کارشناسی ارشد شبکه های کامپیوتری

شبکه های پهن باند

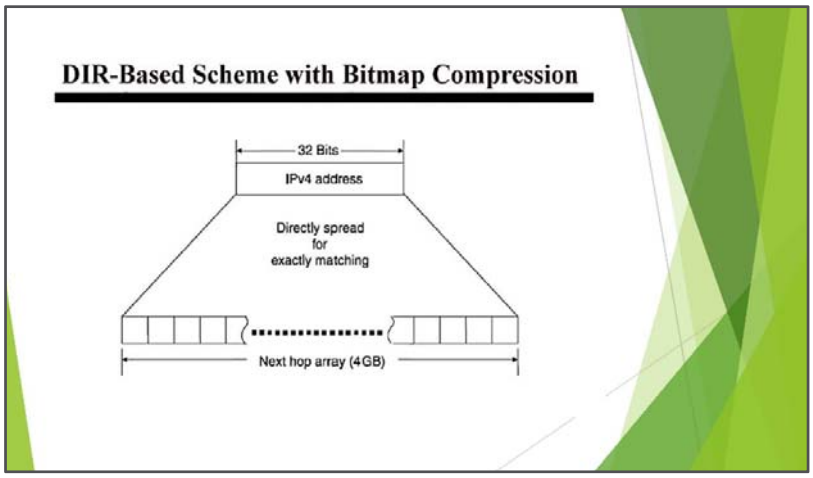
استاد:

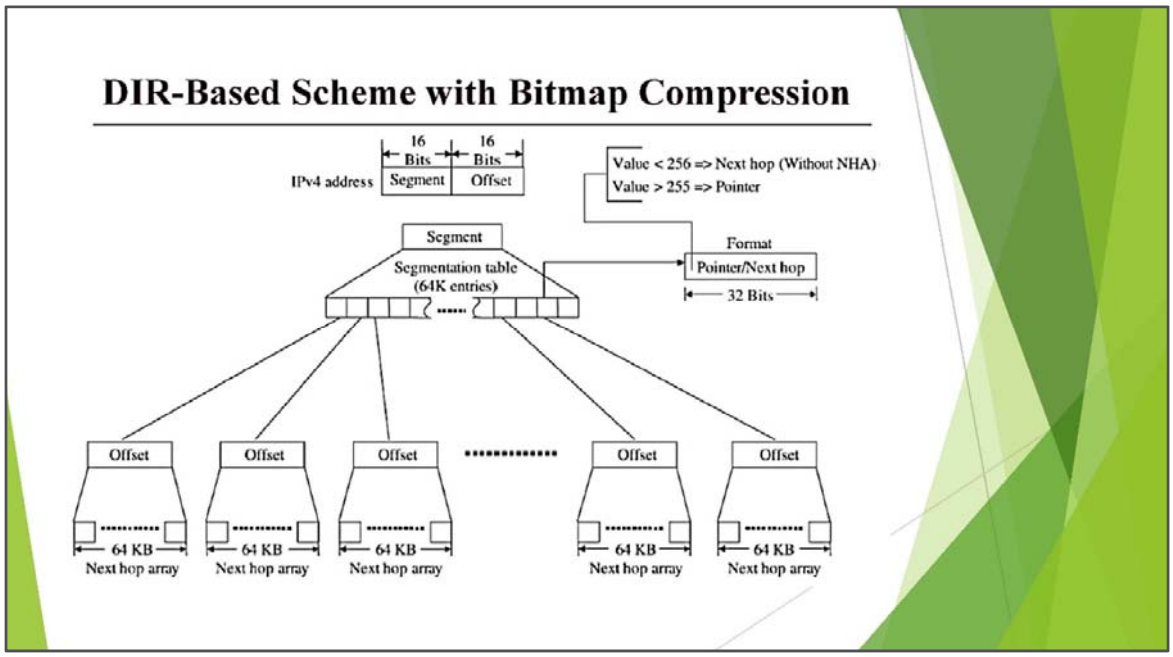
دکتر جبارپور

نویسنده:

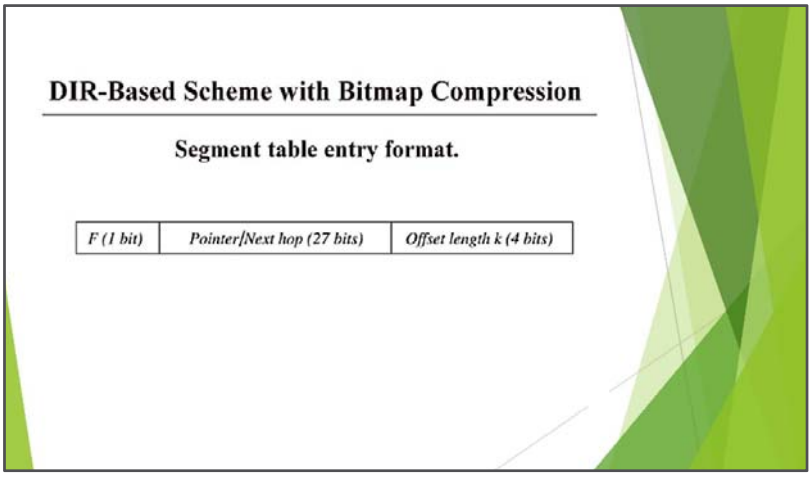
مهندس علیرضا روحانی نژاد

۱۳۹۷

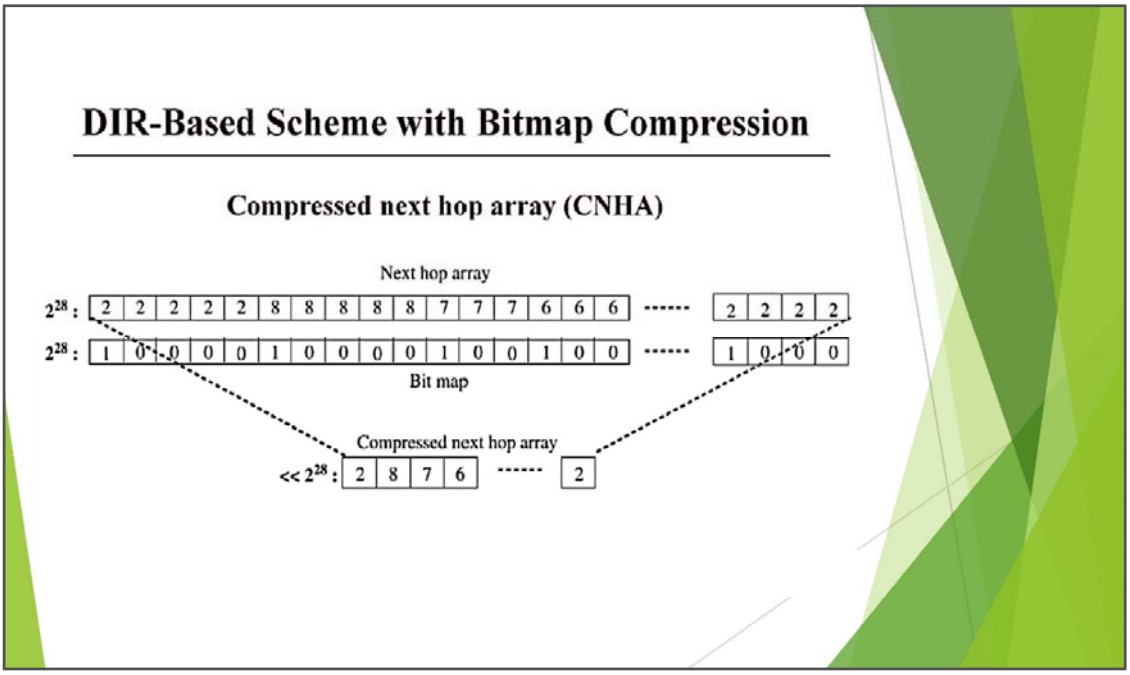




روش ۱۶-۱۶ Direct است. از همان حالت دو سطحی استفاده شده‌است مثل(۸-۲۴) ولی سعی شده اندازه حافظه سطح دوم ثابت نباشد. روش قبلی ۸-۲۴ بود. در این روش سطح ۱۶بار در درخت شکسته می‌شود و همه Prefixها ۱۶بیتی می‌شود. Prefixهای با طول بیشتر از ۱۶بیت در روش قبلی برایشان سطح ۲۴بیتی ایجاد می‌شد و بقیه ۳۲بیتی درنظر گرفته می‌شد. در این روش برای Prefixهای بزرگتر از ۱۶ بایستی ببینیم بزرگترین شاخه آن‌ها دارای چه عمقی است؟ می‌تواند این عمق تا ۳۲بیت هم ادامه داشته‌باشد. در حافظه سوم به‌جای این‌که به‌ازای هر شاخه ۲۱۶ درنظر گرفته‌شود، به اندازه Max عمق درنظر می‌گیریم.اگر عمق k باشد، اندازه 2K می‌شود(خانه حافظه). به این ترتیب در حافظه اول یا Next Hop پیدا می‌شود یا یک Pointer به‌سمت سطح دوم حافظه ایجاد می‌شود. اگر Pointer به سطح دوم داشته باشیم، یک Offset، ۲۴بیتی را مشخص می‌کنیم(چون گفته شد عمق حداکثر برای سطح دوم می‌تواند ۱۶ باشد)؛ سطح اول ۱۶بیت و سطح دوم حداکثر ۱۶بیت.



