Kỷ yếu Hội nghị Quốc gia lần thứ VIII về Nghiên cứu cơ bản và ứng dụng Công Nghệ thông tin (FAIR); Hà Nội, ngày 9-10/7/2015
DOI: 10.15625/vap.2015.000206

TĂNG TỐC ĐỘ ĐỊNH TUYẾN GÓI TIN DỰA TRÊN CÂY ĐA TIỀN TỐ BẰNG PHƯƠNG PHÁP SỬ DỤNG BỘ NHỚ ĐỆM

Nguyễn Mạnh Hùng[1], Phạm Huy Đông[2]

[1]Phòng Sau đại học, Học viện Kỹ thuật Quân sự
[2] Trung tâm TH&ĐL, Đài Truyền hình Việt Nam

Manhhungk12@mta.edu.vn, Dongph@gmail.com

TÓM TẮT - *Trong các hệ thống mạng hiện nay, việc nâng cao tốc độ định tuyến cho các router nhằm nâng cao tốc độ mạng được nghiên cứu và phát triển theo hai hướng chính là: nâng cao chất lượng phần cứng và cải tiến các thuật toán dựa trên phần mềm. Rất nhiều thuật toán dựa vào các cấu trúc dữ liệu Multi-bit Trie, LC-Trie, Prefix Tree, Multiprefix Tree,... đã được các nhà khoa học nghiên cứu, áp dụng vào việc xây dựng bảng định tuyến. Trong bài báo này chúng tôi phân tích và đánh giá hiệu quả định tuyến của cấu trúc dữ liệu cây đa tiền tố và đề xuất kỹ thuật nâng cao hiệu quả định tuyến dựa trên việc sử dụng bộ nhớ đệm. Kỹ thuật đề xuất được đánh giá, so sánh với các kỹ thuật định tuyến dựa trên cây đa tiền tố.*

Từ khóa - *nâng cao tốc độ định tuyến, xây dựng bảng định tuyến động, định tuyến gói tin.*

I. GIỚI THIỆU

Ngày nay, sự phát triển nhanh chóng của Internet phát sinh một vấn đề là làm sao đảm bảo được hiệu suất về thời gian tới đích của các gói tin trong hệ thống mạng, tránh tắc nghẽn? Thực tế đã chứng minh, với một số lượng gói tin đi vào *bộ định tuyến (router)* vô cùng lớn, thì các giải pháp nâng cao tốc độ, hiệu quả định tuyến chính là chìa khóa để giải quyết những khó khăn trên. Để đáp ứng được các đòi hỏi ngày càng cao về chất lượng mạng và nâng cao hiệu quả định tuyến, các nhà phát triển không ngừng nâng cao chất lượng phần cứng của thiết bị mạng, đã cho ra đời các thiết bị mạng có công suất cao, tăng tốc độ chip xử lý, cải thiện và mở rộng băng thông, cải tiến công nghệ cho các thiết bị....

Trong điều kiện hướng nghiên cứu phát triển các công nghệ phần cứng đang dần tiến tới các giới hạn, thì hướng nghiên cứu về các cấu trúc dữ liệu mới và thuật toán xử lý thông tin định tuyến vẫn đang đem lại nhiều kết quả tích cực. Ngày nay, các giải thuật phân loại gói tin hầu hết dựa trên nền tảng phần mềm do ưu thế về tính linh hoạt, mềm dẻo, dễ cài đặt, triển khai cũng như tính kinh tế so với các giải pháp phần cứng. Trong đó, các nghiên cứu tập trung đi sâu vào việc nghiên cứu các cấu trúc dữ liệu (CTDL) sử dụng trong xây dựng bảng định tuyến động của router nhằm mục đích tối ưu hiệu suất về bộ nhớ cũng như về thời gian trong xây dựng, tìm kiếm và cập nhật thông tin cho bảng định tuyến, nghiên cứu đề xuất các CTDL mới để làm bảng định tuyến động (BĐTĐ), các nhà khoa học đã đề xuất các cấu trúc như: *Multi-bit Trie [4, 5, 6, 7], LC-Trie[8, 9], Prefix Tree[1]*... Trong đó, cấu trúc dữ liệu *Cây đa tiền tố Multiprefix Trie (MPT)*, được đề xuất năm 2011 bởi Giáo sư Sun-Yuan Hsieh là một CTDL quan trọng, có nhiều ưu điểm có thể dùng để xây dựng BĐTĐ cho router.

Trong CTDL này, mỗi nút có thể lưu giữ nhiều hơn một tiền tố, qua đó làm giảm số lần truy cập bộ nhớ cần thiết cho các thao tác bảng định tuyến. Nội dung tiếp theo của bài báo gồm: phân tích đặc điểm cấu trúc, các thao tác trên CTDL *MPT* và tính hiệu quả của nó và từ đó chúng tôi đề xuất một số kỹ thuật cải tiến cây MPT.

A. Cây đa tiền tố [2]

Cấu trúc dữ liệu *k-stride* Multiprefix Trie (viết tắt là *k*-MPT, gọi là cây đa tiền tố có bước nhảy *k*), với *k* là số nguyên dương, là một cấu trúc dữ liệu dạng cây, chứa hai loại nút: nút chính: *primary node* (ký hiệu là *p-node*) và 1 nút phụ: *secondary node* (ký hiệu là *s-node*), với các tính chất sau:

P1. Mỗi nút chính p-node *v* chứa các trường sau:

a. $0 \le t \le m$, với t là số tiền tố chứa trong nút v, với $m=O(k)$.

b. t tiền tố, ký hiệu lần lượt là $p_1(v), p_2(v), \ldots p_t(v)$, được lưu trữ theo 1 thứ tự không tăng của độ dài $len(p_1(v)) \ge len(p_2(v)) \ge \ldots len(p_t(v))$.

c. $port(p_t(v))$, là cổng ra (ouput) của $p_i(v)$.

d. $s\_pointer(v)$, là 1 con trỏ trỏ đến 1 cây tiền tố PT chứa các nút phụ s-node, trong đó các nút s-node này chứa các tiền tố có chiều dài $\ge k \,.\, level(v)$, nhưng $\le k \,.\, (level(v) + 1)$. Để thuận tiện, cây biểu diễn bởi con trỏ $s\_pointer(v)$ được gọi là PT của v.

e. Nội dung của $p\text{-}node(v)$ có thể được đại diện đơn giản bởi $(t, p_1(v), p_2(v), \ldots p_t(v), s\_pointer(v))$.

