers* all the prefixes match it, for example, in a 3bits field, 0* covers 00*, 01*, 000, 001, 010 and 011. If A covers B, A is the *prefix node* of B.

longest rule-matched prefix of any prefix (with wildcard or not) determined too. Hence, we can "push" the policy in a prefix to the nodes covered by it and without policy in the period of preprocessing. Then all the leaf nodes contain a policy and no trace back in searching a AM-Trie now. At the same time, the prefix with wildcard '*' in the left part of AM-Trie will never be accessed because their information has been pushed to the nodes they covers in the right part. So, we can delete these node to compress the storage of AM-Trie. Figure 2 illustrates the "push" procedure and the result of compression. The broken line with arrow indicate the push operation, and the red nodes can be delete to compress the AM-Trie. We can save about one half storage after compressing. And the search complexity of compressed AM-Trie is $O(h)$ now.

AM-Trie data structure supports incremental update. The Add operation is just the same as AddNode in AM-Trie creation. The Delete operation is also very simple. Call the search function to locate the policy we want to delete, and then release the storage; if this level contains no policy after deletion, release the whole level. Both Add and Delete operation has the complexity of $O(h)$.

## 2.3 Multidimensional Packet Classification Using AM-Trie

In order to perform multidimensional classification using AM-Trie, we profit from the idea of Aggregated Bit Vector (ABV) [1]. For a $d$-dimension classification problem we build $d$ AM-Tries, each AM-Trie maps to a dimension and its leaf nodes stores the ABV. Because the search operations in different dimension are independent, they can be parallel execution. AM-Trie algorithm is very fit for the architecture of Network Processor, it can run in multiple microengines or hardware threads concurrently and increase the performance enormously. When we get the ABVs of all the dimensions, we AND them together and come out the final classification result.

For a $d$-dimensional classification policy table which has $N$ rules, if the height of generated AM-Trie is $h$ (usually $h \ll W$, where $W$ is the bits of the field), the complexity of AM-Trie algorithm is:

1. The initialization complexity of AM-Trie algorithm is $O(N \log_2 N)$.
   The creation time complexity of AM-Trie is $O(Nh)$, but the initialization of ABV need to sort the policy table. Hence, the initialization complexity $O(N \log_2 N)$.

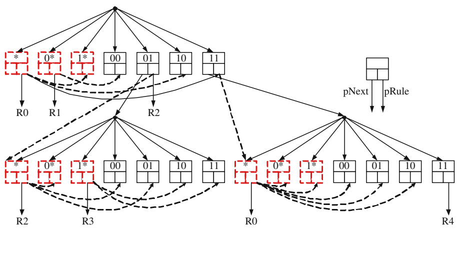

**Fig. 2.** AM-Trie "push" and compression

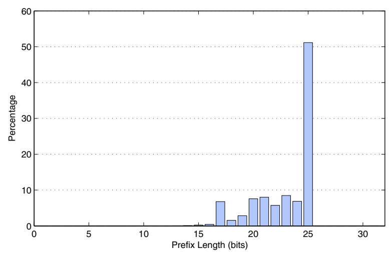

**Fig. 3.** Prefix distribution in a BGP table

2. The search complexity of AM-Trie algorithm is $O(h + d)$.
   In worst case, each AM-Trie needs $h$ comparisons to get the ABV address, different AM-Trie can search concurrently; after that still need to read the $d$ ABVs and AND them to get the final result. So the search complexity of AM-Trie algorithm is $O(h + d)$.
3. The space complexity of AM-Trie algorithm is $O(N^2)$.
   Because there are total $N$ policies, the number of prefix in any field $\leq N$. If a level is $m$-bits, the nodes which have children cost $2^m * sizeof(Node)$ in the child level. In the worst case (all the nodes in the $h - 1$ level have children and all the children do not duplicate), the required storage is less than

   $$[(h-1) * N + 1] * 2^m * sizeof(Node) + N * sizeof(ABV). \qquad (2)$$

   Where $sizeof(Node)$ is the size of AM-Trie Node structure (8 bytes), and $sizeof(ABV)$ is the size of aggregated bit vector ($N$-bits). So the space complexity of AM-Trie algorithm is $O(N^2)$ (Assume $m$ is small enough that $2^m$ not a big number compare to $N$).

But how to choose the the number of level and the stride of each level to gain a better performance? The lower the AM-Trie the faster the algorithm, but the more storage it will cost. If we build a AM-Trie with only one level, the $2^m$ in Equation (2) will be $2^{32}$, we can not afford such a huge memory cost.

In order to optimize the space complexity we can take the distribution of prefix length into account. In the practical policy table, the prefix length distribution is very non-uniformity, Figure 3 is the prefix length distribution of a BGP table which we randomly select from [9]. This table contains 59351 policies, most of them belong to a few kind of prefix length, the 24-bit policies even more than half. A straightforward idea is if we choose the prefix length with high probability as the dividing point, the generated AM-Trie will cost less storage. There are some related work on this topic, we can use a similar way as Srinivasan proposed in [10] to optimize the AM-Trie space complexity.

## 3 Performance Evaluation

We have implemented the AM-Trie Algorithm in single Intel IXP2400 network processor, the eight microengines are assigned as follow: the receive, transmit, buffer manage and scheduling module use one microengine respectively, and the rest 4 microengines carry on parallel AM-Trie classification as well as IP header processing.

