CoPTUA: Consistent Policy Table Update Algorithm for TCAM without Locking

Zhijun Wang, Hao Che, Mohan Kumar, *Senior Member*, *IEEE*, and Sajal K. Das

Abstract—Due to deterministic and fast lookup performance, Ternary Content Addressable Memory (TCAM) has recently been gaining popularity in general policy filtering (PF) for packet classification in high-speed networks. However, the PF table update poses significant challenges for efficient use of TCAM. To avoid erroneous and inconsistent rule matching, the traditional approach is to lock the PF table during the rule update period, but table locking has a negative impact on data path processing. In this paper, we propose a novel scheme, called **Co**nsistent **P**olicy **T**able **U**pdate **A**lgorithm (CoPTUA), for TCAM. Instead of minimizing the number of rule moves to reduce the locking time, CoPTUA maintains a consistent PF table throughout the update process, thus eliminating the need for locking the PF table while ensuring correctness of rule matching. Our analysis and simulation show that, even for a PF table with 100,000 rules, an arbitrary number of rules can be updated simultaneously within 1 second in the worst case, provided that 2 percent of the PF table entries are empty. Thus, CoPTUA enforces any new rule in less than 1 second for practical PF table size with high memory utilization and without impacting data path processing.

为了保证效率，传统的方
法是在更新时上锁

前提是

Index Terms—Network processor, ternary CAM, policy table update, packet classification.

1 Introduction

As Internet applications proliferate and transmission bandwidth increases, network processors used for data path processing in a router need to be able to classify a packet within a few tens of nanoseconds (ns) in order to keep up with multigigabit communication channel (line) rates. In the past few years, significant research efforts have been made on the design of fast packet classification algorithms for both Longest Prefix Match (LPM) and general policy filtering (PF) (e.g., [3], [6], [7], [10], [17], [18]). Unfortunately, most of these approaches neither provide deterministic performance guarantees nor keep up with multigigabit line rates.

An alternative approach which has been gaining popularity is the use of a *ternary content addressable memory* (TCAM) coprocessor to offload the packet classification tasks from the network processor. TCAMs are fully associative memories in which each cell can assume one of three logical states: 0, 1, or don't care (denoted as "x"). The state "x" allows a TCAM to store wildcards in any location in a rule. Each TCAM lookup requires a single clock cycle and a PF table match may require a multiple number of TCAM lookups, depending on the rule size. Thus, TCAM-based packet classification ensures deterministic and fast lookup performance. Indeed, the packet classification processing at OC-192 line rate using a fully programmable network processor and its TCAM coprocessor is reported in [1].

Despite fast lookup performance, the TCAM-based solution poses significant challenges. In addition to high power consumption and relatively large footprint of the TCAM hardware, resource management and database update are also recognized as critical issues. While TCAM hardware and resource management issues have been addressed in [4], [8], [11], [12], [13], [15], [19], [20], [21], the problem of database update has not received much attention. Our goal in this paper is to develop efficient techniques to update TCAM databases.

1.耗电
2.硬件占地较大

The primary source of concern for general PF table update in a TCAM comes from a wide adoption of a class of coprocessors, known as Ordered TCAM or OTCAM [4], in which PF table rules are arranged in an ordered list such that higher priority rules are placed in lower memory addresses. When a search key matches multiple rules, the one in the lowest memory address is selected and the corresponding action in an associated memory is returned. In the worst case, adding a new rule in a PF table may require all the existing rules and their corresponding actions to be moved to new memory locations, causing significant interruption of the data path (i.e., lookup) processing.

一类TCAM是怎么做
的：
高优先级在低的地
址

最坏的情况，FP表里所
有的数据都要移动地
址，使查找的进程中
断，影响效率

Two LPM table update algorithms have been proposed in [16] to minimize the number of rule moves per rule update in an OTCAM. The goal is to minimize the LPM table locking time for rule updates. One of these algorithms is optimal in terms of the worst-case number (at most 16) of rule moves per rule update. As we shall explain in the next section, locking the LPM table for 16 rule moves can actually lead to dropping of about 18 packets at OC-192 line rate. For general PF table update, we conjecture that the worst-case number of rule moves per rule update is $O(N_r)$, where N_r is the total number of rules in the PF table. Consequently, in the presence of multigigabit line rates, attempting to lock a PF table for rule update can significantly impact the performance of data path processing.

- *The authors are with the Center for Research in Wireless Mobility and Networking (CReWMaN), Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX 76019. E-mail: {zwang, hche, kumar, das}@cse.uta.edu.*

Manuscript received 10 Dec. 2003; revised 23 Apr. 2004; accepted 25 May 2004.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0280-1203.

0018-9340/04/$20.00 © 2004 IEEE Published by the IEEE Computer Society

我们提出了一致的策略表更新算法（CoPTUA），它消除了对TCAM PF表锁定的需要，同时确保了规则匹配的正确性。

In this paper, we take a different approach to tackling this problem. Instead of designing efficient algorithms to minimize the number of rule moves and, hence, the locking time, we propose a **C**onsistent **P**olicy **T**able **U**pdate **A**lgorithm (CoPTUA) which eliminates the need for TCAM PF table locking while ensuring the correctness of the rule matching. The idea behind this novel approach is to maintain a consistent and error-free PF table during the update process and avoid inconsistent and/or erroneous rule matching. A PF table is *consistent* if, for each rule move, a search key matching results in the same rule as the one that would be matched before the rule move. Also, for each rule addition or deletion, a search key matching results in the same rule as the one that would be matched just before or after the addition or deletion. This is possible if there exists a small number of empty rule entries. Erroneous rule matching may occur when a rule or its action is being updated. The proposed CoPTUA avoids erroneous rule matching by eliminating direct rule overwriting. This is made possible by decomposing an overwriting operation into three steps: 1) Deactivate a rule by resetting the valid bit, 2) write a new rule, 3) activate the new rule by setting the valid bit. Thus, CoPTUA allows the PF table update process to take place without locking and, at the same time, ensures efficient data path processing.

PFtable

消除直接规则覆盖

怎么做的！

However, the above two requirements tend to increase the number of operations per rule update and require some empty memory entries to be allocated in order to allow consistent rule moves. Our analytical performance study shows that CoPTUA is very efficient in terms of rule update time and memory utilization. In particular, for a PF table with 100,000 rules, the worst-case delay for an arbitrary number of rule updates is less than 1 second, provided that only 2 percent of the PF table entries are empty. Considering the time associated with the rest of the rule update process, from a remote policy server to the management plane and then to the data plane interface, this worst-case delay is negligible. The performance of CoPTUA is also evaluated by simulation. The results show that the maximum update delay is within 0.35 seconds for a TCAM with up to 100,000 rules and 1 percent empty rule entries. Therefore, the proposed solution successfully addresses a critical issue related to the general PF table update, making OTCAM a favorable choice for high performance packet classification. We also show that our proposed approach can be used for consistent PF table update in a WEIghted TCAM (WEITCAM) coprocessor [4] in which there is a weight subfield associated with each rule entry and the rule matching priority is determined by the relative weight assigned to the rule, rather than the memory location of the rule as in OTCAM.

性能方面：如果只有2%的空间空余，最坏的时间不超过1s

加权

加权TCAM，按权重做优先级，而不是地址顺序

The rest of the paper is organized as follows: Section 2 describes the architecture of a TCAM coprocessor. Section 3 identifies the fundamental difficulty in developing fast PF table update algorithms. Section 4 presents our solution for OTCAM PF table update without locking. The performance of CoPTUA is analyzed and simulated in Section 5. Section 6 describes how the proposed technique can be used efficiently for LPM table update and general PF table update in WEITCAMs. Related work is presented in Section 7. Finally, Section 8 concludes the paper.

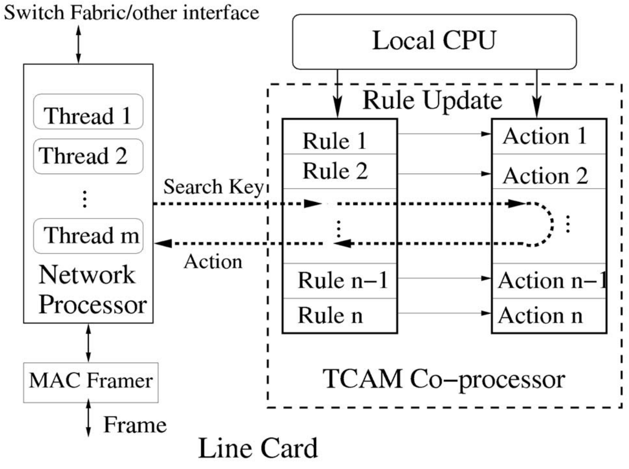


Fig. 1. A network processor and its TCAM coprocessor.

2 TCAM Coprocessor

Fig. 1 shows a typical TCAM coprocessor used for packet classification on behalf of a network processor. The coprocessor contains self-addressable *rules* which map to different memory addresses in an associated memory (normally an SRAM) containing the corresponding *actions*. The TCAM is organized in slots. The number of bits in a slot is fixed (e.g., 64, 72, or 128 bits) as set by the vendors. Depending on the rule size, a rule may take one or more slots. A rule matching is performed for all the rules in parallel. Each parallel matching is done one slot at a time. Hence, for a table where each rule occupies n slots, n TCAM clock cycles are required to get a best matched rule. Therefore, no action can be returned until the n slots are matched. A typical rule for packet classification is composed of 104-bit five tuples: {*source IP address, destination IP address, source port, destination port, protocol number*}. The rules are either arranged in an ordered list or weighted, depending on whether an OTCAM or WEITCAM is in use. A search *key* composed of the same set of subfields, extracted from the header of a packet to be classified is passed from the network processor to the TCAM coprocessor for lookup through the corresponding interface. The matched rule with the highest match priority then results in the corresponding action in the associated memory to be returned to the network processor.

