國立交通大學

網路工程研究所

碩士論文

基於 R*-Tree 的位元圖交集封包分類演算法

Packet Classification Using R*-Tree based Bitmap Intersection

研 究 生：黃鼎峰

指導教授：陳 健 教授

中華民國九十九年六月

基於 R*-Tree 的位元圖交集封包分類演算法

Packet Classification Using R*-Tree based Bitmap Intersection

研 究 生：黃鼎峰　　Student：Ding-Fong Huang

指導教授：陳　健　　Advisor：Chien Chen

國 立 交 通 大 學

網 路 工 程 研 究 所

碩 士 論 文

A Thesis
Submitted to Institute of Network Engineering
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Computer Science and Engineering

June 2010

Hsinchu, Taiwan, Republic of China

中華民國九十九年六月

基於 R*-Tree 的位元圖交集封包分類演算法

Packet Classification Using R*-Tree based Bitmap Intersection

研究生: 黃鼎峰　　指導教授: 陳　健

國 立 交 通 大 學 網 路 工 程 研 究 所

摘要

隨著網際網路的蓬勃發展，網際網路服務如網路安全防護、虛擬私有網路、品質保證等的需求也越來越高。為了達到這些服務，網際網路路由器必須針對收到的封包做出快速的分類，而此過程我們把它稱之為封包分類。封包分類是利用封包標頭中所包含的資訊與路由器中事先定義好的規則做比對，依據比對出來的結果，執行不同的動作；多重欄位的封包分類是一個相當困難的問題，已經有許多的演算法被提出來解決這個問題。在這篇論文中，我們提出了一個透過結合位元圖交集(Bitmap Intersection)演算法與 R*-Tree 封包分類演算法來達到快速封包分類的演算法。透過效能分析以及實驗模擬，我們證明我們所提出的方法無論是在封包分類速度或是所需要的記憶體上，比起這兩種方法都有著顯著的進步。

關鍵字:封包分類、R-Tree、R*-Tree、位元圖交集法

Packet Classification Using R*-Tree based Bitmap Intersection

Student: Ding-Fong Huang Advisor: Dr.Chien Chen

Department of Computer and Information Science
National Chiao Tung University
Hsinchu, Taiwan, 300, Republic of China

Abstract

With the vigorous development of the Internet, there is a stringent speed requirement for processing Internet services such as network security, virtual private network, and quality of service. To accomplish these Internet services, Internet routers must have a fast packet classification capability for incoming packets. Packet classification is the process of identifying a set of pre-defined rules to which a packet matches. Then according to the matched rules, different actions can be performed. Multi-field packet classification generally is a difficult problem, and many different solutions have been proposed to solve this problem. In this thesis, we propose a fast packet classification algorithm that combines bitmap intersection algorithm and R*-Tree algorithm. Through analysis and simulation, we prove that our solution has a significant improvement over both original algorithms in classification speed and memory usage.

Keywords: Packet Classification, R-Tree, R-Tree, Bitmap Intersection*

誌謝

首先，要感謝的是我的指導老師陳健教授，由於老師的督促與指導才能夠完成這篇論文。從進入交大到現在，從老師身上學到解決問題以及研究的方法，重要的是學習到面對事情應有的態度以及觀念。我將會沿用從老師身上所學的東西到我的人生中。

在此，也感謝口試指導委員交通大學的簡榮宏教授、陳俊穎教授以及中興大學的王丕中教授，由於您們的指導與建議才使得本篇論文得以更完善。

接下來要感謝實驗室的學長們:張哲維學長、陳盈羽學長，以及同學們:孫冠宇、莊敬中，由於大家平時在課業與研究上的勉勵以及幫助，使得我的研究所生涯更加充實且順利。同時也要感謝實驗室的學弟，希望你們能夠把握在交大的日子，更加充實自己。

最後，更要感謝我的家人以及女朋友瀞誼，由於你們的照顧以及支持，我才能夠如此順利的完成碩士的學業。在此向你們致上最高的謝意。

我的求學階段即將告一個段落，非常感謝在我求學階段中幫助我的師長、朋友以及家人。感謝你們對我的拉拔以及指導，沒有你們的幫忙不會有現在的我。我將會好好利用你們教給我的東西，面對我未來的人生。

目　　錄

圖目錄

表目錄

一、簡介

網際網路已經成為我們日常生活中的一部分，許多不同的應用也相繼出現在網際網路上，而針對這些越來越多樣化的應用，網際網路服務也變得越來越重要，例如網路安全防護、虛擬私有網路、品質保證等等。要實現這些網路服務，皆需要路由器有能夠分類封包的能力。但隨著網路速度的增加，一般的封包分類演算法已經無法負荷高速的網路速度，如何能夠加速封包比對的速度是目前路由器設計的重要問題。

路由器在執行封包分類的時候，是利用封包標頭的資訊與路由器內部的規則表作比對，以找出對應的規則。規則表是由多個規則所組成的，一個規則由多個欄位組成，這些欄位可能會是一個 Prefix (如 IP source/ destination IP address) 、 一段值域 (如 source /destination port) 、或是一個數值 (如 protocol type) ，另外每個規則各自會有不同的優先權以及所對應的動作，以網路安全防護為例，動作可能是允許或是拒絕接收此封包。

如圖 1 中，路由器中包含一個規則表以及負責處理封包分類的單元，規則表中的規則包含兩個部分，欄位與規則對應的動作。當一個封包到達路由器時，首先會擷取出封包的標頭以取得分類時所需要的資訊，也就是對應到規則的那些欄位。然後封包分類單元利用這些欄位中的資料，去規則表中找尋對應到此封包的規則;所有欄位都必需要符合才是所需要的規則,規則表中可能會有許多規則跟這個封包符合，此時擁有最高優先權的規則才是最後所需要的規則，路由器則依照此規則所預先定義的動作來處理這個封包。

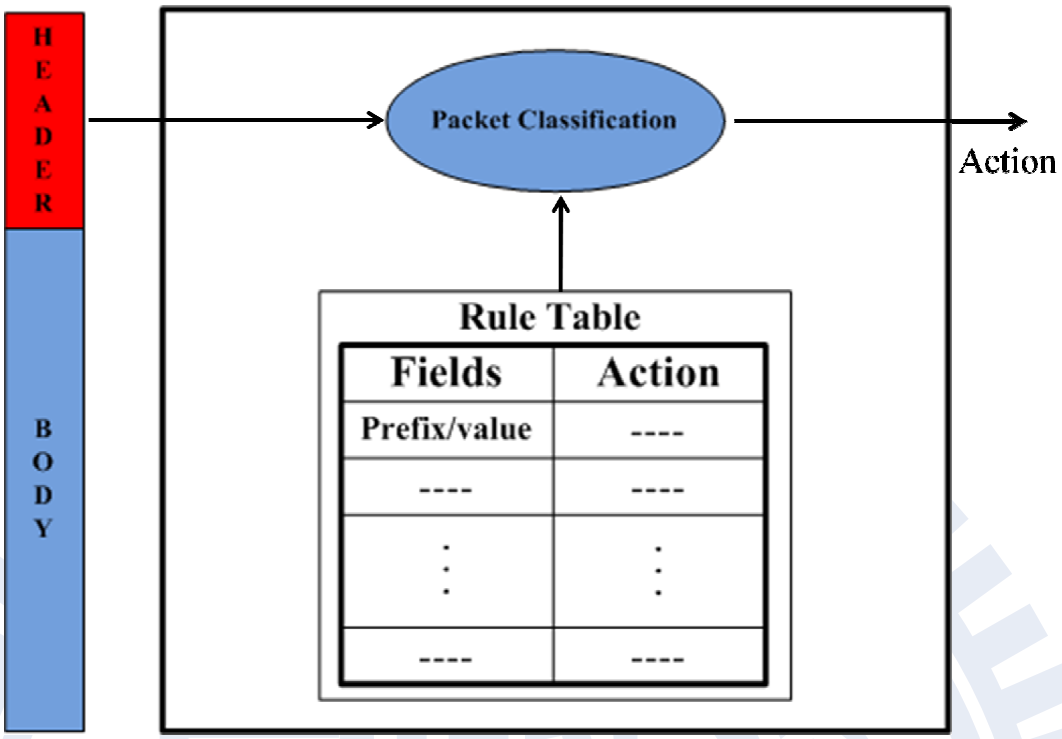

圖 1 路由器中封包分類過程

而封包分類問題主要就是用以解決路由器中，如何把封包與規則作出配對的問題。維度為 d 的封包分類問題可以用以下的方式定義;規則表 R 內會包含維度為 d 的規則之集合，$R=\{R_1,R_2\ldots,R_n\}$，每個規則有各自的優先權。每個規則是由 d 個欄位所組成，$R_i = \{F_{1,i}, F_{2,i}\ldots,F_{d,i}\}$，其中 $F_{j,i}$ 表示規則 i 的第 j 個欄位所設定的值。一個封包 P 包含 d 個標頭資訊，$P = \{P_1,P_2\ldots,P_d\}$，其中 P_i 對應到規則中的第 i 個欄位。當一個封包 P 配對到規則 R_i 時，代表 P 中的 d 個標頭資訊都配對到 R_i 中各自對應的欄位。如果封包 P 配對到多個規則，其中優先權最高的規則為最後的結果。

傳統的封包分類問題可以把它視為在多維圖形空間點位置的搜尋問題 [8]。在多維空間中，我們可以把規則視為存在於空間中的多維矩形，也就是說，規則中的各個欄位代表此矩形在不同維度中所涵蓋的區域，而封包的標頭資訊則是在多維空間中的一個點。如此一來，我們就可以把傳統的封包分類問題看成是在多維空間中搜尋包含某個點

之所有矩形的問題。圖 2 為一個規則表對應到二維空間的範例。此範例中總共包含四個維度為二的規則以及一個封包 P，規則分別被對應到圖形空間中部分的區域。把封包 P 對應到空間後，我們可以得到其所對應的規則，在此範例中即 R3 與 R4，其中又由於 R3 的優先權較高，所以最後會得到到 R3 這個規則。

規則表

規則	F1	F2
R1	10	01
R2	11	0*
R3	*	1*
R4	0*	*

封包	F1	F2
P	01	10

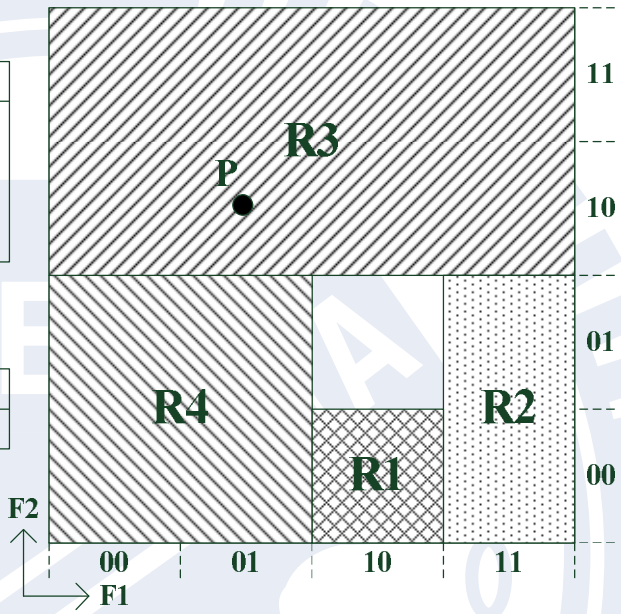

圖 2 規則表對應圖形空間範例

上述的問題是一個很困難的問題。傳統解決方法不是需要大量的儲存空間，就是比對的時間過長。一個好的封包分類演算法必須同時具備快速的分類以及少量的儲存空間的特性。我們觀察到位元圖交集演算法[12]在封包比對的速度上有著不錯的效能，但是當它在處理規則數量大的規則表時，會需要使用大量的記憶體空間，以及在處理封包時需要存取記憶體的次數也會隨著規則數量而增加。存取記憶體次數的上升就代表須要花更多的時間在記憶體存取，這會影響封包分類的速度，讓封包分類的速度下降。另外我們觀察到 R*-Tree 演算法利用樹的架構來索引圖形空間中的物件，即使索引的圖形空間物件數量很大，也還是能夠很有效率的處理這些物件。但是 R*-Tree 在搜尋資料時所花費的時間比較長，是因為它是利用線性搜尋的方式來搜尋節點中的資

訊。總和以上兩種不同演算法的優缺點，我們想說是否能夠利用這兩種演算法各自的優點，去克服彼此的缺點。因此在本篇論文中我們提出了結合與 R*-Tree 封包分類演算法[2]的方法。我們提出的方法主要是要解決圖形空間中的封包分類問題，使用 R*-Tree 來索引圖形空間中的規則，並且利用位元圖交集法來取代原本 R*-Tree 中的線性搜尋。使用 R*-Tree 來索引規則能夠有效率使用記憶體，使用位元圖交集法來搜尋 R*-Tree 能夠提升搜尋時的效能。藉由實驗我們所提出的方法，實驗數據顯示它確實能夠同時滿足分類速度快與儲存空間少的需求。

本篇論文接下來的部分包括:第二章會介紹封包分類問題先前的相關研究；第三章將會提出我們的演算法以及演算法的效能分析比較；第四章中，我們會提出兩個方法來改善我們所提出的演算法；第五章中，我們經由模擬比較我們的方法以及其他文獻中方法的效能；最後第六章則是對於此篇文章的結論以及未來展望。

二、相關研究

在封包分類這個領域中，有許多的論文以及演算法被提出用以解決這個問題，大致上可以分作為下面幾類:Trie-based 演算法、圖形空間演算法、探索式演算法、以及 TCAM-based 演算法等四大類 [7]。我們的方法是屬於圖形空間演算法類別，在此我們簡單扼要地描述幾個圖形空間演算法的相關研究，其他更完整詳細的封包分類演算法介紹可以參考下列論文 [4][6][9]。

HiCuts [10]主要是把所需要的空間依照不同的維度做出切割，每一次的切割會把某個維度切成多個等分的區塊，每個區塊中則會包含在這個切割範圍內的規則，而這些區塊如果所包含的規則數過多的話，會再以同樣的方式切格其他維度，分成更多的區塊。以上步驟將重複直到區塊中的規則數小於一定數量或是所有的維度都被切割完為止。最後再使用一個決策樹來索引這些切割的區塊。在做封包分類的時候，它會依照切割的資訊來搜尋這個決策樹，找到對應的樹葉節點。此節點內會包含可能會對應到此封包的規則，由於其包含的數量不會太多，所以這邊會使用線性搜尋的方法來找到對應的規則，而這個方法的效能主要會受到實際規則表的分佈影響，所以它的效能比較難以被評估。

HyperCuts [11]和上面所提到的 HiCuts 非常的相似，主要的差別在於 HiCuts 切割時一次只會依照一個維度作切割，而 Hyper-Cuts 則是於單次切割中會對多個維度作切割。Hyper-Cuts 也同樣使用一個決策樹來索引它所切出來的空間資訊。在 Hi-Cuts 中決策樹內的節點會記錄著單維的切割資訊，而在 Hyper-Cuts 中則是紀錄多維的切割資訊，也就是它會利用多維陣列來紀錄對應多維切割的資訊。在效能上 Hyper-Cuts

有著不錯的速度，也不需要很大的記憶體空間，但是這個方法對於百搭符號的規則(wildcard rule)有較差的效能。

Lakshman et al. [12] 所提出的位元圖交集 (Bitmap Intersection) 封包分類演算法，其主要的精神是 divide–and-conquer，把封包分類的問題切成數個子問題，最後再把所有的解合併得到真正的結果。這個演算法同樣也是基於圖形空間的封包分類法。假設在同型空間中總共包含 N 個規則，它最多會在每個維度上都產生 2N+1 個不重複的圖形區間，每一個區區間都會包含一個 N 位元長度的位元向量(Bit vector)，位元向量中的第 j 個位元在規則 j 有重複到此圖形區塊的情況下會被設定成 1，反之的話會設成 0。如圖 2.1 中，X1 區間中只有包含 R4，所以第四個位元被設定成 1，而其他規則並沒有包含在此區間中，所以其他的位元被設定為 0。