Before our test we generate the policy table randomly according to the prefix distribution shows in Figure 3. The policy table contains 6 fields (6-dimensional classification), i.e. TCP/IP 5 tuple plus the TOS field. Then we build AM-Trie for each field, all the fields support longest prefix match. In our implementation, the height of all the 6 AM-Tries is no more than 3, according to the search complexity analysis in section 2.3, only 15 memory access needed to classify one packet. In other words, even if implemented with low speed DDR SDRAM

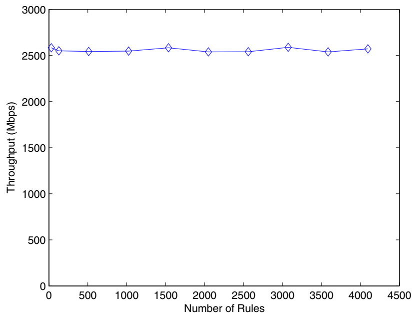

**Fig. 4.** The throughput of AM-Trie on IXP2400, the system achieves 2.5Gbps wire-speed in all condition (number of policies increases from 32 to 4096)

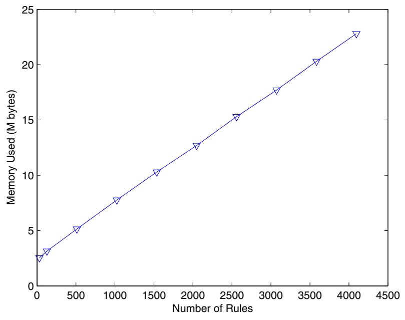

**Fig. 5.** The cost of memory increase linearly as a function of policy number. AM-Trie only used 23M RAM when the size of policy table is 4096.

(150MHz), AM-TRIE may also achieve fast wire-speed (10Mpps). Finally we begin our test using IXIA 1600 traffic generator to generate traffic to IXP2400 (all the packets are 64 bytes).

Figure 4 shows the throughput of the whole system. The system throughput does not decrease when the number of policies increases from 32 to 4096, the throughput maintain above 2.5Gbps (5Mpps). The reasons are: 1) AM-Trie algorithm has good scalability, it can easily be extend to support 256K policies using hierarchical structure if necessary, and still maintain such high speed; 2) IXP2400 is oriented to OC48 (2.5Gbps) network application, the AM-Trie algorithm already makes the most of IXP2400 capability, by the restriction of other system modules (such as receive, transmit, scheduling, etc.), the system performance is unable to enhance more.

Figure 5 demonstrated the memory cost of AM-Trie algorithm as a function of the number of policy (in order to simplify the implementation, we fix the length of ABV to 4096 bits, therefore the memory cost appear linear growth, but not the theoretically space complexity $O(N^2)$). AM-Trie belongs to algorithm of “trade space for time”, it use a lower speed but large capacity and cheap RAM to achieve comparative performance as the expensive and power-slurping hardware (such as FPGA, TCAM, etc.). IXP2400 support 2x64M SRAM and 2G DDR SDRAM, it’s very ample for AM-Trie algorithm.

We are implementing this algorithm on Intel high end network processor IXP2800, which designed for OC192 network applications. The preliminary test result is closed to 10Gbps, anticipated throughput will achieve 10Gbps (20M pps) after code optimization.

## 4 Conclusion and Further Discussion

The rapid growth of Internet needs high-speed packet classification algorithm, but current high-speed classification often use expensive and power-slurping hardware (such as TCAM and FPGA). In this paper, we propose a novel high-speed parallel multidimensional packet classification algorithm, called AM-Trie (Asymmetrical Multi-bit Trie). Our algorithm innovatively use redundant expression for arbitrary prefix length, greatly reduce the search complexity of Trie; use compression to eliminate the trace back in search operation, further enhances the search speed and reduces the storage cost. More important, the AM-Trie is a parallel algorithm, and very fit for the "multi-thread and multi-core" features of Network Processor; the algorithm have good scalability, the increase of policy number nearly has no influence to the algorithm performance. The time complexity of AM-Trie is $O(h + d)$, where $h$ is the height of AM-Trie and $d$ is the dimension of classification; space complex is $O(N^2)$, where $N$ is the policy number.

We also implemented AM-Trie algorithm based on Intel IXP2400. The performance analysis and the test result show that the AM-Trie algorithm can achieves 2.5Gbps wire-speed (all the packets size are 64 bytes, i.e. 5Mpps) in the TCP/IP 6 tuple classification, and it has big performance promotion space in more powerful Network Processor(such as IXP2800 or above). At the same time, our experimental result also indicate that "trade space for time" algorithms on Network Processor using lower speed but cheap and large capacity RAM to obtain high-speed classification is a feasible way.

## References

1. F. Baboescu and G. Varghese, Scalable packet classification, in SIGCOMM, 2001, pp. 199–210.
2. G. V. Sumeet Singh, Florin Baboescu and J. Wang, Packet classification using multidimensional cutting, in SIGCOMM, 2003, pp. 213–224.
3. Florin Baboescu and G. Varghese, Packet classification for core routers: Is there an alternative to cams? in INFOCOM, 2003, pp. 53–63.
4. A. R. Karthik Lakshminarayanan and S. Venkatachary, Algorithms for advanced packet classification with ternary cams, in SIGCOMM, 2005.
5. N. Shah, Understanding network processors, Tech. Rep., 2001.
6. Network processing forum (npf). http://www.npforum.org/
7. Network systems design conference. http://www.networkprocessors.com/
8. T. Wolf and M. A. Franklin, Design tradeoff for embedded network processors, in ARCS, 2002, pp. 149–164.
9. BGP routing table analysis reports. http://bgp.potaroo.net/
10. V. Srinivasan and G. Varghese, Faster IP lookups using controlled prefix expansion, in Measurement and Modeling of Computer Systems, 1998, pp. 1–10.