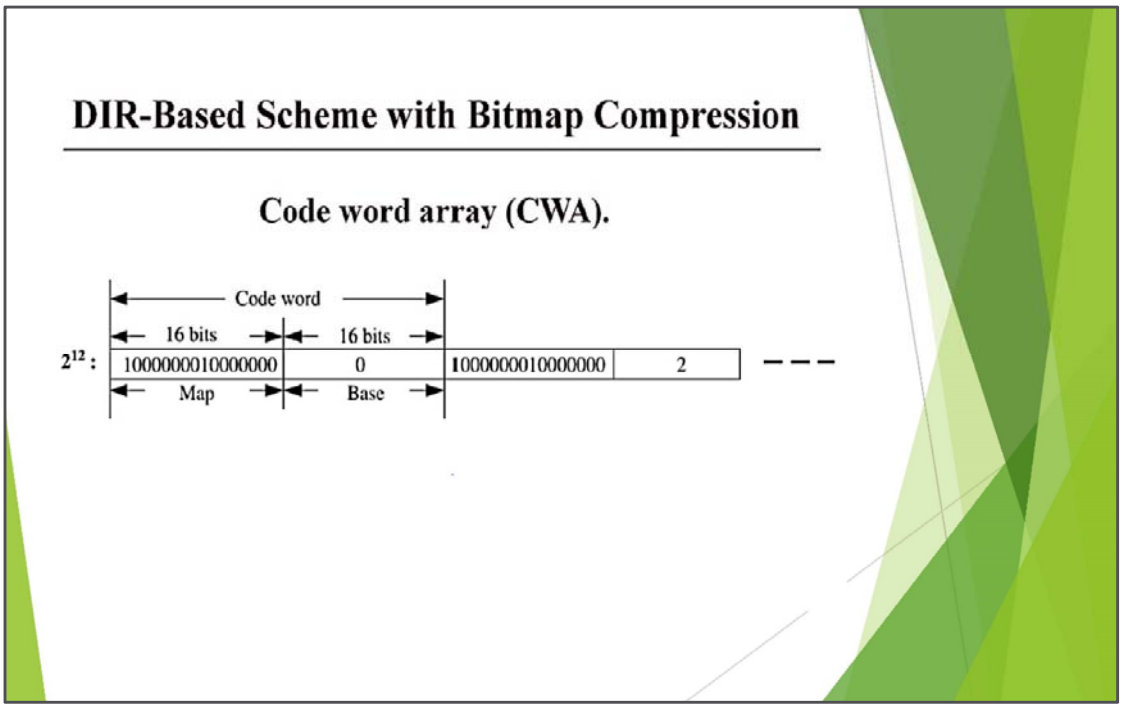
یک بیت به‌عنوان Flag استفاده‌شده اگر صفر باشد فیلد بعدی Pointer یا Next Hopهاست و اگر یک باشد یعنی باید سراغ Offset برویم و از آن‌جا مشخص کنیم کدام خانه حافظه دوم مدنظر ماست.



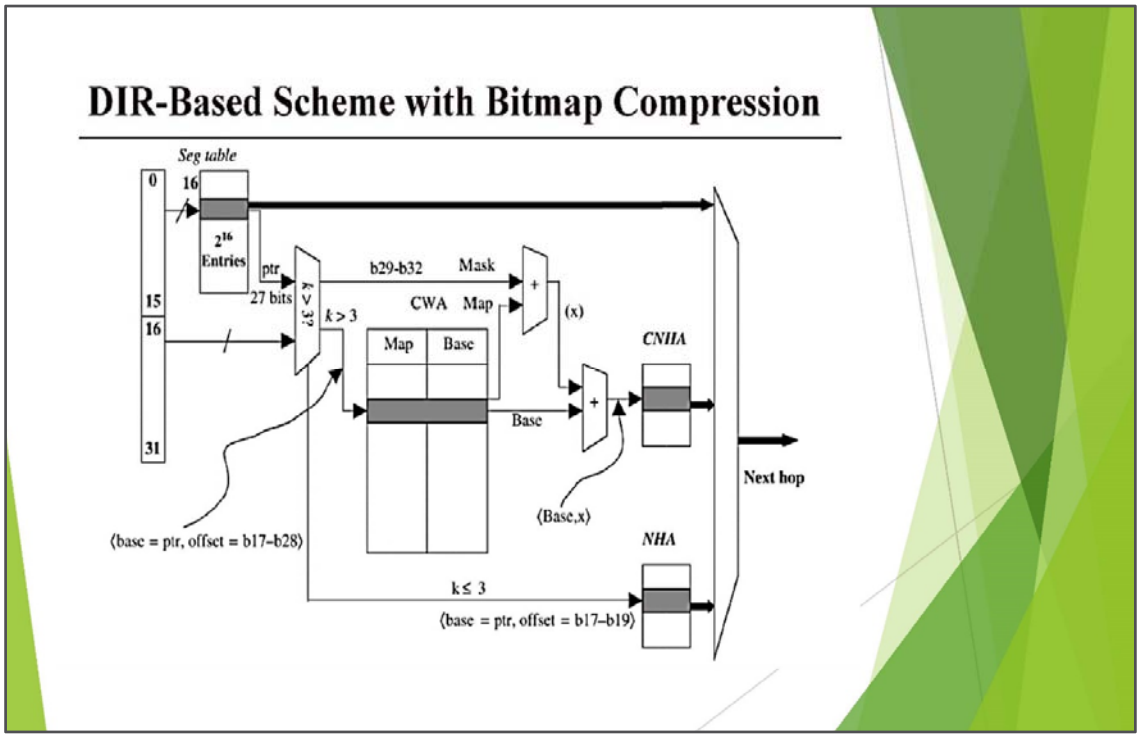
در روش Compressed Next Hop Array یا CNHA، می‌خواهیم فشرده‌سازی ایجاد کنیم. قبلا در سطح دوم حافظه مشاهده کردیم یک‌سری تکرارها را داریم. مثلا در سطح دوم یک Next Hop Array(مطابق با سطر اول شکل اسلاید داریم). حالا به روشی می‌خواهیم روی آن فشرده‌سازی انجام دهیم. یعنی مطابق با شکل Next Hop Arrayای داریم که می‌خواهیم آن‌را فشرده کنیم.روش فشرده‌سازی به‌این صورت است که برای اولین عدد بیت ۱ درنظر می‌گیریم و برای اعداد

تکراری صفر(سطر دوم شکل) طبق شکل به‌ازای ۲، ۱ قرار داده‌است. اگر ۲ تکرار شده به‌جای آن صفر قرار داده‌است.

تعداد بیت‌ها در بدترین حالت ۲۱۶ خواهدبود. چون موضوع‌مان ۱۶-۱۶ Direct است. اگر برای هر Prefix فرض کنیم حجم حافظه سطح دوم ما برابر با ۲۲۸ باشد، ۲۲۸بیت داریم. یک حافظه CNHA ایجاد می‌کنیم که این اعداد را فقط یک‌بار نگهداری می‌کنیم . به‌جای نگهداری Next Hop با حجم و افزونگی زیاد، Bit Map و حافظه فشرده شده را نگهداری می‌کنیم. در انتهای اسلاید ۶ ۷ ۸ ۲ را نشان داده و بایستی تعداد هر یک را نیز در این Bit Map مشخص کنیم.