P2. Bước nhảy k là một số bit sử dụng để phân nhánh trong 1 p-node. Một p-node có số bước là k sẽ có 2^k nút con. Để dễ hình dung, ta gọi $child_0(v), child_1(v), \ldots child_2^{k-1}(v)$ là ký hiệu để biểu diễn cho 2^k con tương ứng với 2^k giá trị có thể có từ chuỗi nhị phân có độ dài k bít:

$\underbrace{00\ldots00}_{k}, \underbrace{00\ldots01}_{k}, \underbrace{00\ldots10}_{k}, \underbrace{00\ldots11}_{k}, \ldots\ldots\ldots$ đến $\underbrace{11\ldots11}_{k}$.

Ví dụ, nếu k=2, sẽ có 4 con là $child_0(v)$, $child_1(v)$, $child_2(v)$ và $child_3(v)$, tương ứng với 4 nhánh có nhãn 00, 01, 10 và 11.

P3. Một p-node có m tiền tố gọi là nút đầy, ngược lại là nút không đầy.

Một p-node được gọi là *nút trong* nếu nó là nút đầy và có nút con và một p-node gọi là *nút ngoài* nếu nó không có nút con nào và nút ngoài có thể là nút không đầy.

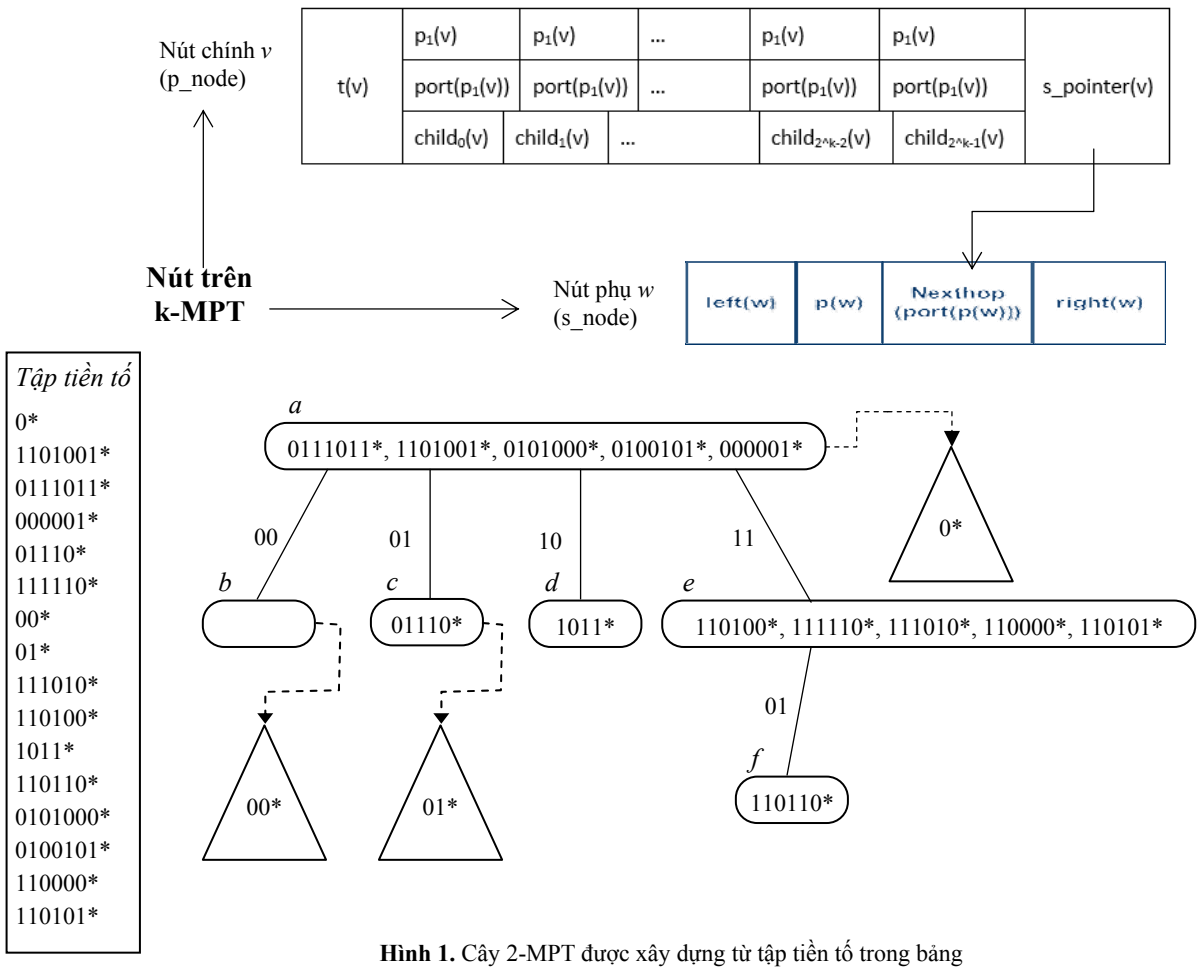
P4. Gọi u và v là hai p-node liên tiếp nhau trên một đường đi trong cây đa tiền tố T. Nếu có hai tiền tố $p_i(u)$ và $p_j(v)$ mà trong đó $p_j(v)$ là tiền tố con của $p_i(u)$, thì $level(u) \le level(v)$.

P5. Mỗi s-node w có các trường sau:

a. $p(w)$, là tiền tố chứa trong w.
b. $port(p(w))$, là cổng output của tiền tố chứa trong w.
c. $left(w)$, là một con trỏ, trỏ đến s-node bên trái của w nếu có, nếu không sẽ là *null*.
d. $right(w)$, là một con trỏ, trỏ đến s-node bên phải của w nếu có, nếu không sẽ là *null*.

Một p-node được gọi là rỗng (empty) nếu nó không chưa tiền tố nào.