The PF table update is generally done via a local CPU/TCAM coprocessor interface. The local CPU resides in the same line card (LC) as the TCAM coprocessor. A user has the choice as to whether or not to lock the network processor/TCAM coprocessor interface while the TCAM database is being updated. Without interface locking, TCAM table lookups via the interface are not interrupted. However, by doing so, the TCAM coprocessor may return inconsistent and/or erroneous results. Locking the interface ensures that the TCAM table lookup always returns correct results, but, during the database update period, all the threads that need

在更新的时候不上锁，查找就不会中断，但是可能查找错误的结果

to access the TCAM coprocessor are suspended, impacting the data path processing performance.

To quantify the performance impact caused by TCAM database locking, let us consider a network processor that needs to support an aggregated line rate of 10 Gbps. Assume the minimum packet size is 49 bytes, then the network processor has to process packets at a maximum rate of about 25 Million packets per second (Mpps) or 40 *ns* per packet time. Further, assume that a TCAM memory width or slot size is 64 bits and a 64-bit PCI bus between the CPU and TCAM coprocessor runs at 66 MHz clock rate (15 *ns* per clock cycle), the same as the PCI for the INTEL IXP2800 network processor [2].

Now, let us estimate the per rule update time in the worst case for a 104-bit five-tuple PF table. In this case, each 104-bit rule takes two 64-bit slots in TCAM. Assume that the action code fits well into one 64-bit associated memory word so that loading an action requires just one access to the TCAM coprocessor. To load the rule and its mask (which must be loaded to set the corresponding wildcard bits), $128 \times 2/64 = 4$ accesses to TCAM coprocessor are needed. So, the estimated total number of TCAM coprocessor accesses for adding a rule is five. This translates into about $15 \times 5 = 75$ *ns*, or about $75/40 \sim 1.9$ packet times. Assume 1,000 rules need to be moved in the worst case to add a new rule in a PF table with 1,000 rules. Then, up to $1.9 \times 1{,}000 = 1{,}900$ incoming packets may get dropped,
？why*1000
because all the threads handling the packets in the network processor will be waiting for TCAM coprocessor access shortly after the TCAM coprocessor is locked (in m packet times in the best case, where m is the total number of threads in the network processor) and all the incoming packets are blocked due to the lack of available threads in handling them.

Locking the interface for an LPM table update can also be harmful. Writing a rule requires two accesses to the TCAM coprocessor to load the rule and its mask and one access to load the action. This translates into about $15 \times 3/40 = 1.1$ packet times. As mentioned in the previous section, up to
16*1.1 =17.6 =18
16 rule moves are needed to add a new rule. Hence, up to 18 packets may be dropped per rule update in the worst case, where all the threads are waiting for LPM access when the TCAM coprocessor is locked. The above estimations clearly demonstrate the limitation in developing fast update algorithms for minimizing the locking time.

3 Complexity of OTCAM Policy Table Update

In this section, we first introduce some useful concepts and mathematical notations to facilitate further discussion.

- *Rule space*: The space of a rule with b bits is defined as a region in b-dimensional space. Each dimension in a rule corresponds to a bit that can assume two values, 0 and 1. A wildcard bit covers the whole space (i.e., 0 and 1) in the dimension corresponding to that bit.

For example, xx constitutes a region which covers the whole of a two-dimensional rule space whereas 11 covers a single point.

- *Rule overlapping*: Two rules A and B are said to overlap with each other if and only if $A \cap B \neq \emptyset$, i.e., they have a common subregion in the rule space.
- *Superset and Subset rules*: A is said to be a superset rule of B (and, hence, B a subset rule of A) if the region covered by B in the rule space is a subregion of that covered by A in the same rule space. This is denoted as $A \supset B$ (or, equivalently, $B \subset A$).
- *Partially overlapping*: Rules A and B are said to partially overlap with each other if they overlap with each other but have no superset-subset relationship.

For example, rules 1x and x0 are partially overlapping with a common point 10. On the other hand, rule 11 is a subset rule of 1x. Clearly, subset-superset relationship is a special case of overlapping relationship.

Overlapping rules can be matched simultaneously and, hence, their relative match priorities need to be determined when they are enforced. Note that a subset rule must have a higher match priority than its superset rules simply because the subset rule would never be matched otherwise. On the other hand, the relative match priorities between two partially overlapping rules need to be specified by the network administrator.

We further introduce the following notations:

- $A \rightarrow B$: A has a lower match priority than B.
- $A < B$: A is in a lower match priority memory location (i.e., in higher memory location) than B in an OTCAM. a比b的地址空间高

Obviously, if $A \rightarrow B$, then we must have $A < B$. If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$. Similarly, $A < B$ and $B < C$ implies $A < C$.

- *Connected rules*: Rules A, B, and C are said to be *connected* if $A \cap B \neq \emptyset$ and $B \cap C \neq \emptyset$.
- *Connected Rule Graph* (CRG): All the connected rules together form a connected rule graph using arrows defined above to link between rules with priority relationship.
- *Source (sink) leaf rule*: In a CRG, a rule is said to be a *source (sink) leaf rule* if there is no lower (higher) match priority rules associated with it. 相当于叶子结点，最高和最低的rule
- *Multiple Match Group*(MMG): In a CRG, all the rules on a directed path from any source leaf rule to any sink leaf rule form an MMG.
- *Independent rules*: Two rules A and B are said to be *independent* of each other if they do not appear in the same MMG, denoted as $A \wedge B = \emptyset$.

Independent rules have no match priority relationship and can be arbitrarily interleaved in an OTCAM. Obviously, any two rules from two different CRGs are independent of each other and, thus, can be arbitrarily interleaved in an OTCAM. 交错

Fig. 2 shows a CRG composed of five rules. The region in the rule space that each rule covers is represented by a horizontal line. Note that $A \cap C \neq \emptyset$, $B \cap C \neq \emptyset$, $C \cap D \neq \emptyset$, $C \cap E \neq \emptyset$, $A \cap D \neq \emptyset$, $B \cap E \neq \emptyset$, $A \cap E = \emptyset$, and $B \cap D = \emptyset$. More specifically, we note that $C \supset E$. Furthermore, A and B are source leaf rules and D and E are sink leaf rules. Hence, there are a total of four MMGs in this CRG. They are: $A \rightarrow C \rightarrow D$, $A \rightarrow C \rightarrow E$, $B \rightarrow C \rightarrow D$, and $B \rightarrow C \rightarrow E$. Rules A and B do not appear in any MMG simultaneously.

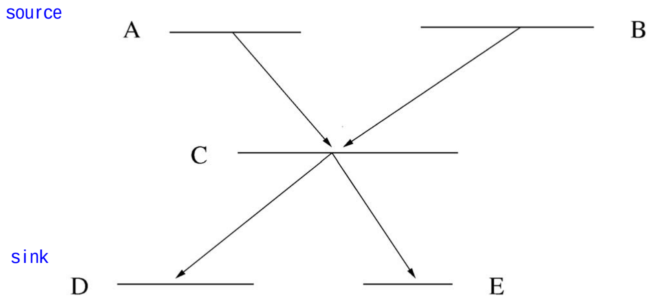


Fig. 2. An example of CRG and MMGs. The match priority increases along the direction of arrows. The region each rule covers is indicated by a horizontal line.

Hence, $A \wedge B = \emptyset$. Similarly, $D \wedge E = \emptyset$. A rule may appear in multiple MMGs, but in one CRG only. Obviously, rules in an MMG must be arranged in an ordered list, whereas independent rules can be interleaved in arbitrary order in an OTCAM.

独立规则可以出现在任何位置

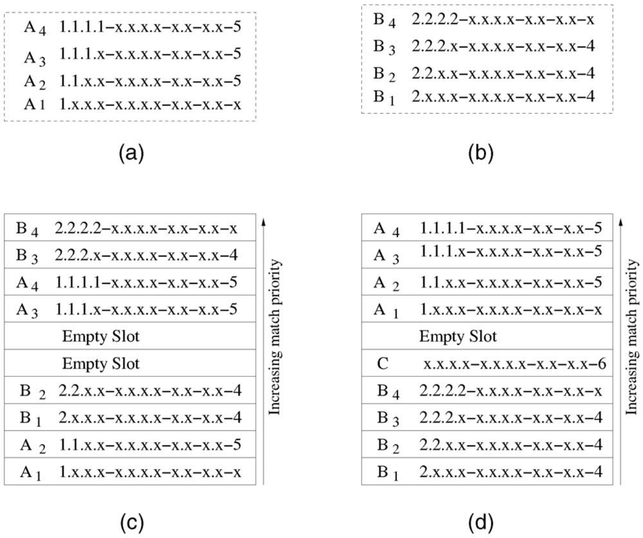
As a special case, for LPM, we note that rules cannot partially overlap with one another. All the rules in an MMG must have a superset-subset relationship and a superset rule must have a shorter prefix length than its subset rules. Hence, the maximum number of rules in one MMG of an LPM table is b, the number of bits in an IP address; thus, $b = 32$ for Internet Protocol version 4 (IPv4). This simple rule structure is fully leveraged in the design of an optimal rule update algorithm [16] in terms of the worst-case performance. By maintaining the empty TCAM slots in the center of the LPM table and placing rules with different prefix lengths in different blocks sequentially and evenly split to the upper and lower half of the OTCAM addresses, it was shown that, for IPv4, in the worst case, $b/2 = 16$ rule moves are required to add a new rule. Since the MMG size for a general PF table can be as large as N_r, the size of the PF table itself, following the same algorithm and logic as in [16], it is easy to show that, in the worst case, at least $N_r/2$ rule moves are required to add a new rule, regardless of what algorithm is used. Even worse, unlike LPM, where the maximum MMG size is fixed, for a general PF table, adding a new rule can cause two MMGs to be merged into one larger MMG. Consequently, keeping empty slots in the center of a PF table does not necessarily lead to an optimal solution in the worst case. This point is demonstrated in the following example.