بخش اول ۱۶بیت همان Bit Mapهاست. مقداری که اینجا قرار گرفته مشخص‌کننده Base موجود در Indexهاست. هر ۱۶بیت، Bit Map را در یک خانه از حافظه(یک کلمه حافظه) ذخیره می‌کنیم. معادل هرکدام را کد کرده و Base Index برایش درنظر می‌گیریم که به آدرس اولین Next Hopی که در آن ۱۶بیت برابر ۱ است، اشاره می‌کند. اگر Next Hop بیت اول ۱ باشد، ۰ را ذخیره می‌کنیم تا به معماری موجود در شکل برسیم.



این معماری یک حافظه ۱۶بیتی سطح اول را به ما نشان می‌دهد که تا عمق ۱۶ را جستجو می‌کند(از صفر تا۳۱). (IPV4: به‌جای ۱تا۳۲، ۰تا۳۱ را درنظر گرفته‌است). اگر Next Hop پیدا نشود، یک اندیس به حافظه سطح دوم که یک k است و نشان‌دهنده حافظه سطح دوم است را نشان می‌دهد.

اگر $K\leq 3$ باشد، Next Hop Array را فشرده‌سازی نمی‌کنند و آن‌را به‌صورت مستقیم در آرایه قرار می‌دهند و اگر $K> 3$ باشد ارزش فشرده‌سازی را خواهدداشت. به این روش CWA می‌گوییم که شامل Bit Map و Base Index است چون Bit Map و Base Indexای را ساخته و یک CHNA را هم به آن اضافه می‌کند.

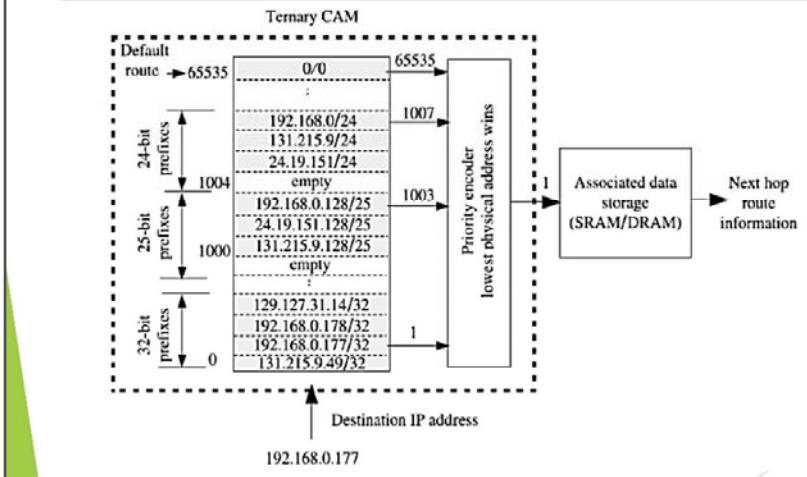
DIR-Based Scheme with Bitmap Compression

Performance. The basic idea of this lookup scheme is derived from the Lulea algorithm, which uses bitmap code to represent part of the trie and significantly reduce the memory requirement. The main difference between BC-16-16 and the Lulea algorithm is that the former is hardware-based and the latter is software-based. The first-level lookup of the BC-16-16 uses direct 16-bit address lookup while the Lulea scheme uses bitmap code to look up a pointer to the next level data structure.

BC-16-16 needs only a tiny amount of SRAM and can be easily implemented in hardware. A large forwarding table with 40,000 routing entries can be compacted to a forwarding table of 450–470 kbytes. Most of the address lookups can be done by one memory access. In the worst case, the number of memory accesses for a lookup is three. When implemented in a pipeline in hardware, the proposed mechanism can achieve one route lookup every memory access. This mechanism furnishes approximately 100M route lookups per second with current 10 ns SRAM.

این روش مشکل Update کردن دارد. اگر تغییری در جدول مسیریابی رخ دهد، بهم ریختگی‌ای در کل حافظه خواهیم داشت.

Ternary CAM for Route Lookup



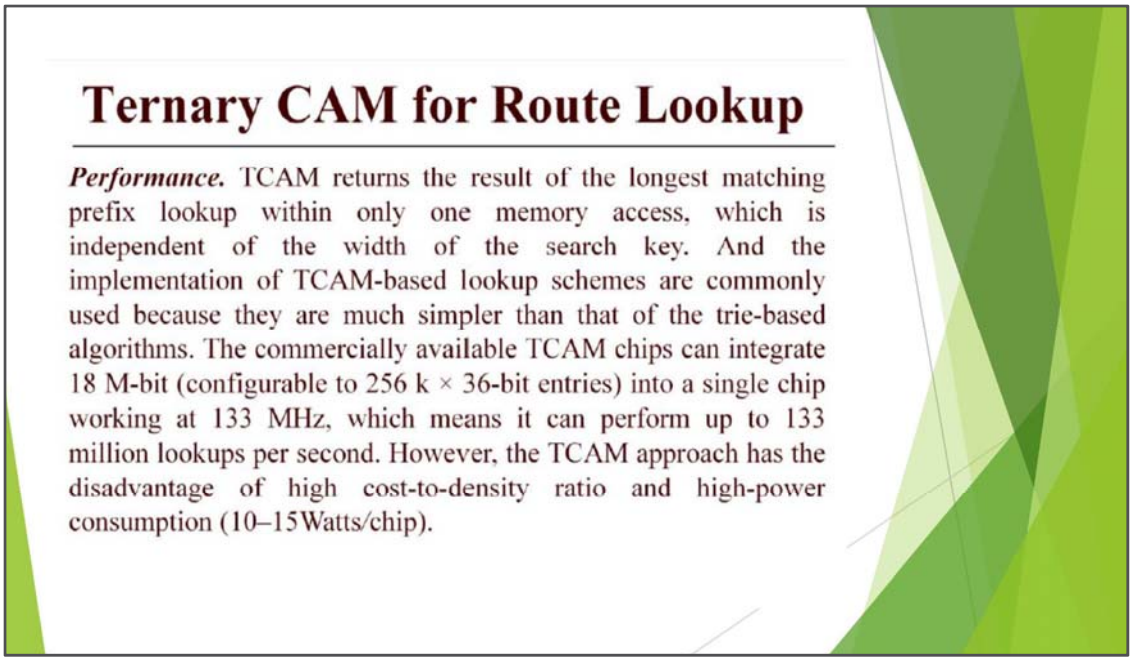
روش‌های سخت افزاری نوع دوم، مبتنی بر حافظه‌های کم هستند. در جدول مسیریابی اعداد ۳۲بیتی داریم و از این ۳۲بیت به تعداد Net IDها، آن‌ها را باید به آدرس مقصد بسته مقایسه کنیم. بنابراین در بعضی ۱۰بیت و برخی ۱۱بیت و برخی بیشتر مقایسه می‌شوند. عملا می‌خواهیم ک ۳۲بیت

را با یک ۳۲بیت مقایسه کنیم. ۳۲بیت اول را از هدر بسته برداشتیم و در جدول مسیریابی قرار داده‌ایم. فقط مقایسه Net IDها برایمان مهم است. طبق Subnet برخی بیت‌ها بی‌اهمیت Do Not Care تلقی می‌شوند. پس باید بتوانیم در حافظه بیت صفر، بیت۱و بیت بی‌اهمیت داشته‌باشیم. برای هر سلول باید بتوانیم این ۳حالت را داشته باشیم. به حافظه‌های کمی که بتوانند این سه حالت را ذخیره‌کنند، Ternary CAM یا CAM گفته می‌شود. در عمل هر Cell با دو بیت ذخیره می‌شود. یک بیت صفر، یک بیت ۱ و یک بیت بی‌اهمیت بودن Cell را مشخص می‌کند. از نظر پیاده‌سازی حافظه‌های Ternary Cam حجم را دوبرابر Ternary می‌کنند چون برای پیاده‌سازی برای ذخیره سه حالت به ۴بیت نیاز خواهیم داشت. (۴=۲۲)

