อัลกอริทึมการจัดประเภทแพ็กเก็ตตามกฎ ด้วยวิธีบิตแมปอินเตอร์เซคชันลุกอัพโดยใช้แคช

A PACKET CLASSIFICATION'S ALGORITHM BITMAP INTERSECTION LOOKUP USING CACHE

ฐาปนา ชูรัตน์

THAPANA CHOORAT

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต

สาขาวิชาเทคโนโลยีสารสนเทศ

คณะเทคโนโลยีสารสนเทศ

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

พ.ศ.2553

KMITL-2010-IT-M-001-006

A PACKET CLASSIFICATION'S ALGORITHM
BITMAP INTERSECTION LOOKUP USING CACHE

THAPANA CHOORAT

A THESIS SUBMITTED IN PATIAL FULFILLMENT

OF THE REQUIREMENT FOR THE DEGREE OF

MASTER OF SCIENCE IN INFORMATION TECHNOLOGY

FACULTY OF INFORMATION TECNOLOGY

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

2010

KMITL-2010-IT-M-001-006

กิตติกรรมประกาศ

วิทยานิพนธ์ฉบับนี้สำเร็จได้อย่างดี ด้วยคำแนะนำ และคำปรึกษาจาก ผศ. อัครินทร์ คุณกิตติ ซึ่งเป็นอาจารย์ผู้ควบคุมวิทยานิพนธ์ ข้าพเจ้ารู้สึกทราบซึ้งในความอนุเคราะห์จากท่านอาจารย์ และขอขอบพระคุณเป็นอย่างสูง

ขอกราบพระคุณคณาจารย์คณะเทคโนโลยีสารสนเทศ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง ทุก ๆ ท่านที่ได้ประสิทธิ์ประสาทวิชาให้กับข้าพเจ้า

ขอขอบคุณ พี่ๆ เพื่อนๆ น้องๆ ในสาขาเทคโนโลยีสารสนเทศ คณะเทคโนโลยีสารสนเทศ ทุกคนที่ให้คำแนะนำต่างๆ และคอยให้กำลังใจเสมอมา

ขอขอบคุณบัณฑิตศึกษาและบัณฑิตวิทยาลัยคณะเทคโนโลยีสารสนเทศที่ให้ความช่วยเหลือในการช่วยดำเนินเรื่องต่างๆ สำหรับการสอบและจัดทำรูปเล่มวิทยานิพนธ์จนแล้วเสร็จ

สุดท้ายนี้ข้าพเจ้าขอกราบขอบพระคุณ บิดา มารดา และครอบครัวของข้าพเจ้าที่เป็นกำลังใจ และให้การสนับสนุนในทุกเรื่องๆ ทำให้ข้าพเจ้าสามารถทำวิทยานิพนธ์ฉบับนี้สำเร็จลุล่วงด้วยดี

คุณค่าและประโยชน์อันพึงมาจากวิทยานิพนธ์ฉบับนี้ ข้าพเจ้าขอมอบแด่ผู้มีพระคุณทุกท่าน

ฐาปนา ชูรัตน์

สารบัญ

สารบัญ (ต่อ)

หน้า

สารบัญ (ต่อ)

สารบัญตาราง

สารบัญรูป

สารบัญรูป (ต่อ)

บทที่ 1
บทนำ

1.1 ความเป็นมาและความสำคัญของปัญหา

รูปแบบของบริการบนระบบอินเตอร์เน็ตปัจจุบันนั้นมีมากมายหลากหลายรูปแบบ แต่ละรูปแบบมีความแตกต่างกันทั้งในเรื่องลักษณะการทำงาน และ ความต้องการด้านคุณภาพของงานบริการ ในขณะที่อุปกรณ์สำคัญของระบบเครือข่ายอย่าง เราเตอร์ (Router) ยังคงให้บริการแบบ best-effort โดยจะส่งแพ็กเก็ตในลักษณะ first come first served ซึ่งไม่เพียงพอจะตอบสนองความต้องการของบริการได้ครอบคลุมทั้งหมด ดังนั้น เราเตอร์ จึงจำเป็นต้องมีกลไกเพิ่มเติมเพื่อมาช่วยให้สามารถตอบสนองความต้องการ เช่น admission control, resource reservation, per-flow queuing, fair scheduling ซึ่งต้องการกลไกเพื่อช่วยแยกแพ็กเก็ตเป็นกระแสที่แตกต่างกัน ซึ่งนั้นหมายความว่า การจัดประเภทแพ็กเก็ตถือเป็นกลไกอย่างหนึ่งที่สำคัญต่อเรื่องดังกล่าว

การจัดประเภทแพ็กเก็ต (Packet classification) คือ กลไกในการตรวจสอบแพ็กเก็ตที่ผ่านเข้ามา โดยพิจารณาจากข้อมูลเฮดเดอร์ของแพ็กเก็ต (Packet's header) ตั้งแต่ 1 ฟิลด์ขึ้นไป เช่น การจัดประเภทแพ็กเก็ตสำหรับ เราเตอร์ พิจารณาจาก Destination IP Address การจัดประเภทแพ็กเก็ตสำหรับไฟวอลล์ พิจารณาจาก Source IP Address, Destination IP Address, Source port, Destination Port เป็นต้น จากนั้นทำการกำหนดเป็นกระแส (Flow) ที่เหมาะสม แพ็กเก็ตที่อยู่ในกระแสเดียวกันจะถูกดำเนินการด้วยกระบวนการเดียวกัน

จากความสำคัญที่กล่าวมานั้นจึงมีงานวิจัยเกี่ยวกับการจัดประเภทแพ็กเก็ตมากมาย แต่ละประเภทมีทั้งข้อดีและข้อด้อย งานวิจัยต่างมุ่งเน้นเพื่อแก้ปัญหาต่างๆ เช่น ความต้องการด้านความเร็วในการค้นหาสูงซึ่งงานวิจัยหลายงานสามารถแก้ปัญหาดังกล่าวได้ แต่กลับติดปัญหาเรื่องการปรับปรุงกฎ ผู้วิจัยจึงมีความสนใจที่จะนำเสนอการจัดประเภทแพ็กเก็ตที่ประสิทธิภาพทั้งด้านความเร็วในการค้นหาและความเร็วในการปรับปรุงกฎ

1.2 ความมุ่งหมายและวัตถุประสงค์ของการศึกษา

เพื่อนำเสนอวิธีการจัดประเภทแพ็กเก็ตที่ประยุกต์ใช้แนวคิดการใช้แคช เพื่อเพิ่มประสิทธิภาพด้านความเร็วในการค้นหา และประสิทธิภาพความเร็วในการปรับปรุงกฎ

1.3 สมมติฐานของการศึกษา

การจัดประเภทแพ็กเก็ตตากฎด้วยวิธี Bitmap Intersection Lookup (BIL) ที่มีความเร็วในด้านการค้นหา แต่หากแพ็กเก็ตที่เข้ามามีลักษณะซ้ำกันเรียงติดกันประสิทธิภาพด้านความเร็วในการค้นหาก็เท่าเดิม ซึ่งในกรณีที่แพ็กเก็ตที่เข้ามามีลักษณะซ้ำกันเรียงติดกันนี้น่าจะสามารถเพิ่มประสิทธิภาพได้โดยการนำหลักการใช้แคชมาประยุกต์ใช้ แต่ด้วยโครงสร้างของการจัดประเภทแพ็กเก็ตด้วยวิธี Bitmap Intersection Lookup (BIL) [1] จะเก็บข้อมูล Bitvector เรียงต่อกัน การค้นหาจะทำการค้นหาเริ่มตั้งแต่ Bitvector ชุดแรกจนถึงชุดสุดท้ายเสมอ การจะนำเอาหลักการใช้แคชมาประยุกต์ใช้นั้นจำเป็นต้องปรับโครงสร้างการเก็บข้อมูล Bitvector ที่เก็บเรียงต่อกันเปลี่ยนเป็นเก็บลงในเพจที่เรียงต่อกันเสียก่อน เพื่อให้สามารถรองรับหลักการใช้แคช ดังนั้นเมื่อมีแพ็กเก็ตที่ซ้ำกับแพ็กเก็ตที่ได้ค้นหาก่อนแล้วควรจะข้ามเพจเริ่มต้น (เพจแรก) ไปค้นหาที่เพจที่เคยพบกฎดังกล่าว ด้านการปรับปรุงกฎเมื่อมีการเก็บข้อมูล Bitvector ลงในเพจที่เรียงต่อกัน ส่งผลให้ไม่ว่ากฎที่เพิ่มเข้ามาใหม่นั้นมี Bitvector รองรับหรือไม่ก็ไม่จำเป็นต้องสร้างโครงสร้างใหม่ทั้งหมด จะทำการสร้าง (allocate) เพจใหม่ขึ้นมารองรับ เพื่อลดเวลาในการปรับปรุงกฎลง ในงานวิจัยฉบับนี้จะเรียกการจัดประเภทแพ็กเก็ตประเภทนี้ว่า "อัลกอริทึมการจัดประเภทแพ็กเก็ตตามกฎด้วยวิธีบิตแมปอินเตอร์เซคชันลุกอัพโดยใช้แคช"