Cấu trúc của 1 nút trên cây k-MPT được biểu diễn một cách cơ bản như sau:



Hình 1. Cây 2-MPT được xây dựng từ tập tiền tố trong bảng

B. Các thao tác trên cấu trúc cây k-MPT [2]

1. Thuật toán chèn 1 tiền tố vào cây: MPT_INSERT (p, v, level)

Input: tiền tố p, nút v, bậc $level$
Output: nút được chèn vào cây

```
1: if v is null then
2:     v := ALLOCATE_P-NODE()
3: if IN_PT(len(p), level) then          // chèn p vào PT của v
```

```
4:      u := ALLOCATE_S-NODE()      // cấp phát một s-node mới
5:      PT_INSERT(p, u, s_pointer(v))   // gọi thủ tục chèn của cây tiền tố
6: else if Is_Full(v) then
7:      if len(p_m(v)) < len(p) then      // len tiền tố cuối của v < len của p
8:          thay thế p_m(v) bằng p
9:          sắp xếp các tiền tố trong v theo thứ tự không tăng của độ dài.
10:         r := GET(p_m(v), k . level, k . (level+1) - 1)
11:         v := child_r(v)
12:         MPT_INSERT(p_m(v), v, level + 1)
13: else
14:         r := GET(p, k . level, k . (level+1) – 1)
15:         v := child_r(v)
16:         MPT_INSERT(p, v, level + 1)
17: else
18: chèn p vào v
19: t(v) := t(v) + 1            // tăng số lượng tiền tố trong v
20: return
```

Thuật toán trên sử dụng một số hàm phụ trợ:

Hàm kiểm tra 1 tiền tố có thuộc PT của 1 nút không?

```
Hàm IN_PT(l, level)
1: if l < k . (level + 1) then
2:     return TRUE
3: else
4:     return FALSE
```

Hàm kiểm tra 1 nút có đầy không?

```
Hàm IS_FULL(v)
1: if v is full then
2:     return TRUE
3: else
4:     return FALSE
```

Độ phức tạp tính toán của thuật toán: $\boldsymbol{O}(W)$

2. Thuật toán Tìm kiếm tiền tố trên cây: MPT_LOOKUP(DA, v, level)

Input: địa chỉ đích cần tìm DA

Output: trả về cổng đích *nexthop* tương ứng của *LMP* nếu tìm thấy

```
// next_hop dùng để lưu lại cổng output của tiền tố khớp tốt hơn hiện tại
// default_route sử dụng để lưu lại cổng output mặc định
1:  level := 0
2:  next_hop := default_route
3:  while v ≠ null do
4:      if có tiền tố trong v khớp với DA then
5:          tìm tiền tố dài nhất p_i(v) khớp với DA
6:          return port(p_i(v))
7:      else
8:          next_hop := PT_LOOKUP(DA, s_pointer(v))
9:      r := GET(DA, k . level, k . (level + 1) - 1)
10:     v := child_r(v)
11:     level := level + 1
12: return next_hop
```

Độ phức tạp tính toán của thuật toán: $\boldsymbol{O}(W^2/k)$

3. Thuật toán xóa một nút trên cây: MPT_DELETE(*p, v, level*)

```
// Thuật toán này sử dụng 2 hàm phụ, FREE_SNODE và FREE_P-NODE, để giải phóng bộ nhớ cho s-node
và p-node, độ phức tạp thời gian O(1)
1:  if v is null then
2:      output "p is not found"
3:  if IN_PT(len(p), level) then
4:      PT_DELETE(p, s_pointer(v))
5:      FREE_S-NODE
6:  else if p is in v then
7:      Xoá p trong v
8:      if v là p-node ngoài then
9:          t(v) := t(v) – 1                    // giảm số lượng tiền tố trong v
10:         if t(v) = 0 và s_pointer(v) = null then
```

```
11:                         FREE_P-NODE(v)
12:       else
13:             tìm tiền tố y trong child_r(v) sao cho len(y) = max{len(p)| p ∈ child_i(v) for 0≤ i≤2k-1}
14:             chèn y vào vị trí tiền tố cuối trong v
15:             v := child_r(v)
16:             MPT_DELETE(y, v, level + 1)
17: else
18:       r := GET(p, k . level, k . (level + 1) - 1)
19:       v := child_r(v)
20:       MPT_DELETE(p, v, level + 1)
21: return
```

Độ phức tạp tính toán của thuật toán: $\boldsymbol{O}(2^k W/k)$

W là độ dài (tính bằng bit) của một địa chỉ đích

B. Hiệu quả định tuyến của cây k-MPT [2]

Dựa vào đặc điểm về cấu trúc và qua nghiên cứu các thuật toán mô tả hoạt động của cây *k-MPT* , chúng tôi có một số nhật xét về hiệu quả định tuyến của CTDL này như sau:

Thứ nhất: Mỗi nút (nút chính + nút phụ) của cây *k-MPT* lưu nhiều tiền tố *(tương ứng là các luật)*, nên so với các CTDL cây khác (như cây tiền tố), với cùng một tập tiền tố, thì chiều cao của *k-MPT* thấp hơn nhiều, do đó tốc độ tra cứu trên cây *k-MPT* sẽ nhanh hơn.

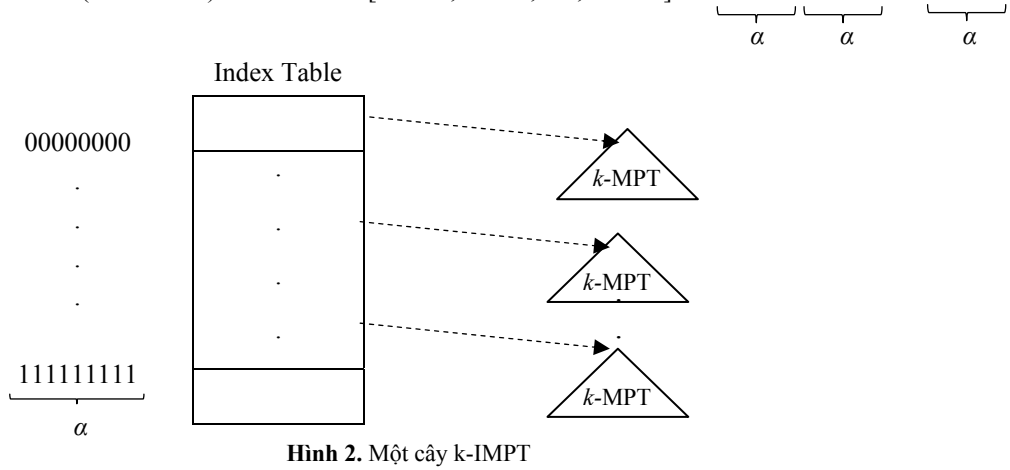
Thứ hai: Trong quá trình tra cứu địa chỉ, *LMP* có thể được tìm thấy tại các nút không phải là *nút lá*. Từ đặc điểm P4, ta khẳng định: nếu trường $p_t(v)$ khớp với *địa chỉ đích*, thì tiền tố lưu trong $p_t(v)$ chính là *LMP*. Khi đã tìm được *LMP* thì thuật toán có thể kết thúc ngay. Mặt khác, do các *tiền tố* sắp xếp theo thứ tự không tăng của độ dài mà quá trình tra cứu, so khớp chỉ diễn ra giữa *địa chỉ đích DA* với những tiền tố nhất định trong nút, chứ không phải với tất cả các tiền tố. Do đó chi phí thời gian tìm kiếm *LMP* giảm khá nhiều.