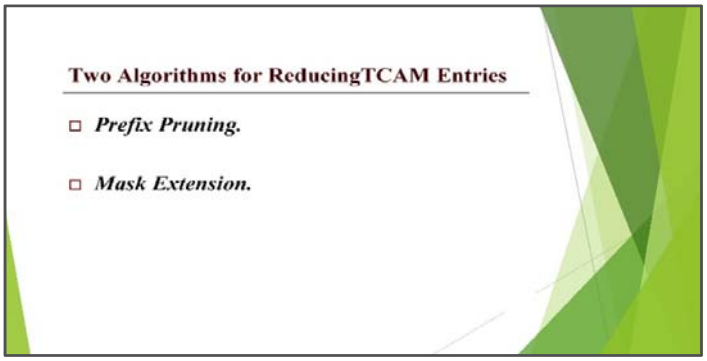
در روش Ternary CAM، Prefixها در حافظه CAM ذخیره می‌شوند. قسمت Net ID را ذخیره می‌کنیم و بقیه بی‌اهمیت‌ها ذخیره خواهندشد. برای مثال فرض کنید در همین اسلاید، بسته‌ای با مقصد 192.168.0.177 را دریافت کنیم. چه می‌شود؟

در جد.ل مسیریابی دقت می‌کنیم: ۳۲بیت این آدرس با خانه آدرس یکی از خانه‌ها تطابق خواهدداشت. ۲۵بیت این آدرس با خانه شماره 1003 مطابقت دارد. ۲۴بیت آن با خانه 1007 تطبیق دارد. همگی خانه‌ها با خانه آخر 00 مطابقت دارند.(پس با خانه اول طبق بحث Longest Match تطابق داریم.

در Ternary CAM در هر خانه حافظه مقایسه‌کننده سخت‌افزاری داریم که Prefixهای جدول را براساس تعداد بیت‌هایش به‌صورت نزولی مرتب می‌کند. بنابراین در Ternary CAM اولویت با حافظه کوچک‌تر خواهد بود. اطلاعات Next Hop در حافظه ذخیره می‌کنیم. برای خانه۱، اطلاعات Next Hop به‌دست می‌آید. پس دو بار مراجعه به حافظه خواهیم داشت. برای این‌که بتوانیم راحت تغییر ایجاد کنیم، می‌توان بین Prefixها، تعدادی خانه خالی هم قرار دهیم تا برای یک Prefix جدید نیاز به تغییر کل خانه‌های حافظه نباشد. این داده‌های موجود در اسلاید به همین دلیل هستند. پس تغییر با دوبار مراجعه به حافظه ممکن است. این روش هم سرعت جستجو و هم سرعت Update کردن را بالا می‌برد.



اشکال این روش مصرف انرژی است.حافظه‌های Ternary CAM حافظه‌هایی هستند که در هر خانه حافظه، مدار سخت‌افزاری وجود دارد. یک مدار مقایسه‌کننده ۳۲بیتی دارند، حال اگر دو میلیون Entry در جدول مسیریابی داشته باشیم و حافظه‌ای با سایز دو میلیون خواهیم داشت. به‌ازای هر Search دو میلیون مقایسه داریم. یک میلیون مقایسه در هر ثانیه نیاز به ۲×۱۰۱۲ مقایسه سخت‌افزاری داریم که انرژی خیلی زیادی مصرف خواهدکرد. پس بایستی به دنبال روشی جهت کاهش مصرف انرژی باشیم.

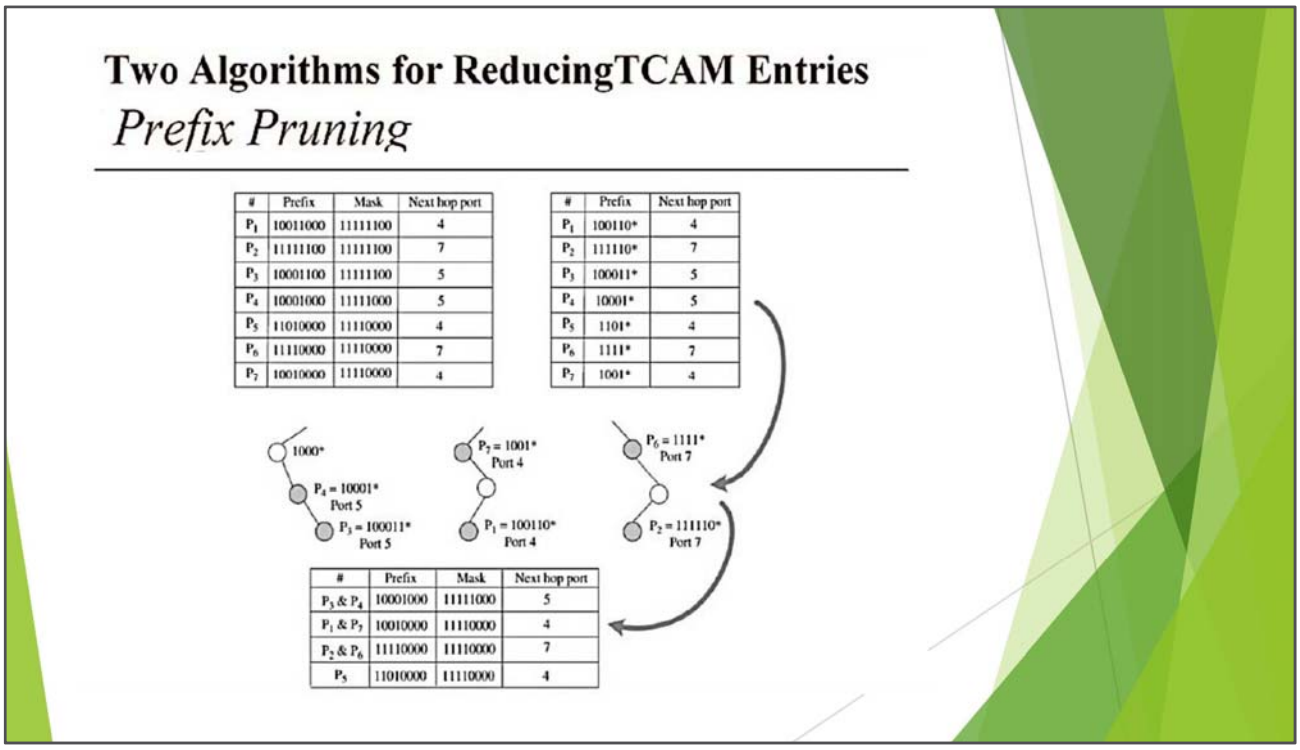


روش‌های زیادی جهت کاهش مصرف انرژی Ternary است:

روش اول کاهش اندازه‌ی جدول است. کم کردن تعداد Entryهای جدول و حذف اضافی‌ها

هرس کردن Prefixها

توسعه Mask

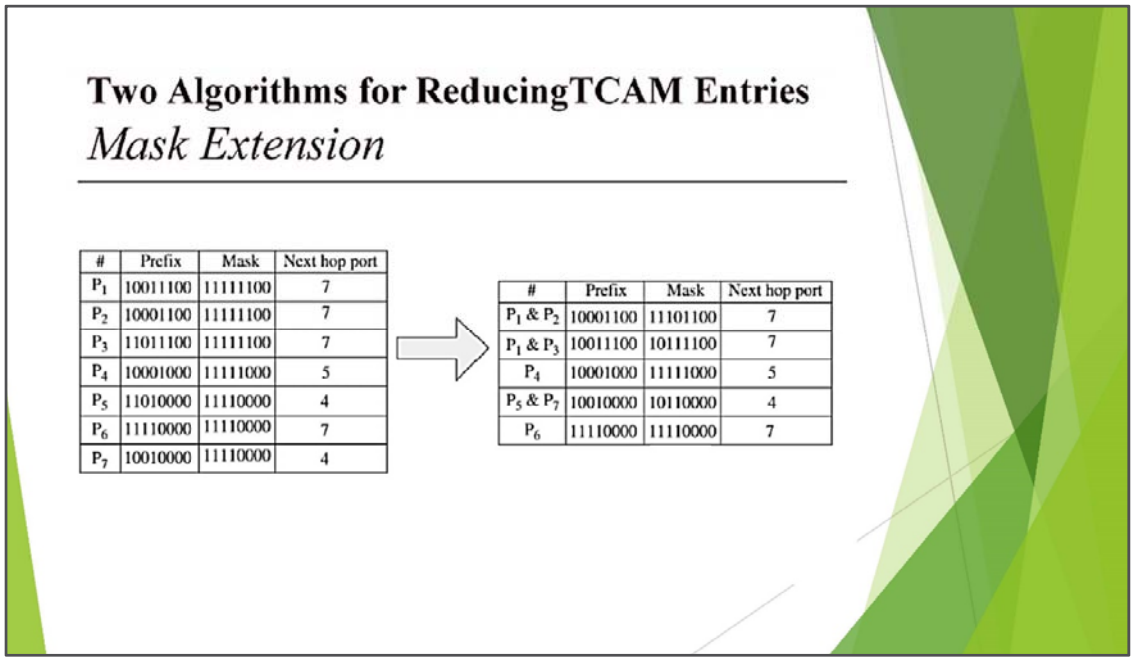


#	Prefix	Mask	Next hop port
P_1	10011000	11111100	4
P_2	11111100	11111100	7
P_3	10001100	11111100	5
P_4	10001000	11111000	5
P_5	11010000	11110000	4
P_6	11110000	11110000	7
P_7	10010000	11110000	4