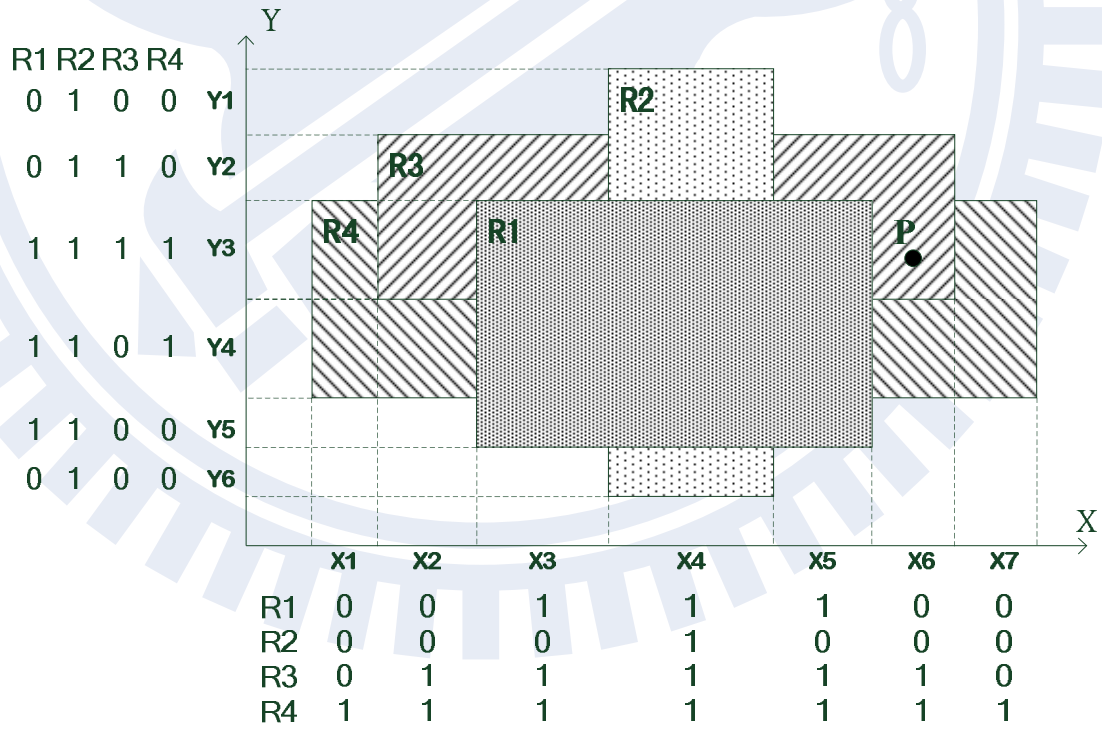

圖 3 位元圖交集法

當封包被送來的時候，首先利用二元搜尋把每個維度中對應到這個封包標頭資訊的圖形區間所屬的位元向量取出來，接下來把取出來的這些位元向量作一個交集的動作，產生另一個的位元向量。在此位元向量中所有被設定的位元就代表比對到的規則，其中優先權最高的規則就是我們想要的規則。由於規則事先以依照其優先權作遞減順序的排列，所以在這位元向量中第一個被設定的位元所對應的規則就是優先權最高的規則。圖 3 呈現一個簡單的例子，假設存在一個封包 P，其對應到的區間分別為 X6 以及 Y3，它們所屬的位元向量分別為 0011 與 1111，則我們得到的交集結果位元向量為 0011，所比對到的規則是 R3 和 R4，因為 R3 的優先權比較高，所以得到的規則就為 R3。

這個方法因為把不同維度的比對分開執行，所以可以同時的對所有維度作比對的動作，這可以利用實作在硬體的方式來達到高速的封包分類。此方法在規則數量變大的時候則會有所需的儲存空間變大以及比對所需要的時間變多等缺點，原因是因為當規則數變大的時候就代表需要更長的位元向量以及更多的圖形區間，所以所需要的儲存空間會變大;而由於位元向量變長，所以做比對的時候取出位元向量的時間會變長，導致比對的時間變長。

R*-Tree 演算法 [2] 是一個用來索引圖形空間中物件的一個演算法， Maindofer et al. [3]提出了把 R*-Tree 使用在圖形空間的封包分類上的想法。R*-Tree 為 R-Tree 的延伸，它本身的架構跟 R-Tree [1]十分的類似。首先，我們在此介紹 R-Tree，它是用階層式架構來索引圖形空間物件，透過搜尋 R-Tree 我們能夠知道某個區域中包含那些圖形空間物件。圖形空間物件代表空間中的一個區域，而且此區域必須要是矩形的，在 R-Tree 中圖形空間物件會被視為最小包含區塊(Minimal Bounding Region，MBR)，如圖 4 中的 a、b、c 與 d 這四個 MBR，每

個 MBR 都分別對應到圖形空間中的不同區域，這些區域有可能相互重疊。數個 MBR 會構成一個更大的 MBR，這些 MBR 又會再構成一個更大的 MBR，直到最大的 MBR 包含到整個圖形空間中所有的 MBR 為止。每個 MBR 所包含的下一層的 MBR 為此 MBR 的子 MBR。以圖 4 中左邊圖為例，MBR B 是由 a、b、c、d 四個子 MBR 所構成，其所包含的圖形空間區域是由構成它的 MBR 所決定，MBR B 所包括的範圍為能夠容納其所屬 MBR 的最小區域。R-Tree 中利用樹資料結構來紀錄空間中的 MBR 資訊，樹中的節點代表著某個 MBR，深度越深的節點代表越底層的 MBR。 R-Tree 中的每個節點紀錄一個 MBR 所包含下一層所有子 MBR 的圖形空間資訊以及在記憶體中的對應這些子 MBR 的節點的指標(Pointer)。若一個 MBR 內所包含的子 MBR 的個數為 X，則 $m \le X \le M$，其中 m 表示節點中包含 MBR 數量的最少個數，M 表示最大的個數。圖 4 的右邊圖為節點的架構圖，節點 B 對應到右邊圖中的 MBR B，它記錄下一層的子 MBR 資訊，它們為 a、b、c、與 d 所包含空間的範圍資訊以及所對應節點的指標。

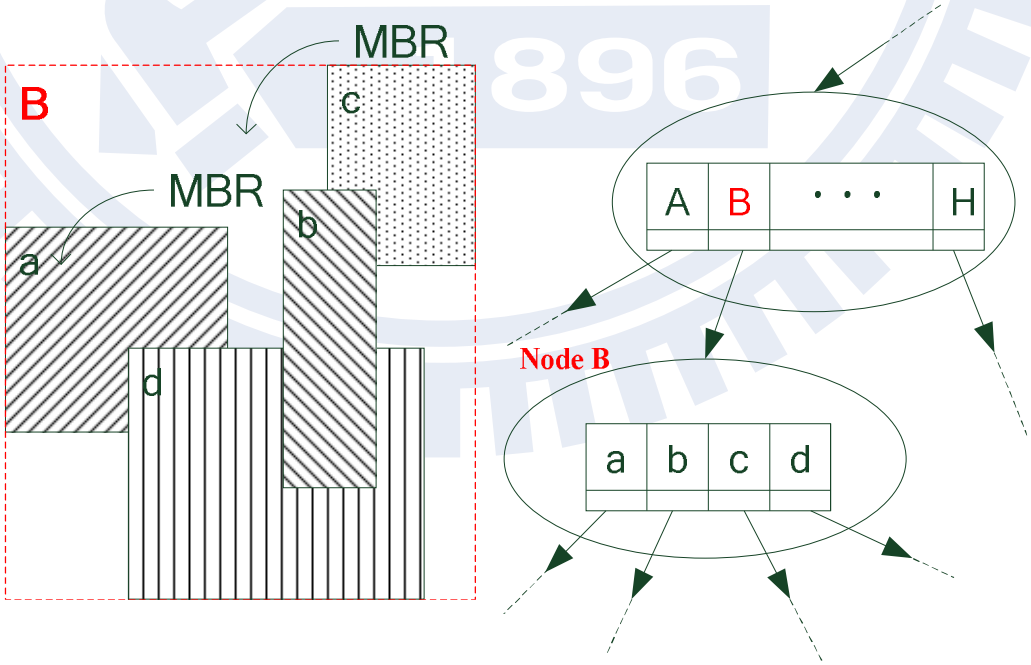

圖 4 R*-Tree 節點架構

在 R-Tree 的建立上，是透過把 MBR 依序的插入到 R-Tree 的方式來建立，在插入 R-Tree 時，會先根據插入後對節點體積的改變大小來決定目前要插入的這筆資料所要插入的樹葉節點(Leaf Node)，其中改變體積最小的就是這筆資料所要插入的樹葉節點。在插入後假若此樹葉節點內的 MBR 數量超過 M 則需要執行分裂的動作，把此樹葉節點切成兩個節點。之後則必須因應此次的插入調整 R-Tree 內的其他節點，如果過程中根節點(Root Node)因為包含過多 MBR 而分裂成兩個節點，那就必須要產生一個新的根節點，新的根節點會包含從舊根節點分裂出來的這兩個節點，調整完成後插入的動作也就完成了。

當我們想要搜尋空間中某個區塊的所包含的圖形空間物件時，首先，搜尋的動作從樹中的根節點開始，藉由節點中所儲存的 MBR 的圖形空間資訊，找出根節點中與搜尋區塊在空間上重疊到的 MBR，再到這些 MBR 對應的指標所指到的下一層節點中搜尋，重複以上動作直到到達樹葉節點為止，在樹葉節點中也是執行相同的動作，找出與搜尋區塊重疊到的 MBR，這些 MBR 就為最後的搜尋結果。例如在搜尋圖 4 中的 B 節點時，會使用線性搜尋的方式，依序地讀取節點中 MBR 的空間資訊以及比對是否與搜尋的區域重疊。假設搜尋的空間與 c 重疊的話，則會繼續往 c 對應的節點作搜尋。

R*-Tree 與 R-Tree 兩者十分的相似，它們使用相同的資料結構，但兩者相較，R-Tree 在搜尋上效能較差，是因為 R-Tree 中 MBR 之間重疊比率較高，這會導致搜尋時需要走較多的路徑，如圖 4 中，若我們想要搜尋的物件屬於 b 的子 MBR，並且它的位置位於 b 與 d 重疊的區塊中，在搜尋節點 B 時我們並不知道物件是位於 b 還是 d 的子 MBR，因此這兩個 MBR 我們都需要搜尋，但事實上 d 的子 MBR 中並沒有我們想要的物件，所以搜尋 d 會造成多餘的花費，這個問題會讓搜尋速

度降低。R*-Tree 利用以下四個性質來最佳化它的效能:

1. 應該要最小化 MBR 所包含的區域，在此所要最小化的區域為此 MBR 包含可是其子 MBR 所不包括的區域，如圖 4 中節點 B 的空白區域。
2. MBR 彼此的重疊區域必須最小化，這會減少搜尋時所需要經過的路徑數量。
3. MBR 的邊也應該要最小化，這裡所指的邊為 MBR 所有的邊線長之總和，在空間的體積固定的情況下，正方形的邊線總長為最小的，換句話說最小化邊線總長會讓 MBR 的形狀更接近正方形。由於正方形比起其他形狀更容易被包裝，因此能夠降低包裝此 MBR 所需要的空間，進而減少整體 MBR 的體積。
4. 空間利用度應該要最佳化，空間利用度指的是一個 MBR 內能夠包含的子 MBR 數量，節點中 MBR 的數量越接近 M 代表空間利用度越高，這會讓樹的高度降低，因此能夠使得搜尋時的花費降低，增進搜尋時的效率。

基於以上四個特性，R*-Tree 在決定 MBR 要插入的樹葉節點的演算法以及 MBR 的分裂演算法上作了更改，以符合這四個特性。並且另外增加了強制重新插入的機制，此機制會發生在插入資料時，當樹中某個階層第一次發生 MBR 超載時會採用此機制來處理超載的 MBR。此機制會把 MBR 中部分的子 MBR 取出，然後再重新插回到樹當中。此機制會選擇離 MBR 中心較遠的子 MBR 作重新插入，由於較外側的子 MBR 被重新插入了，所以 MBR 的形狀會更加趨近正方形，這正滿足第三項特性，所以透過強制重新插入能夠增加效能。透過上述的修改，R*-Tree 建樹演算法所建立出的樹比起 R-Tree 建樹演算法建出的樹，MBR 彼此重疊的情況將會比較少。因此在搜尋時，就能夠避免掉先前

所提到因為 MBR 彼此重疊而在搜尋時需要搜尋多餘路徑的情況。所以在搜尋時 R*-Tree 能夠比 R-Tree 更快速的得到結果。

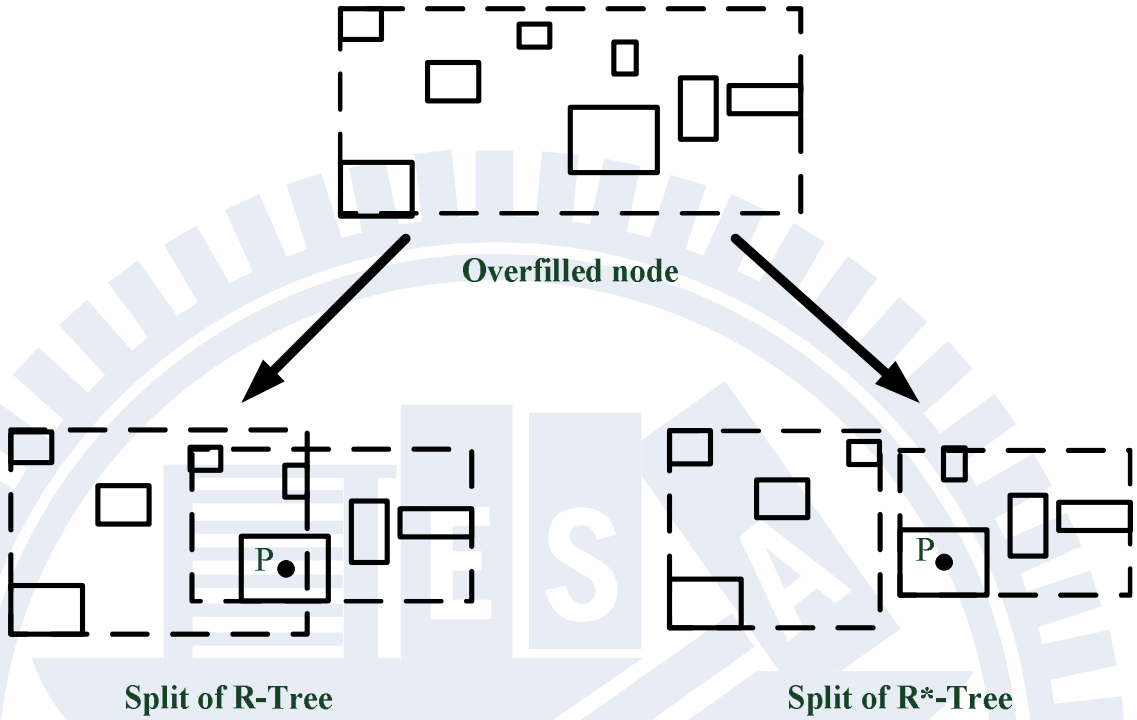

圖 5 R-Tree 以及 R*-Tree 在節點分裂上的不同

圖 5 中為 R-Tree 以及 R*-Tree 在分裂同一個 MBR 過多的節點後所得到的不同結果。左邊為 R-Tree 的結果，由 R-Tree 所分裂出來的兩個子節點會有互相重疊的情況發生，這會影響到在搜尋時的效能，假若搜尋的位置在這重疊的區域中，則這兩個子節點都需要被搜尋，這會花費比較多的時間。右邊為 R*-Tree 使用先前所提到了較好的分類演算法以及強制重新插入等機制分類節點後的結果，可以明顯的看出來，分裂出來的兩個節點並不會有重疊的情況發生。假設我們所要搜尋的位置為圖中 P 所標示的位置，R-Tree 在搜尋時就必須要搜尋分裂出來的這兩個 MBR，因為 P 所在的位置位於這兩個 MBR 重疊的區域中。而 R*-Tree 在搜尋時只需要搜尋右側的 MBR，因為 P 只與右側的 MBR 有重疊。這就是 R*-Tree 在搜尋效能會比 R-Tree 還要來的好的原因。

利用 R*-Tree 能夠解決圖形空間封包分類問題，把分類規則對應到 R*-Tree 上被索引的圖型物件，封包標頭資訊對應到搜尋的區塊，如此一來我們就可以使用 R*-Tree 這個演算法來解決此問題。當封包到達路由器時，利用 R* -Tree 的搜尋演算法，搜尋到的物件就是適合的規則，之後再從適合的規則中找出優先權最高的規則，就是符合的規則。R*-Tree 這個方法的優點在於只需要很小的儲存空間，但是它的比對速度就比較慢。