Thứ ba: Việc lưu trữ nhiều tiền tố trong một nút giúp *k-MPT* giảm chi phí lưu trữ thông tin. Mặt khác, nhìn chung các tiền tố được lưu trữ trong các nút có mức càng cao (càng xa *nút gốc*) thì có độ dài càng bé, do đó chi phí lưu trữ của các nút ở mức cao ít hơn chi phí lưu trữ của các nút ở mức thấp. Do đó, nếu việc cấp phát bộ nhớ lưu trữ cho các nút được lập trình linh hoạt hơn, ta có thể tiết kiệm được dung lượng lưu trữ.

Thứ tư: Việc lưu giữ nhiều tiền tố trong một nút của cây và việc phân loại các tiền tố theo thứ tự không tăng của độ dài làm giảm số nút trên cây, giảm số lần truy cập bộ nhớ, và giảm chi phí tìm kiếm vị trí để chèn và xóa các tiền tố. Đặc biệt trong thao tác xóa tiền tố, việc tìm kiếm tiền tố thay thế (để đảm bảo tính chất của cây) có chi phí thấp. Việc đảm bảo được các tính chất của cây *k-MPT* sau các thao tác cập nhật có ý nghĩa quan trọng, đảm bảo hiệu quả của hoạt động định tuyến của Router.

C. Kỹ thuật phân hoạch cây k-MPT [2]

Khi *bước nhảy k* tăng thì chiều cao của cây *k-MPT* giảm, tuy nhiên số nhánh con và sự phức tạp của các quá trình xử lý tăng lên, làm hiệu suất định tuyến trung bình giảm. Với mục đích làm giảm chiều cao của cây *k-MPT* mà không tăng *bước nhảy k*, một kỹ thuật được đề xuất là phân hoạch cây *k-MPT*. Ý tưởng nhằm thực hiện phân hoạch cây *k-MPT* thành một số các *k-MPTs* có chiều cao thấp hơn, tạo nên một cấu trúc dữ liệu mới gọi là Cây *đa tiền tố chỉ mục có bước nhảy k* (*k-Stride Index Multiprefix Tree* - gọi tắt là *k*-IMPT), dựa trên cây *k*-MPT đã trình bày. Cây *k*-IMPT phân hoạch một cây *k*-MPT thành nhiều cây *k*-MPTs có chiều cao thấp hơn, danh sách các gốc được lưu giữ trong mảng một chiều (index table) như sau: *tab*[00…0, 00…1, … , 11…1]



Hình 2. Một cây k-IMPT

Với một chiều dài α cố định, bảng chỉ mục có không quá 2^{α} phần tử, mỗi phần tử $tab[b_0b_1...b_{\alpha-1}]$ trỏ tới gốc 1 cây *k*-MPT con mà có chứa các tiền tố với tiền tố con chung dạng $b_0b_1...b_{\alpha-1}$ có độ dài α bít (xem hình 2).

Để thực hiện các thao tác bảng định tuyến (tìm kiếm, chèn, xóa) trong một *k*-MPT, trước hết chúng ta đối chiếu với mảng chỉ số và thực hiện các thao tác này trong cây *k*-MPT tương ứng.

Ví dụ: nếu chèn 1 tiền tố *p* vào một *k*-MPT, đầu tiên chúng ta lấy α bít của p để xác định giá trị chỉ mục của gốc trong mảng.

Nếu $len(p) \geq \alpha$ chắc chắn *p* được chèn vào cây k-MPT có gốc là giá trị chỉ mục.

Ngược lại nếu $len(p) < \alpha$ thì ta phải mở rộng tiền tố *p* thành một tập tiền tố có độ dài α. Ví dụ, với p = 101000* và α = 8 (tức là 8 bít đầu của tiền tố biểu diễn giá trị của chỉ mục). Vì tiền tố 101000* mở rộng thành 10100**00**, 10100**01**, 10100**10**, và 10100**11**, và do đó tiền tố 101000* được chèn vào 4 cây *k*-MPT được biểu diễn bởi *tab*[10100000], *tab*[10100001], *tab*[10100010] và *tab*[10100011].

Quá trình thực hiện các thao tác của bảng định tuyến *(chèn, tra cứu địa chỉ và xóa)* trên cây *k-MPT* đều được bắt đầu bằng việc xác định gốc của cây con *k-MPT*, bằng cách xác định giá trị thập phân của α bit đầu tiên của tiền tố sẽ trả lại giá trị tương ứng của gốc trong mảng chỉ mục, sau đó thực hiện các hoạt động chèn, tra cứu hoặc xóa tiền tố trên cây *k-MPT* con đó.

Việc mở rộng tiền tố và chèn vào các cây tương ứng như trên có thể dẫn tới sự bùng nổ số lượng tiền tố được lưu giữ trên cây. Tức là số tiền tố được lưu giữ lớn hơn nhiều so với số lượng tiền tố đầu vào, và việc lưu giữ các tiền tố trùng lặp gây ra tốn kém bộ nhớ và xử lý phức tạp. Chúng ta phải chọn một giá trị α phù hợp để giảm bộ nhớ cần thiết, α không nên quá lớn (vì với α cố định, sẽ có 2^{α} cây k-MPT được tạo ra), nhưng nếu α quá nhỏ thì hiệu quả làm giảm chiều cao của cây cũng không cao. Trên thực tế, sau quá trình thử nghiệm, chọn giá trị α bằng độ dài tiền tố ngắn nhất của bảng định tuyến được đánh giá là một sự lựa chọn phù hợp.

III. ĐỀ XUẤT KỸ THUẬT TĂNG TỐC ĐỘ ĐỊNH TUYẾN DỰA TRÊN CÂY ĐA TIỀN TỐ SỬ DỤNG BỘ NHỚ ĐỆM

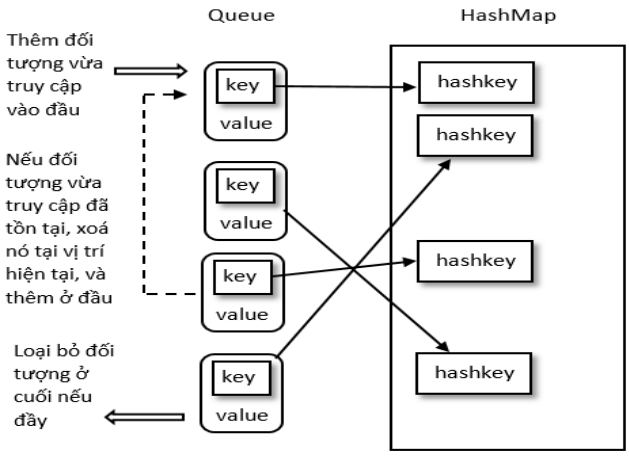
A. Kỹ thuật tăng tốc cho k-MPT sử dụng bộ nhớ đệm (Cache)

Một lượng dữ liệu được truyền đi trong hệ thống mạng có thể có rất nhiều gói tin có trường *địa chỉ đích* giống nhau. Mặc dù việc nhận các *địa chỉ đích* gói tin đến của router là ngẫu nhiên, nhưng một *địa chỉ đích* có thể bị tra cứu lặp lại nhiều lần trong một khoảng thời gian lân cận. Để hạn chế sự tra cứu lặp lại đó, chúng tôi đề xuất kỹ thuật sử dụng *bộ nhớ đệm cache*, để lưu kết quả tra cứu của một số *địa chỉ đích* gói tin vừa được tra cứu.