#	Prefix	Next hop port
P_1	100110*	4
P_2	111110*	7
P_3	100011*	5
P_4	10001*	5
P_5	1101*	4
P_6	1111*	7
P_7	1001*	4

#	Prefix	Mask	Next hop port
P_3 & P_4	10001000	11111000	5
P_1 & P_7	10010000	11110000	4
P_2 & P_6	11110000	11110000	7
P_5	11010000	11110000	4

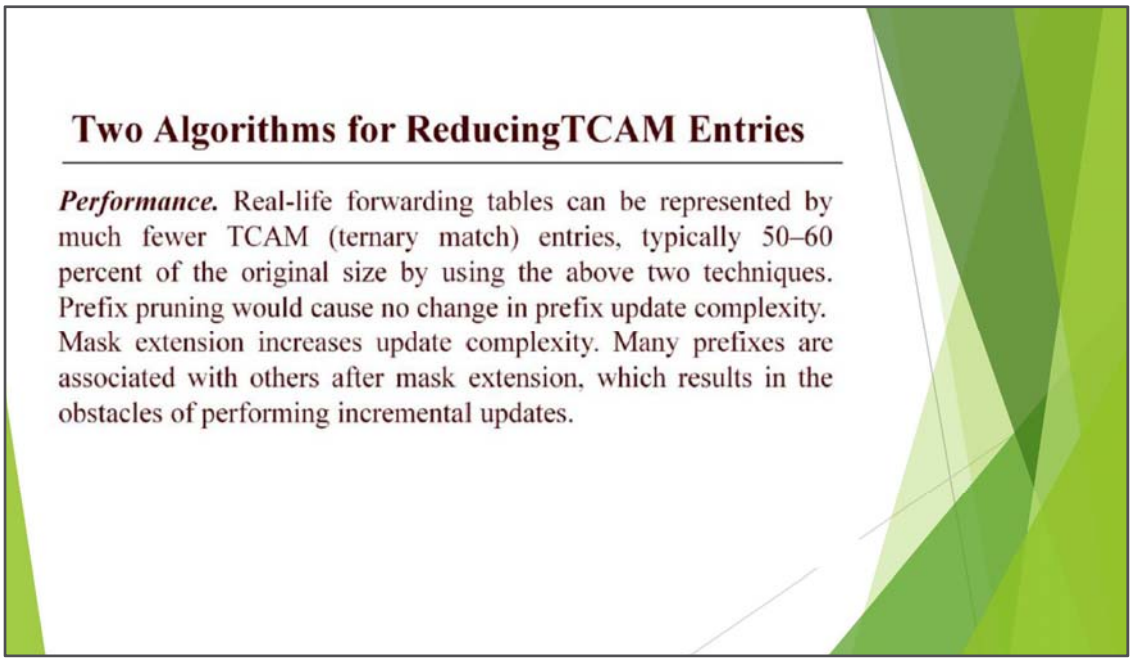
در روش Prefix Pruning از روی درخت Binary شاخه‌هایی را پیدا می‌کنیم که بودن Prefix در درخت تاثیری در جستجو نداشته باشد. به‌عنوان مثال اگر Prefixهای P3 و P4 را نگاه کنیم می‌توانیم بگوییم P3 زیرمجموعه‌ی P4 است. به این معنی که اگر Prefixهایی با P3 تطابق داشته باشند، حتما با P4 هم تطابق دارند و خروجی آن‌ها با هم یکسان هستند(خروجی پورت۵). پس حذف P3 تاثیری در نتیجه‌ی جستجو نخواهدداشت. در این مثال جدول هفت عنصری به چهار عنصری کاهش پیدا کرده‌است. کارایی این روش کاملا به جدول مسیریابی‌مان بستگی دارد. با توجه به این‌که اندازه جدول خیلی بزرگ باشد و تعداد پورت‌های روتر محدود است(حداکثر ۲۰۰پورت) پس به‌طور متوسط در صورت داشتن خروجی یکسان می‌توانیم از دو میلیون Prefix، ده‌هزار Prefix داشته باشیم.



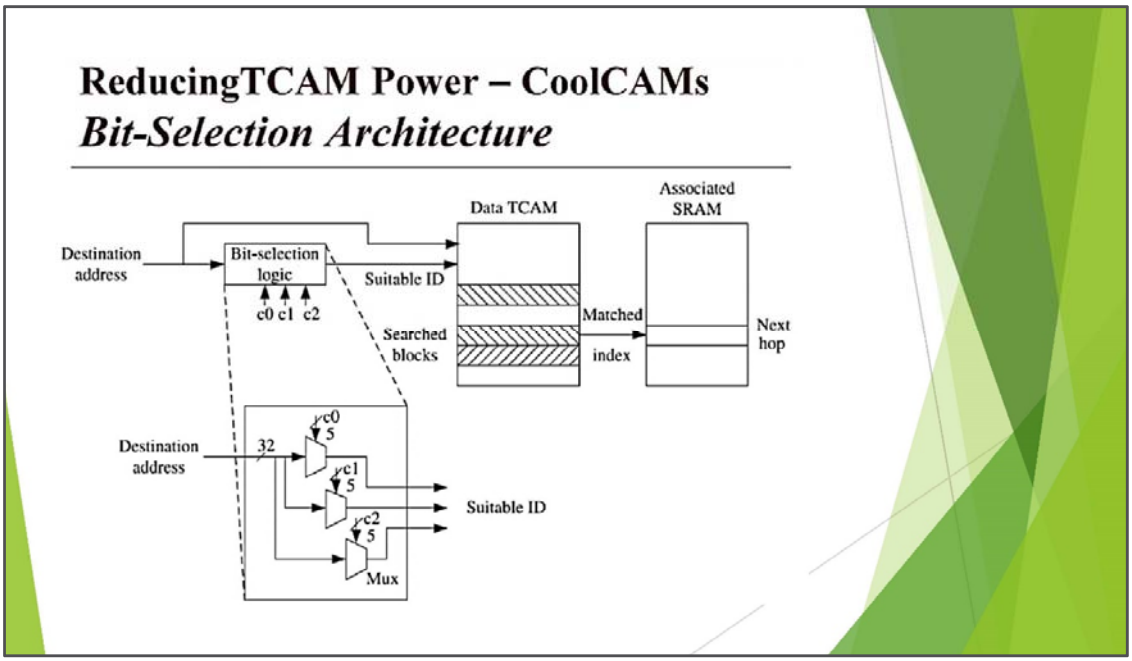
روش دوم: در این روش دنبال دو Prefix هم‌طول هستیم که پورت خروجی یکسان داشته‌باشند ولی Prefixها فقط در یک بیت با هم اختلاف داشته‌باشند. صفر و یک بودن یک بیت عملا در نتیجه جستجو موثر نیست. می‌توانیم دو مورد Prefix را در هم ادغام کنیم و یک بیت را بی‌اهمیت تلقی کنیم. عملکرد به محتوای جدول مسیریابی بستگی خواهدداشت.

نکته: به Subnetها توجه کنید. می‌توانیم بین آن‌ها یک بیت صفر هم داشته باشیم. درحالی‌که در حالت عادی همه‌ی Subnetها ۱ هستند و بعد از آن تمام آن‌ها صفر می‌شوند.