三、R*-Tree 與位元圖交集混合法

3.1 基本想法

在前面的章節有提到，位元圖交集法有著快速的封包比對速度，並且它可以使用硬體平行處理的特性加速比對的速度。然而，它在規則數量多的情況下會有記憶體需求過高的缺點，它的空間複雜度是 $O(dN^2)$，其中 d 代表維度的個數，N 代表規則的數目。也就是說，當規則數量成長時，所需要的儲存空間會以二次方的方式往上成長。除此之外，規則數變多也就代表位元向量會隨之變長，這會導致比對時所需要的記憶體讀寫次數增加，進而讓比對速度變慢。

另外一種 R*-Tree 的演算法，在空間利用度上有著不錯的效能。當規則數量成長時，所需要的空間只會以線性的方式成長。然而在比對的速度上這個方法就比較慢了，主要的原因是因為 R*-Tree 在節點中找尋重疊的 MBR 時是以線性搜尋的方式來搜尋。舉例來講，考慮一個樹高為 4，維度為 5 的 R*-Tree，其中每個節點包含的 MBR 數量為 10 個，假設在理想狀況下，搜尋的動作必須要經過四個節點，總共需要 200 次的記憶體讀寫才能得到結果，所以在比對上的速度相對就比較慢了。

為了解決上面所提到的問題，我們提出一個結合這兩種方法的演算法，希望可以利用 R*-Tree 在空間利用度的優點去克服位元圖交集法的問題，以及利用位元圖交集法比對速度上的優勢來加速 R*-Tree 演算法。

我們的方法是基於 R*-Tree 封包分類法，把位元圖交集法使用在各個節點上。我們觀察到，在圖形封包分類的問題當中，搜尋時必定是

搜尋一個點，利用此特性能夠將這兩種演算法加以結合。我們將 R*-Tree 中的每個節點想成一個獨立的圖形空間，裡面包含著對應此節點的子 MBR。將這些 MBR 當作規則。因為搜尋時必定搜尋一個點，我們就可以在節點上使用位元圖交集法，來替代原本 R*-Tree 演算法中搜尋重疊 MBR 的方法。此外，由於節點中所包含的 MBR 數量並不多，這表示位元圖交集法並不會使用到過多的記憶體空間，雖然整體 MBR 的數量會超過原本規則的數量，但是它們會被切分成較小的群體，每個群體使用極小的位元圖來代表，整體來看所需要的記憶體空間比起位元圖交集法來小得多。

3.2 實際作法

我們的方法基本上分成兩個部分，建立 R*-Tree 以及處理封包比對。第一部分主要是把所有的規則建立成一個 R*-Tree，而第二部分是利用這個建立好的 R*-Tree 來搜尋封包所對應的規則。

在建立R*-Tree這部分，我們會先把規則轉換成MBR的資料格式，其中包含規則在每個維度中對應到圖形空間的開始位置與結束位置以及規則的相關資訊(如優先權、比對後採取的動作)。之後利用原本 R*-Tree 的演算法把這些 MBR(即 規則)依序的插入到樹當中，直到規則被完全插入為止。接下來我們使用 Depth-First-Search 或是 Breadth-First-Search 的方式依序拜訪樹中的節點。每到一個節點時，就把 MBR 資訊轉換成位元圖交集法中所使用的位元向量，並且重新記錄到此節點之中。在拜訪完所有的節點後，建立 R*-Tree 的步驟就正式完成。圖 6 中顯示了我們方法對於圖 4 中的範例所建立的節點架構，我們使用位元圖交集法把圖 4 中四個 MBR 圖形資訊轉換成位元向量，並且把這些位元向量紀錄在節點 B 中。

在處理封包比對這部分，我們把封包標頭中用來比對的欄位資訊

取出，然後利用此資訊詢問我們所建立的 R*-Tree 來找出規則。一開始從樹的根節點開始，注意在這邊與 R*-Tree 方法不同的是，這邊是利用位元圖交集法來找出與封包所代表的詢問點重疊的 MBR。如圖 6，我們使用位元圖交集法快速地搜尋節點 B 中與詢問點重疊的 MBR，它的搜尋速度比起使用線性搜尋要快，所以我們能夠快速地得到結果。在結果位元向量中被設定為 1 的位元所對應的 MBR 就是我們要找的 MBR。找到所有重疊的 MBR 後，繼續拜訪這些 MBR 所對應的節點，重複以上動作直到拜訪到樹葉節點為止。在樹葉節點也是使用相同的方式找出重疊的 MBR，而這些 MBR 就會對應到適合的規則，其中優先權最高的就是最後的比對結果。

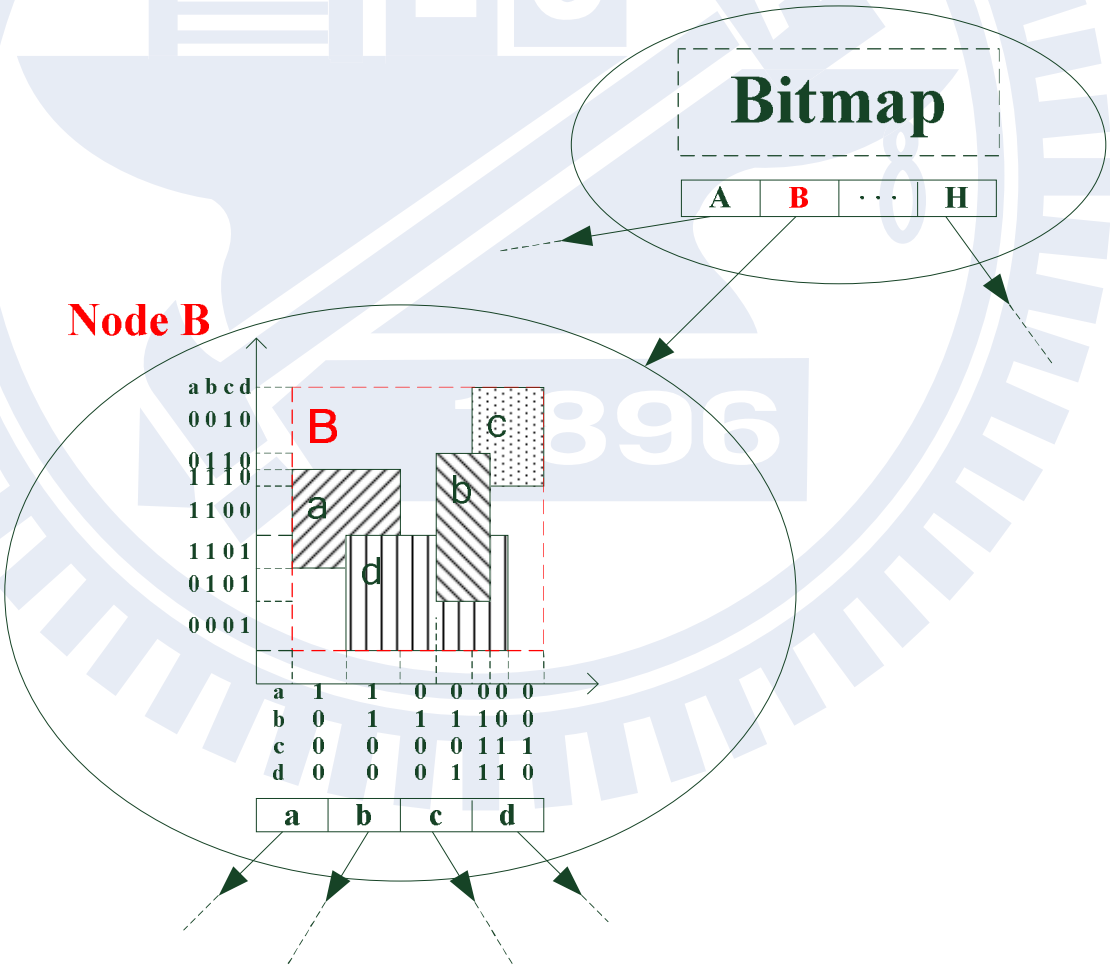

圖 6 節點架構

3.3 效能分析

這邊我們將分析並比較我們的方法、位元圖交集演算法、以及 R*-Tree 封包分類演算法，在時間平均複雜度與空間複雜度上的差異。

在比對速度方面，我們所提出的演算法封包比對速度的複雜度如下:

$$\theta(\log_M N \times d \times (\log_2 M + M / w)) \quad (1)$$

式子中的 d 表示維度的個數，N 表示規則的數目，w 表示記憶體通道的寬度，M 表示一個節點中所包含的 MBR 數量。做一次封包比對的過程中，必須要拜訪 $\log_M N$ 個節點(即 樹高)才能夠找到規則。並且每個節點中要花費 $\log_2 M$ 的時間找到封包對應的區間以及花費 M/w 的時間取出位元向量，這兩個動作總共要做 d 次。其中 M 可以視為常數。從式子(1)中可以了解到，M 越小搜尋時每個節點處理的時間會縮短，但是需要搜尋的節點數量會上升。然而，M 越大搜尋時需要搜尋的節點數量會下降，但是每個節點的處理時間則會上升，因此 M 對於搜尋的速度會有重大的影響。在第五章，我們將會利用模擬來探討 M 對於效能的影響。而下面是位元圖交集法比對的複雜度:

$$\theta(d \times (\log_2 N + N / w)) \quad (2)$$

在這個方法中，要針對每個維度作出總共 d 次搜尋位元向量的動作，每一次都需要 $\log_2 N$ 的時間找出區間的位置以及 N/W 的時間從記憶體中取出位元向量。當規則數目 N 數量大的時候，花在記憶體存取的時間會遠大於找出區間位置的時間。而 R*-Tree 的複雜度如下:

$$\theta(\log_M N \times d \times M \times (\log_2 S / w)) \quad (3)$$

此方法中，同樣也必須拜訪 $\log_M N$ 個節點，每個節點中，搜尋重疊 MBR 的動作總共需要 $d \times M \times (\log_2 S / w)$ 的時間，其中 S 為維度所在空間的範圍大小，以 IP 位置為例，S 所代表的數值為 2^{32}。可以看得出來我們的方法複雜度相對於另外兩個方法是比較好的。

在空間複雜度方面，我們提出的方法所的空間複雜度如下:

$$O(N \times d \times M) \quad (4)$$

我們的方法中樹中節點個數為 O(N/M)，每個節點內位元向量所需要的空間為 $d \times M^2$，空間複雜度為所有節點所需要記憶體空間的總和。而 R*-Tree 演算法的空間複雜度如下:

$$O(N \times d \times \log_2 S) \quad (5)$$

R*-Tree 中 O(N/M)也同樣表示節點數目，節點內紀錄 MBR 資訊則需要 $d \times M \times \log_2 S$ 的空間，S 代表每個維度中，所在空間範圍大小。而 $\log_2 S$ 則代表儲存每個 MBR 的空間資訊所需記憶體的大小，以 source/ destination 位址為例，總共需要 64 位元儲存 MBR 在空間中的開始以及結束位置。在此 $M \times log_2 S$ 也同樣能夠視為常數，所以這兩種方法的空間複雜度相當。位元圖交集法的空間複雜度如下:

$$O(d \times N^2) \quad (6)$$

相較於前兩種方法，此方法的空間複雜度比較差，當規則數量上升時，需要的記憶體空間會以二次方的方式成長。在規則數量大的時候，需要大量的記憶體空間，這會影響封包分類的效能。

四、強化 R*-Tree 與位元圖交集混合法

更進一步的分析我們所提出的方法，影響封包比對效能的關鍵因素就是在搜尋 R*-Tree 時所經過的路徑數量，當比對封包所經過的路徑數量上升時，其所花費的時間也會上升。這是因為拜訪的節點數量上升所導致的，當需要拜訪的節點數量越多，也就表示需要執行越多次的位元圖交集演算法，因此需要更多的時間。

封包比對時所產生的路徑大致上能夠分成兩個種類，必需要經過的路徑以及不需要經過的路徑。必須要經過的路徑指的是通往結果所必需要搜尋的路徑，在搜尋的過程中，為了能夠取得結果，這些路徑是必須要被經過的，所以花費在這些路徑上的時間是必須的。而不需要經過的路徑指的就是那些在搜尋中不會得到結果的路徑，雖然這些路徑最後並不會得到結果，但是在搜尋的過程中必須要經過這些路徑才能知道這個事實，所以這會產生額外的花費，造成不必要的浪費。

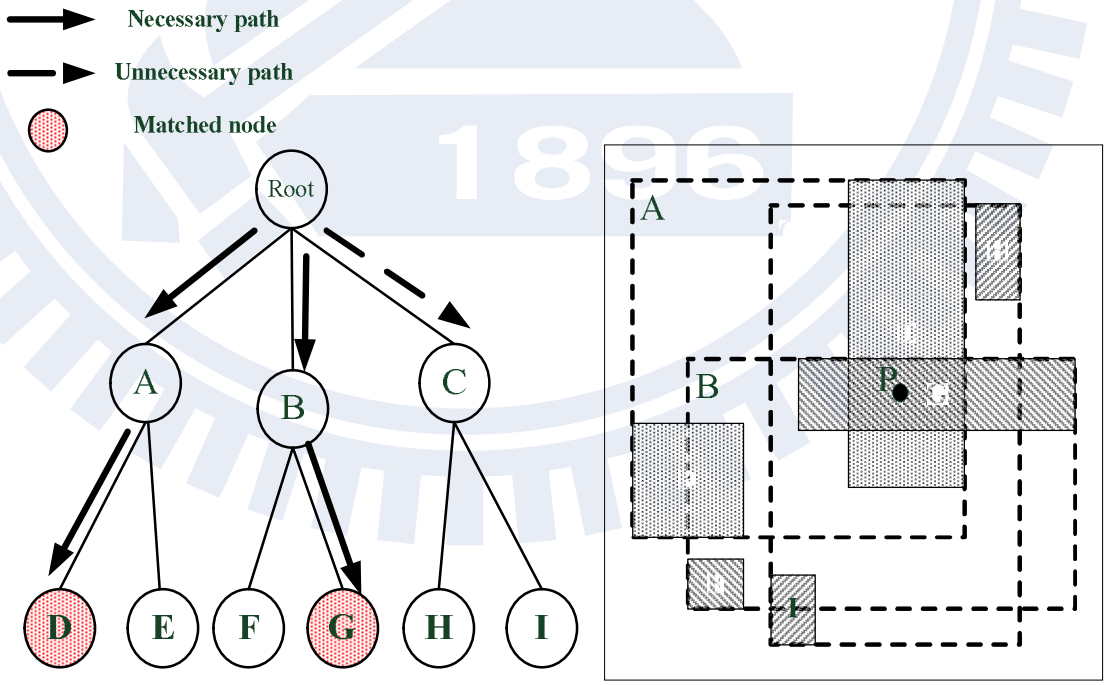

圖 7 必要路徑以及不必要的路徑

圖 7 為必要路徑以及不必要路徑的一個範例，圖中包含一個高度為 2 的 R*-Tree，並且顯示出封包 P 比對時搜尋所經過的路徑。可以看到，我們分別在節點 D 與 G 中找到了對應的規則。為了到達這兩個節點所必要經過的路徑就是必要路徑。另外我們可以看到另外有一條路徑經過節點 C，這條路徑是不必要路徑，因為在這條路徑中我們並沒有得到對應的規則，所以這條路徑其實並不需要被搜尋。

透過減少封包比對時所經過的路徑數量，能夠有效的減少比對時所花費的時間，在此針對兩種不同種類的路徑，分別提出了相對應的演算法來嘗試減少路徑的數量。針對不需要經過的路徑我們利用了布隆過濾器 (Bloom Filter)來試著減少不需要路徑的數量。此外我們利用多重 R*-Tree 的方式把必需要經過路徑分散到多顆 R*-Tree 當中，降低搜尋單顆 R*-Tree 所必需要經過的路徑數量。

4.1 布隆過濾器

首先，簡單介紹布隆過濾器，它是一個特殊的資料結構，透過它能夠檢察一個元素是否屬於一個集合之中，它有著非常好的空間利用度，以及所花費的查詢時間十分短的優點。一個布隆過濾器是由一個長度為 N 的位元向量與 M 個雜湊(hash)函式所構成。在布隆過濾器被使用之前，必須要先設定此布隆過濾器。所謂的設定就是把集合中所有的元素事先的用以下描述的方法餵到布隆過濾器中。對於每一個要被放入布隆過濾器的元素，它都會先行利用所包含的 M 個雜湊函式運算取得 m 個值域介於 0 與 N 之間的數值，然後把位元向量中位置對應這些數值的位元都設成 1。在設定完所有的元素後，設定就完成了，如此就能夠使用此布隆過濾器。布隆過濾器的檢查步驟如下，首先把要檢查的元素利用此 M 個雜湊函示運算取得 m 個值域介於 0 與 N 之間的數值，之後檢查位元向量中位置對應這些數值的位元，假若所有位元都