1.4 ทฤษฎีหรือแนวคิดที่ใช้ในการวิจัย

การจัดประเภทแพ็กเก็ตตามกฎด้วยวิธี Bitmap Intersection Lookup (BIL) ในการค้นหานั้นจะเสียเวลา AND ของ Bitvector ตั้งแต่ชุดแรกจนถึงเพจสุดท้าย หรือจนกว่าจะพบกฎ ทำให้เวลานานเพื่อที่จะกฎที่ตรงกับแต่ละแพ็กเก็ต ถึงแม้ว่าแพ็กเก็ตที่เข้ามาจะซ้ำกันแพ็กเก็ตก่อนหน้าจึงมีแนวความคิดในการนำแคชมาใช้ โดยเก็บ Bitvector แบ่งเป็นเพจ และเก็บค่าของเพจที่เคยค้นหาพบไว้ในแคช ถ้าการค้นหาครั้งถัดไปซ้ำกับการค้นหาครั้งก่อนหน้าก็ให้นำค่าเพจในแคชมาใช้ได้ ทำให้เพิ่มความเร็วในการทำงานขั้นตอนการค้นหาได้

1.5 การเปรียบเทียบระหว่างวิธีการที่นำเสนอกับวิธีการแบบพื้นฐาน

การเปรียบเทียบประสิทธิภาพของการจัดประเภทแพ็กเก็ตโดยจะทำการทดลองเปรียบเทียบความเร็วของการค้นหาและการปรับปรุงกฎ กับการจัดประเภทแพ็กเก็ตตามกฎด้วยวิธี Bitmap Intersection Lookup ด้านการวัดประสิทธิภาพด้านความเร็วในการค้นหากฎ ทดสอบด้วยการจำลองข้อมูลแพ็กเก็ตที่ใช้ในการค้นหาจำนวน 100,000 แพ็กเก็ต จากฐานข้อมูลกฎ 1,000, 2,000, ถึง 10,000 กฎ แบ่งออกเป็น 3 ลักษณะ คือ 1) Repeat first rule 2) Repeat last rule และ 3) Random rule โดยแต่ละลักษณะมีรายละเอียดดังนี้

1) Repeat first rule กำหนดให้แพ็กเก็ตตรงกับกฎข้อแรกในฐานข้อมูลกฎเรียงติดกัน

2) Repeat last rule กำหนดให้แพ็กเก็ตตรงกับกฎข้อสุดท้ายในฐานข้อมูลกฎเรียงติดกัน

3) Random rule กำหนดให้แพ็กเก็ตโดยการสุ่มกฎจากฐานข้อมูลโดยไม่ให้แพ็กเก็ตที่ตรงกับกฎข้อเดียวกันเรียงติดกัน

นำข้อมูลดังกล่าวทำการค้นหาเปรียบเทียบประสิทธิภาพด้านการค้นหาของทั้งสองวิธี ด้านการวัดประสิทธิภาพด้านความเร็วในการปรับปรุงกฎทำการเพิ่มกฎและลบกฎตั้งแต่จำนวน 1,000 – 10,000 กฎ โดยกำหนดให้โครงสร้างเริ่มต้นสามารถรองรับจำนวนกฎเท่ากัน

1.6 ขอบเขตการวิจัย

งานวิจัยในครั้งนี้ต้องการนำเสนอแนวคิดของวิธีการจัดประเภทแพ็กเก็ตตามกฎที่มีประสิทธิภาพด้านความเร็วในการค้นหาและด้านความเร็วในการปรับปรุงกฎ พิสูจน์โดยทำการพัฒนาอัลกอริทึมด้วยตัวแปรภาษา GCC บนระบบปฏิบัติการ FreeBSD เปรียบเทียบประสิทธิภาพกับการจัดประเภทแพ็กเก็ตตามกฎด้วยวิธี Bitmap Intersection Lookup (BIL) การทดลองจะทำการจำลองฐานข้อมูลกฎจำนวน 1,000, 2,000, ..., 10,000 กฎ และแพ็กเก็ตที่ใช้ในการค้นหาจำนวน 100,000 แพ็กเก็ต โดยแบ่งเป็น 3 ลักษณะ คือ 1) Repeat first rule 2) Repeat last rule 3) Random rule ตามที่ได้อธิบายข้างต้น การเปรียบเทียบประสิทธิภาพประกอบด้วย ประสิทธิภาพด้านความเร็วในการค้นหา ประสิทธิภาพด้านความเร็วในการปรับปรุงกฎ และประสิทธิภาพด้านการใช้เนื้อที่เก็บข้อมูล

1.7 ขั้นตอนของการศึกษา

บทที่ 1 กล่าวถึงความเป็นมาของงานวิจัย ความมุ่งหมายและวัตถุประสงค์ สมมติฐาน ทฤษฎีที่ใช้ ขอบเขตของการวิจัย และขั้นตอนการศึกษา

บทที่ 2 กล่าวถึงความหมายของการจัดประเภทแพ็กเก็ตตามกฎ ความต้องการของการจัดประเภทแพ็กเก็ตตามกฎ การกำหนดค่าของกฎให้อยู่ในรูปแบบ Prefix และ ตัวอย่างวิธีการจัดประเภทแพ็กเก็ตตามกฎ

บทที่ 3 อัลกอริทึมการจัดประเภทแพ็กเก็ตตามกฎด้วยวิธีบิตแมปอินเตอร์เซคชันลุกอัพโดยใช้แคช บทนี้กล่าวถึงแนวคิดและสถาปัตยกรรม

บทที่ 4 การวิเคราะห์ทฤษฎี ตามกรณีที่ตั้งสมมติฐานด้านการค้นหา ด้านการปรับปรุงกฎ และการใช้เนื้อที่ในการเก็บข้อมูล

บทที่ 5 การทดลองและผลการทดลอง เพื่อแสดงประสิทธิภาพด้านความเร็วในการค้นหา และ ด้านความเร็วในการปรับปรุงกฎ ของวิธีที่นำเสนอกับการจัดประเภทแพ็กเก็ตเดิม

บทที่ 6 เป็นบทสรุปผลการวิจัยและข้อเสนอแนะ

บทที่ 2
ทฤษฎีพื้นฐานที่ใช้ในการวิจัย
การจัดประเภทแพ็กเก็ตตามกฎ

ในหัวข้อนี้จะกล่าวถึงทฤษฎีที่เกี่ยวข้องในการวิจัย ซึ่งเนื้อหาในบทนี้จะกล่าวถึงความหมาย การวัดประสิทธิภาพ การกำหนดค่าของกฎให้อยู่ในรูปแบบ Prefix ความต้องการของการจัดประเภทแพ็กเก็ต เนื้อหาในบทนี้จำเป็นสำหรับการศึกษาและการประเมินประสิทธิภาพของการจัดประเภทแพ็กเก็ตตามกฎ

2.1 ความหมายของการจัดประเภทแพ็กเก็ต

การจัดประเภทแพ็กเก็ต (Packet classification) คือ กลไกในการตรวจสอบแพ็กเก็ตที่ผ่านเข้ามา โดยพิจารณาจากข้อมูลเฮดเดอร์ของแพ็กเก็ต (Packet's header) ตั้งแต่ 1 ฟิลด์ขึ้นไป เช่น การจัดประเภทแพ็กเก็ตสำหรับเราเตอร์พิจารณาจาก Destination IP Address การจัดประเภทแพ็กเก็ตสำหรับไฟวอลล์พิจารณาจาก Source IP Address, Destination IP Address, Source port, Destination Port เป็นต้น จากนั้นทำการกำหนดเป็นกระแส (Flow) ที่เหมาะสม แพ็กเก็ตในกระแสเดียวกันจะถูกดำเนินการด้วยกระบวนการเดียวกัน

กระแส (Flows) [3] ถูกระบุโดยกฎ ใช้ในการจัดประเภทแพ็กเก็ตที่เข้ามา และเรียกที่รวบรวมกฎว่า classifier แต่ละกฎสามารถระบุได้หนึ่งกระแส ซึ่งประกอบด้วยเงื่อนไขและกระบวนการดำเนินการ

2.2 การวัดประสิทธิภาพของการจัดประเภทแพ็กเก็ต

การจัดประเภทแพ็กเก็ตตามกฎเป็นเครื่องมือที่ใช้บนเครือข่ายอินเตอร์เน็ต ซึ่งต้องทำงานบนช่องทางการสื่อสารที่มีความเร็วสูงด้วยกฎที่มีขนาดใหญ่ ดังนั้นการจัดประเภทแพ็กเก็ตตามกฎจึงต้องมีประสิทธิภาพสูง โดยประสิทธิภาพจะพิจารณาด้านต่างๆ ดังต่อไปนี้ [3]

2.2.1 มีความเร็วในการค้นหาสูง

ผู้ใช้ต้องการความเร็วจากบริการ ยิ่งต้องทำงานอยู่บนเครือข่ายและช่องทางการสื่อสารที่มีขนาดใหญ่ กลไกอย่างการจัดประเภทแพ็กเก็ตยิ่งต้องมีความเร็วในการค้นหาสูง เพื่อให้สามารถตอบสนองความต้องการของผู้ใช้ได้

2.2.2 มีความเร็วในการปรับปรุงกฎสูง

การจัดประเภทแพ็กเก็ตตามกฎนั้น โครงสร้างข้อมูลถูกสร้างจากกฎที่ถูกกำหนดไว้ ซึ่งกฎมีการเปลี่ยนแปลงตลอดเวลา ทั้งจากผู้ดูแลเครือข่ายหรือโปรโตคอล หากการปรับปรุงกฎช้าส่งผลให้การจัดประเภทแพ็กเก็ตช้าตามไปด้วย ดังนั้นจำเป็นต้องมีการปรับปรุงที่มีความเร็วสูงเช่นกัน