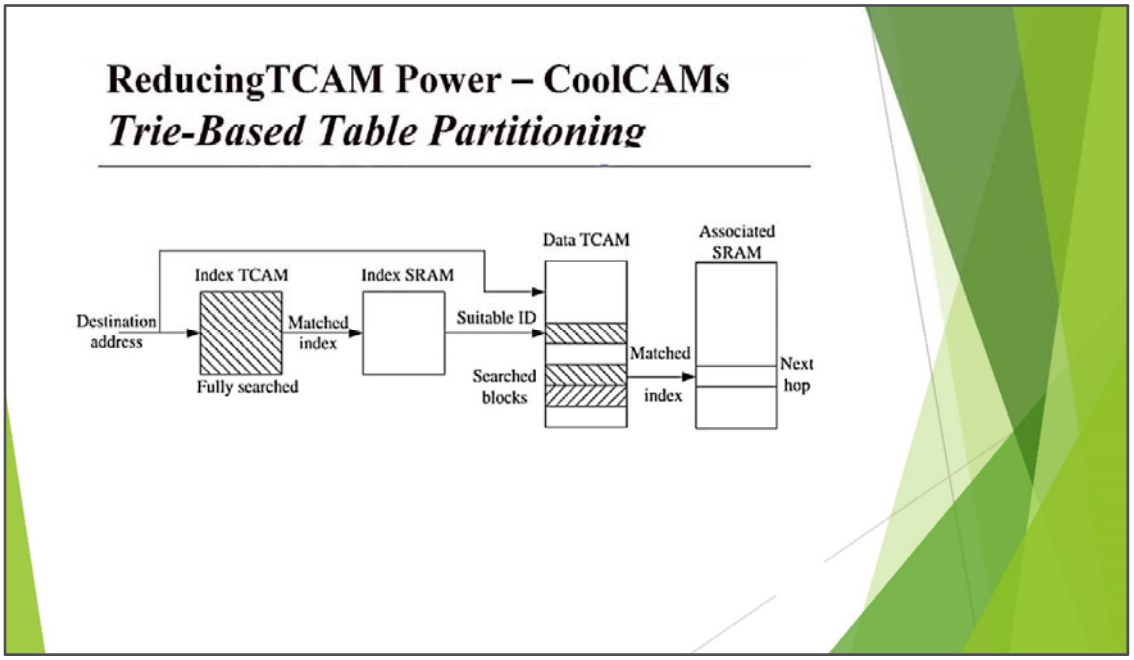
با دو روش گفته‌شده می‌توانیم درصدی اندازه‌ی جدول را کاهش دهیم ولی همچنان سایز جدول می‌تواند بزرگ باشد چون گفتیم کاملا بستگی به Prefix دارد.



باید روش‌هایی پیدا کنیم که روش‌های جستجو را کم کند. اساس این روش‌ها این است که Prefixهایی با ویژگی یکسان را در یک بلوک حافظه ذخیره کند.



اولین کار، بلوک‌بندی حافظه است. بعد از آن Prefixهایی که ویژگی یکسان دارند را در یک بلوک ذخیره کنیم وقتی آدرس مقصد را از روی هدر بسته برمی‌داریم کافی است این ویژگی‌ها روی آدرس مقصد بررسی شوند و از روی آن‌ها بلاک حافظه برگزیده شود. مثلا طبق شکل حافظه را به دو بلوک تقسیم می‌کنیم:Prefixهای با بیت اول۱ در بلوک بالا و Prefixهای با بیت اول صفر در بلوک پایین تعریف می‌کنیم. حالا آدرس مقصد را از روی بسته برمی‌داریم و می‌خواهیم Prefix را سرچ کنیم. اگر بیت اول آدرس مقصد ۱ است، فقط بلوک بالا را فعال می‌کنیم و برعکس. به‌این ترتیب ۵۰٪ می‌توانیم در مصرف حافظه صرفه‌جویی کنیم.اگر دو بیت را درنظر بگیریم بایستی حافظه ۴بلوک شود و اگر سه بیت را درنظر بگیریم بایستی حافظه ۸بلوک شود و...این‌که کدام بیت‌ها را درنظر بگیریم بستگی به جدول مسیریابی خواهدداشت. باید بیت‌هایی درنظر بگیریم که سایز بلوک‌ها هم‌اندازه بشود. براساس اطلاعات آماری مشخص کنیم کدام بیت‌ها را می‌خواهیم انتخاب کنیم. هرگاه نیاز به به‌روزرسانی جدول باشد ممکن است این بیت‌ها ذخیره کنند. مثلا اگر فرض کنید سه بیت انتخاب شود، ۸بلوک داشته باشیم، براساس اطلاعات آماری جدول سعی می‌کنیم به ۸گروه مساوی تقسیم انجام گردد. براساس آمار، بیت ۱م، ۵م و ۱۰م(مثل شکل) و C0، C1 و C2 نام‌گذاری می‌شوند. حال آدرس را از هدر بسته برمی‌داریم. سه بیت را نگاه می‌کنیم، شماره بلوک مشخص می‌شود. در همان بلوک سرچ می‌کنیم و سایز بلوک‌ها را غیرفعال می‌کنیم.



در روش Cool CAM به‌جای این‌که با انتخاب بیت شماره ID بلوک تعیین شود از Prefixهای ذخیره شده در بلوک یک Index ایجاد می‌کنیم. اول یک Index کوچک داریم که نشان می‌دهد پورت خروجی هر آدرس مقصد در کدام بلوک مشخص شده، درواقع Index شماره ID بلوک حافظه را نشان می‌دهد و آدرس مقصد را در این بلوک، جستجو می‌کند.

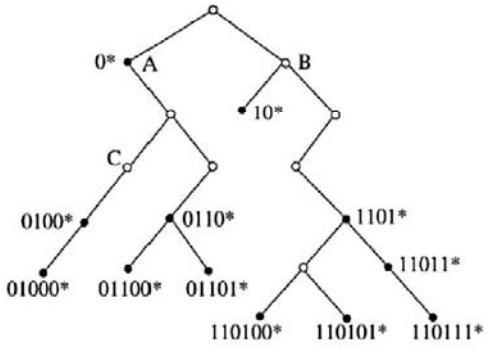
جهت ساخت Index روش‌های مختلفی وجود دارد. Bit Selection هم قابل استفاده است.

دو روش برای ساخت Index، هر دو براساس درخت پیشنهاد می‌کنیم: برای ذخیره سازی Prefix از درخت(Tree) استفاده می‌کنیم.(صفر باشد سمت چپ درخت و ۱ سمت راست)

ReducingTCAM Power – CoolCAMs
Trie-Based Table Partitioning

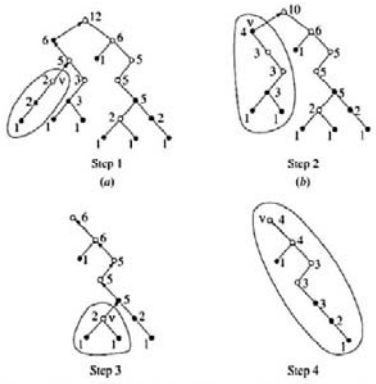
0*
0100*
01000*
0110*
01100*
01101*
10*
1101*
110100*
110101*
11011*
110111*

(a)