Việc sử dụng bộ nhớ cache để tăng tốc độ định tuyến được chia thành 2 hướng nghiên cứu chính: 1) áp dụng cache cho tập luật trong bảng định tuyến (tập luật nào được sử dụng nhiều sẽ được lưu vào cache) như sử dụng *Rule Caching[10], Popular Rule Caching*..., và 2) áp dụng cache cho việc định tuyến gói tin đến địa chỉ đích (địa chỉ đích nào được định tuyến đến nhiều sẽ được lưu vào cache) như *Digest Cache[11], LFU cache*. Trong bài báo, chúng tôi sử dụng kết hợp hàng đợi và bảng băm để giúp tăng tốc độ tìm kiếm trong cache khi định tuyến gói tin dựa vào địa chỉ đích, dựa vào ưu thế về thời gian tìm kiếm của bảng băm.

1. Cách xây dựng cache

Cache được thiết kế dùng một bảng băm để lưu các khoá [key] phục vụ tra cứu trong cache và một hàng đợi sắp xếp theo một trật tự nhất định để lưu các giá trị gói tin cần định tuyến (Tiền tố địa chỉ đích và Cổng đích nexthop). Khi cache đầy, sẽ xoá các phần tử cuối hàng đợi (ít dùng nhất) ra khỏi cache và đưa phần tử mới vào đầu hàng đợi. Việc tìm kiếm các key trong bảng băm của cache sẽ nhanh hơn các CTDL khác.



Hình 3. Mô hình sử dụng bộ nhớ cache

Để thao tác với cache, dùng hàm **put** để đưa 1 phần tử vào cache, còn hàm **exists** để kiểm tra 1 phần tử đã có trong cache hay chưa. Mô hình hoạt động của cache được thiết kế như sau:

2. Hoạt động tra cứu khi áp dụng cache

Khi gói tin đi vào Router và yêu cầu tra cứu địa chỉ, trước hết Router sẽ kiểm tra xem *địa chỉ đích* của gói tin cần tra cứu có trong cache không, nếu có thì lấy thông tin *nexthop* tương ứng với *địa chỉ đích* đó đã được lưu trong cache làm kết quả, ngược lại nếu *địa chỉ đích* đó không có trong cache, thì thực hiện tra cứu địa chỉ đó trên cây k-MPT, sau đó đưa thông tin *địa chỉ đích* vừa tra cứu với *nexthop* tương ứng thu được vào cache. Khi cache đầy, cần loại bớt những dữ liệu cũ, ít sử dụng hơn và cập nhật dữ liệu mới hơn cho cache.

Tuy nhiên sẽ nảy sinh vấn đề: lựa chọn dung lượng bao nhiêu cho cache để đạt hiệu quả tối ưu, vì chi phí tra cứu địa chỉ khi sử dụng kỹ thuật này bao gồm chi phí tìm kiếm trong cache, chi phí cập nhật cache.

Vấn đề nữa là khi có luật cần thêm hoặc xoá, thì khi tiến hành thêm hoặc xoá tiền tố trong cây, ta phải tiến hành tìm *kiếm tiền tố này trong cache* để cập nhật lại *nexthop* hoặc xoá. Để đảm bảo rằng dữ liệu trong cache luôn lưu giữ đúng giá trị mới nhất khi bảng định tuyến có sự thay đổi.

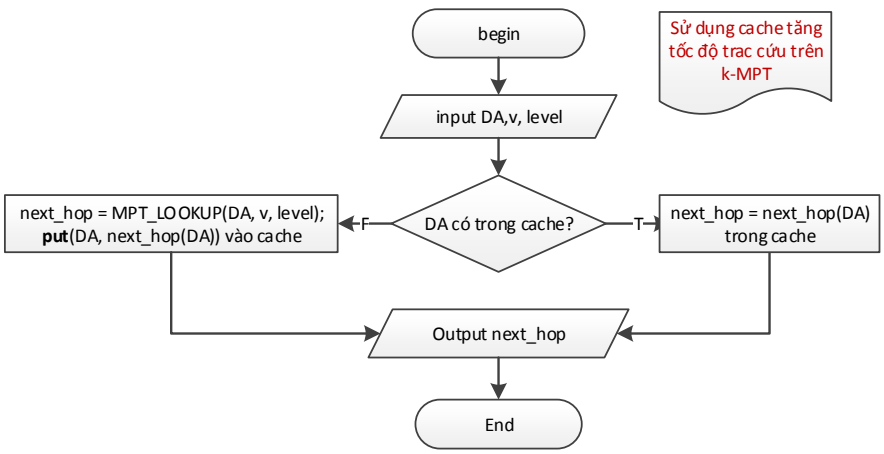
Thực tế kiểm nghiệm cho thấy với lượng gói tin ít, cache ít phát huy tác dụng, nhưng với số lượng gói tin đến lớn, cache tỏ ra khá hiệu quả khi định tuyến vì số lần tra cứu trên cây *k-MPT* phải thực thi ít hơn.

B. k-MPT có sử dụng cache:

Để áp dụng cache trong việc hỗ trợ định tuyến trong cây k-MPT, chúng tôi tiến hành cài đặt bộ nhớ cache theo thiết kế ở trên vào thuật toán tìm kiếm (định tuyến gói tin) MPT_LOOKUP của cây k-MPT [2]. Khi một *địa chỉ đích* được tra cứu, trước tiên chúng ta sẽ tìm kiếm địa chỉ này có tồn tại trong cache hay không (bằng kỹ thuật tìm kiếm hàm băm theo thiết kế ở trên). Nếu địa chỉ này có tồn tại trong cache, cổng đích *(nexthop)* của nó được trả về và hoàn tất quá trình tìm kiếm. Nếu địa chỉ đích không nằm trong cache, sẽ tiến hành tìm kiếm địa chỉ này trên cây theo thuật toán MPT_LOOKUP gốc. Xảy ra 2 trường hợp

- Nếu không tìm thấy địa chỉ đích trong cây, thông báo không tìm thấy và kết thúc tìm kiếm.