為 1 的話表示此元素屬於這個集合，假若有一個或一個以上為 0 的話表示不存在此集合中。

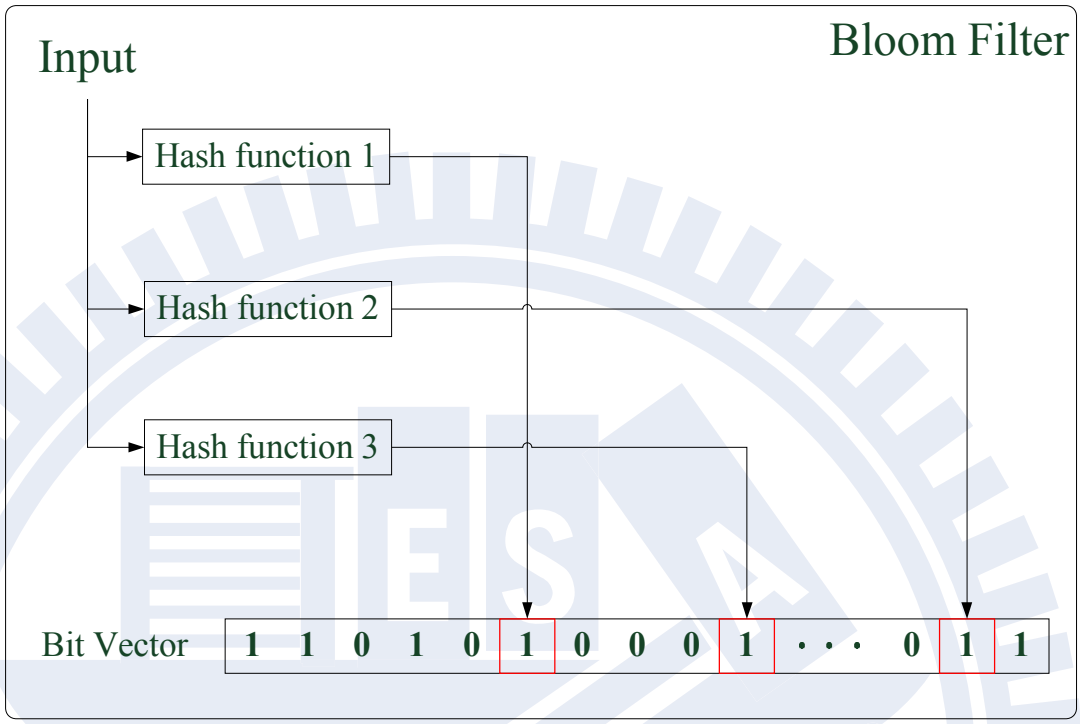

圖 8 Bloom filter with a match

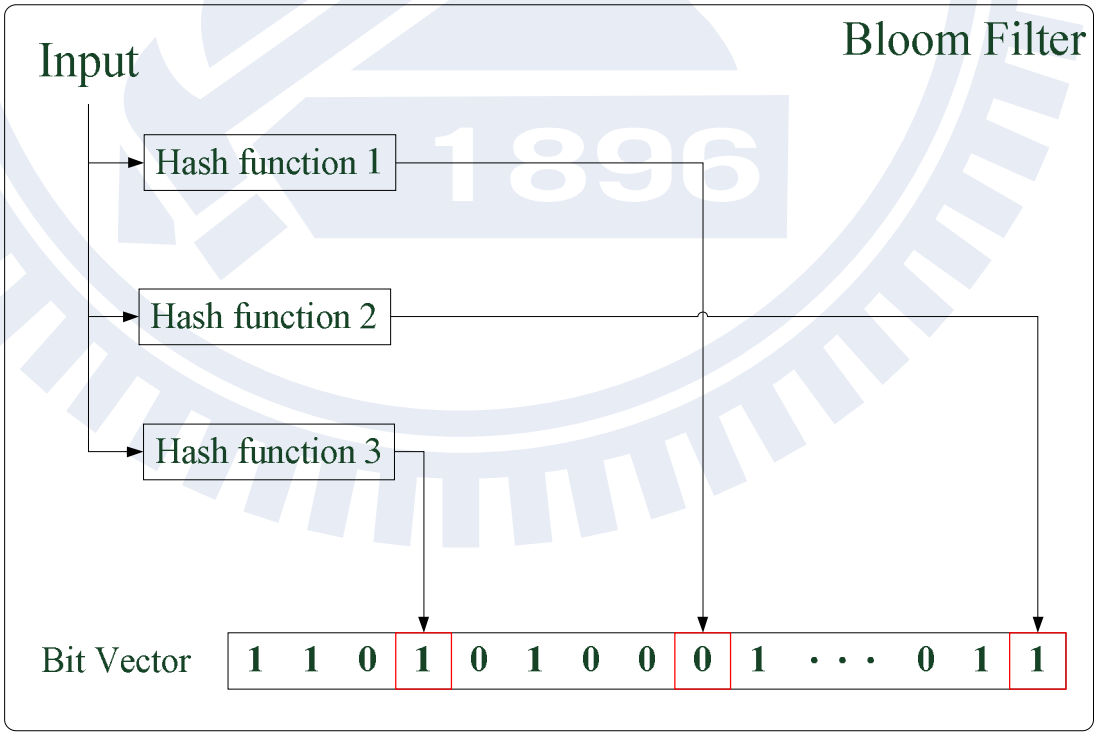

圖 9 Bloom filter without a match

圖 8 以及圖 9 分別代表布隆過濾器處理屬於其集合的元素以及不屬於其集合的元素的情況。圖中的布隆過濾器包含 3 個雜湊函式以及一個位元向量,每個輸入資料會先經過此 3 個函式取得 3 個不同的位置，每個位置對應到位元向量中的一個位元。接下來會檢查這些對應這些位置的位元，如圖 8 中，所有的位元都為 1 的話，就表示此輸入的資料屬於此集合。若如圖 9 中，其中有一個位元為 0 的話，就表示此輸入的資料不屬於此集合。布隆過濾器也有著一定的錯誤判別率，假設有一個不屬於集合中的元素，它經過雜湊函式所得到的位置都正好落在被設定為 1 的位元上時，此時布隆過濾器會把它判定為集合中的元素，但是實際上它並不是。這就是誤判發生的原因。但是只會把不屬於此集合的元素誤判為集合中的元素，原本屬於此集合中的元素是不會被誤判為不屬於此集合中的元素。

利用布隆過濾器我們能夠減少不需要的路徑數量,大致上的想法是利用布隆過濾器區分出目前搜尋的路徑是否會得到結果，也就是說在搜尋較高的階層時先行利用布隆過濾器把那些不需要的路徑區分出來，然後就能夠節省搜尋這條路徑所需要的時間。針對 R*-Tree 高度較高的節點，像是階層為一或二的節點使用布隆過濾器。在完成 R*-Tree 的建置之後，針對較高的節點，把屬於此節點下的所有規則用來設定針對此節點所產生的布隆過濾器，並且把完成設定的布隆過濾器儲存在此節點中。當搜尋的路徑經過此節點時，會先利用節點內的布隆過濾器檢察目前搜尋的封包是否屬於此節點所包含的規則的集合，如果答案是肯定的，就讓搜尋的動作繼續，若是否定的，就能夠事先中斷此搜尋路徑，避免掉不必要的浪費。

在實際的作法上,假若我們要對樹中的某一個節點加入布隆過濾器。首先我們會收集此節點之下所包含的所有規則，這些規則會位於此節

點之下的樹葉節點中。接下來使用這些規則來設定布隆過濾器。而一個節點會包含數個布隆過濾器，每個布隆過濾器會對應到規則中不同的欄位。當使用規則來設定節點中的布隆過濾器時，會依照規則欄位所對應到的布隆過濾器，依序的使用欄位中的值來設定這些布隆過濾器。當我們要使用節點中的布隆過濾器來判定封包時，首先把欄位的值從封包中取出，再依序的利用對應這些欄位的布隆過濾器檢查。若封包屬於節點中規則的集合，則所有的布隆過濾器都會判定封包屬於此欄位的集合。若有一個或以上的布隆過濾器得到否定的答案，則表示不屬於集合中。

但是這裡有一個問題必須要解決，由於在規則中的欄位是以 Prefix 的形式所儲存的，而送進來的封包資訊卻是以數值的形式存在，因此使用 Prefix 所訓練出來的布隆過濾器，是無法直接的用來處理數值型式的封包資訊。為了解決這個問題我們可以利用 Prefix expansion 把 Prefix 展開成數值的形式，然後利用這些展開的數值來訓練布隆過濾器，所產生的布隆過濾器就能夠用來區別封包是否屬於這些規則的集合。但是這會產生另外一個問題，考慮所要展開的欄位為一個 IP prefix，並且其 prefix length 為 2，完全展開此 prefix 總共會產生 2 的 30 次方，約莫 10 憶個數值，這會對布隆過濾器的造成巨大的影響。為了解決這個問題，我們把 Prefix 切成數個子集合，每個集合收集 prefix length 屬於同一個區間的 Prefix。假設切成四個集合，第一個集合包含 prefix length 為 0~8 的 Prefix，第二個集合包含 prefix length 為 9~16 的 Prefix，第三個集合包含 prefix length 為 17~24 的 Prefix，第四個集合包含 prefix length 為 25~32 的 Prefix。如此在展開 Prefix 時就不需要完全展開，只需要展開到集合中最長的 prefix length 即可，這可以有效的減少展開 prefix 的影響。而針對這四個集合分別使用不同的布隆過濾器來處理。

在檢察封包時，這四個布隆過濾器都必須要用來檢查封包中的同一個欄位，對於第一個布隆過濾器使用封包中網路位置的前 8 個位元來檢查，第二個使用前 16 位元，以此類推。當其中有一個過濾器表示出此網路位置屬於對應於它的集合，就表示此網路位置屬於規則中此欄位的集合。若沒有過濾器表示，則代表此網路位置不屬於集合中。下圖 10 為使用 Prefix expansion 的一個例子，可以看到總共有三個 Prefix，長度分別為6、7與14。在此我們把Prefix分成兩個子集合，0~8與9~16。接下來就如圖中所示，Prefix 長度為 6 與 7 的 extend 到 Prefix 長度 8，而 Prefix 長度 14 的 extend 到長度 16。在完成 Prefix expansion 後，不同的 Prefix 集合會設定對應此集合的布隆過濾器，Prefix 長度為 8 的集合使用 Prefix 前 8 個 bits 去設定，而長度為 16 的集合則是使用前 16 個 bits 作設定。

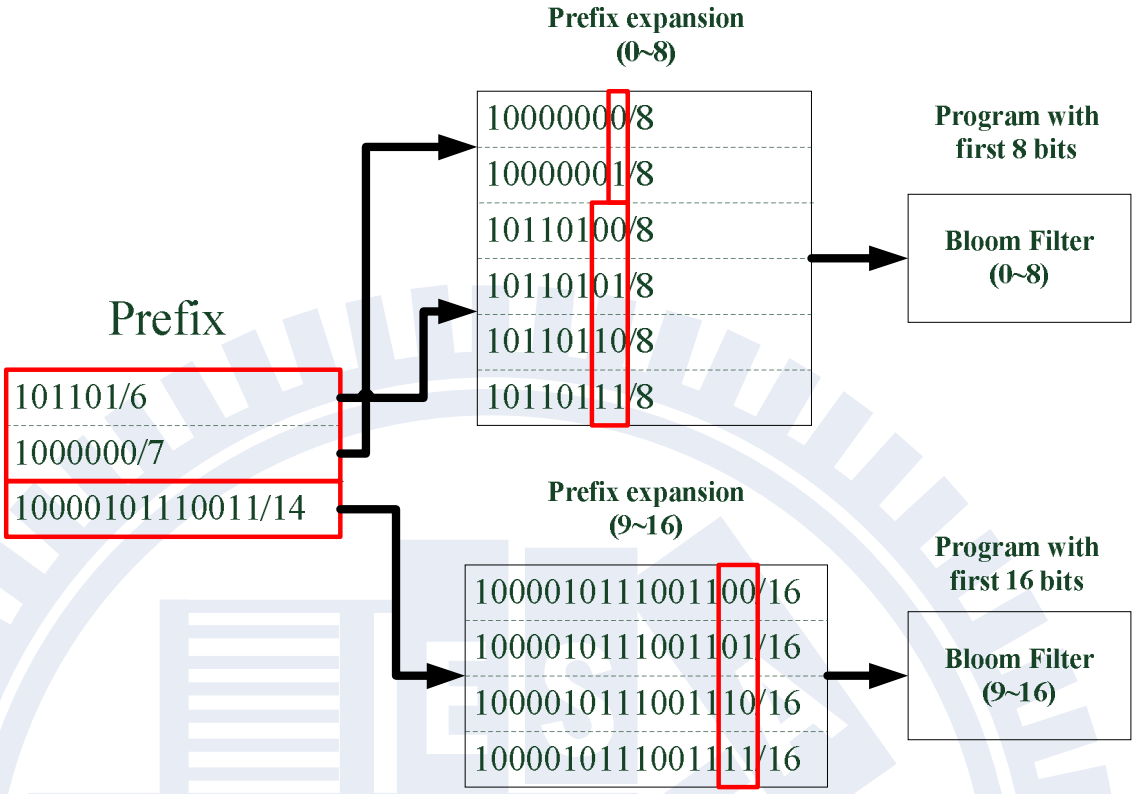

圖 10 Prefix expansion

但是使用布隆過濾器有以下缺點，首先是布隆過濾器需要執行多次的雜湊函式來檢查元素是否存在於集合當中，這些函式的運算也是需要花費時間，這將會是額外的花費，假使解省的路徑數無法換回這些額外的花費的話，會使得封包比對的時間上升，降低執行的效率。另外，布隆過濾器也是需要額外的記憶體空間紀錄位元向量，這會讓整體所使用的記憶體上升。

4.2 多重 R*-Tree

在此我們希望能夠降低必需要經過路徑之數量，首先我們觀察在執行封包分類時，那些必須要經過路徑之特性。我們發現這些路徑到最後都會走到的結果大致上能夠分為兩種。第一種是此封包最後會比對到的結果，也就是優先權高的比對結果，此種結果在空間中通常只覆蓋較小的區域。而路徑最後達到的另外一種結果是屬於百搭符號的規

則，這種結果的優先權較低，並且其規則所包含的區域也比較大。一般的情況下，比對所得到的結果中優先權高的結果數量會小於優先權較低的結果。換句話說，在比對時優先權較低的結果所經過的路徑佔較多的數量，但是比對出來的結果卻是優先權高的規則。這表示在搜尋時會花費較多的時間在那些最後不會比對到的規則上。另外觀察不同封包所得到的結果，會發現優先權高的結果通常會指向不同的規則，但是優先權較低的結果常會對到相同的百搭符號規則。這主要是因為百搭符號規則所覆蓋的範圍較大，所以在查詢時容易查詢到這些百搭符號規則。因此，我們認為可以利用這兩種特性來降低搜尋到優先權較低所搜尋的路徑數量，以降低封包比對時所需要的時間花費。

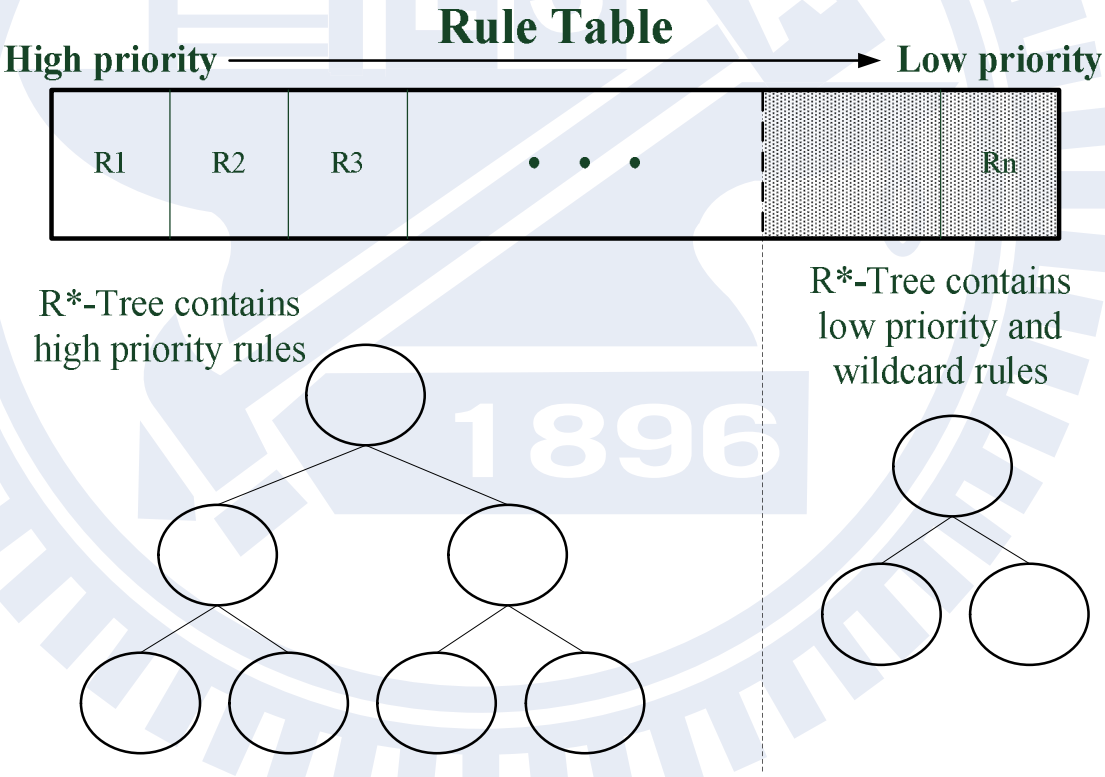

圖 11 依據規則表建立多個 R*-Tree

綜合以上兩個我們所觀察到的性質，我們提出以下的方法來增進比對時的效能。從這兩個性質可以看出，在執行封包比對的時候所搜尋的路徑大部分是為了規則表中所佔比例較少的百搭符號規則，而且這