2.2.3 ใช้เนื้อที่ในการเก็บข้อมูลต่ำ

การจัดประเภทแพ็กเก็ตทำงานกับกฎที่มีขนาดใหญ่ แต่มีอุปกรณ์ที่ใช้นั้นมีหน่วยความจำขนาดจำกัด ดังนั้น การจัดประเภทแพ็กเก็ตต้องใช้หน่วยความจำให้น้อย และมีประสิทธิภาพที่สุด

2.3 การกำหนดค่าของกฎให้อยู่ในรูป Prefix

วิธีการของการจัดประเภทแพ็กเก็ตตามกฎหลายๆวิธีต้องทำการกำหนดค่าของกฎให้อยู่ในรูปแบบ Prefix ซึ่งข้อมูลที่ถูกแปลงค่านั้นจะอยู่ในรูปเซตของ Prefix ซึ่งแต่ละเซตมีสมาชิกไม่เกิน 2•W – 2 ตัว [3] โดยมีขนาด Prefix เท่ากับ ขนาดเฮดเดอร์ฟิลด์ (W) ของแพ็กเก็ต เช่น กำหนดให้แต่ละกฎมีขนาด Prefix เท่ากับ 4 บิต กฎ 1 มีค่าเท่ากับ 4-7 ดังนั้น สมาชิกจะประกอบด้วยเซตของ Prefix เป็น {01**}, กฎ 2 มีค่าเท่ากับ 1- 14 ดังนั้นจะประกอบด้วยเซตของ Prefix เป็น {0001, 001*, 01**, 10**, 110*, 1110} เป็นต้น

จากรูปแบบ value/mask ทำการกำหนดค่าให้อยู่ในรูปแบบของ Prefix ได้ดังนี้ value เท่ากับ 0100 และ mask เท่ากับ 1100 นำ value และ mask มาทำการ intersection ด้วยตรรกะ AND ผลลัพธ์ คือ 0100 = 4 เป็นสมาชิกแรก จากนั้น สมาชิกตัวสุดท้ายเท่ากับ inverse ของ mask เท่ากับ 0011 = 3 บวกกับสมาชิกตัวแรก 3 + 4 = 7 ดังนั้นค่า prefix ประกอบด้วยช่วงของ 4-7 ทำให้การเก็บข้อมูลกฎสามารถเก็บรูปแบบของช่วงหรือเก็บในรูปแบบ value/mask ก็ได้ เนื่องจากเวลานำไปใช้เราต้องทำการแปลงให้อยู่ในรูปแบบ Prefix ตัวอย่างเก็บข้อมูลกฎ ในตารางที่ 2.1 แสดงกฎจำนวน 6 กฎที่ทำการแปลงค่าเป็น Prefix โดยกฎแต่ละข้อประกอบด้วยฟิลด์จำนวน 2 ฟิลด์ คือ ฟิลด์ F1 และ F2

ตารางที่ 2.1 ตัวอย่างของกฎในรูปของ Prefix [3]

Rule	F1	F2
R1	00*	00*
R2	0*	01*
R3	1*	0*
R4	00*	0*
R5	0*	1*
R6	*	1*

2.4 ตัวอย่างของวิธีการจัดประเภทแพ็กเก็ตตามกฎ

2.4.1 Linear Search

รูปแบบโครงสร้างข้อมูลเป็นแบบ Linked-list [3] ของกฎที่เรียงลำดับตามความสำคัญจากมากไปน้อย จะค้นหาโดยการนำแพ็กเก็ตไปเปรียบเทียบกับโครงสร้างข้อมูลดังกล่าวแบบตามลำดับ (sequential search) จนกว่าจะพบกฎที่ตรงกัน อัลกอริทึมนี้ใช้พื้นที่ได้ไม่ดีนัก เวลาในการจัดประเภทขึ้นอยู่กับจำนวนกฎ

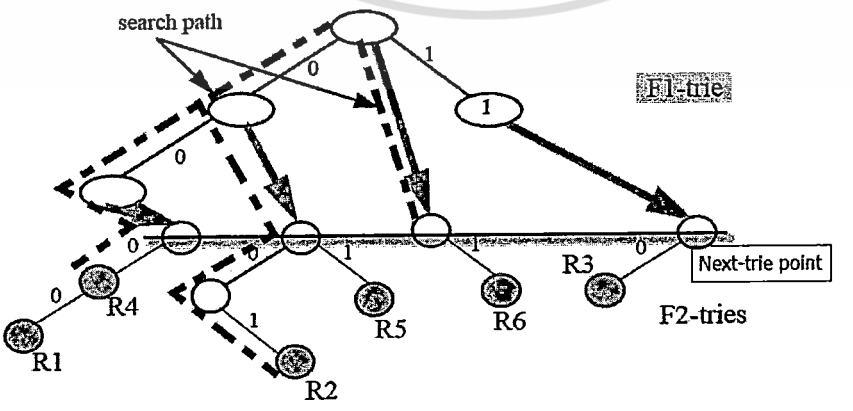
Time complexity เท่ากับ O (N) และ Storage complexity เท่ากับ O (N)

2.4.2 Hierarchical Tries

รูปแบบโครงสร้างข้อมูลเป็นแบบต้นไม้ D – dimensional ของ hierarchical radix tries [5] เป็นการแผ่ออกอย่างง่ายของโครงสร้างข้อมูล 1 dimension ยกตัวอย่างดังนี้ หากมี D มากกว่า 1 (มากกว่า 1 ฟิลด์) เราจะเริ่มสร้างจาก dimensional แรกก่อน โดยเรียกว่า F1-trie นำชุดของ prefix $\{R_{j1}\}$ (บิตในฟิลด์ที่หนึ่งเรียงตามลำดับนั้นเอง) มาสร้างเป็นโหนดใน F1-trie ความเป็นไปได้ของบิต คือ 0, 1 และ * จากโหนดแรกสุดสร้างโหนดลูกซ้ายและขวา 0,1 ตามลำดับแต่ละละดับชั้นเทียบกับตำแหน่งบิตในชุดของ prefix $\{R_{j1}\}$ และเมื่อเจอบิตที่เป็น * จะทำการสร้างโหนดที่เรียกว่า Next-trie point (และเมื่อเป็นบิตตัวสุดท้ายด้วย) และทำแบบนี้กับ dimensional ถัดไปจนครบ บางครั้งถูกเรียกว่า Multi-level tries, Back-tracking search tries หรือ tries-of-tries ดังรูปที่ 2.1 โดยยึดตามกฎในตารางที่ 2.1

จากรูปที่ 2.1 การค้นหาจะนำแพ็กเก็ตเปรียบเทียบไปทีละโหนดตามเลเวลของ Hierarchical tries จากเส้นประ จะเห็นทางการค้นหาของแพ็กเก็ต (000,010) จากเส้นทางเห็นได้ว่าความเป็นไปได้ในการค้นหานั้นมีโอกาสที่จะไม่เจอกฎที่ตรงก็จำเป็นต้องใช้เส้นทางใหม่

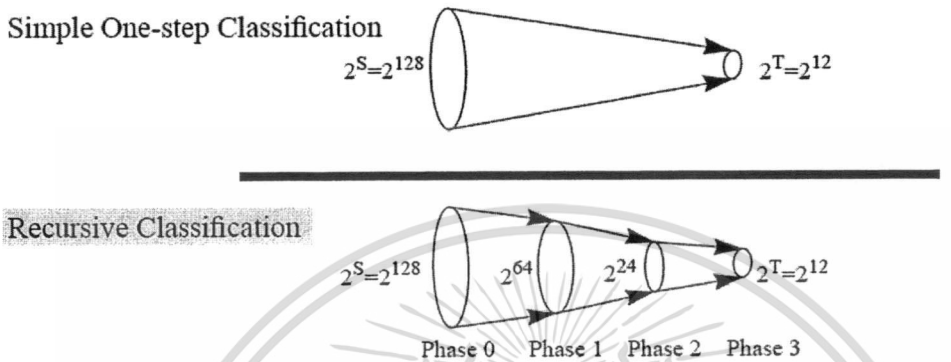
Time complexity เท่ากับ O (WD) และ Storage complexity เท่ากับ O (N•D•W)



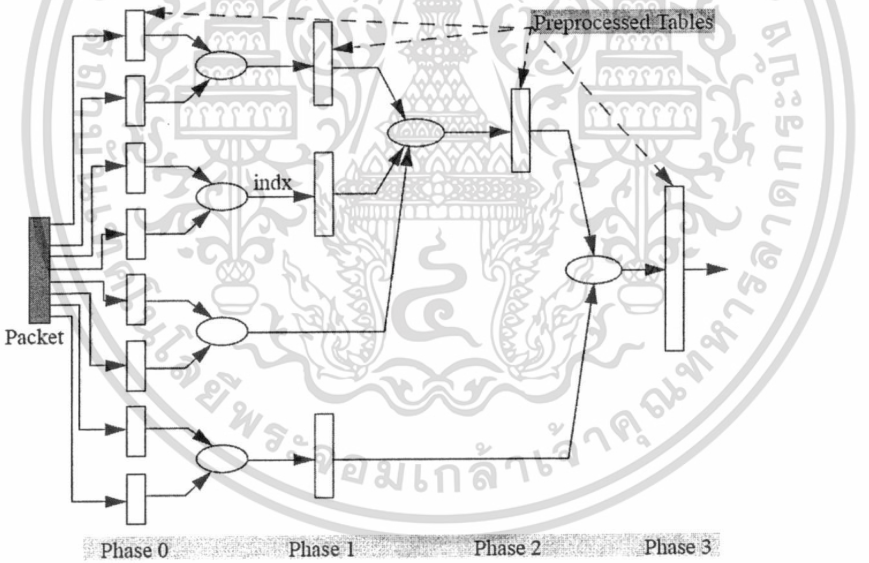
รูปที่ 2.1 Hierarchical Tries

2.4.3 Recursive Flow Classification (RFC)

RFC [4] เป็นอัลกอริทึมแบบฮิวริสติก สำหรับการจัดประเภทแบบหลายฟิลด์ จากรูปที่ 2.2 การจัดประเภทแพ็กเก็ตทำโดยการ Mapping เฮดเดอร์จาก S บิต ไปเป็น T บิต โดย T < S ในการที่ Map แบบครั้งเดียวเลยนั้นไม่สามารถทำได้ในความเป็นจริง ดังนั้น จึงต้องมีการแบ่งออกเป็น Phase (รูปที่ 2.3) โดยที่ Phase I จะนำค่าที่ส่งมาจาก Phase I-1 มาคำนวณเป็นอินเดก