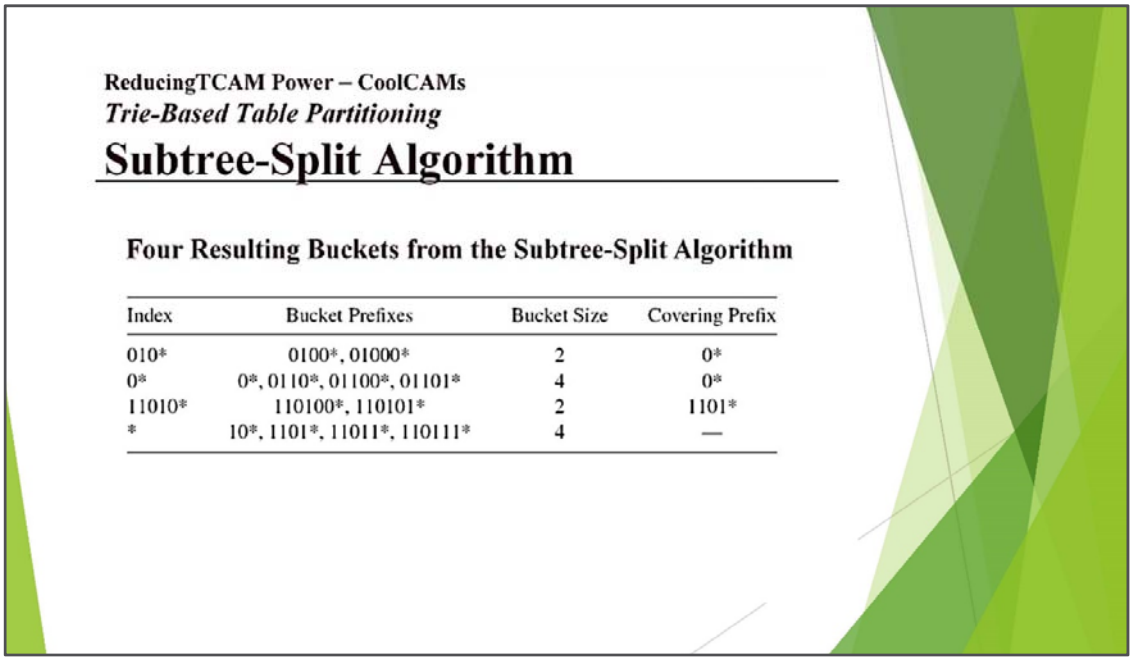


(b)

ReducingTCAM Power – CoolCAMs
Trie-Based Table Partitioning
Subtree-Split Algorithm



Four iterations of the subtree-split algorithm (with parameter b set to 4) applied to the 1-bit trie from previous figure.



به روش (Subnet) اگر درخت به چند زیر درخت تقسیم گردد و هر پارتیشن در بلوک ذخیره شود، می‌شود به ازای هر پارتیشن یک یا چند Index درنظر گرفته شود. دراین روش به هر نود از درخت باینری یک Value اختصاص می دهیمValue برابر با تعداد Prefixهای مجموعه شامل خودش و فرزندانش خواهد بود. نود پایانی۱ نود بالاتر ۲

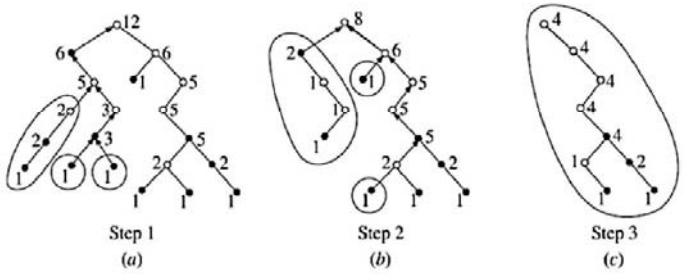
اگر Prefix باشند مقدار می‌گیرند. نود توپر مقدار می‌گیرد و نود توخالی مقدار نمی‌گیرد.

مجموع فرزندان می‌شود مقدار موردنظر ما. حال درخت را به روش Pre order(NLR) (اول ریشه، بعد سمت چپ و درنهایت سمت راست). وقتی Value یک نود کوچک‌تر یا مساوی سایز بلوکمان شد، همچنین این Value از نصف سایز بلوک بزرگ‌تر باشد این شاخه را می‌توانیم از درخت جدا کنیم و آدرس را به عنوان ID می‌توانیم ذخیره کنیم وPrefixها را ذخیره کنیم.

در اسلاید فرض شده است سایز بلوک برابر ۴ باشد پس اگر Value بین۲ تا ۴ باشد می‌توانیم شاخه را جدا کنیم. از ریشه شروع کرده است: مقدار ۱۲ و بزرگتر از ۴ است نمی‌توانیم آن را جدا کنیم. سمت چپ می‌رویم و تا جایی ادامه می‌دهیم که مقدار به ۲ برسد. به ۲ رسید می‌تواند جدا کند. ۵ را نمی‌تواند چون باید از ۴ کمتر باشد. این شاخه را می‌توان جدا کرد. آدرس گروه اول این شاخه را می‌توان جدا کرد آدرس گروه اول این شاخه برابر است با ۰۱۰۱ به عنوان آدرس در TCAM استفاده می‌کنیم و به همین ترتیب تا آخر.

ReducingTCAM Power – CoolCAMs
Trie-Based Table Partitioning

post-order split algorithm



Three iterations of the post-order split algorithm (with parameter *b* set to 4) applied to the 1-bit trie from previous figure.

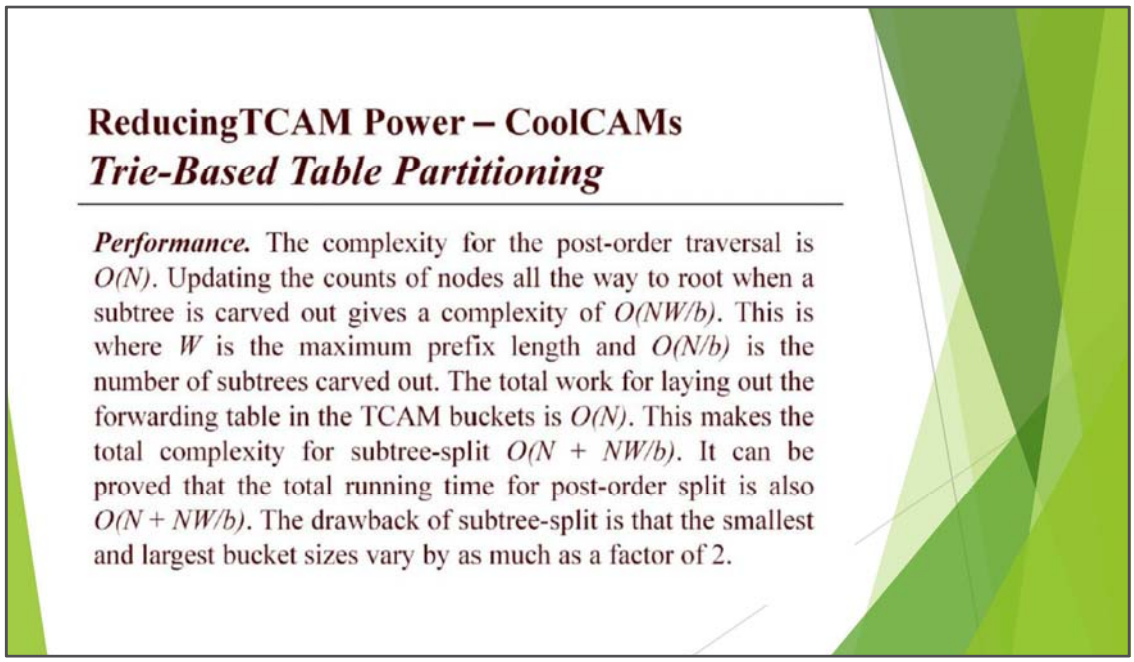
ReducingTCAM Power – CoolCAMs
Trie-Based Table Partitioning

post-order split algorithm

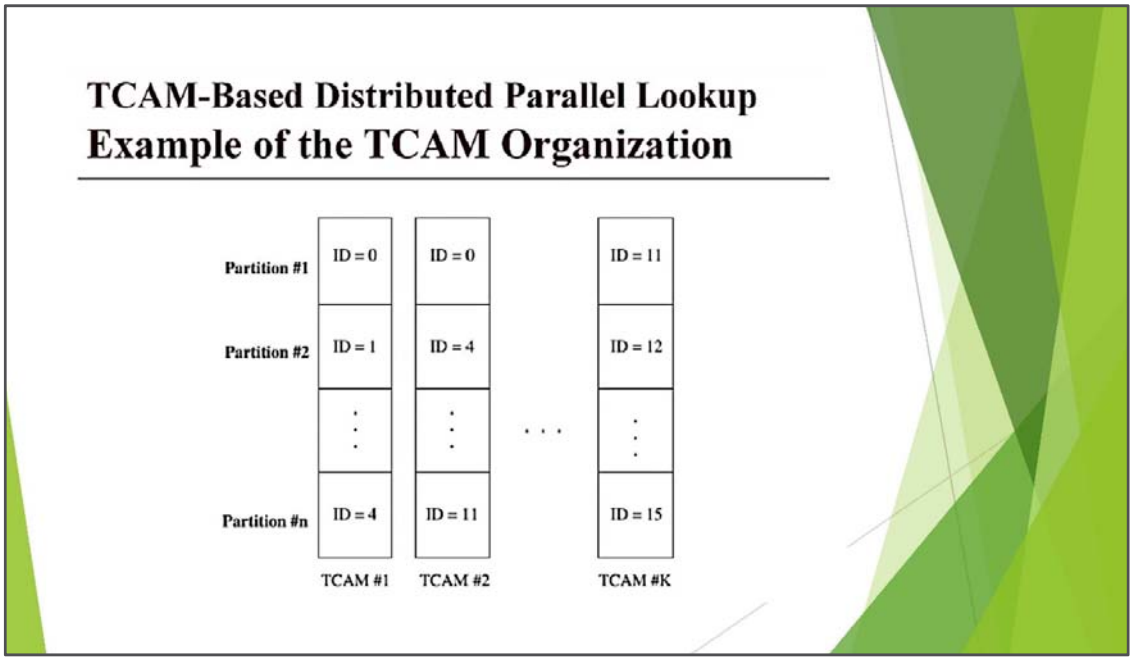
Three Resulting Buckets from the Post-Order Split Algorithm