些規則的優先權較低，所以通常不會是我們最後所得到的結果。所以我們的想法是透過把這些百搭符號規則從規則表中移出，另外再使用一個 R*-Tree 去處理這些規則。也就是說會產生兩顆 R*-Tree，第一顆樹負責普通的規則，第二顆樹負責百搭符號規則。

圖 11 中為使用多重 R*-Tree 的例子，在開始建立 R*-Tree 之前，我們依照規則表中的優先權把規則表中切成大小不同的兩塊。第一塊包含優先權較高的規則，其中包含的規則數量較多。第二塊包含優先權較低的規則以及百搭符號，其中包含的規則數量較少。其中第一塊所包含的規則的優先權完全高於第二塊中的規則。接下來這兩塊規則表各自使用 R*-Tree 與位元圖交集混合法的建樹方法，建立成兩顆 R*-Tree。透過上述的方法，建立出多重的 R*-Tree。

當我們把這些規則移出後，原本搜尋中為了這些規則所搜尋的路徑就會被分攤到兩顆樹當中，這可以讓搜尋第一個樹所花費的時間降低。另外考慮到百搭符號通常其優先權都比較低，所以在決定所要移出的規則的方法時，我們可以直接移出規則表中優先權低的規則。這會帶來兩個好處，第一，因為百搭符號通常都是優先權最低的那些規則，所以可以移出大部分的百搭符號規則，處理上也較容易。第二，透過這個方法，我們能夠確保所有在第一個樹中的規則之優先權皆高於第二顆樹中所包含的規則。這可以讓我們在比對封包時，不需要每次都搜尋兩顆樹。假若我們能夠在第一顆樹中得到結果，由於第一顆樹中所有的規則其優先全都高於第二顆樹中的規則，我們就不需要再去搜尋第二顆樹。

在此我們提出這兩種方法來嘗試增進我們方法在執行上的效能。這兩種不同的方法的執行效能會在第五章中用實驗的方式來驗證這兩者是否會對執行的效率有幫助。

五、實驗結果

本篇論文中我們使用 ClassBench [5]這個專門使用在封包分類問題上的工具來測試我們方法的效能。此工具包含規則產生器以及流量產生器兩部分，其中規則產生器可以產生出反映出真實情況的規則表，而流量產生器會產生用來測試分類方法的流量。另外 ClassBench 提供總共三種不同種類的設定檔，分別反映出不同性質的規則表，在產生規則時使用不同的設定檔，就能夠產生出對應其性質的規則表。而這三種不同性質的設定檔所代表的種類如下所示:

1. Access Control List (acl): 在此，acl 代表的是在路由器中所執行的 VPN 或是 NAT 所使用的規則表。
2. Firewall (fw): 針對防火牆應用的規則表。
3. IP Chain (ipc): 這種設定反映出在基作業系統上所執行的 VPN 或是 NAT 所使用的規則表。

我們我使用的平台規格如表 1 中所示，並且我們使用 C 來實作。

表 1 實驗平台規格

參數	數值
中央處理器	Intel C2D E8400
處理器時脈	3.00GHz
記憶體容量	4.00GB
作業系統	WindowXP

在這邊我們所用來測試的規則包含五個維度，分別對應到封包標頭中的五個欄位(即 Source/Destination address、 Source/ Destination Port、 Protocol)。在此章節中我們會在不同的參數設定下實驗三種不同的演算法，分別是位元圖交集法、R*-Tree 分類法與 R*-Tree 與位元圖混合法。首先，我們會實驗這三種不同的演算法在不同的性質的規

則表下的封包比對時需要的記憶體存取次數以及記憶體的需求。接下來是觀察節點中 MBR 的數量對比對速度上的影響。最後是實驗布隆過濾器以及多重 R*-Tree 這兩種方法對於效能上的影響。實驗數據都是處理十萬個封包後，其結果平均所得到的數值。

5.1 不同種類規則表 (acl、fw、ipc)

在此將會實驗位元圖交集法、R*-Tree 分類法以及 R*-Tree 與位元圖混合法在處理不同規則表與不同規則表大小時封包比對時記憶體存取次數，以及記憶體的需求。主要的目的是比較我們的方法與其他的方法在效能上的差異。規則表的種類為先前所提到的 acl、fw、和 ipc，針對每種規則表都會測試不同數量的規則對效能的影響為何。詳細的實驗參數如表 2 中所示

表 2 不同種類規則表實驗參數

參數	數值
規則數量	1k~10k
測試封包數量	100k
規則表種類	acl, fw, ipc
規則的維度	5
節點中 MBR 最大數量	64
節點中 MBR 最小數量	32
記憶體頻寬	32 bit

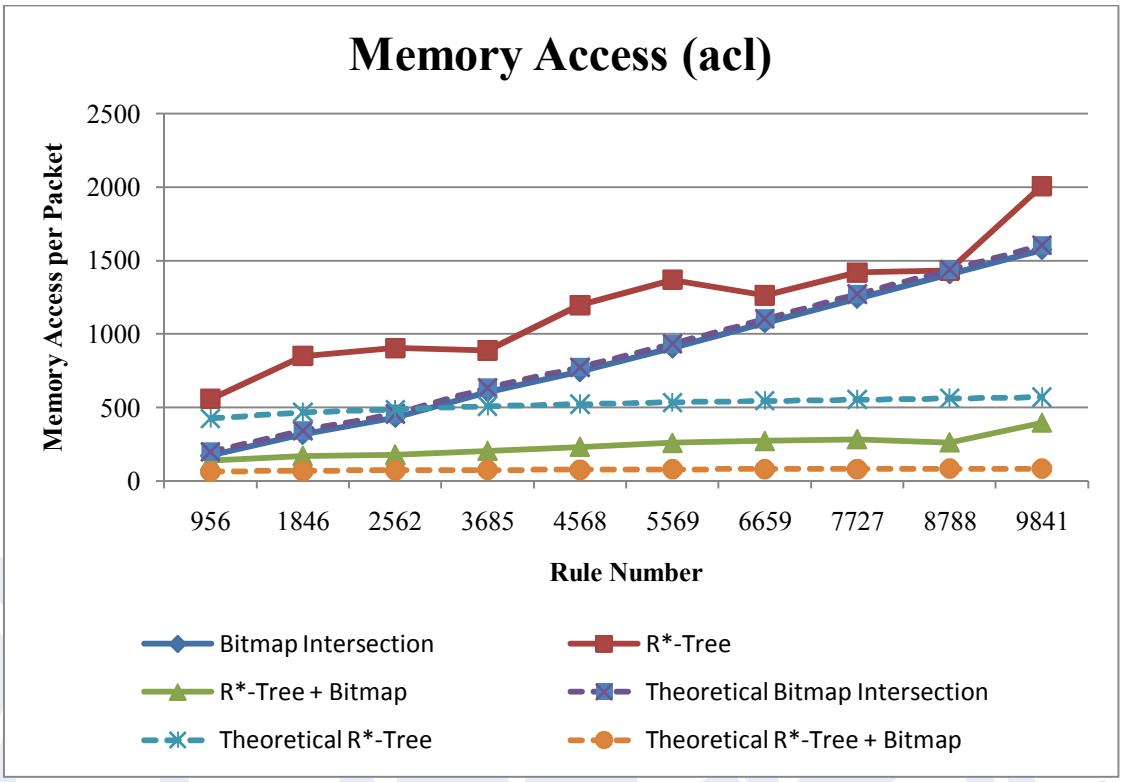

圖 12 Memory Access (acl rule table)

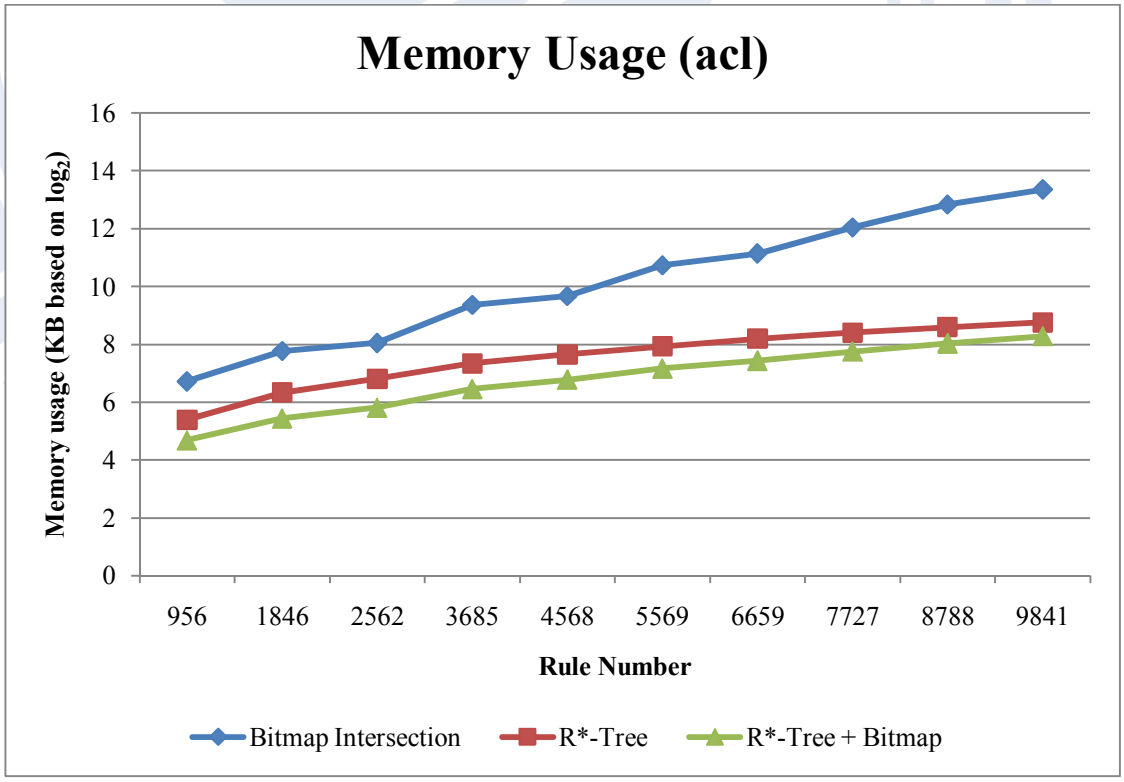

圖 13 Memory usage (acl rule table)

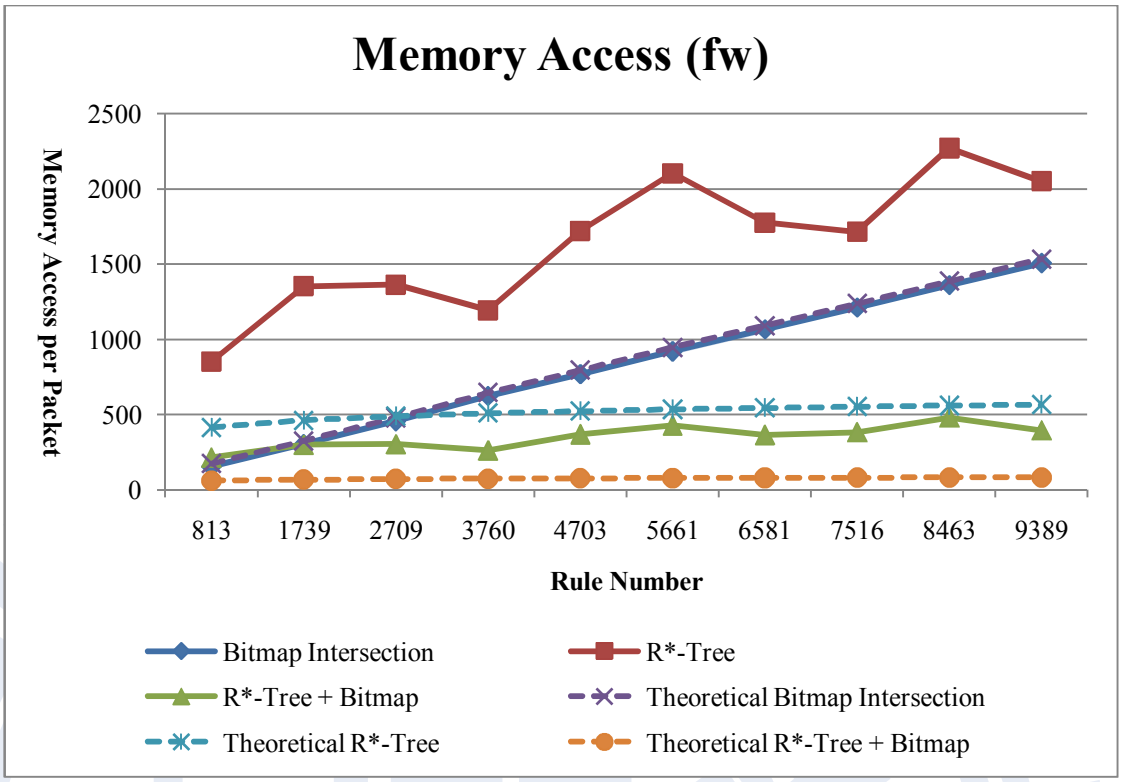

圖 14 Memory Access (fw rule table)

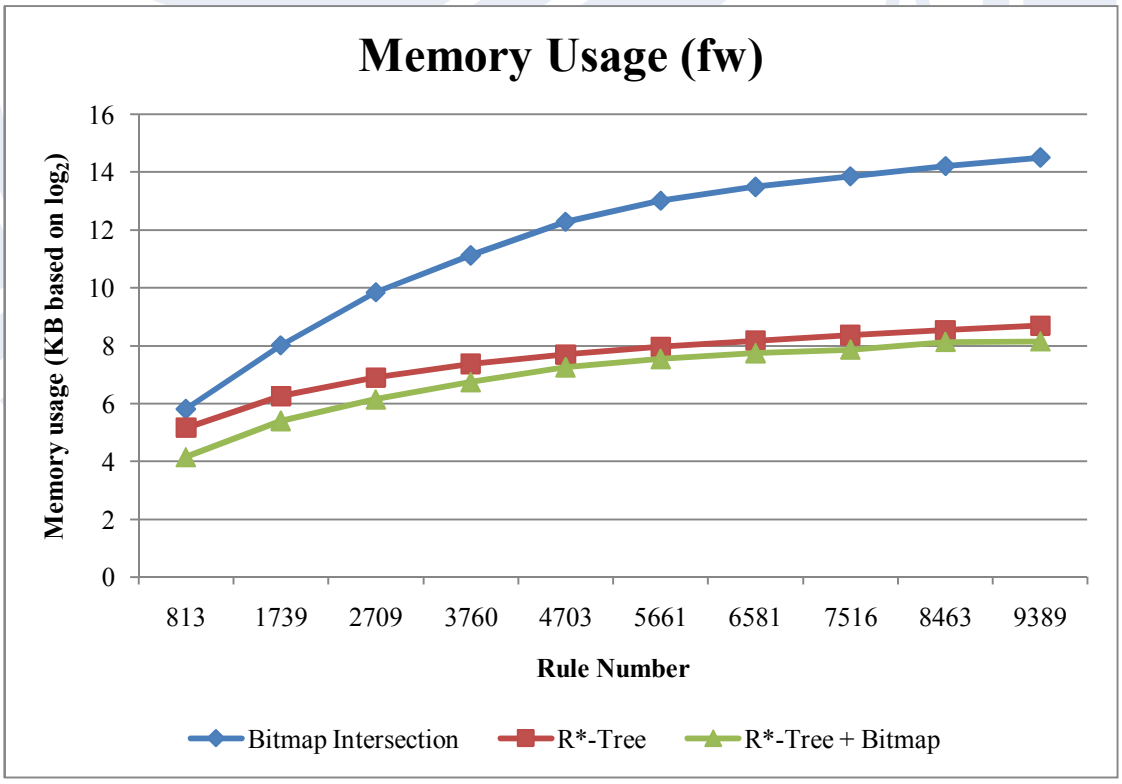

圖 15 Memory usage (fw rule table)