รูปที่ 2.2 การ Mapping เฮดเดอร์บิตของแพ็กเก็ตจาก S บิต ไปเป็น T บิต



รูปที่ 2.3 กระแสแพ็กเก็ต (Packet Flow) ใน RFC

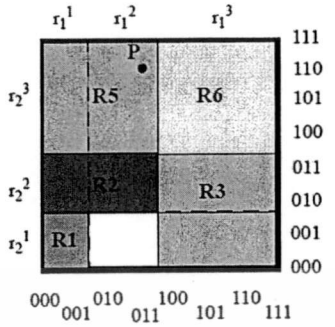
ขั้นตอนการเปรียบเทียบมีดังนี้

1. นำแพ็กเก็ตมาแบ่งออกเป็น block เล็กๆ ตามใน Phase 0 จากนั้นนำค่าไปเปรียบเทียบกับอินเด็กใน Preprocessed table ของ Phase ดังกล่าว นำค่าที่ได้ไปคำนวณหาอินเด็กของ Phase ถัดไป
2. ใน Phase ถัดไปจะนำค่าที่ส่งมาจาก Phase ก่อนหน้ามาทำเหมือนในขั้นตอนแรก
3. ใน Phase สุดท้ายจะได้ค่าเป็น classID เพื่อชี้ไปยังกฎในฐานข้อมูล

Time complexity เท่ากับ O (D) และ Storage complexity เท่ากับ O (N^D)

2.4.4 Bitmap-intersection

Bitmap-intersection [3] เป็นการจัดประเภทแพ็กเก็ตแบบหลายฟิลด์ ทำโดยการแบ่งกฎออกเป็นช่วงใส่ลงบนแกนของกราฟ



รูปที่ 2.4 แนวคิดของ Bitmap-intersection

จากนั้นสร้าง bitmap ขนาดเท่ากับ N บิต โดยตำแหน่งของบิตแต่ละบิตใน bitmap จะตรงกับหมายเลขของกฎและหากกฎใดอยู่ในช่วงดังกล่าวบิตในตำแหน่งเดียวกับกฎจะมีค่าเท่ากับ 1 แต่หากไม่จะมีค่าเป็น 0 ในตำแหน่งดังกล่าว ตัวอย่างดังรูปที่ 2.5 ใน dimension 1 r_1^1 มีค่า bitmap เท่ากับ 110111 เนื่องจากในช่วงของ r_1^1 ตรงกับกฎ R1, R2, R4, R5, R6 ในกราฟด้านบน

Dimension 1

r_1^1	{R1,R2,R4,R5,R6}	110111
r_1^2	{R2,R5,R6}	010011
r_1^3	{R3,R6}	001001

Dimension 2

r_2^1	{R1,R3,R4}	101100
r_2^2	{R2,R3}	011000
r_2^3	{R5,R6}	000111

รูปที่ 2.5 โครงสร้างของ Bil table

Query on P(011,010): 010011 Dimension 1 bitmap
000111 Dimension 2 bitmap

Result ⟶ 000011

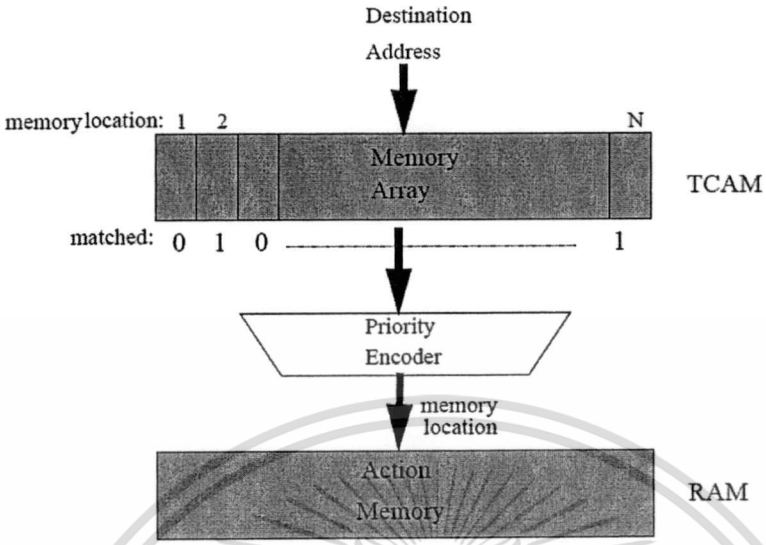
R5 Best matching rule

รูปที่ 2.6 การค้นหากฎ

การค้นหานำแพ็กเก็ต P (011,010) เทียบในกราฟพบว่ามีได้ค่าเท่ากับ dimension 1 เป็น r_1^2 และ dimension 2 เป็น r_2^3 นำ bitmap จากทั้งสอง dimension มาทำการ intersection กันได้ผลลัพธ์เป็น 000011 ตำแหน่งที่ 5, 6 มีค่าเป็น 1 แสดงว่าแพ็กเก็ตดังกล่าวตรงกับกฎ 5, 6 เลือกตามลำดับความสำคัญ Best matching rule คือ R5 (จากบิตตัวแรกที่เป็น 1 ตรงกับกฎที่ 5)

Time complexity เท่ากับ O (D•W + N/memwidth) และ Storage complexity เท่ากับ O (D•N^2)

2.4.5 Ternary CAMs (Ternary Content-addressable memory)



รูปที่ 2.7 โครงสร้างของวิธีการ Ternary CAM

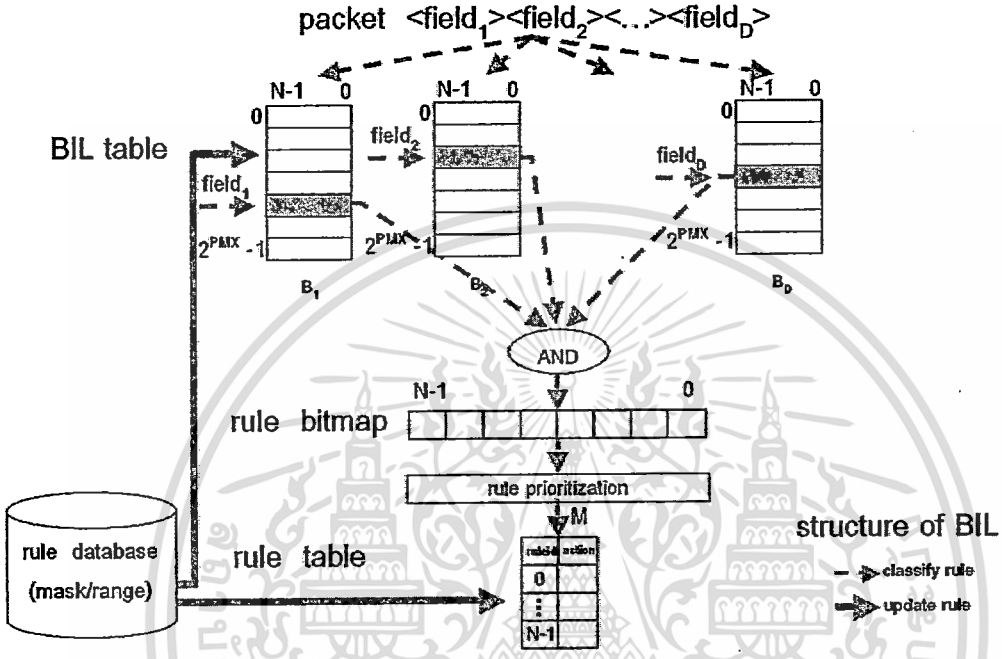
เป็นการจัดประเภทแพ็กเก็ตทำงานเฉพาะบนฮาร์ดแวร์ ซึ่งใช้หน่วยความจำที่มีจำนวน Transistor มากกว่าหน่วยความจำทั่วไป โดยแต่ละบิตบน TCAM [2] จะประกอบด้วยบิต 0 หรือ 1 หรือ * แต่ก็มีข้อด้อย คือ TCAM ใช้พลังงานสิ้นเปลื้องและราคาแพงกว่าหน่วยความจำทั่วไป การค้นหาจะนำเฮดเดอร์ของแพ็กเก็ต ทำการ lookup แบบขนานทุกแอดเดรสของหน่วยความจำ TCAM และตรงกับกฎที่บิตมีค่าเท่ากับ 1 ทำการเลือกกฎที่มีความสำคัญสูงสุดเป็นผลลัพธ์

Time complexity เท่ากับ O (1) และ Storage complexity เท่ากับ O (N)