最坏的情况，
LPM 16个
PF N/2个

Fig. 3 shows how two MMGs can merge into one MMG by simply adding one rule which partially overlaps with some of the rules from both MMGs. The two MMGs $\mathbf{M}_A$ and $\mathbf{M}_B$ are shown in Fig. 3a and Fig. 3b with five-tuple rules $A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow A_4$ and $B_1 \rightarrow B_2 \rightarrow B_3 \rightarrow B_4$, respectively. A search key can match rules in either $\mathbf{M}_A$ or $\mathbf{M}_B$, but not both. If $\mathbf{M}_A$ and $\mathbf{M}_B$ belong to two different CRGs, rules in $\mathbf{M}_A$ are independent of those in $\mathbf{M}_B$ and, hence, they can be placed independently in the table, as shown in Fig. 3c, i.e., the empty rule entries are placed in the center.

Now, suppose a new rule C, which partially overlaps with A_1 and B_4 as shown in Fig. 3d, is to be added. Assume $B_4 \rightarrow C$ and $C \rightarrow A_1$. After rule C is added, however, $\mathbf{M}_A$ and $\mathbf{M}_B$ merge into one MMG and all the rules in $\mathbf{M}_A$ must be moved to the higher match priority memory locations than C and all the rules in $\mathbf{M}_B$ must be moved to the lower match priority memory locations than C, thereby resulting in all the existing rules being rearranged as shown in Fig. 3d. This example demonstrates that having empty slots in the center of the table does not help to minimize the number of rule moves per rule update in the worst case, as far as a general PF table is concerned. In summary, we conclude that the number of rule moves required for adding a new rule in the worst case for any fast update algorithm is no better than $N_r/2$ for a general PF table with N_r rules.

在表中间放空位置不能减少会泽的移动数量(在最坏情况下)



A_4	1.1.1.1–x.x.x.x–x.x–x.x–5
A_3	1.1.1.x–x.x.x.x–x.x–x.x–5
A_2	1.1.x.x–x.x.x.x–x.x–x.x–5
A_1	1.x.x.x–x.x.x.x–x.x–x.x–x

(a)

B_4	2.2.2.2–x.x.x.x–x.x–x.x–x
B_3	2.2.2.x–x.x.x.x–x.x–x.x–4
B_2	2.2.x.x–x.x.x.x–x.x–x.x–4
B_1	2.x.x.x–x.x.x.x–x.x–x.x–4

(b)

B_4	2.2.2.2–x.x.x.x–x.x–x.x–x
B_3	2.2.2.x–x.x.x.x–x.x–x.x–4
A_4	1.1.1.1–x.x.x.x–x.x–x.x–5
A_3	1.1.1.x–x.x.x.x–x.x–x.x–5
	Empty Slot
	Empty Slot
B_2	2.2.x.x–x.x.x.x–x.x–x.x–4
B_1	2.x.x.x–x.x.x.x–x.x–x.x–4
A_2	1.1.x.x–x.x.x.x–x.x–x.x–5
A_1	1.x.x.x–x.x.x.x–x.x–x.x–x

Increasing match priority

(c)

A_4	1.1.1.1–x.x.x.x–x.x–x.x–5
A_3	1.1.1.x–x.x.x.x–x.x–x.x–5
A_2	1.1.x.x–x.x.x.x–x.x–x.x–5
A_1	1.x.x.x–x.x.x.x–x.x–x.x–x
	Empty Slot
C	x.x.x.x–x.x.x.x–x.x–x.x–6
B_4	2.2.2.2–x.x.x.x–x.x–x.x–x
B_3	2.2.2.x–x.x.x.x–x.x–x.x–4
B_2	2.2.x.x–x.x.x.x–x.x–x.x–4
B_1	2.x.x.x–x.x.x.x–x.x–x.x–4

(d)

Fig. 3. A new MMG is formed by combining two MMGs and a new rule. (a) MMG $\mathbf{M}_A$. (b) MMG $\mathbf{M}_B$. (c) Original table. (d) Table after rule C is inserted.

4 Proposed Solution for TCAM Policy Table Update without Locking

The previous section demonstrated that locking the TCAM PF table for rule update can be harmful. In this section, the proposed *C*onsistent *P*olicy *T*able *U*pdate *A*lgorithm (CoPTUA) is described in detail. The goal is to eliminate the need for locking the TCAM table while ensuring consistent and error-free rule matching during a rule update process. In CoPTUA, a batch of rules is updated together to minimize the update delay. A rule update process includes three steps: 1) deleting rules that need to be removed, 2) rearranging the remaining rules, and 3) adding new rules. The idea behind CoPTUA is to maintain the PF table consistency and ensure error-free rule matching during the update process. The PF table *consistency* is maintained if, for each rule move, a search key matching results in the same rule as the one that would be matched before the rule move, as well as, for each rule addition or deletion, a search key matching results in the same rule as the one that would be matched just before or after the addition or deletion. *Error-free rule matching* is achieved if direct rule overwriting can be avoided for rule update. CoPTUA meets both conditions and allows the PF table update process to take place without locking and

一致性：更新前后查询结果都一样

避免规则的直接覆盖

yet poses zero impact on the data path processing. The following subsections present CoPTUA in detail.

4.1 Hardware Capability

We summarize here the TCAM coprocessor capabilities required in our solution, which hold true for most of the existing TCAM coprocessors:

1. Each TCAM rule entry has a valid bit associated with it. To activate a rule entry, this valid bit needs to be set. Otherwise, the rule entry is considered inactive or empty and it will never be matched. Consequently, the deletion of a rule is nothing more than resetting the valid bit and adding a rule does not take effect until this valid bit is set.
2. After a rule is matched, resetting the valid bit has no effect on the action return process. In other words, deleting a rule cannot stop the return of the action for that rule to the network processor if a match for that rule occurs prior to the deletion operation.
3. Resetting the valid bit for a best matched rule between two successive partial key matches causes the rule to not be matched. Instead, the second best rule is matched.
4. The TCAM is dual port, accessible both from a local CPU and a network processor simultaneously.

TCAM是双端口，可从本地访问CPU和网络处理器同时进行。

4.2 Update without Policy Table Lock

There are two possible types of incorrect rule matching during the update process without PF table locking: 1) erroneous rule matching and 2) inconsistent rule matching. Erroneous rule matching may occur if a rule gets a match while it or its corresponding action is partially updated. Inconsistent rule matching means that the rule that gets a match is not the best matched rule. Inconsistent rule matching may occur when a key matching takes place in the middle of a rule update process, which does not guarantee table consistency until the process finishes. In what follows, we identify the conditions for avoiding erroneous and inconsistent rule matching.

Erroneous rule matching can be avoided if the following condition is met: No update related operations are performed on a rule and/or its corresponding action if the valid bit of that rule is set, with the exception of delete operations, i.e., resetting the valid bit. To meet this condition, all that needs to be done is to avoid overwriting an existing rule with its valid bit set. To this end, the overwriting operations need to be decomposed into three steps as follows:

如果设置了规则的有效，则没有对该规则相应的更新操作。删除操作例外

- *Step* 1: A delete process, which involves only a single operation to reset the valid bit of the existing rule.
- *Step* 2: A write process, which involves multiple operations to add a new rule and its corresponding action.
- *Step* 3: Setting the valid bit for the new rule.

Based on capabilities (2) and (3) in the previous section, Step 1 cannot cause any erroneous rule matching. Capability (1) ensures that Step 2 also meets the condition. Finally, Step 3 obviously meets the condition. Note that a *writing* process over an empty slot only includes Step 2 and Step 3.

Inconsistent rule matching can be avoided if the following conditions are met: 1) For each rule move, a search key matching results in the same rule as the one that would be matched before the rule move and 2) for each rule addition or deletion, a search key matching results in the same rule as the one that would be matched just before or after the addition or deletion.

规则的移动、删除、插入前后，匹配的规则是一样的

Any PF table update algorithm that meets the above conditions guarantees that the PF table update process poses zero impact on the data path (or TCAM lookup) process, thus eliminating the need for TCAM PF table locking. In the next section, we propose such an algorithm for general TCAM PF table update.

我们要寻找一个满足上述要求的算法即可

4.3 Consistent PF Table Update for OTCAM

In what follows, we simply use *move* to represent a rule move process which is composed of a write process to write a rule to a new location and then a delete process to delete the rule from its old location. As we shall see in the next section, for search key matching which requires n clock cycles, the delete process must be delayed by $n-1$ clock cycles to ensure consistent rule matching. Similarly, we simply use *write* to represent a write process. Before describing the proposed algorithm, let us first present two theorems.

Theorem 1. *After a rule is deleted from a PF table, the consistency for all the remaining rules in the PF table is maintained.*

Proof. Deleting a rule can only cause the release of a match priority relationship among the rest of the rules. Hence, any rules with original match priority relationship either still preserve the same relationship or become independent as a result of the removal of some other rule(s). In either case, the remaining rules can stay in their original memory locations without causing inconsistency. □

Note that, when adding a new rule, care must be taken to ensure that it will not be in conflict with the match priority relationship for the existing rules. For example, assume $A \rightarrow B$. It is not allowed to simply add rule C and expect to have $B \rightarrow C \rightarrow A$, i.e., reverse the priority relationship between A and B. In this paper, we assume that, to reverse the match priority of two existing rules, the following procedure is followed: 1) One delete process to remove one of the two rules; 2) one add process to add that rule back at a different location which reverses the priority relationship between the two rules. This implies that to go from $A \rightarrow B$ to $B \rightarrow C \rightarrow A$ involves one delete process and two add processes instead of a simple add process to add C. With this assumption, we immediately have the following result:

Theorem 2. *Adding a new rule does not change the match priority relationship among the existing rules in an MMG.*

CoPTUA is based on the above two theorems. The basic idea is the following: Given a batch of updates to be performed including one or multiple rule deletions and/or additions, CoPTUA first deletes all the rules which do not appear in the final configuration that is calculated in the

? ?