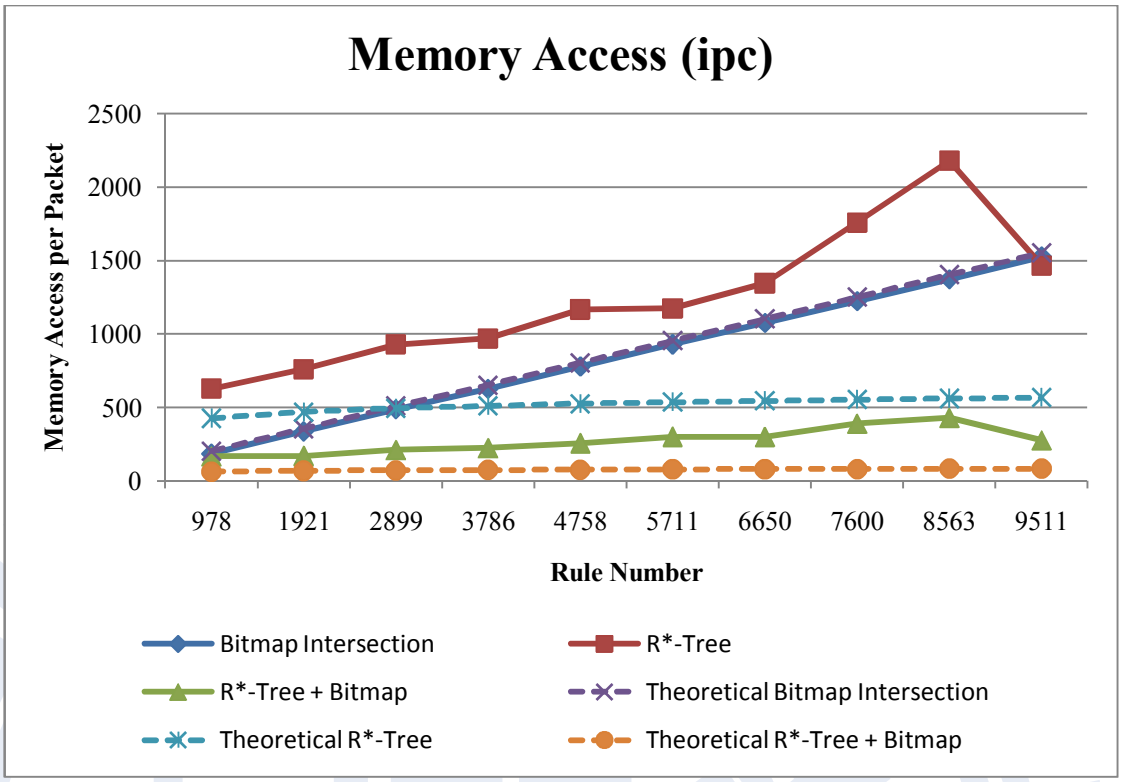

圖 16 Memory Access (ipc rule table)

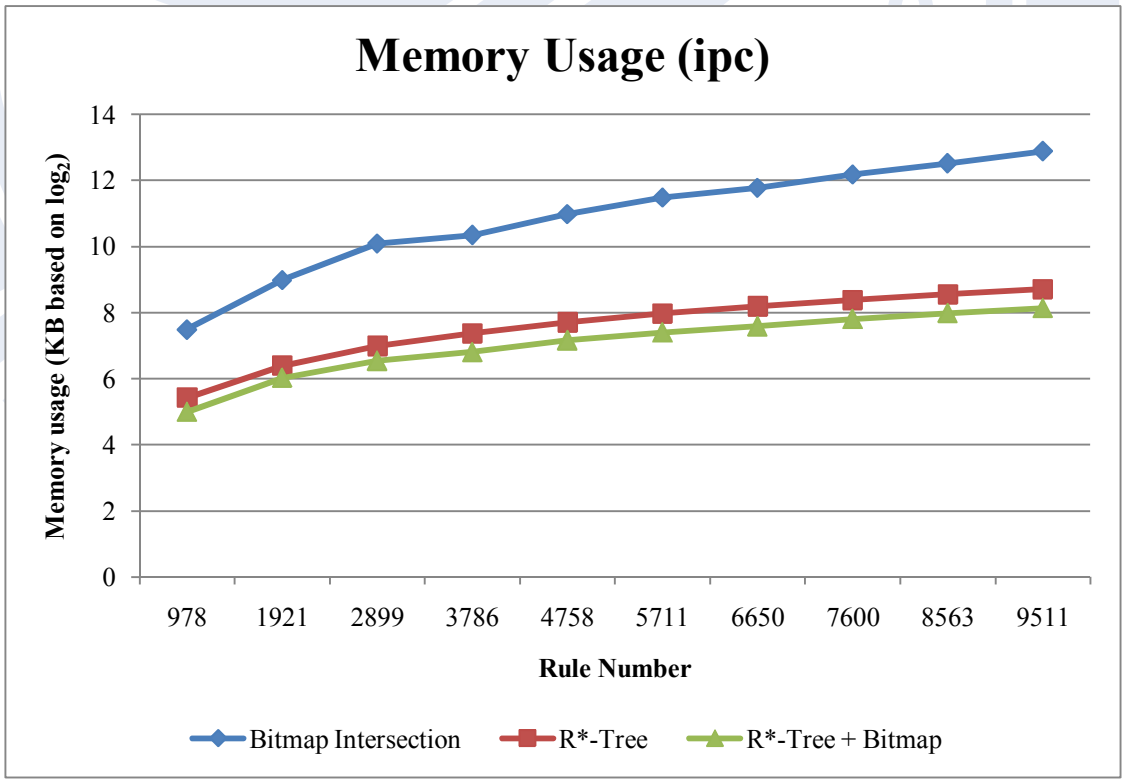

圖 17 Memory usage (ipc rule table)

圖 12、圖 14 以及圖 16 分別是 acl、fw 與 ipc 的實驗結果，包含不同演算法在不同大小規則表的記憶體存取次數，代表的是執行一個封包分類所需要存取記憶體的次數。首先我們觀察圖 12，acl 規則表下的實驗數據，在規則表數量較小的時候，我們所提出的方法在記憶體存取次數上皆優於其他兩種演算法，而位元圖交集法也有不錯的效能，其效能與我們的方法相當，而 R*-Tree 演算法比起其他兩者，在效能上就比較差了。當規則的數量成長時，可以看得出來位元圖交集法的效能開始下降，原因是因為位元向量變長，存取位元向量所需要的記憶體存取次數變多所導致。反觀我們所提出的方法，雖然記憶體存取次數有所上升，但是比起位元圖交集法，我們方法的上升的幅度比較小。在規則數為一萬筆時，我們的方法封包比對的效能大約是位元圖交集法的 3.95 倍，是 R*-Tree 的 5 倍。另外我們可以觀察到一個有趣的現象，就是在我們的方法以及 R*-Tree 演算法會有規則數上升可是記憶體存取次數反而下降的情形發生。這種情況的發生是因為在我們的方法中影響封包分類效能的因素並不只是規則的數目，而是被規則在圖形空間中分佈的情況所影響。若規則與規則間重疊的比率很高的話，就代表 R*-Tree 中 MBR 的重疊率也會上升，這會導致在執行封包比對時需要搜尋更多的路徑，進而使得封包比對需要存取記憶體的次數上升。一般來說，當一個空間中放入越多的圖形空間物件，物件與物件之間互相重疊的可能性也越高。但是考慮這些物件的分佈情況，例如比較下列兩種狀況，第一種為物件數量較小，但是它們大多都集中在一個較小的區域中。第二種情況是物件數目較多，單是它們分散在一個廣大的區域中。由於第一種情況下，物件彼此會互相重疊的可能性較高，在使用 R*-Tree 搜尋時需要花費更多的時間。反觀第二種情況，雖然物件數量較多，但是分散在廣大的空間中，彼此重疊的可能性較小，在

搜尋時所花費的時間也會較小。所以在這才會有這種情形發生。

接下來觀察圖 14，fw 規則表下的實驗數據，在此實驗數據下的結果與 acl 有些微的不同。在規則數量小的時候，可以觀察到是位元圖交集法效能最好，我們所提出的方法效能較差，可是差距十分的小。當規則數量增加，這裡也會呈現與 acl 規則表相同的現象，就是位元圖交集法的記憶體存取次數會隨著規則數量的上升而增加。同樣的我們的方法也是，但是幅度也比較小。當規則數為一萬筆時，我們所提出的方法效能是位元圖交集法的 3.8 倍，是 R*-Tree 的 5.18 倍。另外比較 acl 與 fw 的結果，可以發現到，我們的方法在 fw 規則表下效能較差，主要的原因為 fw 規則表的規則彼此的重疊比率較高，這會讓比對時搜尋較多的路徑，而降低效能。反觀位元圖交集法，在不同的規則表下所表現出的效能都差不多，是因為影響位元圖交集法的主要因素為規則的數量，而不是規則的分佈。在這裡 R*-Tree 也會發生規則數上升而記憶體存取次數卻下降的現象，其原因同為先前所解釋的。

觀察圖 16，ipc 規則表的實驗結果。在此所呈現的結果與 acl 所呈現的結果類似，在規則數量小時，同樣也是我們所提出的方法效能最好，而位元圖交集法效能稍差。並且當規則數量上升時也呈現相同的現象，隨著規則數量上升而效能下降，下降的幅度也是我們的方法最小。當規則數為一萬筆時，我們所提出的方法效能是位元圖交集法的 5.5 倍，是 R*-Tree 的 5.3 倍。綜合以上三種實驗數據，我們所提出的方法在規則數量小時的效能與位元圖交集法相當，但是隨著規則數量的上升，我們的方法所呈現的效能就比其他方法好，當規則數量到達一萬筆時，平均效能是位元圖交集法的 4.42 倍，是 R*-Tree 的 5.16 倍。

不同演算法在不同種類規則表的記憶體需求分別呈現在圖 13、圖 15 以及圖 17 中。由於結果差距比較大，所以這裡的結果取了一個 $\log_2$

讓實驗數據方便觀察。在記憶體的使用上，我們所提出的方法在不同規則表所呈現出來的結果都非常的相似，當然也是有些許的不同，但是大致上的趨勢是相同的。但是位元圖交集法所呈現的結果差異就很大，在規則數量為一萬筆的情況下 acl 規則表需要大約 10MB，fw 規則表需要 20MB，ipc 規則表需要 7MB。而我們的方法以及 R*-Tree 在不同規則表所使用的記憶體則差不多，我們的方法大約為 300KB，R*-Tree 為 420KB。從中可以很明顯的看得出來，我們的方法在記憶體的使用上比起其他方法有效率。實驗結果證明，位元圖交集法的記憶體需求會隨著規則數量的上升而成平方的成長，而我們的方法以及 R*-Tree 則是呈現線性的成長。

5.2 R*-Tree 節點數量對效能的影響

接下來的實驗主要目的是觀察 R*-Tree 中節點內所包含的 MBR 數量的多寡對封包比對速度的影響。我們將會實驗不同種類以及大小的規則表。透過觀察記憶體存取次數、樹的高度與搜尋時經過的節點數量來探討節點中 MBR 數量對於效能上的影響。其他詳細的實驗參數會列在表 3 中。

表 3 R*-Tree 節點數量實驗參數

參數	數值
規則數量	1k, 10k
測試封包數量	100k
規則表種類	acl, fw, ipc
規則的維度	5
節點中 MBR 最大數量	8, 16, 32, 64, 128, 256, 512
節點中 MBR 最小數量	4, 8, 16, 32, 64, 128, 256
記憶體頻寬	32 bit