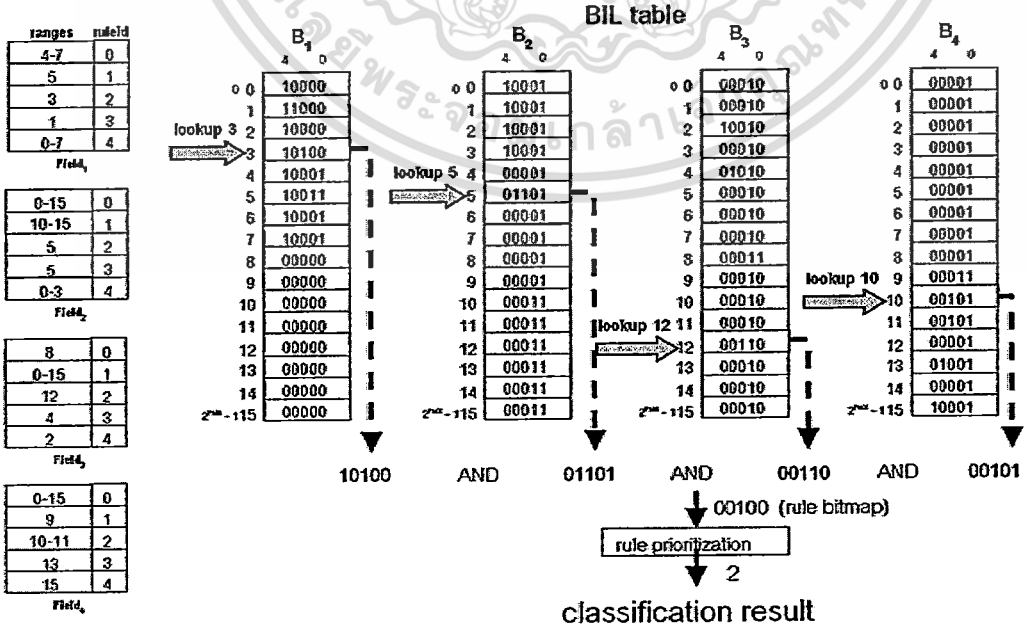
2.5 Bitmap Intersection Lookup (BIL)

สถาปัตยกรรมของวิธีการ Bitmap Intersection Lookup [1] จะประกอบด้วย BIL table จำนวน D ตาราง แต่ละตารางประกอบด้วยแอดเดรสจำนวน 2^{PMX} แอดเดรส ซึ่งแต่ละแอดเดรสประกอบด้วยบิตจำนวน N บิต ตั้งแต่บิต 0 ถึงบิต N-1 เรียกว่า Bitvector และตำแหน่งแต่ละบิตของ Bitvector จะตรงกับตำแหน่งของกฎใน rule table โดยค่าบิตที่เป็น “1” คือ บิตที่อยู่ในแอดเดรสที่ตรงกับค่าของกฎและค่าบิตที่เป็น “0” คือ บิตที่ไม่อยู่ในแอดเดรสที่ตรงกับค่าของกฎ การจัดเรียงลำดับความสำคัญของกฎจะจัดเรียงจากมากไปน้อย ในการเพิ่มกฎจะนำค่าของกฎซึ่งถูกกำหนดในรูปของมาสค์ (Mask) หรือช่วง (Range) ทำการแปลงค่าให้อยู่ในรูปของ Prefix เพื่อใช้สำหรับ Set Bitvector ใน BIL table และจัดเก็บแอ็คชันของกฎแต่ละข้อลงใน rule table ต่อไป

สำหรับการค้นหาจะแบ่งเฮดเดอร์ของแพ็กเก็ตออกเป็น Block ขนาดเล็กๆ จำนวน D blocks นำค่าของแต่ละ Block ไป Lookup ใน BIL table และนำผลลัพธ์ที่ได้จากการ Lookup คือ Bitvector จากแต่ละตาราง ทำการ Intersection ด้วยตรรกะ AND โดยอัลกอริทึมจะเลือกกฎที่มีลำดับความสำคัญสูงสุดเป็นผลลัพธ์และกำหนดแอ็คชันของแพ็กเก็ตจากข้อมูลในตาราง rule table ดังรูปที่ 2.8



รูปที่ 2.8 โครงสร้างของวิธี Bitmap Intersection Lookup



รูปที่ 2.9 ตัวอย่างการค้นหาและปรับปรุงกฎ

1. การปรับปรุงกฎ ในตาราง Field1- Field4 แสดงฐานข้อมูลกฎที่กำหนดอยู่ในรูปของช่วงทั้งหมด 5 ข้อ คือ ruleid 0 - ruleid 4 โดยกฎแต่ละข้อจะประกอบด้วยฟิลด์จำนวน 4 ฟิลด์ ซึ่งแต่ละฟิลด์มีขนาด 4 บิต กำหนดให้ PMX คือขนาดของ Block เท่ากับ 4 บิต ดังนั้น BIL table แต่ละตารางจะประกอบด้วยแอดเดรสจำนวน $2^4 = 16$ แอดเดรส คือ หมายเลขแอดเดรส 0 ถึง หมายเลขแอดเดรส 15 ซึ่งแต่ละแอดเดรสจะมีขนาดเท่ากับจำนวนกฎเท่ากับ 5 บิต ตั้งแต่ บิต 0 ถึงบิต 4 ในการเพิ่มกฎจะนำค่าของกฎซึ่งถูกกำหนดในรูปของช่วงทำการแปลงค่าให้อยู่ในรูปของ Prefix เพื่อใช้สำหรับ Set Bitvector ใน BIL table เช่น ที่ Field1 พบว่า ruleid 0 มีค่าเท่ากับ 4 - 7 (0100-0111) ดังนั้นจะประกอบด้วยเซตของ Prefix เป็น {01**} หรือ {0100/1100} (ในรูปของ value/mask) นำ Prefix ที่ได้ทำการ Set Bitvector ในตาราง B1 ตั้งแต่แอดเดรส 4 - 7 ด้วยอัลกอริทึมในรูปที่ 2.7 (ในกรณีการลบกฎจะนำ Prefix ที่ได้ทำการ Reset Bitvector ด้วยอัลกอริทึมในรูปที่ 2.8) โดยจากฐานข้อมูลกฎสามารถสร้างเป็น Bitvector ดังที่แสดงในตาราง B1-B4

```
1. for (j = (value & mask);j <= ((value & mask) + (~mask));j++) {
2.    setBitvector(i,j,ruleid);
3. }
```

รูปที่ 2.10 อัลกอริทึม Set Bitvector

```
1. for (j = (value & mask);j <= ((value & mask) + (~mask));j++) {
2.    resetBitvector(i,j,ruleid);
3. }
```

รูปที่ 2.11 อัลกอริทึม Reset Bitvector

2. การค้นหาทำการแบ่งเฮดเดอร์ของแพ็กเก็ตออกเป็น Block ขนาดเท่ากับ PMX เท่ากับ 4 บิต นำค่าของแต่ละ Block ซึ่งมีค่าเป็น 3, 5, 12 และ 10 ไป Lookup ใน BIL table นำ Bitvector ซึ่งได้จากการ Lookup ในตาราง B1 - B4 ทำการ intersection ด้วยตรรกะ AND โดยอัลกอริทึมจะเลือกกฎที่มีลำดับความสำคัญสูงสุดเป็นผลลัพธ์ ดังนั้นจากตัวอย่างดังกล่าวแพ็กเก็ตจะถูกจัดประเภทตรงกับกฎข้อที่ 2

การวัดประสิทธิภาพเบื้องต้น [1]

กำหนดให้

N คือ จำนวนกฎทั้งหมด

D คือ จำนวนเฮดเดอร์ฟิลด์

F_i คือ ขนาดของแต่ละเฮดเดอร์ฟิลด์ โดยที่ 1 <= i <= D

WORD คือ ขนาดบิตที่ทำการประมวลผลในแต่ละครั้ง

ถ้าให้ F_i มีขนาดเท่ากันทุกฟิลด์ โดยแต่ละฟิลด์มีขนาดเท่ากับ W บิต เพราะฉะนั้นขนาดบิตทั้งหมดจะเท่ากับ D • W บิต

ทำการแบ่ง F_i ออกเป็นขนาด $P_{i,j}$ บิต โดยกำหนดให้

M_i คือ จำนวน Block ที่แบ่งใน F_i โดยที่ $\sum_{j=1}^{M_i} P_{i,j} = F_i$

พิจารณาที่ F_i ใดๆ เพียง 1 ฟิลด์ และที่ $P_{i,j}$ ของ F_i มีขนาดเท่ากัน เท่ากับ PM_i บิต เพราะฉะนั้น

$$M_i = \frac{F_i}{PM_i}$$

กำหนดให้ PM_i มีขนาดเท่ากันทุกฟิลด์ เท่ากับ PMX บิต และ MX คือ จำนวน Block ภายในแต่ละฟิลด์ เพราะฉะนั้น $MX = \frac{W}{PMX}$ และจำนวนตารางจะเท่ากับ $\frac{D \cdot W}{PMX}$ หรือ D•MX

1. เวลาในการค้นหา

```
1.  อ่านค่าเฮดเดอร์ฟิลด์ของแพ็กเก็ตจากฐานข้อมูลแพ็กเก็ต เก็บลงในบัฟเฟอร์ → t_read
2.  for (i=0; i<D; i++ ) {
3.      for (j=0; j<MX; j++ ) {
4.          แบ่งเฮดเดอร์ฟิลด์ที่ i ของแพ็กเก็ตเป็น Block โดยเลื่อนบิตไปทาง
                ขวาครั้งละ PMX บิต และเก็บค่าลงในหน่วยความจำ → t_shiftaverage
5.      }                                                          } t_div
6.  }
7.  for (i=0; i<n/WORD; i++ ) {
8.      for (j=0; j< D•MX; j++ ) {
9.          นำค่าของเฮดเดอร์ฟิลด์ที่ถูกแบ่งในหน่วยความจำไป Lookup ใน BIL table และ
            นำผลลัพธ์ คือ Bitvector ในตำแหน่ง WORD ที่ i จากุ BIL table j มา AND กัน
            → t_look + t_and                                       } t_sproc
10.     }                                                          } t_search
11.     if (ผลลัพธ์จากการ AND != 0) {
12.         คำนวณหมายเลขของกฎจากตำแหน่งบิตที่มีค่าเป็น 1 บิตแรกใน Bitmap และ
            ส่งค่า Action จาก rule table กลับ
13.     }
14. }
15. ส่งค่า Default Action กลับ
```