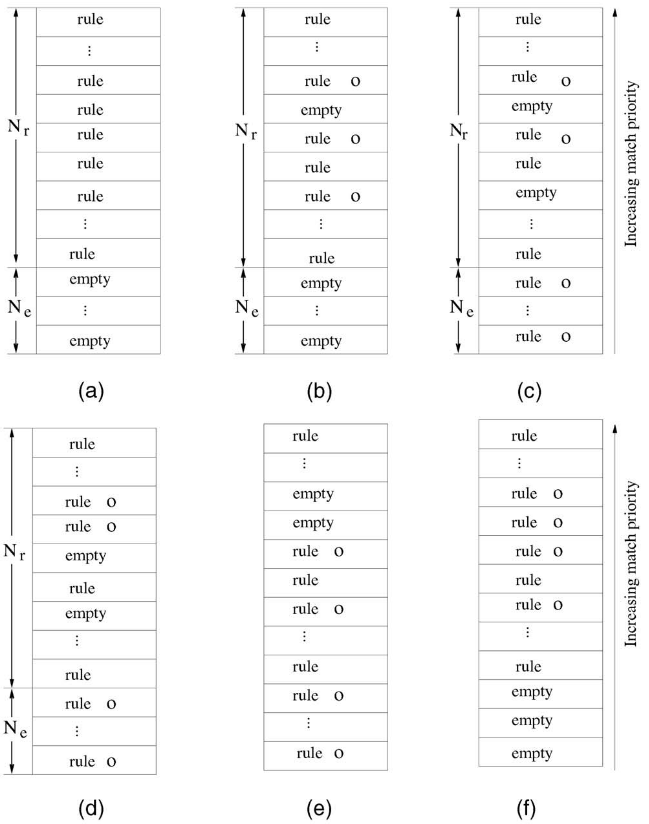


Fig. 4. The table configuration in OTCAM with all empty entries at the bottom. (a) Original table with N_r rules. (b) After some rules are deleted, all relevant rules are marked as "o." (c) After N_e relevant rules with lowest match priority are moved into the empty entries at the bottom. (d) After the remaining relevant rules are moved toward the top. (e) All relevant rules are in order. (f) The last configuration by moving relevant rules toward the top and adding all new incoming rules.

control plane, i.e., those rules which need to be deleted. Every rule deletion results in a partially updated consistent PF table, according to Theorem 1. Then, the existing rule orders are rearranged to the final configuration without adding the new rules but with the corresponding rule entries allocated. Note that rule rearrangement must follow a given procedure to ensure table consistency. This will result in a consistent intermediate configuration, which is equivalent to the configuration before the rearrangement, i.e., the configuration just after the rules were deleted. This is true because all the rules whose relative orders have been changed due to this rearrangement must be those rules which have no priority relationship. Otherwise, their relative orders are not changed in the final configuration, according to Theorem 2. Finally, CoPTUA adds the new rules. CoPTUA is described in detail below.

In CoPTUA, all the N_e empty rule entries are kept at either the top or the bottom of the PF table. Now, suppose initially all the rules are placed at the top of the PF table (i.e., lower memory addresses) as shown in Fig. 4a.

empty位置在top

First, delete all the rules which do not appear in the final configuration, resulting in an intermediate configuration as shown in Fig. 4b.

Second, a procedure needs to be specified to properly rearrange the existing rules before any new rules can be added. To this end, note that rules from different CRGs can be interleaved in arbitrary order. Hence, only those rules which are in the same CRG to which any new incoming rule belongs may have to be rearranged. These rules are defined to be *relevant* and all others are *irrelevant* with respect to the rules to be added. To facilitate further discussion, we mark all the relevant rules with "o" in Fig. 4b, Fig. 4c, Fig. 4d, Fig. 4e, and Fig. 4f.

和要加入的规则相关的用0表示

Now, the rearrangement procedure is as follows: First, move the relevant rules in *increasing match priority order* in which the lower match priority rules are moved before higher match priority rules to the available lowest match priority location in the N_e empty memory space at the bottom. In the case that a particular entry in the N_e rule entries is supposed to be taken by a newly added rule, that entry is left empty. The intermediate configuration after this rearrangement is shown in Fig. 4c. Next, the relevant rules in the top N_r entries are moved as closely toward the top as possible in a decreasing match priority order. This is because, after pushing the relevant rules close to the top, some empty entries are released and then the relevant rules can be rearranged close to the bottom. This creates at least N_e empty entries below all relevant rules in the top N_r entries as shown in Fig. 4d. The following step is to rearrange all the relevant rules in the top N_r rule entries with all empty entries in the lowest priority locations. In this configuration, the structure of the relevant rules in the top N_r entries in Fig. 4d is now identical to the one in Fig. 4b except that there is no empty entry among the relevant rules. Hence, the same rearrangement procedure, as in Fig. 4c and Fig. 4d is iteratively applied to the top entries until all the relevant rules are placed below all the empty entries, as shown in Fig. 4e. The empty entries here do not include those allocated to the new rules which are yet to be added. Subsequently, all the relevant (or irrelevant) rules are moved toward the top (or bottom) to fill all the available empty entries in decreasing (or increasing) match priority order, depending on which one requires the smaller number of moves.

在与新加入的规则相关联的规则中，优先级较低的Ne个移动到底部。

把剩余的相关规则经可能的往顶部推，中间空出来给相关的规则

Finally, the new rules are added to the preallocated empty rule entries in decreasing match priority order. Fig. 4f gives the final configuration when all the relevant rules are moved to the top.

CoPTUA is formally stated in Fig. 5 for a table with all rules at the top. The update process for a table with all rules at the bottom is similar. Note that, in this case, the relevant rules move to the top empty entries following a decreasing priority order and the relevant rules move to the bottom of the table following an increasing priority order.

Now, we use an example to illustrate how CoPTUA works. Assume that a PF table is composed of three MMGs belonging to three different CRGs, i.e., $\mathbf{M}_D$ $\{D_3 \rightarrow D_2 \rightarrow D_1\}$, $\mathbf{M}_E$ $\{E_3 \rightarrow E_2 \rightarrow E_1\}$, and $\mathbf{M}_F$ $\{F_2 \rightarrow F_1\}$, as shown in Fig. 6a. Suppose that a batch of updates includes the deletion of E_2 and additions of G and H. Further assume:

1. The deletion of E_2 breaks $\mathbf{M}_E$ into two separate single-rule CRGs.
2. The addition of rule G merges E_3 and E_1 back into one new MMG M_G, i.e., $E_1 \rightarrow G \rightarrow E_3$.

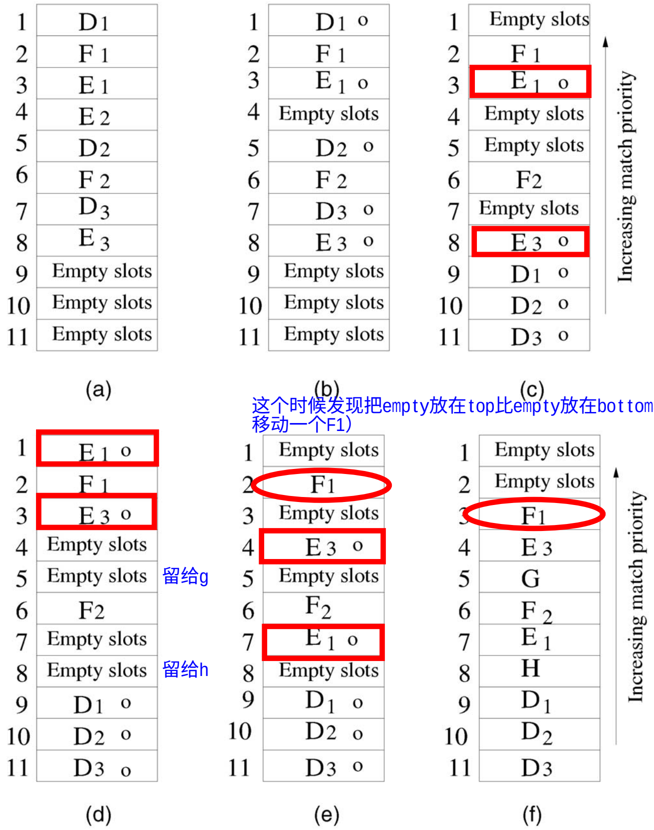
```
CoPTUA Procedure{
 Initially, all the rules are located at the top
 Delete all the rules which are not in the final configuration
 WHILE (Number of relevant rules above the empty entries $\neq 0$)
 {
   Move the relevant rules which are above the empty entries into the
      empty entries at the bottom of the table in increasing match
      priority order. If a rule to be written is not a rule in the original
      table, an empty entry is kept at that location
   Move all the relevant rules except the ones which have been moved
     to the bottom toward the top in decreasing match priority order
 }
 IF (Number of moves of relevant rules to the top $<$ Number of moves
     of irrelevant rules to the bottom)
   Move all the relevant rules towards the top in decreasing match
     priority order
 ELSE
   Move all the irrelevant rules towards the bottom in increasing match
     priority order
 Write all the new rules to their respective empty entries in decreasing
   match priority order
}
```

Fig. 5. CoPTUA for a table with all rules on the top.

3. The addition of rule H further merges all the rules in M_D and M_G into one MMG, i.e., $D_3 \rightarrow D_2 \rightarrow D_1 \rightarrow H \rightarrow E_1 \rightarrow G \rightarrow E_3$.

By running CoPTUA, four major intermediate steps are identified which correspond to four consistent PF table formats, as shown in Fig. 6b, Fig. 6c, Fig. 6d, and Fig. 6e. First, E_2 is deleted, as depicted in Fig. 6b. The table is partially updated as rule E_2 is deleted. In Fig. 6c, the three relevant rules with the lowest match priorities are moved to the bottom of the table in increasing match priority order. The rule order relationship is kept the same as in Fig. 6b. In Fig. 6d, the relevant rules at the top are moved toward the top end in decreasing match priority order. The order does not change in this step. In Fig. 6e, all the relevant rules are in their final order and below all empty entries. Note again that empty entries at 5 and 8 are kept for the two new rules and should not be considered as empty entries here. The relative order for rules E_1 and E_3 is reversed, which does not introduce inconsistency since they are independent at this point. Since moving all the irrelevant rules to the bottom requires a smaller number of moves than that of moving all the relevant rules to the top, F_1 is moved toward the bottom. Finally, the new rules G and H are added to complete the batch updates, resulting in the final configuration, as shown in Fig. 6f.