i	$Index_i$	Bucket Prefixes	Size	Covering Prefix
1	010*, 01100*, 01101*	0100*, 01000*, 01100*, 01101*	4	0*, 01100*, 01101*
2	0*, 10*, 110100*	0*, 0110*, 10*, 110100*	4	0*, 10*, 110100*
3	1*	110101*, 1101*, 11011*, 110111*	4	—

در روش Post order Split Algorithm: پیمایش به صورت Post order است. اول چپ بعد راست و بعد ریشه. وقتی سر جمع Prefixهای پیمایش شده، دقیقا برابر سایز بلوک شد، آن زیر شاخه را جدا کرده و در یک بلوک حافظه ذخیره می‌کنیم. به همین دلیل ممکن است یک زیرشاخه با سایز بلوک نداشته باشیم بنابراین باید پیمایش را ادامه دهیم تا مجموع آنها به اندازه سایز یک بلوک شود. به همین دلیل در این روش حافظه بهینه‌تر از روش قبلی استفاده می‌شود و سایزهای ناقص بلوک را نخواهیم‌داشت ولی باید به ازای هر چند زیرشاخه، یک بلوک بسازیم و Prefixهای ریشه‌شان را به عنوان Index ذخیره کنیم. حسن این روش این است که حافظه TCAM بهینه می‌شود ولی طول Indexها افزایش خواهدداشت. برای مثال در درختی که ساختیم به مقدار ۵ می‌رسیم که از ۴ بزرگ‌تر است بعد به ۲ می‌رسد که کمتر است. این زیرشاخه را نگه می‌داریم تا با زیر شاخه‌های دیگر به مجموع ۴ آنها را برسانیم. دوباره پیمایش می‌کنیم به دو زیر شاخه می‌رسیم که مقدار ۱ دارند که به مجموع ۴ برسیم.

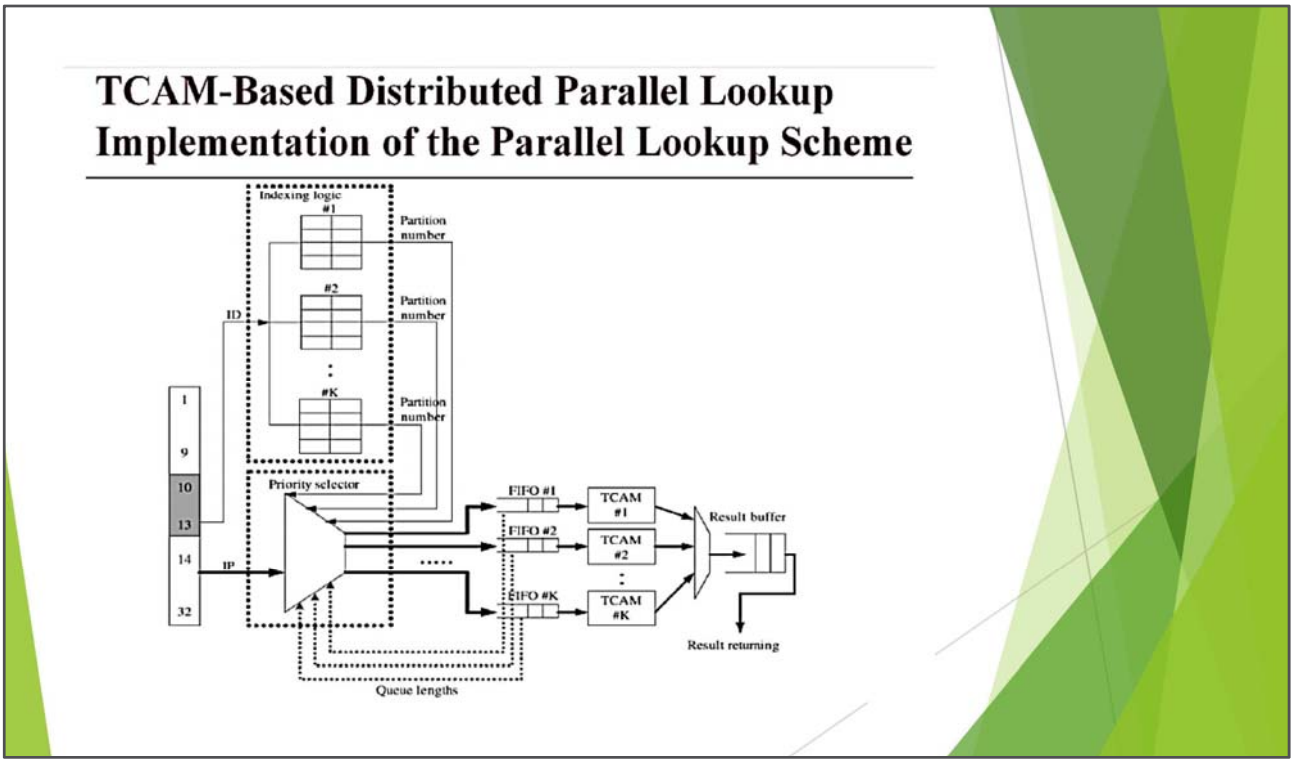


بحث Partitioning مطرح شده است.



گفته‌شد جدول مسیریابی را می‌توان به ۱۶بلوک تقسیم کرد. حال K حافظه TCAM را درنظر بگیرید که هر کدام به N بلوک تقسیم شده‌اند. بنابراین N×K تعداد کل بلوک‌هاست. به‌شرطی که N×K بزرگ‌تر از ۱۶ باشد. می‌توان بعضی بلوک‌ها را در دو مکان حافظه هم ذخیره کرد. مثلا بلوک صفر می‌تواند هم در حافظه‌ی اول ذخیره شود و هم در حافظه‌ی دوم. از آن‌جایی که این حافظه‌ها دارای سخت‌افزار مجزا هم هستند می‌توانیم به‌صورت همزمان آن‌ها را فعال کنیم درنتیجه وقتی جستجو در حافظه‌ی اول انجام می‌دهیم به‌صورت همزمان جستجو در سایر حافظه‌ها هم می‌توانیم انجام دهیم و سرعت جستجو در جدول مسیریابی را می‌توانیم Kبرابر کنیم.

ایده Distributed Parallel Lookup:یعنی Kتا TCAM به‌صورت موازی داشته‌باشیم و عمل جستجو در جدول مسیریابی را به‌صورت موازی همزمان در این Kتا بتوانیم انجام دهیم. در هر TCAM هر لحظه فقط یک بلوک می‌تواند فعال باشد پس از نظر مصرف انرژی هم این روش قابل قبول است.



در اینجا مدار سخت‌افزاری این روش را ساخته. چهار بیت ۱۳و۱۲،۱۱،۱۰ را به‌عنوان Index درنظر گرفته. یک مدار Index داریم که مشخص می‌کند کدام بلوک کجای حافظه ذخیره شده‌است. حال باتوجه به نتیجه این مدار اولا باید خانه حافظه مربوطه فعال شود ثانیا بلوک مربوطه هم باید فعال شود. مدار منطقی موجود در اسلاید این کار را برای ما انجام می‌دهد. اگر بلوکی تکرار شده باشد، این مدار سعی می‌کند با فیدبک گرفتن از اندازه، صف‌های جستجوی حافظه را روی صف کمتر قرار دهد. با این روش Kتا سرچ موازی خواهیم داشت.