รูปที่ 2.12 อัลกอริทึมในการค้นหา

จากรูปที่ 2.12 อัลกอริทึมการค้นหาจะเริ่มการทำงานโดยอ่านค่าเฮดเดอร์ฟิลด์ของแพ็กเก็ตจากฐานข้อมูลแพ็กเก็ต เก็บลงในบัฟเฟอร์ ดังชุดคำสั่งของบรรทัดที่ 1 และทำการแบ่งเฮดเดอร์ฟิลด์ (ซึ่งมีจำนวนทั้งหมด D ฟิลด์) เป็น Block ขนาดเล็กๆ โดยเลื่อนบิตไปทางขวาครั้งละ PMX บิตและเก็บค่าเหล่านั้นลงในหน่วยความจำ โดยจำนวน Block ทั้งหมดจะเท่ากับ $D \cdot MX$ หรือ $\frac{D \cdot W}{PMX}$ ดังชุดคำสั่งของบรรทัดที่ 2 - 6 ในลำดับถัดไปจะนำค่าของเฮดเดอร์ฟิลด์ที่ถูกแบ่งในหน่วยความจำไป Lookup ใน BIL table และนำผลลัพธ์ คือ Bitvector มากระทำตรรกะ AND ครั้งละ WORD ดังชุดคำสั่งของบรรทัดที่ 8 - 10 โดยหากพบว่าผลลัพธ์ที่ได้จากการ AND มีค่าไม่เท่ากับ 0 อัลกอริทึมจะทำการคำนวณหมายเลขของกฎจากตำแหน่งบิตที่มีค่าเป็น 1 บิตแรกใน Bitmap และส่งค่า Action จาก rule table กลับไป ดังชุดคำสั่งของบรรทัดที่ 11 – 13 โดยอัลกอริทึมจะกระทำตรรกะ AND ทั้งหมด $\frac{n}{WORD}$ ครั้ง ดังชุดคำสั่งของบรรทัดที่ 7 - 14

กำหนดให้

ชุดคำสั่งของบรรทัดที่	1 - 15	ใช้เวลาในการทำงานเท่ากับ t_{search}
ชุดคำสั่งของบรรทัดที่	1	ใช้เวลาในการทำงานเท่ากับ t_{read}
ชุดคำสั่งของบรรทัดที่	2 - 6	ใช้เวลาในการทำงานเท่ากับ t_{div}
ชุดคำสั่งของบรรทัดที่	4	ใช้เวลาในการทำงานเท่ากับ $t_{shiftaverage}$
ชุดคำสั่งของบรรทัดที่	7 - 14	ใช้เวลาในการทำงานเท่ากับ t_{sproc}
ชุดคำสั่งของบรรทัดที่	9	ใช้เวลาในการทำงานเท่ากับ $t_{look} + t_{and}$

เพราะฉะนั้น $t_{search} = t_{read} + t_{div} + t_{sproc}$

โดยที่ $t_{div} = \sum_{i=1}^{D} \left(\sum_{j=1}^{M_i} t_{shiftaverage} \right)$

พิจารณาที่ขนาดฟิลด์เท่ากัน และขนาด PM_i เท่ากัน ดังนั้น

$$t_{div} = MX \cdot D \cdot t_{shiftaverage} = \frac{D \cdot W}{PMX} \cdot t_{shiftaverage}$$

และ $t_{sproc} = \frac{D \cdot W}{PMX} \cdot (t_{look} + t_{and}) = \frac{D \cdot W}{PMX} \cdot t_{la}$

$$= \Pr ob_{found}(n) \cdot \frac{n}{WORD} \cdot \left(\frac{D \cdot W}{PMX} \cdot t_{la} \right) \quad (1)$$

โดยที่ $\Pr ob_{found}(n)$ คือ ความน่าจะเป็นในการค้นหาว่าข้อมูลนี้ตรงกับกฎที่ n

กรณี Best Case, $\Pr ob_{found}(n) \cdot \frac{n}{WORD} = 1$

กรณี Worst Case, $\Pr ob_{found}(n) \cdot \frac{n}{WORD} = \frac{N}{WORD}$

ดังนั้นจากอัลกอริทึม จะใช้เวลาในการค้นหาต่อแพ็กเก็ตโดยมีสมการทั่วไป คือ

$$t_{search} = t_{read} + \frac{D \cdot W}{PMX} \cdot t_{shiftaverage} + \Pr ob_{found}(n) \cdot \frac{n}{WORD} \cdot \left(\frac{D \cdot W}{PMX} \cdot t_{la} \right)$$

$$= t_{read} + \frac{D \cdot W}{PMX} \cdot \left(t_{shiftaverage} + \Pr ob_{found}(n) \cdot \frac{n}{WORD} \cdot t_{la} \right) \quad (2)$$

กรณี Best Case จะใช้เวลาในการค้นหาต่อแพ็กเก็ต โดยมีสมการ คือ

$$t_{search} = t_{read} + \frac{D \cdot W}{PMX} \cdot \left(t_{shiftaverage} + t_{la} \right) \quad (3)$$

และ กรณี Worst Case จะใช้เวลาในการค้นหาต่อแพ็กเก็ต โดยมีสมการ คือ

$$t_{search} = t_{read} + \frac{D \cdot W}{PMX} \cdot \left(t_{shiftaverage} + \frac{N}{WORD} \cdot t_{la} \right) \quad (4)$$

2. เวลาในการปรับปรุงกฎ

```
1.  รับคำสั่งของกฎจากฐานข้อมูลกฎ หรือจากผู้ดูแลระบบ เก็บลงในบัฟเฟอร์
2.  แปลความหมายคำสั่งของกฎที่อ่านมา และเก็บค่าแต่ละฟิลด์ลงในหน่วยความจำ
3.  เก็บค่า ruleid, Action และคำสั่งลงใน rule table, กำหนดค่า tmp = 0 และ d = 0
4.  for (i=0;i<D; i++) {
5.          tmp +=d และ แปลงค่าของกฎให้อยู่ในรูปของ Prefix (value/mask) ➔ t_conv
6.          current = first
7.          While (current != NULL) {
8.                  d=temp
9.                  for (k=0;k < MX; k++) {
10.                         สำหรับ current->Prefix แบ่ง Prefix (value/mask) ของฟิวด์ i เป็น Block โดย
                            เลื่อนบิตไปทางขวาครั้งละ PMX บิต และเก็บค่าลงในตัวแปร value และ
                            mask ➔ t_shiftaverage
11.                         for (l=value AND mask ;l<=( value AND mask)+~mask; l++) {
12.                                 set Bitvector ที่ตำแหน่งบิต ruleid แอดเดรสหมายเลข l
                                ใน BIL table d ➔ t_set
13.                         }
14.                         d++
15.                 }
16.                 current = current->next
17.         }
18. }
```

t_prep : บรรทัด 1–3
t_proc : บรรทัด 4–18
t_add : บรรทัด 1–18

รูปที่ 2.13 อัลกอริทึมการเพิ่มกฎ

จากรูปที่ 2.13 อัลกอริทึมการเพิ่มกฎจะเริ่มการทำงานโดยรับคำสั่งของกฎจากฐานข้อมูลกฎหรือจากผู้ดูแลระบบเก็บลงในบัฟเฟอร์ ทำการแปลความหมายคำสั่งของกฎที่อ่านมาและเก็บค่าแต่ละฟิลด์ลงในหน่วยความจำและเก็บค่า ruleid, Action และคำสั่งลงใน rule table ดังชุดคำสั่งของบรรทัดที่ 1 - 3 โดยอัลกอริทึมจะทำการแปลงค่าของกฎให้อยู่ในรูปของ Prefix (value/mask)

ซึ่งข้อมูลที่ถูกแปลงค่าแล้วจะอยู่ในรูปเซตของ Prefix ซึ่งแต่ละเซตจะมีสมาชิกไม่เกิน 2•W – 2 ตัว ดังชุดคำสั่งของบรรทัดที่ 5 ในลำดับถัดไปจะทำการแบ่ง Prefix (value/mask) เป็น Block โดยเลื่อนบิตไปทางขวาครั้งละ PMX บิตและเก็บค่าลงในตัวแปร value และ mask ดังชุดคำสั่งของบรรทัดที่ 10 และนำตัวแปร value และ mask มาคำนวณหาหมายเลขแอดเดรสต่ำสุดและหมายเลขแอดเดรสสูงสุด ซึ่งจะมีจำนวนแอดเดรสไม่เกิน 2^{PMX} แอดเดรส เพื่อทำการ set Bitvector ในตาราง BIL table ต่อไป ดังชุดคำสั่งของบรรทัดที่ 11 - 13