A salient feature of CoPTUA is that the worst-case rule update performance is independent of the number of rules to be updated in a batch. This feature allows CoPTUA to yield an upper bound on the update delay performance, which is independent of rule structures and update patterns, as we shall see in Section 5.



	(a)	(b)	(c)	(d)	(e)	(f)
1	D1	D1 o	Empty slots	E1 o	Empty slots	Empty slots
2	F1	F1	F1	F1	F1	Empty slots
3	E1	E1 o	E1 o	E3 o	Empty slots	F1
4	E2	Empty slots	Empty slots	Empty slots	E3 o	E3
5	D2	D2 o	Empty slots	Empty slots	Empty slots	G
6	F2	F2	F2	F2	F2	F2
7	D3	D3 o	Empty slots	Empty slots	E1 o	E1
8	E3	E3 o	E3 o	Empty slots	Empty slots	H
9	Empty slots	Empty slots	D1 o	D1 o	D1 o	D1
10	Empty slots	Empty slots	D2 o	D2 o	D2 o	D2
11	Empty slots	Empty slots	D3 o	D3 o	D3 o	D3

这个时候发现把empty放在top比empty放在bottom移动的要少（只用移动一个F1）留两个空位就行

Fig. 6. An example of CoPTUA update process. (a) Original table. (b) After E_2 is deleted and the relevant rules are marked as "o." (c) After the three lowest relevant rules are moved to the bottom. (d) After the remaining relevant rules are moved toward the top. (e) After the top relevant rules are moved below the empty entries. (f) Final configuration after the irrelevant rules are moved toward the bottom and the new rules G and H are written.

4.4 Proof of Correctness of CoPTUA

To prove the correctness of CoPTUA, we need to introduce two lemmas. First, we note that, in the middle of a move process, a rule may be duplicated just after it has been written to a new location, while the same rule in the old location is yet to be deleted. The following lemma states under what condition duplicated rules may coexist without causing inconsistent rule matching.

复制 共存

首先，我们注意到，在移动过程中，规则可能在写入新位置后重复，而旧位置中的相同规则尚未删除。

Lemma 1. *For any two rules $A \rightarrow B$, if there is at least one copy of B such that all the copies of $A < B$, then the PF table consistency is maintained.*

Proof. In this case, any search key which matches both A and B will result in the return of the action associated with B, which is desired. □

虽然有复制，但只要复制后，相关联的规则之间的顺序保持不变

Fig. 7 shows three configurations with duplicated rules. Here, $A \rightarrow B \rightarrow C$, $A \wedge D = \emptyset$, and $B \wedge D = \emptyset$. It is easy to check that all the rules in the three configurations satisfy the condition of Lemma 1 and, hence, all three tables are consistent.

Lemma 2. *Assume a search key matching takes n clock cycles. In a rule move process, the rule in the old location cannot be deleted until the nth clock cycle after the rule has been written to its new location.*

假设搜索关键字匹配需要n个时钟周期。在规则移动过程中，在将规则写入新位置之后的第n个时钟周期之前，不能删除旧位置中的规则

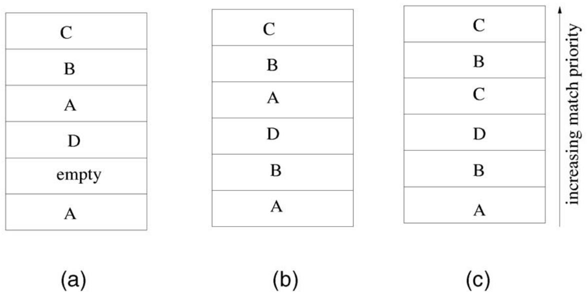


Fig. 7. Some examples of consistent table configurations with duplicated rules satisfying the condition of Lemma 1. Here, $A \rightarrow B \rightarrow C$, $A \wedge D = \emptyset$, and $B \wedge D = \emptyset$.

Proof. Let t_{ms} and t_{me} be the time instants the search key match starts and ends, respectively. Note that $t_{me} - t_{ms} = n$. Let t_a be the instant the rule is activated at its new location and t_d be the instant the rule is deleted at the old location. There are two different match cases: 1) $t_{ms} \le t_a$ and 2) $t_{ms} > t_a$. In the first case, to be consistent, we must have $t_{me} \le t_d$. Consistency is guaranteed if $t_{me} = t_{ms} + n \le t_a + n \le t_d$. In the second case, $t_{ms} > t_a$, for which the rule at the new location is already valid at the beginning of the search key match process. In summary, consistent rule matching is guaranteed as long as $t_a + n \le t_d$. □

Theorem 3. *CoPTUA maintains PF table consistency throughout the update process.*

Proof. As shown in Fig. 5, the first phase in the update process is to delete all the rules which do not appear in the final configuration. According to Theorem 1, the PF table consistency is guaranteed in this phase. The second phase is an iterative rearrangement process. In this process, only relevant rules may have to be rearranged and all the irrelevant rules are not affected. Relevant rules are moved from the top (bottom) toward the bottom (top) in increasing (decreasing) match priority order. This process satisfies the condition in Lemma 1, provided that each move satisfies the condition in Lemma 2. The third phase is to either move all the relevant rules to the top in decreasing match priority order or all the irrelevant rules to the bottom in increasing match priority order. Just as in phase two, the PF table consistency is guaranteed for each move since the conditions in Lemmas 1 and 2 are satisfied. The last phase is to add all the new rules with match priority relationship in decreasing order. After each new rule is added, the PF table is a partially updated consistent table because each new rule is located in their preallocated final location. Moreover, adding new rules with match priority relationship in decreasing order ensures that, if the matched rule is a new rule, it is the best one possible. □

5 Performance Analysis and Evaluation

As mentioned in Section 1, to ensure error-free and consistent rule matching, CoPTUA tends to require a larger number of operations and also a larger number of empty memory entries than traditional approaches based on database locking. Hence, there are two critical concerns for CoPTUA, i.e., rule update time and memory efficiency. These concerns are resolved in this section by analytical and simulation studies under various rule structures.

TABLE 1
Parameter Definition

N	PF table size in the units of the number of rule entries
N_r	maximum number of rules in the PF table ($N_r < N$)
R	number of relevant rules for each batch update
N_e	number of empty rule entries ($N_e = N - N_r$)
α	percentage of rule entries which are empty ($\alpha = N_e/N_r$)
β	ratio between number of empty entries and number of relevant rules (i.e., $\beta = N_e/R$)
W	number of write operations per batch update
D	number of delete operations per batch update
W_w	worst-case number of write operations per batch update
D_w	worst-case number of delete operations per batch update
n	number of clock cycles for each search key match
t_w	number of clock cycles for writing a rule to TCAM coprocessor
t_d	number of clock cycles for deleting a rule from TCAM coprocessor

5.1 Number of Write and Delete Operations per Batch Update

In CoPTUA, the number of write and delete operations in a batch update process is dependent on the number of relevant rules and the number of available empty entries in the PF table. Here, we give analytical upper bounds on the required number of write and delete operations for a batch rule update. Table 1 lists the definitions for all the necessary parameters.

给出一个添加、删除操作的上阶

First, consider a batch update process that does not delete rules but only adds some rules to the PF table with all empty rule entries at the bottom. The batch update process is executed following the process described in the previous section. The main part of the update operation is an iterative process, as illustrated in Fig. 4c and Fig. 4d. The first step of each iteration is to write at least N_e lowest priority relevant rules into the empty entries at the bottom of the table and to delete up to N_e redundant rules. The number of either write or delete operations is at most N_e in this step. The second step of each iteration is to move the remaining relevant rules toward the top of the PF table. This step costs up to $R - iN_e$ write or delete operations in the ith iteration. Hence, the total number of operations for either write or delete is at most $(R - (i-1)N_e)$ in the ith iteration. The iterative process continues until all the relevant rules are moved to their final order and below all empty entries. The maximum number of iterations is $\lceil 1/\beta \rceil$. Then, the relevant (or irrelevant) rules are moved to the top (or bottom), whichever requires fewer moves. This step costs no more than the minimum of R and $N_r - R$ rule moves. Finally, the new rules are added to the preallocated empty entries and these write operations have been accounted for in the

这里的迭代是指 一匹操作里 有多少组相关的规则，

这里的迭代指上面的c-d步骤

e步骤

iterative process. Hence, the total number of write and delete operations per batch update is

$$\begin{aligned} W = D &= \sum_{i=1}^{\lceil 1/\beta \rceil} (R - (i-1)N_e) + min(R, N_r - R) \\ &= \frac{1}{2}\left(1 + \left\lceil \frac{1}{\beta} \right\rceil\right) R + min(R, N_r - R). \end{aligned} \tag{1}$$

The larger the β value is, the smaller the number of write and delete operations required for each batch update. When $\beta \geq 1$, both W and D reach their minimum value $min(2R, N_r)$.

Now, if the batch update process also includes the deletion of N_d rules, the update process first deletes these rules. Hence, N_d extra delete operations are required and

$$\begin{aligned} D &= \frac{1}{2}\left(1 + \left\lceil \frac{1}{\beta} \right\rceil\right) R + min(R, N_r - R) + N_d \\ &\leq \frac{1}{2}\left(3 + \left\lceil \frac{1}{\beta} \right\rceil\right) R + min(R, N_r - R). \end{aligned} \tag{2}$$

From (1) and (2), we note that the number of write or delete operations is proportional to the number of relevant rules. In the worst case, all the rules in the table are relevant, i.e., $R = N_r$ and $\beta = \alpha$. Then, the worst-case upper bounds on the numbers of write and delete operations are as follows:

$$W_w = \frac{1}{2}\left(1 + \left\lceil \frac{1}{\alpha} \right\rceil\right) N_r, \tag{3}$$

$$D_w = \frac{1}{2}\left(3 + \left\lceil \frac{1}{\alpha} \right\rceil\right) N_r. \tag{4}$$

Although the above results are derived based on the assumption that all the empty rule entries are at the bottom of the PF table, the results also hold true when they are at the top of the table. This is because the two cases are symmetric and no extra operations are required for one versus the other.