- Nếu tìm thấy địa chỉ đích, trả về cổng đích *nexthop* và lưu thông tin địa chỉ đích và cổng đích vừa tìm thấy vào cache. Nếu cache đầy, tiến hành loại bỏ phần tử ít dùng nhất trong cache.



Hình 4. Sơ đồ thuật toán

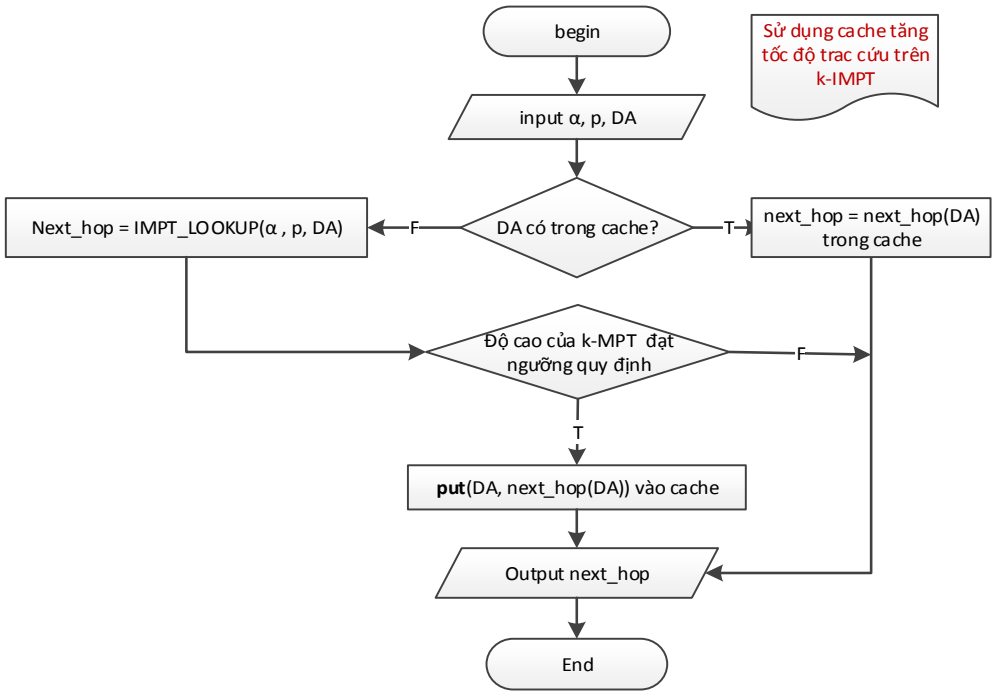
Khi áp dụng cache, mỗi khi bảng định tuyến có sự thay đổi, tuyến đã đưa vào cache có thể không còn đúng nữa, ta phải tiến hành tìm luật này trong cache rồi update lại theo luật mới thêm. Việc tìm kiếm trong cache dựa vào hàm băm không bị ảnh hưởng nhiều về mặt thời gian.

C. k-IMPT có sử dụng cache

Tương tự ý tưởng dùng cache cho *k-MPT*, khi tiến hành sử dụng cache cho *k-IMPT*, chúng ta có thể sử dụng 1 bộ nhớ cache cho tất cả các cây phân hoạch *k-MPTs* thành phần theo cách tương tự cho cây *k-MPT* nguyên thuỷ. Tuy nhiên nếu cây *k-IMPT* có độ cao thấp, thì việc sử dụng cache cho nó sẽ kém hiệu quả, nên chúng ta sẽ sử dụng bộ nhớ cache cho những cây *k-IMPT* có độ cao tương đối lớn. Toàn bộ quá trình tìm kiếm trên các cây *k-IMPT* sẽ sử dụng thuật toán IMPT_LOOKUP [2], các tình huống xảy ra cũng tương tự khi sử dụng cache cho cây *k-MPT*:

- Nếu không tìm thấy địa chỉ đích trong các cây k-IMPT, thông báo không tìm thấy và kết thúc tìm kiếm.

- Nếu tìm thấy địa chỉ đích, trả về cổng đích *nexthop* và kiểm tra độ cao của cây k-IMPT mà địa chỉ được tìm thấy, nếu cây này có độ cao đạt ngưỡng theo quy định thì lưu thông tin địa chỉ đích và cổng đích vừa tìm thấy vào cache. Nếu cache đầy, tiến hành loại bỏ phần tử ít dùng nhất trong cache.



Hình 5. Sơ đồ chi tiết thuật toán

IV. CÀI ĐẶT THỬ NGHIỆM VÀ ĐÁNH GIÁ

Để đảm bảo sát với ứng dụng thực tế, chương trình sử dụng các bộ dữ liệu được tạo bằng bộ công cụ ClassBench do David E. Taylor, Jonathan S. Turner thuộc Phòng Nghiên cứu ứng dụng, Khoa Khoa học Máy tính, Đại học Washington, Saint Louis tạo ra [http://www.arl.wustl.edu/classbench]. Bộ dữ liệu bao gồm các tập luật và các tập tham số gói tin được sinh bởi bộ công cụ trên có dữ liệu đầu vào là những bộ dữ liệu thực của các nhà cung cấp dịch vụ Internet. Đây là bộ công cụ được cộng đồng nghiên cứu sử dụng để đánh giá các thuật toán và các thiết bị phân loại gói tin.

A. Đánh giá hiệu quả của thuật toán xây dựng cây có sử dụng cache

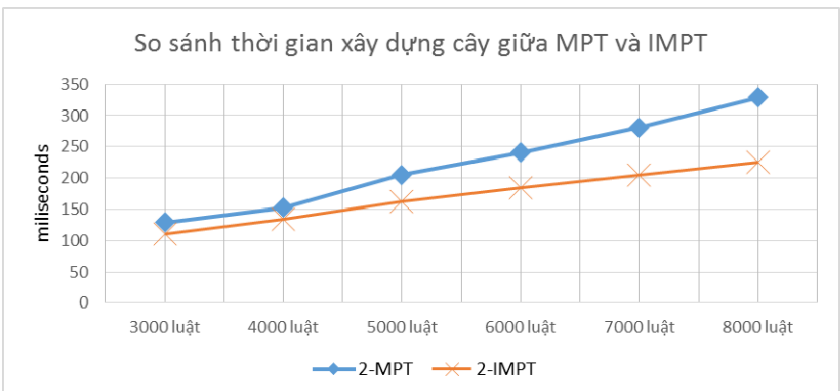
Các thí nghiệm trên bộ cơ sở dữ liệu tiền tố đề cập ở trên, thực hiện với cây *k-MPT* và *k-IMPT* (biến k=2), thực hiện trên cùng một máy tính BXL Core I5 4430 có tốc độ 3.0 GHz và bộ nhớ RAM là 8GB; đơn vị tính thời gian là mili giây.