กำหนดให้

ชุดคำสั่งของบรรทัดที่ 1-18 ใช้เวลาในการทำงานเท่ากับ t_{add}

ชุดคำสั่งของบรรทัดที่ 1-3 ใช้เวลาในการทำงานเท่ากับ t_{prep}

ชุดคำสั่งของบรรทัดที่ 5 ใช้เวลาในการทำงานเท่ากับ t_{conv}

ชุดคำสั่งของบรรทัดที่ 10 ใช้เวลาในการทำงานเท่ากับ $t_{shiftaverage}$

ชุดคำสั่งของบรรทัดที่ 12 ใช้เวลาในการทำงานเท่ากับ t_{set}

ชุดคำสั่งของบรรทัดที่ 4-18 ใช้เวลาในการทำงานเท่ากับ t_{proc}

เพราะฉะนั้น $t_{add} = t_{prep} + t_{proc}$

B_{min} คือ หมายเลขแอดเดรสต่ำสุด

B_{max} คือ หมายเลขแอดเดรสสูงสุด

N_{node} คือ จำนวนสมาชิกของ Prefix (value/mask) ที่ได้จากการแปลงค่าของกฎ โดยที่ $1 <= N_{node} <= (2 \cdot W-2)$

$N_{address} = B_{max} - B_{max} + 1$

$= 2^{mask_d}$; $mask_d$ คือจำนวนบิตที่จะ net mask, $1 <= d <= \frac{D \cdot W}{PMX}$

ดังนั้น $t_{proc} = D \cdot \left[t_{conv} + N_{node} \cdot \frac{W}{PMX} \cdot \left(t_{shiftaverage} + N_{address} \cdot t_{set} \right) \right]$ (5)

กรณี Best Case, $N_{address} = 1$, $N_{node} = 1$

กรณี Worst Case, $N_{address} = 2^{PMX}$, $N_{node} = 2 \cdot W-2$

ดังนั้นจากอัลกอริทึม จะใช้เวลาในการเพิ่มกฎต่อข้อ โดยมีสมการทั่วไป คือ

$$t_{add} = t_{prep} + D \cdot \left[t_{conv} + N_{node} \cdot \frac{W}{PMX} \cdot \left(t_{shiftaverage} + N_{address} \cdot t_{set} \right) \right] \quad (6)$$

กรณี Best Case จะใช้เวลาในการเพิ่มกฎต่อข้อ โดยมีสมการ คือ

$$t_{add} = t_{prep} + D \cdot \left[t_{conv} + \frac{W}{PMX} \cdot \left(t_{shiftaverage} + t_{set} \right) \right] \quad (7)$$

กรณี Worst Case จะใช้เวลาในการเพิ่มกฎต่อข้อ โดยมีสมการ คือ

$$t_{add} = t_{prep} + D \cdot \left[t_{conv} + (2 \cdot W - 2) \cdot \frac{W}{PMX} \cdot \left(t_{shiftaverage} + 2^{PMX} \cdot t_{set} \right) \right] \quad (8)$$

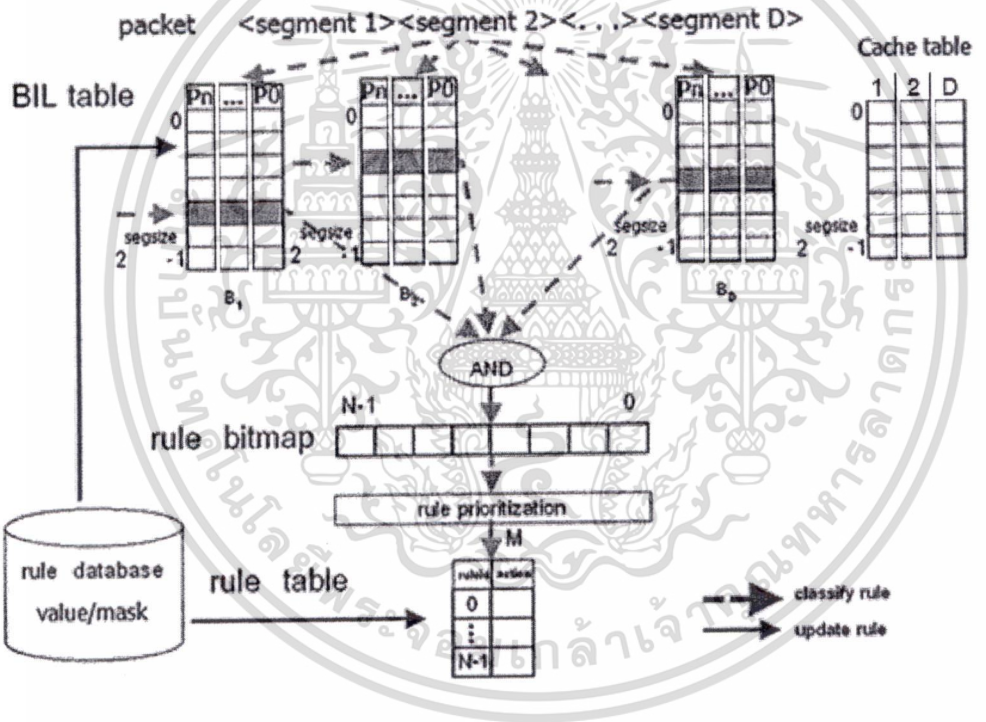
จากอัลกอริทึมการเพิ่มกฎในรูปที่ 2.13 ชุดคำสั่งในบรรทัดที่ 12 จะทำการ Set Bitvector ในตาราง BIL table ในขณะที่อัลกอริทึมการลบกฎจะทำการ reset Bitvector ในตาราง BIL table โดยกำหนดให้เวลาในการปรับปรุงกฎคือ t_{update} เพราะฉะนั้น $t_{update} = t_{add} = t_{del}$

Time complexity เท่ากับ $O\left(\frac{D \cdot W}{PMX} \bullet \frac{N}{WORD}\right)$ และ Storage complexity เท่ากับ $O\left(\frac{D \cdot W}{PMX} \bullet 2^{PMX} \bullet N\right)$

บทที่ 3
อัลกอริทึมการจัดประเภทแพ็กเก็ตตามกฎด้วยวิธีบิตแมปอินเตอร์เซคชันลุกอัพโดยใช้แคช

อัลกอริทึมการจัดประเภทแพ็กเก็ตตามกฎด้วยวิธีบิตแมปอินเตอร์เซคชันลุกอัพโดยใช้แคช เป็นการนำวิธีการจัดประเภทแพ็กเก็ตตามกฎด้วยวิธี Bitmap Intersection Lookup (BIL) มาประยุกต์ร่วมกับหลักการใช้แคช เพื่อเพิ่มประสิทธิภาพด้านความเร็วในการค้นหา

3.1 สถาปัตยกรรมวิธีบิตแมปอินเตอร์เซคชันลุกอัพโดยใช้แคช


packet
<segment 1><segment 2><. . .><segment D>
Cache table
BIL table
Pn ... P0
segsize
AND
N-1
0
rule bitmap
rule prioritization
M
rule database
value/mask
rule table
classify rule
update rule

รูปที่ 3.1 แสดงสถาปัตยกรรมของวิธีบิตแมปอินเตอร์เซคชันลุกอัพโดยใช้แคช

สถาปัตยกรรมของอัลกอริทึมการจัดประเภทแพ็กเก็ตตามกฎด้วยวิธีบิตแมปอินเตอร์เซคชันลุกอัพโดยใช้แคช หรือ BIL using Cache ประกอบด้วย BIL table จำนวน D ตาราง โดยแต่ละตารางประกอบด้วยเพจเรียงต่อกัน มีความกว้างเท่ากับ WORD (WORD คือ ขนาดบิตที่ทำการประมวลในแต่ละครั้ง) จำนวนเพจใน BIL table จึงเท่ากับ (N/WORD) ปัดเศษขึ้น เช่น N เท่า 100 กฎ, WORD เท่ากับ 32 บิต ดังนั้นใน BIL table มีจำนวน เพจ เท่ากับ 100/32 = 3.125 ปัดเศษขึ้นเท่ากับ 4 เพจ แต่ละเพจประกอบด้วยแอดเดรสจำนวน $2^{SegmentSize}$ แอดเดรส (SegmentSize คือ

ขนาดบิตของ segment ที่ถูกแบ่งจากแพ็กเก็ตเฮดเดอร์) เริ่มตั้งแต่ 0 ถึง $2^{SegmentSize} - 1$ แต่ละแอดเดรสประกอบด้วยบิตเรียงต่อกัน เรียกว่า Bitvector โดยบิตที่เรียงกันจะเรียงตรงกับตำแหน่งของกฎตั้งแต่ P*WORD ถึง (P+1)*WORD - 1 เช่น เพจ 0 ประกอบด้วยบิตที่ตรงกับตำแหน่ง (0*32) = 0 ถึง (1*32)-1 = 31 บิตที่มีค่าเป็น "1" คือ บิตที่อยู่ในแอดเดรสที่ตรงกับค่าของกฎ บิตที่มีค่าเป็น "0" คือ บิตที่ไม่อยู่ในแอดเดรสที่ตรงกับค่าของกฎ การเรียงลำดับความสำคัญของกฎเรียงจากมากไปน้อย ตาราง BIL table ถูกสร้างขึ้นจากฐานข้อมูลกฎ (Rule database) ซึ่งเก็บอยู่ในรูปแบบของ value/mask หรือ ช่วง (range) จากนั้นทำการกำหนดค่าให้อยู่ในรูปแบบของ Prefix เพื่อใช้ในการเปลี่ยนค่าของ Bitvector ในตาราง BIL table ต่างๆ แอ็คชันของกฎแต่ละข้อถูกเก็บลงใน Rule table ในส่วนของ Cache table ประกอบด้วยตารางจำนวนเท่ากับ BIL table แต่ละตารางประกอบด้วยแอดเดรสจำนวนเท่ากับ BIL table คือ $2^{SegmentSize}$ แอดเดรส โดยในแต่ละแอดเดรสเก็บ page counter ใช้เป็นเงื่อนไขในการพิจารณาสำหรับการค้นหาแพ็กเก็ต