Fig. 8 plots the functional relationship between D_w and W_w and α. Note that the number of write operations is N_r in the worst case if the PF table is locked for rule update. CoPTUA can achieve the same performance in terms of the number of write operations if $\alpha \geq 100\%$. As α decreases, both D_w and W_w increase. For instance, for $\alpha = 1\%$, $W_w = 50.5N_r$ and $D_w = 51.5N_r$. The following subsection quantifies the maximum delay per batch update.

5.2 Upper Bound on Worst-Case Delay per Rule Update

According to Lemma 2, in a move process, the deletion of a rule at its old location must be delayed $n-1$ clock cycles after the rule is activated at its new location. Hence, an extra delay of $t_{mv} = n-1$ clock cycles needs to be added to each move. We include this delay into each write time.

In practice, one can avoid t_{mv} delay for most of the rule moves by writing a batch of rules to their corresponding new locations before deleting them. For example, for the rule moves as shown in Fig. 4c, when the lowest N_e number of rules are moved into the empty entries at the bottom, one may write and activate all N_e rules in their new locations before deleting all the redundant rules from their old locations. In this case, at most one extra delay of $t_{mv} = n-1$ clock cycles is involved for up to N_e moves. However, in the following worst-case analysis, we assume that each move incurs an extra delay of $t_{mv} = n-1$ clock cycles. The worst-case upper bound for update process time, t_u, can be expressed as:

批量写就只有一次delay

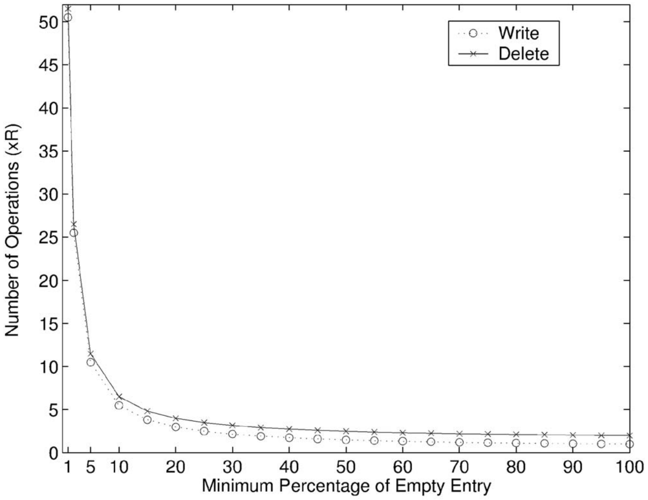


Fig. 8. The number of write and delete operations versus the percentage of empty rule entries in the worst case.

$$t_u = W_w(t_w + t_{mv}) + D_w t_d. \tag{5}$$

仍假设move每次都需要delay

Clearly, the smallest possible batch update interval is t_u in the worst case. Hence, the worst-case delay per rule update at this batch update interval is $2t_u$. This is because a new rule may come just after the last update process begins and it is updated and activated at the end of the next update process.

这是因为新规则可能在最后一次更新过程开始后到来，并在下一次更新过程结束时更新并激活。

Equations (1)-(5) quantitatively characterize the relationship among all the parameters involved. They can be used to guide the OTCAM coprocessor resource provisioning. A designer can adjust any of these parameters to obtain a bounded maximum update delay. For example, if the rule enforcement can tolerate a longer delay, the higher TCAM utilization can be achieved. On the other hand, increasing the local CPU write speed can raise the OTCAM utilization without incurring longer update delay.

Now, let us estimate the worst-case delay per rule update by plugging in the parameter values based on the state-of-the-art technologies. As stated in Section 2, Intel IXP2800 PCI bus is 64-bit wide and runs at 66 MHz clock rate (15 ns per clock cycle). We assume that the same CPU interface is available for the TCAM coprocessor. Then, each write operation requires five clock cycles for a 104-bit rule with 64-bit action, plus one clock cycle for activating the rule. Then, $t_w = 15 \times 6 = 90\ ns$. For simplicity, assume the TCAM clock rate is also 66 MHz (in general, it is larger) and a search key matching takes two TCAM clock cycles, so $t_{mv} = 15\ ns$. Each delete takes one clock cycle, i.e., $t_d = 15\ ns$. Fig. 9 depicts the result for the maximum delay per rule update (i.e., two t_u) versus α with $N_r = 100,000$ rules.

The maximum delay per rule update using CoPTUA is a little above 1.2 seconds at $\alpha = 1\%$. This delay reduces to less

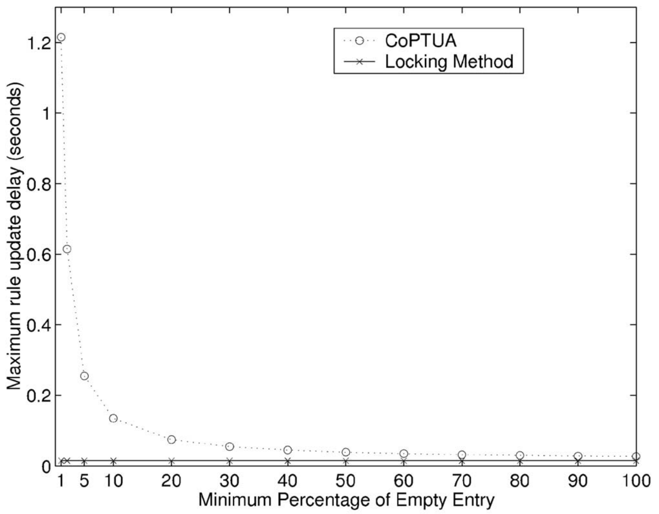


Fig. 9. The maximum delay versus the percentage of empty rule entries in the worst case for $N_r = 100,000$.

than one second when $\alpha \geq 2\%$. Given that the policy enforcement is either controlled manually by network administrators or by a remote policy server, enforcing a rule usually takes seconds to minutes to accomplish. Therefore, this maximum delay is negligible. Also plotted in Fig. 9 is the maximum delay per rule update when an algorithm based on locking is used. This is about 0.015 seconds, independent of α. During the lock period (0.0075 seconds, half of the maximum delay per rule update), however, up to 0.1875 million packets can be dropped, assuming that the network processor handles 10 Gbps line rate, resulting in significant performance degradation.

The above analysis clearly demonstrates the viability and importance of CoPTUA for TCAM PF table update. With CoPTUA, an OTCAM can then provide true maximum and deterministic throughput performance guarantee for data path processing.

5.3 Performance Evaluation by Simulation

The previous section presented the analytical upper bound on the worst-case delay per rule update as a function of the percentage of empty rule entries. In this subsection, we study the maximum delay per rule update by simulation.

The real PF tables available today are generally small, ranging from a few tens to a few thousand rules in a PF table. For such small databases, the update delay in CoPTUA is negligible even if all the rules are relevant. To test the performance of CoPTUA under large database systems, we adopt an approach used in [3], [6], [7], where large numbers of five-tuple PF tables are synthesized using small real databases as seeds. We synthesize up to 100,000 five-tuple rules based on a small real database with 195 rules and some other rule statistics, as observed in [3], [6], [7].

使用小型真实数据库作为种子合成大量五元组PF表

In our seed database, about 40 percent of both source and destination IP addresses are wildcarded. Among these, about 15 percent are 0 length prefixes (i.e., the whole address is wildcarded) and other prefix lengths are 8, 16, 24 bits. About 30 percent of the port numbers have wildcard bits. The protocol number is specified in all the rules and there are only four types of protocols: TCP (Transport Control Protocol), UDP (User Datagram Protocol), IP (Internet Protocol), and ICMP (Internet Control Message Protocol). In our synthesized database, all the rule subfields except protocol number have a 50 percent chance having wildcard bits and 20 percent of the subfields that have wildcard bits are all-wildcarded subfields. We vary the probability, P_w, for the protocol subfield to be all-wildcarded between 1 percent and 5 percent. If the subfield value is an exact number (i.e., no wildcard bit in the subfield), it has a 50 percent chance of being picked from one of eight possible values and a 50 percent chance of being picked from one of the remaining 248 values. The time for writing a rule takes 90 ns and, for deleting a rule, takes 15 ns, the same as for Intel IXP 2800.

全通配

The simulation results shown in this paper are based on the above parameter setup. Our simulation study with various other parameter setups (not shown in this paper) concludes that the most important factor affecting the update delay performance for CoPTUA is the average number of overlapping rules per rule for a given N_r. In other words, the update delay performance for CoPTUA is insensitive to the change of parameter setups as long as the number of overlapping rules per rule is fixed. The larger the average number of overlapping rules per rule, the larger is the number of rule moves for each batch update and, consequently, the larger is the update delay. Hence, although our simulation parameter setup may not faithfully mimic the possible parameter setup for future real world PF tables, it is expected to provide useful data which reflects the actual performance of CoPTUA. For this reason, we simply use P_w as a tuning knob to generate a wide range of average number of overlapping rules per rule, with all other parameters fixed.

平均重叠规则数

In our simulation, the maximum N_r number of rules may be supported in a PF table and at least 1 percent of N_r (i.e., $N_e = 0.01N_r$) rule entries are kept empty. The rule update requests are assumed to follow a Poisson arrival process and the average update request rate is set to 100 per second. Each update request has 50 percent probability of adding a new rule and 50 percent probability of deleting an existing rule. The actual update process is such that, after an update period (the time between the beginning and end of an update), the next update process starts immediately as long as there is at least one update request in the request queue. If there is more than one request in the queue, all the requests will be processed as a batch in the next update process. The update delay for each rule addition is defined as the interval between the time the rule is activated and the time the request is received. For each pairing of N_r and P_w values, 20,000 updates are simulated and the maximum delay is collected.