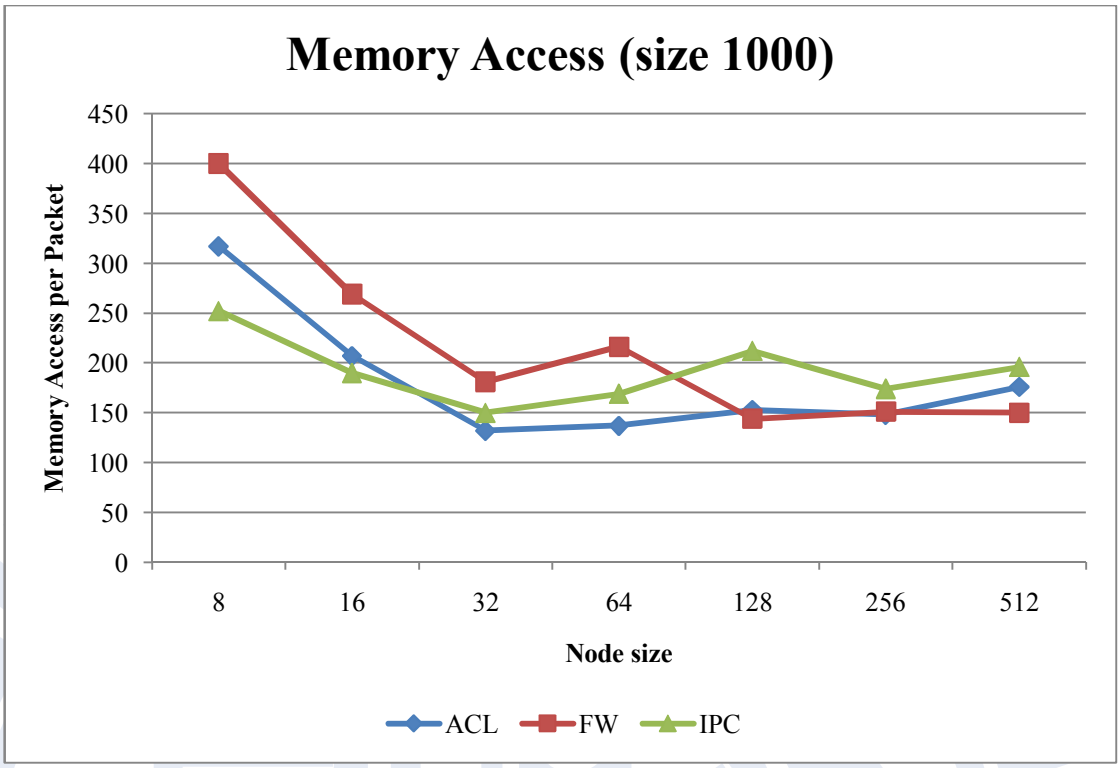

圖 18 Response time of different node size with rule size 1000

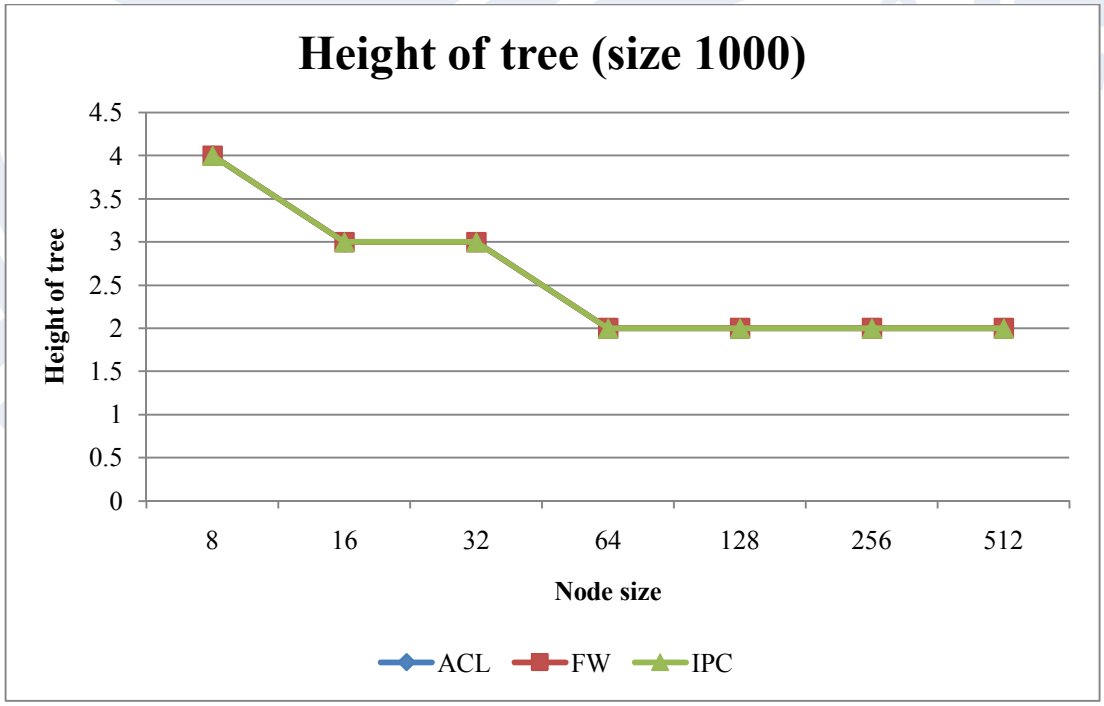

圖 19 Tree height of different node size with rule size 1000

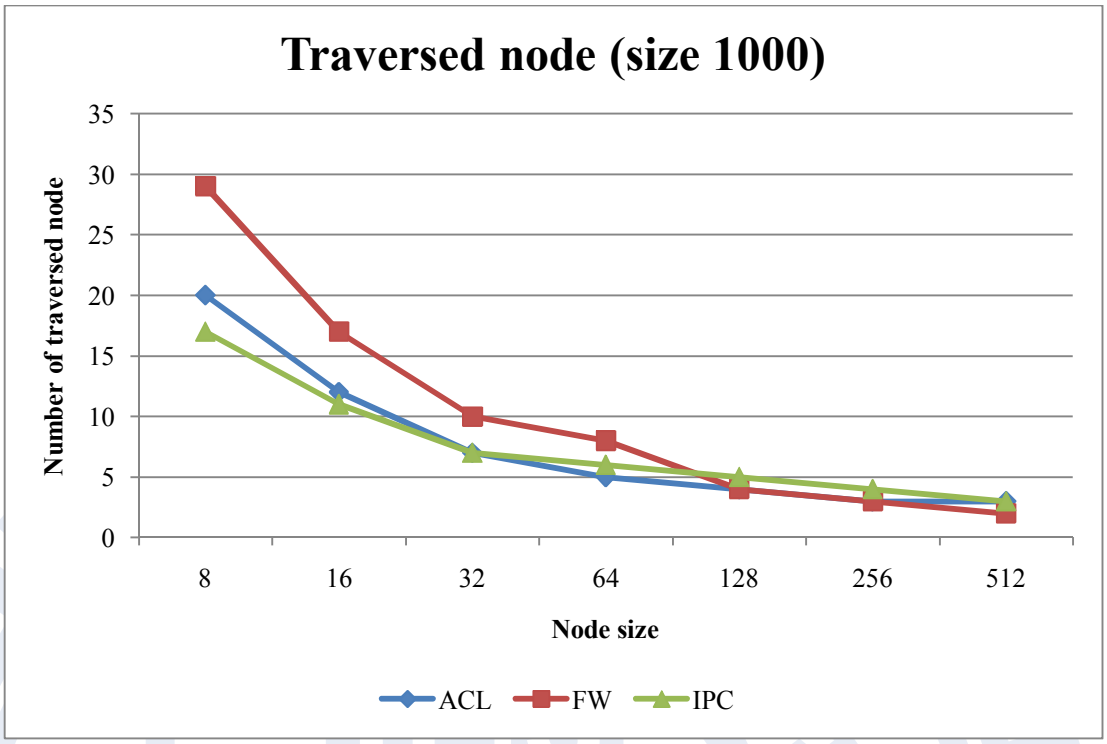

圖 20 Memory Access of different node size with rule size 1000

圖 18、圖 19 與圖 20 分別是在規則數量為 1000 筆時，不同節點大小對記憶體存取次數，樹高以及比對時所搜尋的節點數量的結果。從中可以發現，節點所包含的 MBR 數量對效能的影響十分的大，在 fw 的規則表下，效能的差距有 2.78 倍之多，其他的規則表也有 2 倍左右的差距。另外我們發現到 R*-Tree 的樹高直接影響到效能，樹高越高的 R*-Tree 在封包比對的時候需要花費較多的時間才能得到結果。主要的原因我們可以從搜尋經過的節點數得知，觀察 acl 的實驗數據，當樹高為 4 也就是節點大小為 8 時，每次的封包比對平均需要經過 29 個節點才能得到結果。在樹高為 3 也就是節點大小為 16 時，平均只需要經過 16 個節點就能得到結果。在搜尋節點所花費的記憶體存取相當的情況下，經過的節點數量越少，所需要的記憶體存取也就越少。接下來比較節點大小 32 與 64 的數據，節點大小 64 的樹高以及經過節點數量都比節點大小 32 的要小，但是他們的記憶體存取次數卻是節點大小 32

的比較好。這是因為處理單一個節點所需要的記憶體存取變多所導致，由於節點內 MBR 數量上升，搜尋時需要更多記憶體存取才能得到結果，因此才會發生樹高以及經過的節點變少，可是記憶體存取次數不降反升的情況。另外，我們觀察在相同樹高下不同節點大小對於效能的影響。觀察節點大小 64、128、256 與 512 這四組數據，它們的樹高皆為 2。隨著節點大小的上升，比對時所經過的節點數量會下降，但是記憶體存取次數卻會上升，是因為上面所提到的情況所導致。整體來說，在規則數量小情況下，節點大小為 32 時會有最好的效能。

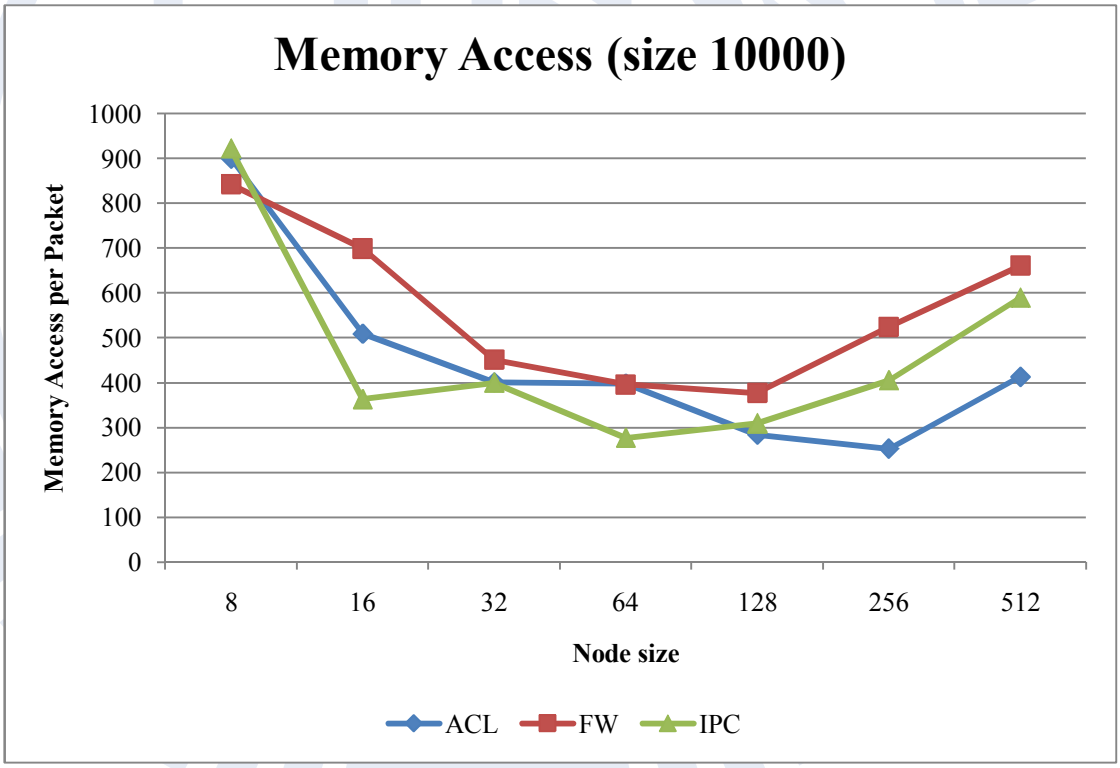

圖 21 Memory Access of different node size with rule size 10000

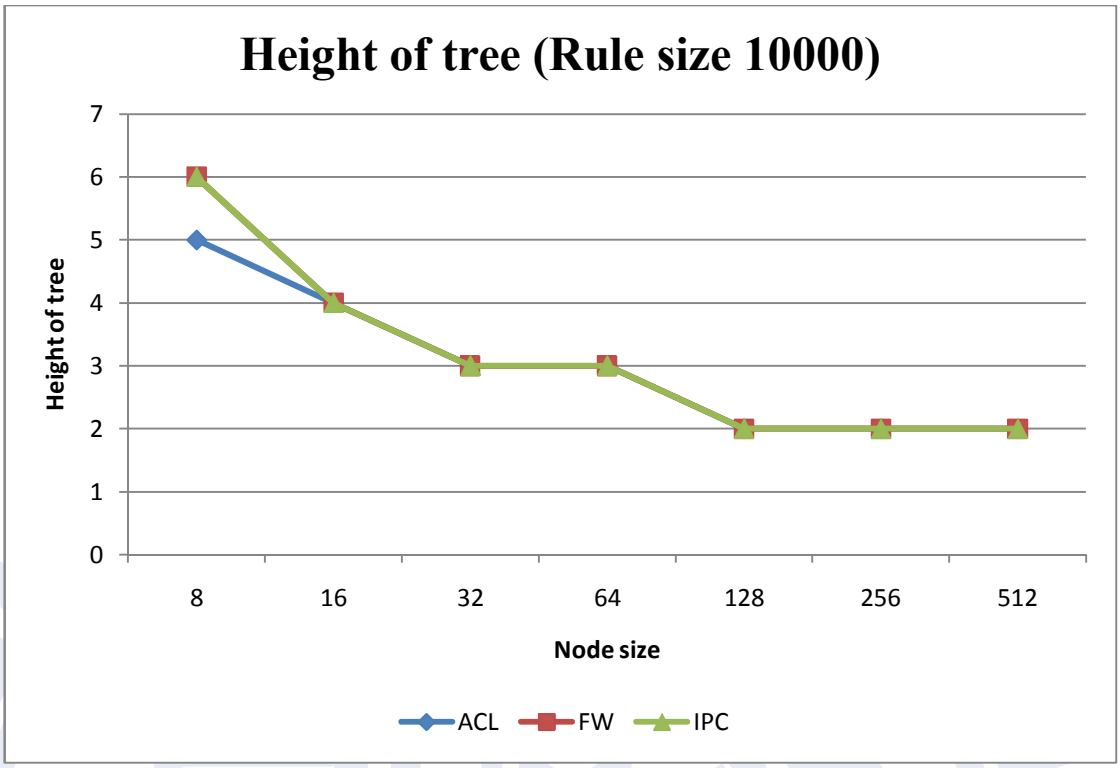

圖 22 Tree height of different node size with rule size 10000

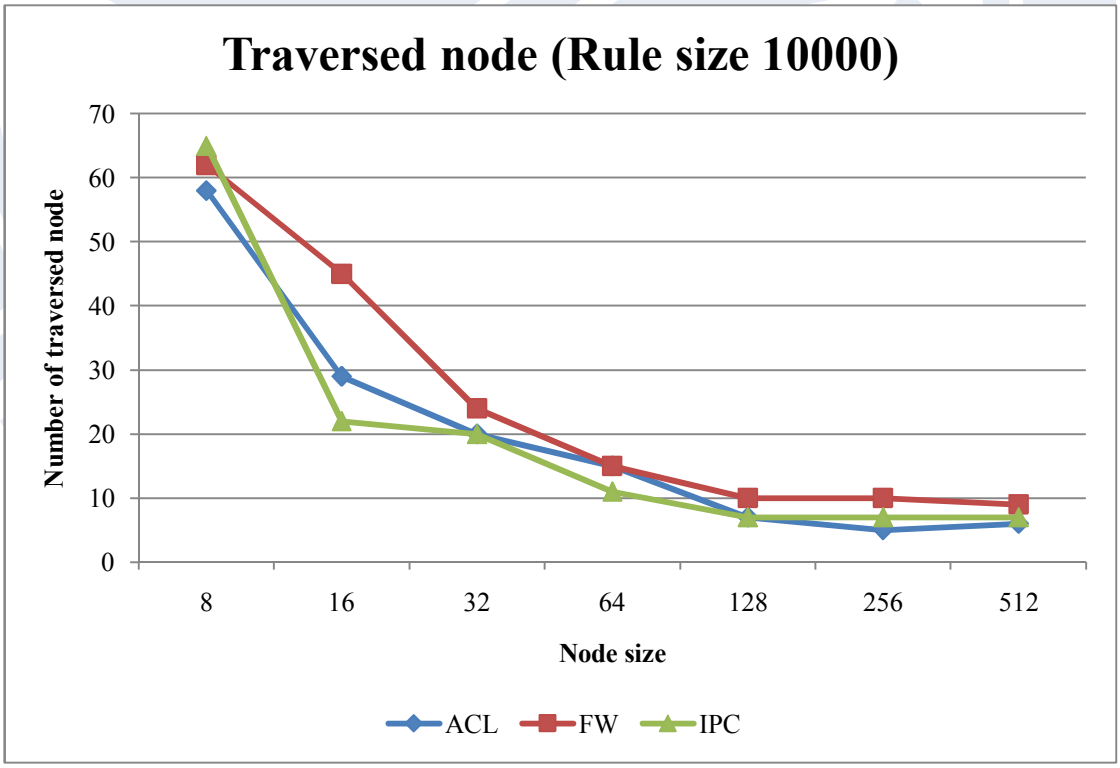

圖 23 Traversed node of different node size with rule size 10000

接下來我們實驗在規則數量較多時，節點大小對於我們所提出的方法的影響。圖 21、圖 22 以及圖 23 為規則數量一萬筆時的實驗結果。從結果我們可以知道，在規則數量較大的情況下，節點大小對於效能也是有非常大的影響。針對 acl 規則表討論，依據節點大小的不同其效能會差到 3.56 倍，其他種類的規則表也有三倍左右的差距。對於規則數量較大的情況，R*-Tree 的高度也同樣對效能有所影響，樹高越低，封包比對時所需要經過的節點數量也越少，封包比對的記憶體存取次數也越少。當然其中也有發生樹高下降而記憶體存取次數卻上升的情況，其主要原因也是因為搜尋單一節點所花費的記憶體存取變多所導致此情況的發生。接下來觀察固定樹高下不同節點大小對於效能的影響，在此我們同樣觀察節點大小 128、256 以及 512 這三組數據，它們的樹高皆為 2。在規則數為一萬筆時，當節點大小上升時，比對時所經過的節點數量同樣也是會下降，但是下降的幅度很小。因此在記憶體存取次數上都呈現變多的結果，是因為所需要搜尋的節點數量沒有變少，並且搜尋單一節點所需要的記憶體存取變多所導致。最後，從以上實驗的結果我們可以發現到，我們所提出的方法在節點大小 32 與 64 之間的時候會有最好的效能。

5.3 布隆過濾器與多重 R*-Tree 對效能的影響

最後我們將會比較我們所提出用以改善 R*-Tree 與位元圖混合法的兩種演算法在效能上是否有改善。這兩種方法分別利用布隆過濾器以及多重 R*-Tree 來改善效能。我們使用布隆過濾器來減少花費在不必要搜尋路徑上所花的時間，以及使用多重 R*-Tree 來嘗試降低搜尋必要路徑的數量。會分別實驗 acl、fw 以及 ipc 這三種不同種類的規則表，觀察封包比對的記憶體存取次數的差別。布隆過濾器的實驗中，我們將只會使用 Source / Destination IP address 這兩個欄位來設定布隆過濾器，

是因為其他欄位規則與規則之間的差異較小，使用布隆過濾器的效果有限，而 IP address 的差異較大，使用布隆過濾器的效果較好。此外使用越多的欄位代表須要加入更多的布隆過濾器，在搜尋時需要檢查的布隆過濾器也越多，這會成為額外的花費，所以在此我們只使用這兩個欄位。另外多重 R*-Tree 實驗中，我們設定移到第二顆樹的規則為原本規則表中後面 5%的規則。我們有實驗 5%、10%、15%以及 20%，這些設定的所呈現的結果是相當的，並且隨著比例的調高，其效能有下降的趨勢。原因為越多的規則被放到第二顆樹之後，就代表處理封包時更容易搜尋到第二顆樹，這會破壞多重 R*-Tree 中不需要搜尋兩棵樹的特性，所以效能未因此而下降，因此在這邊我們呈現在移出 5%規則到第二顆樹後對於效能上的影響。其他詳細的實驗數據將會列在表 4 中。