Thuật toán cải tiến k-IMPT khi xây dựng cây sẽ phân hoạch thành nhiều cây con, do đó chiều cao của các cây được giảm đi khá nhiều, thời gian xây dựng cây cũng diễn ra nhanh hơn so với cây k-MPT. Chúng tôi đã tiến hành đo thời gian xây dựng cây với số luật biến thiên từ 3.000 đến 8.000 luật, kết quả thể hiện ở bảng 1:

Bảng 1. Bảng đo thời gian xây dựng cây của 2-MPT và 2-IMPT*(đơn vị miliseconds)*

Số lượng luật:	3000 luật	4000 luật	5000 luật	6000 luật	7000 luật	8000 luật
2-MPT	129.3	152.8	205.5	240.88	280.83	329
2-IMPT	110.8	133.63	161.88	185.14	205.5	225.42

Từ kết quả trên, ta có biểu đồ so sánh thời gian xây dựng cây của 2-MPT và 2-IMPT như sau:



Hình 6. Thời gian xây dựng cây của 2-MPT so với 2-IMPT

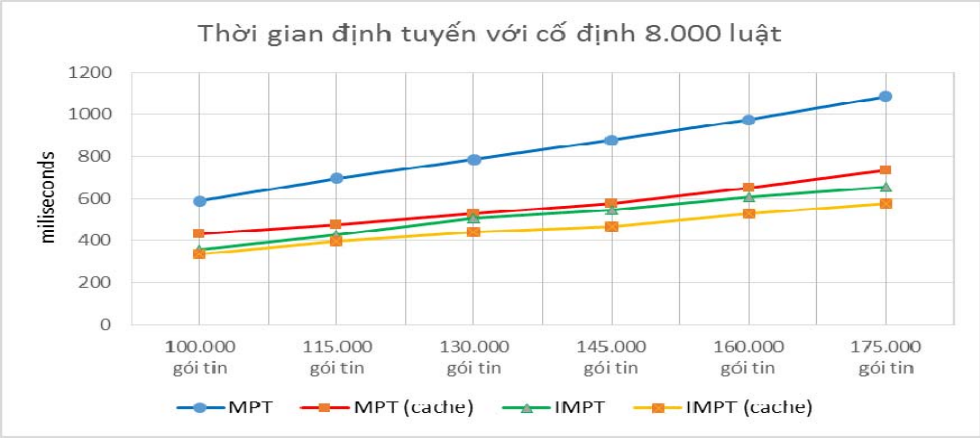
B. Đánh giá hiệu quả của việc định tuyến gói tin dựa trên cây k-MPT và k-IMPT cây có sử dụng cache

Để chứng minh hiệu quả thuật toán tra cứu của 2-IMPT so với 2-MPT, cùng với thuật toán 2-MPT và 2-IMPT có sử dụng bộ nhớ cache, chúng tôi cố định 8000 luật và cho biến thiên số lượng gói tin đến. Chi phí thời gian tra cứu trung bình đo được thể hiện trong bảng 2:

Bảng 2. Bảng đo thời gian tra cứu của 2-MPT và 2-IMPT với 8000 luật *(đơn vị miliseconds)*

Số gói tin: ->	100.000 gói tin	115.000 gói tin	130.000 gói tin	145.000 gói tin	160.000 gói tin	175.000 gói tin
MPT	589.4	695.6	786.4	877.25	975	1085.8
MPT (cache)	430.6	476	528	577.25	652.5	733.6
IMPT	358.2	426.8	505.43	546.33	608	657.83
IMPT (cache)	338	395.6	439.8	467	529.6	575.17

Từ bảng trên, ta có biểu đồ so sánh, thể hiện hiệu quả của thuật toán 2-MPT và 2-IMPT cùng với khi chúng sử dụng thêm bộ nhớ cache như sau:



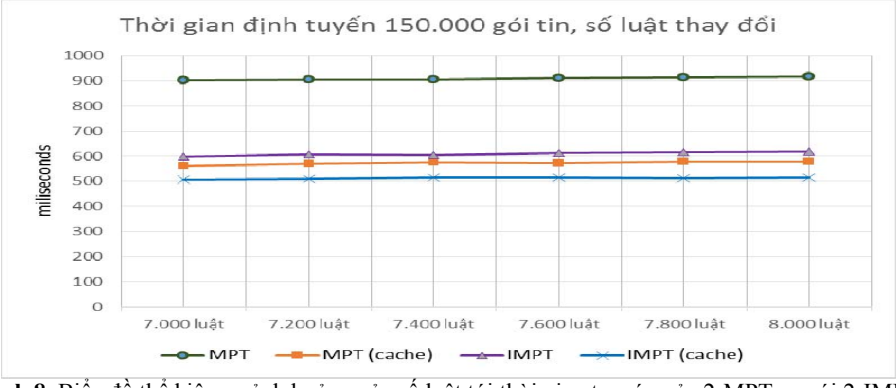
Hình 7. Hiệu quả tra cứu của 2-MPT so với 2-IMPT

Tiếp theo, chúng tôi đã thực hiện đo thời gian tra cứu của 2-IMPT so với 2-MPT với số lượng gói tin đến cố định là 150.000 gói tin, số lượng luật biến thiên từ 7000 đến 8000 luật, chi phí thời gian tra cứu trung bình thu được như bảng 3 dưới đây:

Bảng 3. Bảng đo thời gian tra cứu 150.000 gói tin đến của 2-MPT và 2-IMPT *(đơn vị miliseconds)*

Số luật: ->	7.000 luật	7.200 luật	7.400 luật	7.600 luật	7.800 luật	8.000 luật
MPT	903	906.25	904.83	912.5	915	916.25
MPT (cache)	560	570	576.5	573.25	578.75	579.5
IMPT	599.2	608	604.5	611.75	616.6	619
IMPT (cache)	507	508.6	514	514.75	511.5	514.2

Từ bảng trên, ta có biểu đồ thể hiện sự ảnh hưởng của số lượng luật trong bảng định tuyến tới chi phí tra cứu của 2-MPT và 2-IMPT như sau:



Hình 8. Biểu đồ thể hiện sự ảnh hưởng của số luật tới thời gian tra cứu của 2-MPT so với 2-IMPT

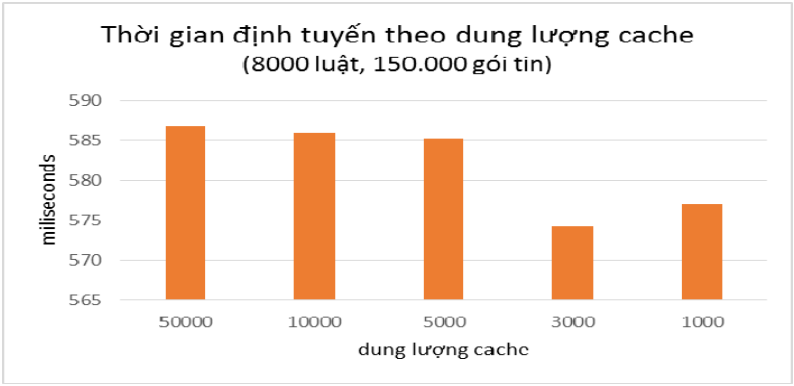
Từ các biểu đồ trên, chúng ta nhận thấy, tốc độ tra cứu của các thuật toán 2-MPT và 2-IMPT được cải thiện đáng kể khi sử dụng bộ nhớ đệm (cache). Tuy nhiên việc lựa chọn cache phải nằm trong một khoảng nào đó ở mức độ

phù hợp để tránh lãng phí bộ nhớ và tối ưu tốc độ cho thuật toán. Chúng tôi cũng tiến hành thử nghiệm với cố định 8000 luật và tra cứu 150.000 gói tin đến của bộ dữ liệu thử nghiệm trong mục 6.1.

Trong quá trình đo kết quả thử nghiệm, để đơn giản chúng tôi sử dụng đơn vị đo của cache được tính theo *block* thông tin, mỗi *block* được thể hiện cho một bộ gồm khoá Key và một bản ghi lưu thông tin của địa chỉ đích và *nexthop* tương ứng. Kết quả đo thời gian tra cứu được thể hiện trong bảng sau:

Bảng 4. Bảng đo thời gian tra cứu 150.000 gói tin đến của 2-MPT theo dung lượng cache *(đơn vị miliseconds)*

Dung lượng cache *(block)*	Thời gian tra cứu 150.000 gói tin của 2-MPT *(đơn vị miliseconds)*
50000	586.8
10000	586
5000	585.3
3000	574.2
1000	577



Hình 9. Biểu đồ thể hiện sự ảnh hưởng của dung lượng cache tới thời gian tra cứu của 2-MPT

V. KẾT LUẬN

Bài báo đã trình bày tổng quan về bài toán xây dựng bảng định tuyến động nhằm tăng tốc độ định tuyến gói tin trong router. Phân tích đặc điểm cấu trúc, các thao tác và tính hiệu quả của CTDL cây *k-MPT* và *k-IMPT*. Trên cơ sở các phân tích về cây *k-MPT* và *k-IMPT*, bài báo đã đề xuất kỹ thuật tăng tốc độ tra cứu cho *k-MPT* và *k-IMPT* sử dụng bộ đệm. Thử nghiệm kỹ thuật đề xuất trên bộ dữ liệu chuẩn[3], kết quả cho thấy việc sử dụng bộ đệm cho quá trình định tuyến có hiệu quả rõ rệt, nhất là với lượng gói tin cần định tuyến lớn.

VI. TÀI LIỆU THAM KHẢO

[1] M. Berger, "IP Lookup with Low Memory Requirement and Fast Update", Proc. IEEE High Performance Switching and Routing, pp. 287-291, June 2003.

[2] Sun-Yuan Hsieh,Senior Member, IEEE, Yi-Ling Huang, and Ying-Chi Yang, "Multiprefix Trie: A New Data Structure for Designing Dynamic Router-Tables", IEEE TRANSACTIONS ON COMPUTERS, 2011.

[3] David E. Taylor, Jonathan S. Turner, *ClassBench: A Packet Classification Benchmark*, IEEE INFOCOM, 2005.

[4] Sartaj Sahni, Kun Suk Kim, Efficient construction of variable-stride multibit tries for IP lookup, Proceedings Symposium on Applications and the Internet, 2002.

[5] Sartaj Sahni, Kun Suk Kim, Efficient Construction Of Multibit Tries For IP Lookup, ACM Transactions on Networking, 2003.

[6] Sartaj Sahni, Kun Suk Kim, Efficient Construction Of Fixed-Stride Multibit Tries For IP Lookup, Proceedings 8th IEEE Workshop on Future Trends of Distributed Computing Systems, 2001.

[7] Yi Jiang, Fengju Shang, Research on Multibit-Trie Tree IP Classification Algorithm, International Conference on Communications, Circuits and Systems Proceedings, 2006.

[8] Lee, Chae-Y. ; Park, Jae-G., IP Lookup Table Design using LC-trie with Memory Constraint, Journal of Korean Institute of Industrial Engineers, Volume 27, Issue 4, 2001.

[9] Jing Fu, Olof Hagsand and Gunnar Karlsson, Improving and Analyzing LC-Trie Performance for IP-Address Lookup, JOURNAL OF NETWORKS, VOL. 2, NO. 3, JUNE 2007.

[10] Nitesh B. Guinde, Roberto Rojas-Cessa and Sotirios G. Ziavras, Packet Classification using Rule Caching, IEEE International Conference on Information, Intelligence, Systems and Applications, July 2013.

[11] Francis Chang, Wu-chang Feng, Wu-chi Feng, Kang Li, EfficientPacket Classification with Digest Caches, in Proc. of the Third Workshop on Network Processors & Applications (NP3), February 2004.

IMPROVING PACKET ROUTING SPEED BASED ON MULTIPREFIX-TRIE BY USING CACHING

Nguyen Manh Hung, Pham Huy Dong

Abstract - *In current networks, improving routing speed for the router to enhance the network speed was researched and developed in two main directions: improved hardware quality and improved algorithms based on software. Many algorithms based on the data structure as Multi-bit trie, LC-Trie, Prefix Tree, Multiprefix Trie ... have been researching by scientists, applied to the construction of the router-tables. In this paper we analyzed and evaluated the effectiveness of the routing of Multiprefix Tree data structure and proposed a technique to improve the efficiency of routing based on the use of caching. Technical proposals are evaluated, compared with routing techniques based on multi-prefix tree.*

Keywords *- improving routing speed, building dynamic routing table, routing packets.*