每次更新周期中间就进行下一次更新，且把当前更新请求队列里的所有请求合并成一次 批操作

每个规则添加的更新延迟：激活规则的时间与接收请求的时间之间的间隔。

Fig. 10 shows the average number of overlapping rules per rule versus N_r for different P_w values. As expected, the average number of overlapping rules per rule increases as N_r increases. For a given N_r value, a larger P_w value results in a larger average number of overlapping rules per rule. This is because, as P_w becomes larger, a larger number of rules will have an all-wildcarded protocol subfield, making it more likely for rules to overlap with each other. For

随着Pw变大，大量规则将具有全通配的协议子字段，

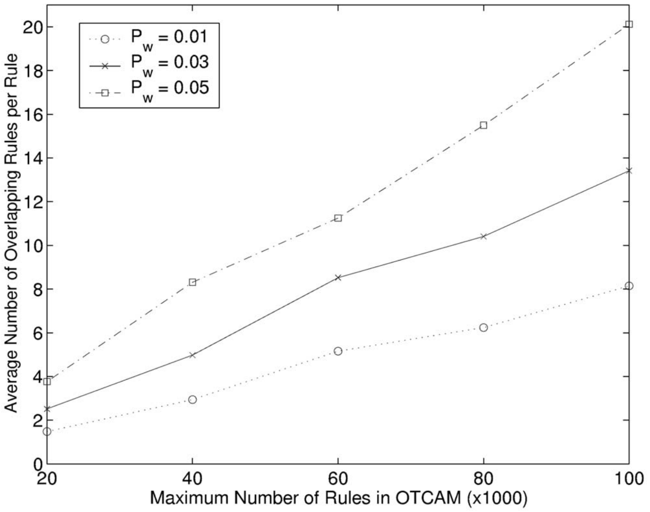


Fig. 10. The average number of overlapping rules per rule.

example, at $P_w = 0.05$, the average number of overlapping rules per rule increases from below 5 to more than 20 as N_r increases from $20,000$ to $100,000$.

Fig. 11 depicts the maximum number of relevant rules per update versus N_r. The maximum number of relevant rules per update is 20-60 percent of N_r. For example, at $N_r = 100,000$, the maximum number of relevant rules is about 60,000 for $P_w = 0.05$, i.e., 60 percent of N_r. This indicates that the number of relevant rules in the worst case is on the order of N_r.

Fig. 12 shows that the maximum rule update delay at $N_r = 100,000$ is about 0.35 seconds, much smaller than the theoretical upper bound (about 1.2 seconds) derived in the previous section.

The above results clearly indicate that the rule update delay using CoPTUA is negligible for a PF table of size as large as 100,000 rule entries and memory utilization as high as 99 percent. Therefore, in practice, for any PF table size in an OTCAM, using CoPTUA for rule update causes zero impact on the data path processing, while ensuring minimum rule update delay and providing high memory utilization.

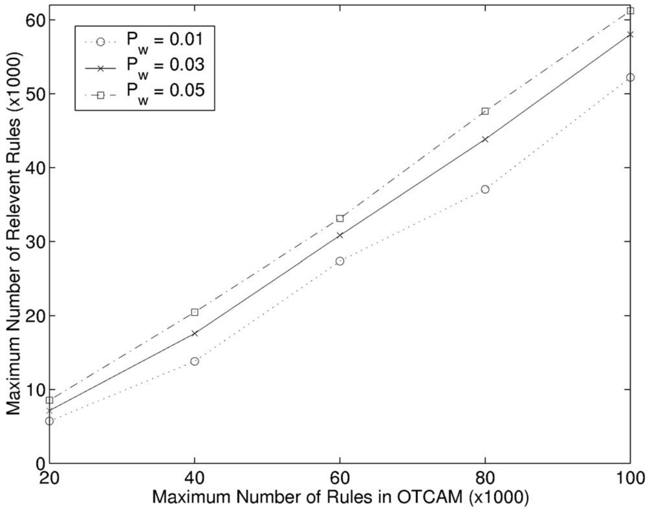


Fig. 11. Maximum number of relevant rules per update.

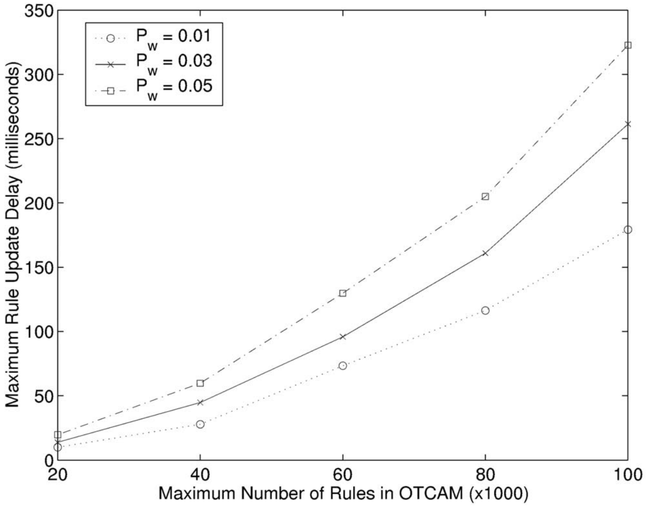


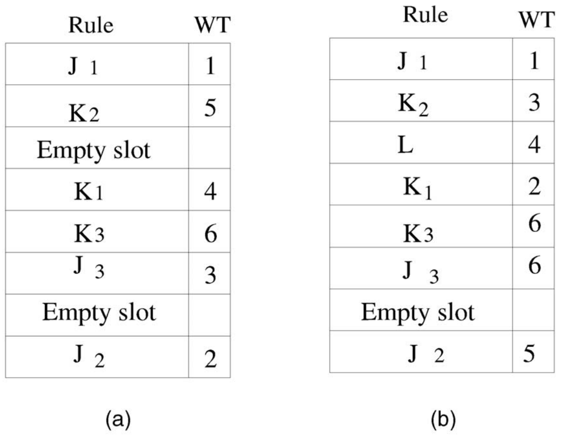
Fig. 12. Maximum rule update delay.

6 Consistent Rule Update for LPM and WEITCAM-Based Policy Table

The number of rule moves for LPM table in the worst case is much smaller than that for PF table update. Thus, the LPM table has a much smaller chance of getting an inconsistent or erroneous rule matching without table locking during the update process. However, if consistent and error-free LPM must be maintained without TCAM locking, the CoPTUA should be used.

Since the LPM table update is a special case of the general policy table update, CoPTUA can be directly applied to the LPM table update. However, as mentioned before, any algorithm that meets the two conditions in Section 4.2 does not require table locking while imposing zero impact on data path processing. It can be easily verified that the two algorithms proposed in [16] satisfy the consistency condition. The error-free condition is met as long as the overwriting follows the three-step procedure specified in Section 4.2. Hence, the two algorithms in [16] can be easily modified to allow rule updating without locking the LPM table. The added operations are valid bit set/reset to avoid direct rule overwriting and the $n-1$ cycles of waiting period in a rule move process to satisfy the conditions in Lemma 2. As the maximum number of rule moves is 16 for the algorithms proposed in [16], the maximum number of added delete operations is 16 and the maximum waiting interval is 16 clock cycles, which amounts to only about 480 ns extra delay per rule update. In contrast, locking the LPM table for the move of 16 rules can affect the data processing of up to 18 packets at OC-192 line rate, as mentioned in Section 2.

CoPTUA works even better for WEITCAMs [4]. For policy table update in a WEITCAM, no extra empty rule entries are required, meaning that the policy table can be fully utilized. Again, given a batch of updates to be performed including one or multiple rule deletions and additions, the rule deletions are performed first, which will



(a)

Rule	WT
J 1	1
K2	5
Empty slot	
K1	4
K3	6
J 3	3
Empty slot	
J 2	2

(b)

Rule	WT
J 1	1
K2	3
L	4
K1	2
K3	6
J 3	6
Empty slot	
J 2	5

Fig. 13. A WEITCAM table with six rules, three weight bits for each rule. (a) Original table. (b) After rule L is inserted.

not cause any inconsistency. To add a rule, instead of having to move some of the existing rules around, as is the case for an OTCAM, the weight values of some of the existing rules may need to be changed. To maintain consistency, the weight values for the existing rules must be updated before the new ones are added. Since a weight value update requires only one clock cycle, it is valid to match either the new or the old weight value. In other words, rule weight subfield overwriting is allowed and no erroneous search key matching can occur while the weight value is being updated.

To ensure consistency while the weights are being changed, changing the weights to larger (smaller) values (here, a larger weight indicates a higher match priority), it must be executed in decreasing (increasing) match priority order. After all the rules in the policy table are set to their final weight, the new rules can then be written and activated to finalize the configuration.

Let us look at an example as shown in Fig. 13. The update process is to add a rule L into a policy table. Assume that $L \cap J_1 \neq \emptyset$, $L \cap J_2 \neq \emptyset$, $L \cap K_1 \neq \emptyset$, and $L \cap K_2 \neq \emptyset$, and the two MMG M_J and M_K initially belong to different CRGs. Rule L has match priority relationships as follows: $J_1 \rightarrow L \rightarrow J_2$ and $K_2 \rightarrow L \rightarrow K_3$. After L is added, the possible new weights for these rules are shown in Fig. 13b. In this case, the weights for J_2 and J_3 are increased which must be updated in decreasing match priority order, i.e., first update J_3 and then J_2. For K_1 and K_2, the weight values are to be reduced and updated in the increasing match priority order, i.e., K_1 must be updated before K_2. Finally, the new rule L is added with weight value 4.

7 Related Work

Only a few published research papers addressed the TCAM memory resource management issues. McAuley and Francis [9] first proposed using TCAM for routing table lookup and discussed some update issues related to the OTCAM. Shah and Gupta [16] proposed two algorithms on the rule table update in the context of the LPM table using OTCAM. One of these algorithms is considered to be optimal in terms of the worst-case number of LPM table operations per entry update.