表 4 布隆過濾器以及多重 R*-Tree 實驗參數

參數	數值
規則數量	1k ~ 10k
測試封包數量	100k
規則表種類	acl, fw, ipc
規則的維度	5
節點中 MBR 最大數量	64
節點中 MBR 最小數量	32
布隆過濾器誤判機率	0.001
布隆過濾器判定欄位	Source IP , destination IP
Prefix 集合數量	4 (0~8, 9~16, 17~24, 25~32)
多重 R*-Tree 中規則移到第二顆樹的比例	5%
記憶體頻寬	32 bit

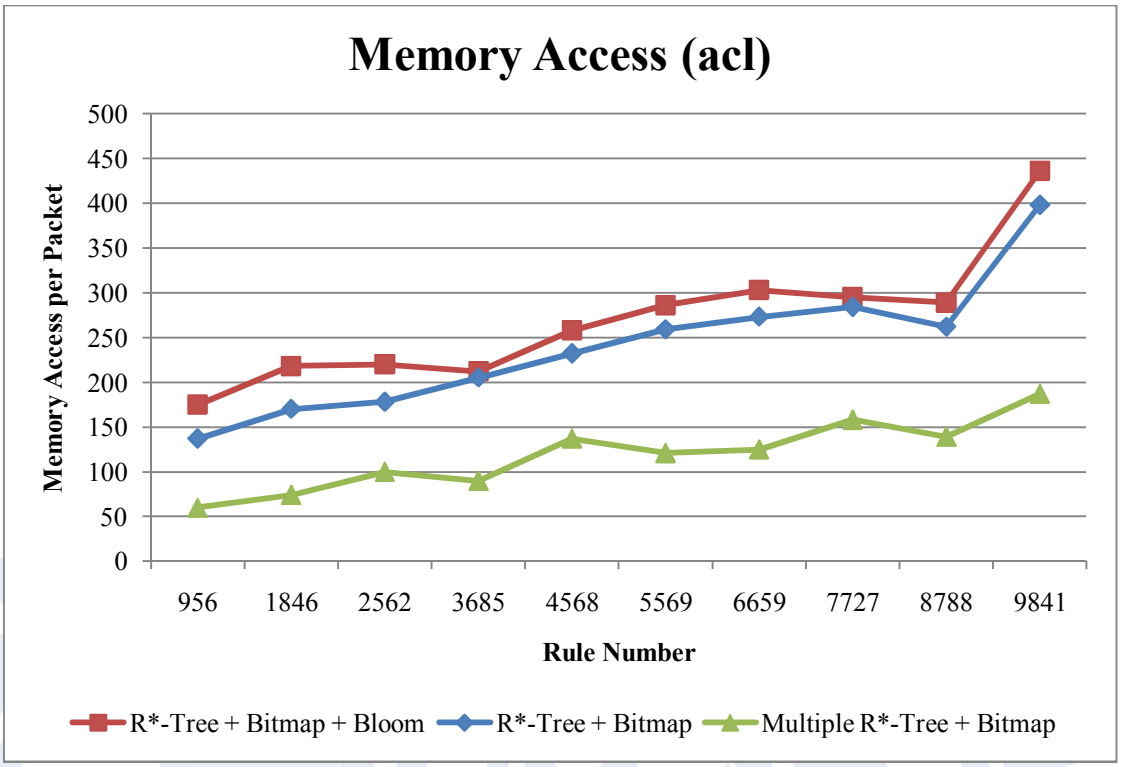

圖 24 Memory Access (acl rule table)

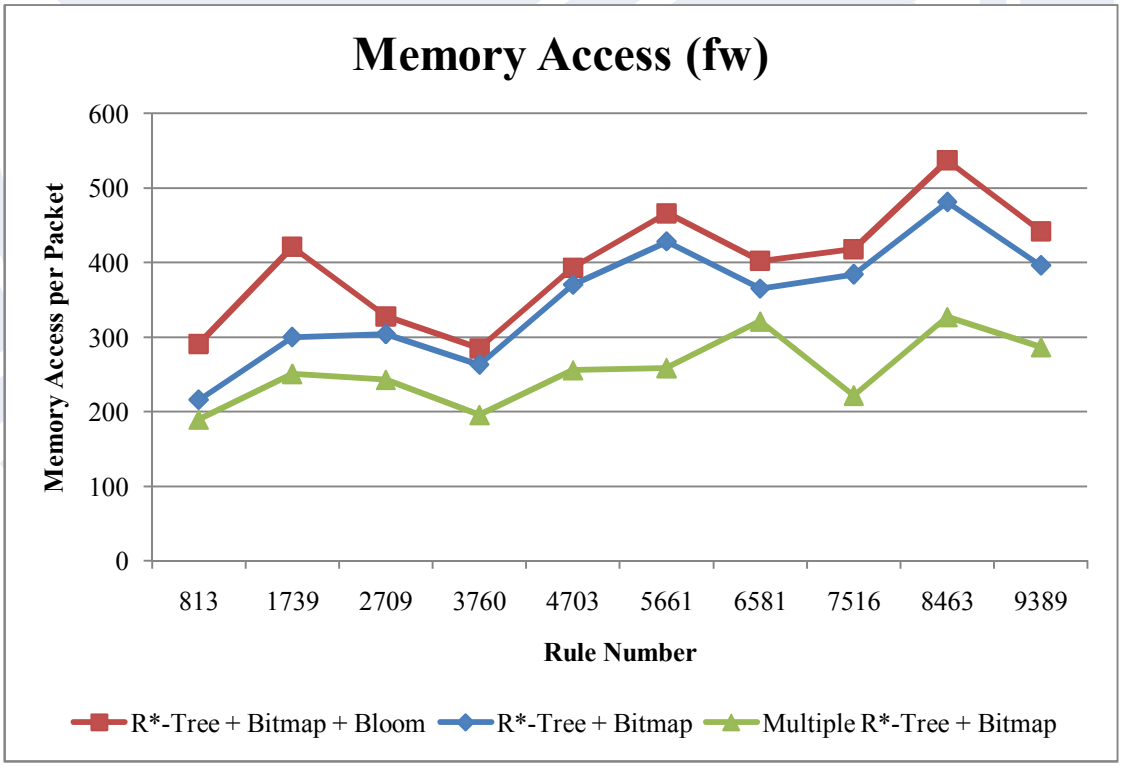

圖 25 Memory Access (fw rule table)

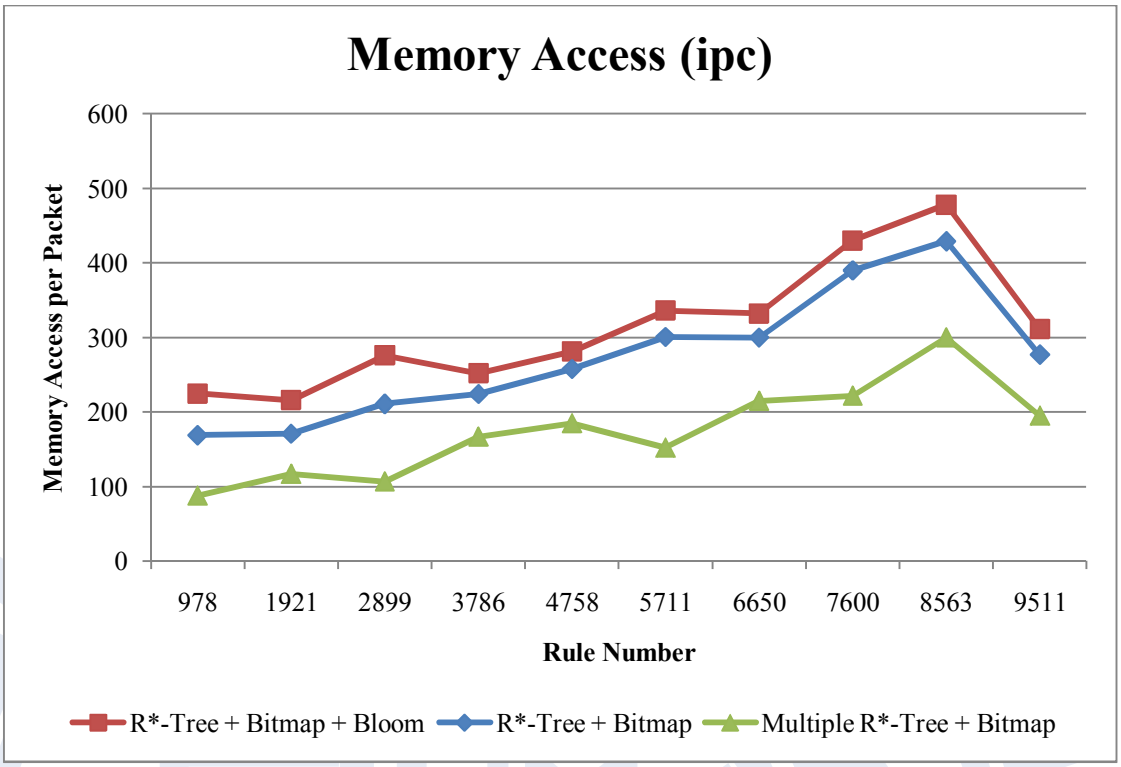

圖 26 Memory Access (ipc rule table)

針對 acl、fw 與 ipc 的實驗結果分別被列在圖 24、圖 25 以及圖 26 當中。不同規則表所展現出的實驗結果都有相同的趨勢，使用布隆過濾器的效能比較差，使用多重 R*-Tree 的效能比較好。首先我們探討加入布隆過濾器為何會讓記憶體存取次數下降。我們發現主要原因是因為在封包比對所搜尋的路徑中，不必要的路徑所佔的比例十分的少，布隆過濾器確實能夠讓搜尋時避免掉這些路徑，但是布隆過濾器的運算花費卻超過它所省下的那些花費，所以整體的記憶體存取次數才會因而上升。接下來我們觀察多重 R*-Tree 的實驗數據，可以發現到加入多重 R* -Tree 後效能比原本好上很多，平均起來大約有 30%左右增進。這表示多重 R*-Tree 的方法能夠讓搜尋時所經過的必要路徑數目分攤到不同的 R*-Tree 中，大部分的封包都能夠在第一顆樹就找到對應的規則，不需要再去搜尋第二顆樹，因此搜尋時所經過的路徑就能夠減少，這會使搜尋時所花費的記憶體存取有效的減少。

綜合以上實驗結果，在封包處理速度上，我們所提出的方法在規則表小的時候有著與位元圖交集法有著相當的效能。在處理規則數量較大的規則表時，我們所提出的方法效能會比其他的演算法更好。此外，我們所提出的方法所需要的記憶體空間是遠勝於位元圖交集法的，所以在封包處理的過程中，我們所提出的方法需要存取記憶體的次數比位元圖交集法要來的少。整體來說我們的演算法無論是在比對速度或是記憶體的使用上都優於 R*-Tree 演算法以及位元圖交集法。另外我們所提出用來改善我們方法的演算法中，從結果來看多重 R*-Tree 確實是能夠讓比對的效能更加上升。

六、結論與未來展望

本篇論文中，我們提出了一個能夠同時解決位元圖交集法空間複雜度太高與 R*-Tree 封包分類法比對速度慢這兩個問題的演算法。主要的方法是透過結合以上兩種演算法，利用彼此的優點來克服缺點。我們也分析並且比較 R*-Tree、位元圖交集法與我們所提出的方法在空間複雜度以及時間複雜度上的理論值。此外我們還另外提出兩種方法來更加強化我們所提出的演算法。第一種方法利用布隆過濾器嘗試減少封包比對時所搜尋的不要要路徑的數量。而第二種方法，我們使用多重 R*-Tree 來分散必要的路徑到多顆 R*-Tree 之上。在論文的最後，我們利用 ClassBench 這套能反映現實情況的工具來實驗我們的方法在不同特性的規則表下的效能。實驗結果指出，我們的方法在比對效能上平均是位元圖交集法的 4.42 倍，並且是 R*-Tree 演算法的 5.16 倍。在記憶體的使用上，我們的方法在處理一萬筆規則時大約只需要 300KB 的記憶體，R*-Tree 需要約 420KB，然而位元圖交集法則需要 10MB 左右的空間。我們的演算法的比對效能以及記憶體都比其他兩個演算法還要好，並且比對過程中的記憶體存取次數也比位元圖交集法要來的少。我們所提出用來改善我們演算法的兩種方法，其中布隆過濾器並不能改善封包比對的速度，但是多重 R*-Tree 卻是能夠讓比對的速度更快，大約有 30%的增進。接下來的研究方法，是想透過硬體或是軟體的平行處理來加速我們的方法。我們可以利用硬體平行處理的性質來加速節點內搜尋重疊 MBR 的速度，以及使用軟體的多重執行緒，同時的去搜尋 R*-Tree 中的多個節點，這些方法都夠提升我們的方法的效能。另外會繼續實驗在處理規則數量大並且規則彼此重疊很嚴重的狀況時，布隆過濾器是否能讓封包分類的效能上升。

參考文獻

[1] A. Cuttman, “R-Tree: A Dynamic Index Structure for Spatial Searching,” in proceedings of SIGMOD, pages 47-57, 1984.

[2] B. Seeger, H.-P. Kriegel, N. Beckmann, and R. Schneider, “The R*-tree: An Efficient and Robust Access Method for Points and Rectangles,” in proceedings of SIGMOD, pages 322-331, 1990.

[3] C. Maindorfer and T. Ottmann, “Is the Popular R*-Tree Suited for Packet Classification?,” in proceedings of IEEE NCA, pages 168-176, 2008.

[4] D. Taylor, “Survey & Taxonomy of Packet Classification Techniques,” *ACM Comput. Surv.*, vol. 37, no. 3, pages 238–275, 2005.

[5] D.E. Taylor and J.S. Turner, “ClassBench: A Packet Classification Benchmark,” in proceedings of IEEE INFOCOM, 2005.

[6] G. Varghese, “Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices,” Morgan Kaufmann publishers, 2005.

[7] H. J. Chao and B. Liu, “High Performance Switches and Routers,” Wiley-IEEE Press, 2007.

[8] M.H. Overmars and A.F. van der Stappen, “Range Searching and Point Location Among Fat Objects,” *Journal of Algorithms*, vol. 21, no. 3, pages 629-656, November 1996.

[9] P. Gupta and N. McKeown, “Algorithms for Packet Classification,” *IEEE Network*, vol. 15, no. 2, pages 24-32, March/April 2001.

[10] P. Gupta and N. McKeown, “Packet Classification using Hierarchical Intelligent Cuttings,” in proceedings of *IEEE Micro*, vol.20, no.1, pages 34-41, January /February 2000.

[11] S. Singh, F. Baboescu, G. Varghese, and J. Wang, “Packet

Classification using Multidimensional Cutting," in proceedings of SIGCOMM, pages 213-224, 2003.

[12] T.V. Lakshman and D. Stiliadis, "High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching," in proceedings of SIGCOMM, pages 191-202, 1998.

[13] C.R. Hsu, C Chen and C.Y. Lin, "Fast Packet Classification using Bit Compression," in proceedings of IEEE Global Telecommunication Conference, 2005.

[14] P. Cupta and N. McKeown, "Packet Classification on Multiple Fields," in proceedings of *Sigcomm, Computer Communication Review*, pages 147-160, September 1999.

[15] F. Baboescu, S. Singh and G. Varghese, "Packet Classification for Core Routers: is there an alternative to CAMs," in proceedings of *Infocom*, vol. 1 pages 53-63, March 2003.

[16] W.T. Chen, S.B. Shin and J.L. Chiang, "A Two-stage Packet Classification Algorithm," in proceedings of *Conference on AINA*, page 762-767, March, 2003.

[17] X. Sun and Y.Q. Zhao, "Packet Classification using Independent Sets," in proceedings of *ISCC*, vol. 1, page 83-90, June/July 2003.

[18] M. Waldvogel, G. Varghesez, J. Turnerz and B. Platnery, "Scalable High Speed IP Routing Lookups," in proceedings of *ACM Sigcomm*, vol.27, issue 4, page 25-36, October 1997.

[19] K. Lakshminarayana, A. Rangarajan and S. Venkatachary, "Algorithms for Advanced Packet Classification with Ternary CAMs," in proceedings of SIGCOMM, page 22-26, August, 2005.