The power consumption issue is addressed in [15] and [20]. In these methods, the TCAM device is divided into multiple blocks to accommodate an LPM table. Only the power for the block that is being searched is turned on and each match key only needs to search one of these blocks to find the best matched route, thus reducing the power consumption.

Some research efforts have been put on the TCAM table compaction. Liu [11] described two route compacting techniques to reduce the size of an LPM table in an OTCAM to increase the TCAM utilization. He also introduced a range encoding scheme for efficient range matching [12]. Lysecky and Vahid [14] extended Liu's work to perform the TCAM minimization dynamically in the update processor rather than via the network. Lunteran and Enghersen [13] proposed a packet filter rule encoding scheme to reduce the rule length in TCAM. The proposed approach was reported to reduce the rule length significantly.

8 Conclusions

In this paper, we proposed a **Co**nsistent **P**olicy **T**able **U**pdate **A**lgorithm (CoPTUA) for general policy table update in an ordered ternary content address memory (OTCAM). Instead of attempting to minimize the number of rule moves to reduce the locking time, CoPTUA maintains policy table consistency after each rule move, thus eliminating the need for locking the policy table while ensuring the correctness of the rule matching. Thus, the use of CoPTUA for rule update poses zero impact on data path processing.

Our worst-case analysis showed that, even for a policy table with 100,000 rules, an arbitrary number of rules can be updated simultaneously in less than one second, provided that no less than 2 percent of the rule entries are empty. The simulation study showed that the maximum update delay is less than 0.35 seconds for a PF table with 100,000 rules and at least 1 percent empty rule entries. These imply that, with CoPTUA, any new rule can be enforced in less than one second for any practical PF table sizes. Although the proposed technique is targeted at the PF table update in an OTCAM, we demonstrated that the proposed technique can work even better for the PF table update in a WEITCAM.

References

[1] "AMCC Ships 10-Gbit/s Processor," *Light Reading,* 25 Mar. 2002.

[2] M. Adiletta, M.R. Bluth, D. Bernstein, G. Wolrich, and H. Wilkinson, "The Next Generation of Intel IXP Network Processors" *Intel Technology J.,* vol. 6, no. 3, pp 6-18, 2002.

[3] F. Baboescu and G. Varghese, "Scalable Packet Classification," *Proc. ACM SIGCOMM,* 2001.

[4] H. Che, Y. Wang, and Z. Wang, "A Rule Grouping Technique for Weight-Based TCAM Coprocessors," *Proc. 11th Hot Interconnects (HOTI),* 2003.

[5] A. Feldman and S. Muthukrishnan, "Tradeoff for Packet Classification," *Proc. INFOCOM,* 2001.

[6] P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," *Proc. ACM SIGCOMM,* 1999.

[7] P. Gupta and N. McKeown, "Packet Classification Using Hierarchical Intelligent Cuttings," *Proc. Seventh Hot Interconnects (HOTI),* 1999.

[8] N.F. Huang, W.E. Chen, C.Y. Lou, and J.M. Chen, "Design of Multi-Field IPv6 Packet Classifiers Using Ternary CAMs," *Proc. IEEE GLOBECOM,* 2001.

[9] M. Kobayashi, T. Murase, and A. Kuriyama, "A Longest Prefix Match Search Engine for Multi-Gigabit IP Processing," *Proc. Int'l Conf. Comm. (ICC),* 2000.

[10] T.V. Lakshman and D. Stidialis, "High Speed Policy-Based Packet Forwarding Using Efficient Multi-Dimensional Range Matching," *Proc. ACM SIGCOMM,* 1998.

[11] H. Liu, "Routing Table Compaction in Ternary CAM," *IEEE Micro,* vol. 22, no. 1, pp. 58-64, 2002.

[12] H. Liu, "Efficient Mapping of Range Classifier into Ternary-CAM," *Proc. 10th Hot Interconnects (HOTI),* 2002.

[13] J.V. Lunteren and A.P.J. Engbersen, "Multi-Field Packet Classification Using Ternary CAM," *Electronics Letters,* vol. 38, no. 1, pp. 21-23, 2002.

[14] R. Lysecky and F. Vahid, "On-Chip Logic Minimzation," *Proc. 40th Conf. Design Automation,* 2003.

[15] R. Panigrahy and S. Sharma, "Reducing TCAM Consumption and Increasing Throughput," *Proc. 10th Hot Interconnects (HOTI),* 2002.

[16] D. Shah and P. Gupta, "Fast Updating Algorithms for TCAMs," *IEEE Micro,* pp 36-47, 2001.

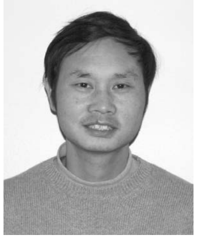
[17] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and Scalable Layer Four Switching," *Proc. ACM SIGCOMM,* 1998.

[18] V. Srinivasan, S. Suri, and M. Waldvogel, "Packet Classification Using Tuple Space Search," *Proc. ACM SIGCOMM,* 1999.

[19] S. Sharma and R. Panigrahy, "Sorting and Searching Using Ternary CAMs," *Proc. 10th Hot Interconnects (HOTI),* 2002.

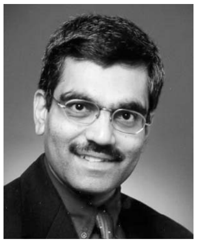
[20] F. Zane, G. Narlikar, and A. Basu, "CoolCAM: Power-Efficient TCAMs for Forwarding Engines," *Proc. IEEE INFOCOM,* 2003.

[21] K. Zheng, C. Hu, H. Lu, and B. Liu, "An Ultra High Throughput and Power Efficient TCAM-Based IP Lookup Engine," *Proc. IEEE INFOCOM,* 2004.

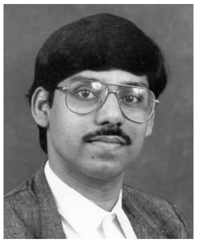


Zhijun Wang received the MS degree in electrical engineering from Pennsylvania State University, University Park, in 2001. He is working toward the PhD degree in the Computer Science and Engineering Department at the University of Texas at Arlington. His current research interests include data management in mobile networks and peer-to-peer networks, mobile computing, and networking processors.

Hao Che received the BS degree from Nanjing University, Nanjing, China, in 1984, the MS degree in physics from the University of Texas at Arlington, in 1994, and the PhD degree in electrical engineering from the University of Texas at Austin in 1998. He was an assistant professor of electrical engineering at Pennsylvania State University, University Park, from 1998 to 2000 and a system architect with Santera Systems, Inc., Plano, Texas, from 2000 to 2002. Since September 2002, he has been an assistant professor of computer science and engineering at the University of Texas at Arlington. His current research interests include network architecture and design, network resource management, multiservice switching architecture, and network processor design.



Mohan Kumar received the PhD (1992) and MTech (1985) degrees from the Indian Institute of Science and the BE degree (1982) from Bangalore University in India. He is an associate professor of computer science and engineering at the University of Texas at Arlington. His current research interests are in pervasive computing, wireless networks and mobility, active networks, mobile agents, and distributed computing. Recently, he has developed or codeveloped algorithms for active-network-based routing and multicasting in wireless networks and caching prefetching in mobile distributed computing. He has published more than 95 articles in refereed journals and conference proceedings and supervised master's and doctoral theses in the areas of pervasive computing, caching/prefetching, active networks, wireless networks and mobility, and scheduling in distributed systems. He is on the editorial board of *The Computer Journal* and he has guest edited special issues of several leading international journals, including *MONET* and *WINET* issues and the *IEEE Transactions on Computers*. He is a cofounder of the IEEE International Conference on Pervasive Computing and Communications (PerCom) and served as the program chair for PerCom 2003 and is the general chair for PerCom 2005. He has also served on the technical program committees of numerous international conferences/workshops. He is a senior member of the IEEE. Prior to joining The University of Texas at Arlington in 2001, he held faculty positions at the Curtin University of Technology, Perth, Australia (1992-2000), the Indian Institute of Science (1986-1992), and Bangalore University (1985-1986).



Sajal K. Das received the BS degree in 1983 from Calcutta University, the MS degree in 1984 from the Indian Institute of Science, Bangalore, and the PhD degree in 1988 from the University of Central Florida, Orlando, all in computer science. He is currently a professor of computer science and engineering and also the founding director of the Center for Research in Wireless Mobility and Networking (CReWMaN) at the University of Texas at Arlington (UTA). Prior to 1999, he was a professor of computer science at the University of North Texas (UNT), Denton, where he founded the Center for Research in Wireless Computing (CReW) in 1997 and also served as the director of the Center for Research in Parallel and Distributed Computing (CRPDC) during 1995-1997. He was a recipient of the UNT Student Association's Honor Professor Award in 1991 and 1997 for best teaching and scholarly research, UNT's Developing Scholars Award in 1996 for outstanding research, UTA's Outstanding Faculty Research Award in Computer Science in 2001 and 2003, and the UTA College of Engineering Research Excellence Award in 2003. An internationally known computer scientist, he has visited numerous universities, research organizations, government, and industry labs worldwide for collaborative research and invited seminar talks. He is also frequently invited as a keynote speaker at international conferences and symposia. His' current research interests include resource and mobility management in wireless networks, mobile and pervasive computing, wireless multimedia and QoS provisioning, sensor networks, mobile internet architectures and protocols, parallel processing, grid computing, performance modeling, and simulation. He has published more than 250 research papers in these areas, directed numerous industry and government funded projects, and holds four US patents in wireless mobile networks. He serves on the editorial boards of the *IEEE Transactions on Mobile Computing*, *ACM/Kluwer Wireless Networks*, *Parallel Processing Letters*, and the *Journal of Parallel Algorithms and Applications*. He is vice chair of the IEEE TCPP and TCCC Executive Committees and on the advisory boards of several cutting-edge companies.