การค้นหาเริ่มโดยการนำแพ็กเก็ตแบ่งเป็น segment เล็กๆ จำนวน D segments นำแต่ละ segment ไปอ่านค่า page counter จากตาราง Cache table ตาม segment ดังกล่าวเพื่อใช้เป็นเงื่อนไขสำหรับการค้นหาในตาราง BIL table หาก page counter มีค่าเท่ากันทั้งหมด จะเริ่มค้นหาจากเพจดังกล่าวที่เก็บใน page counter หากไม่เท่ากันทำการค้นหาเริ่มต้นที่เพจแรก โดยการอ่านค่า Bitvector จาก BIL table ที่ละเพจแต่ละตาราง นำ Bitvector ที่ได้มาทำการ intersection ด้วยตรรกะ AND เรียกผลลัพธ์นี้ว่า rule bitmap หาก rule bitmap มีค่าไม่เท่ากับ 0 แสดงว่าแพ็กเก็ตดังกล่าวตรงกับกฎในฐานข้อมูลกฎ ทำการเขียนหมายเลขเพจปัจจุบันลงใน page counter ของแต่ละ segment ในตาราง Cache table และหาค่ากฎที่มีความสำคัญสูงสุดเพื่อส่งค่าแอ็คชันจาก Rule table ต่อไป

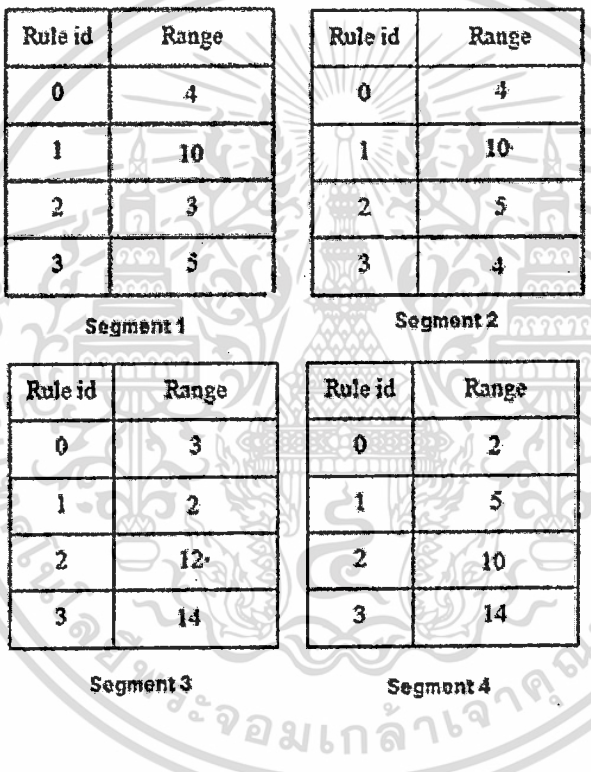
3.1.1 การสร้าง BIL table

สมมติให้ segment ของแพ็กเก็ตมีขนาดเท่ากับ 4 บิต และความกว้างของเพจเท่ากับ 2 บิต ในฐานข้อมูลกฎเก็บค่าดังต่อไปนี้

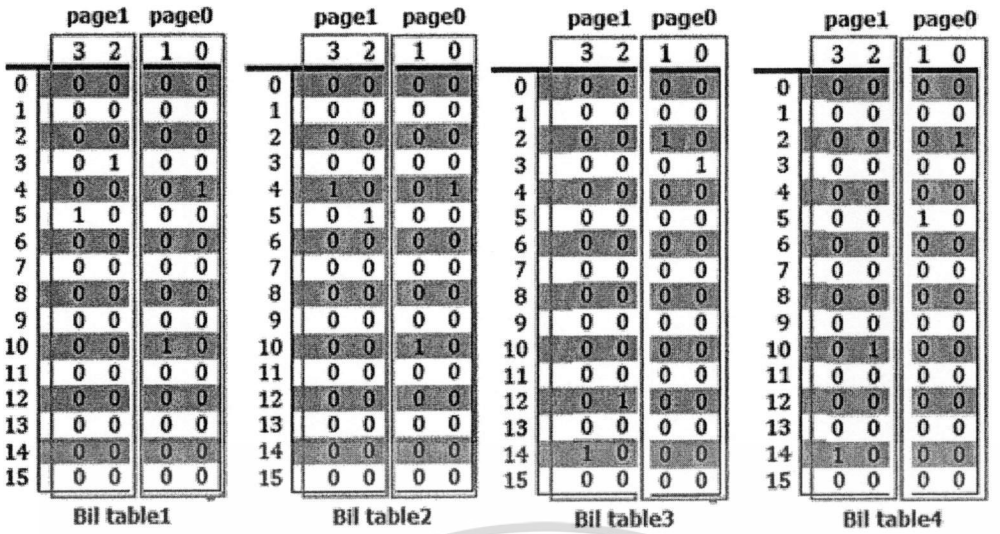
ตาราง 3.1 สมมติการเก็บข้อมูลในฐานข้อมูลกฎ

Rule id	Action	Value	Mask
0	Accept	4.4.3.2	15.15.15.15
1	Drop	10.10.2.5	15.15.15.15
2	Accept	3.5.12.10	15.15.15.15
3	Drop	5.4.14.14	15.15.15.15

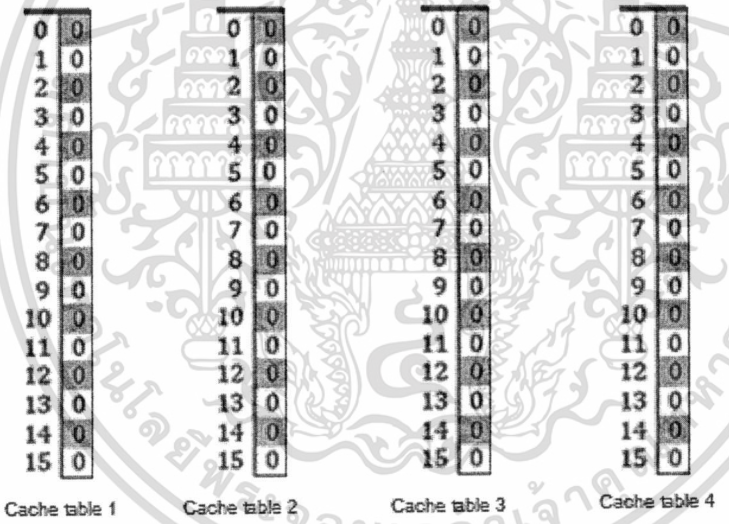
เริ่มต้นโดยการสร้างตาราง BIL table เบื้องต้นจำนวน 4 ตาราง โดยในแต่ละตารางประกอบด้วย เพจ จำนวน 2 เพจ มีแอดเดรสตั้งแต่ 0 ถึง 15 มีค่า Bitvector เริ่มต้นเท่ากับ "0" การสร้างตาราง Cache table โดยในแต่ละตารางประกอบด้วยแอดเดรสตั้งแต่ 0 ถึง 15 มีค่า page counter เริ่มต้น เท่ากับ "0" สำหรับการเปลี่ยนค่า Bitvector นั้นจำเป็นต้องกำหนดค่ากฎให้อยู่ในรูปแบบของ Prefix ด้วยวิธีที่ได้แสดงไว้ข้างต้น เช่น rule id 0 ใน Segment แรก value คือ 4 (0100) mask คือ 15 (1111) แปลงให้อยู่ในรูปแบบช่วงของ Prefix ได้เป็น 0100 นั้นคือ 4 จากนั้นทำการเปลี่ยนค่า Bitvector ที่ตรงกับตำแหน่งของกฎที่ 0 ใน BIL table1 แอดเดรสที่ 4 ให้มีค่าเท่ากับ "1" รูปที่ 3.2 แสดงการแปลงข้อมูลกฎให้อยู่ในรูปแบบ Prefix



รูปที่ 3.2 แสดงรูป Prefix



รูปที่ 3.3 แสดงการเก็บข้อมูลใน BIL table



รูปที่ 3.4 แสดงการเก็บข้อมูลใน Cache table

3.1.2 การค้นหา (lookup)

การค้นหาเริ่มโดยการนำแพ็กเก็ตแบ่งเป็น segment เล็กๆ จำนวน D segments นำแต่ละ segment ไปอ่านค่า page counter จากตาราง Cache table ตาม segment ดังกล่าวเพื่อใช้เป็นเงื่อนไขสำหรับการค้นหาในตาราง BIL table หาก page counter มีค่าเท่ากันทั้งหมด จะเริ่มค้นหาจากเพจดังกล่าวที่เก็บใน page counter หากไม่เท่ากันทำการค้นหาเริ่มต้นที่เพจแรก โดยการอ่านค่า Bitvector จาก BIL table ที่ละเพจแต่ละตาราง นำ Bitvector ที่ได้มาทำการ intersection ด้วยตรรกะ AND เรียกผลลัพธ์นี้ว่า rule bitmap หาก rule bitmap มีค่าไม่เท่ากับ "0" แสดงว่าพบว่าตรง

กับกฎในฐานข้อมูลกฎ ทำการเขียนหมายเลขเพจปัจจุบันลงใน page counter ของแต่ละ segment ในตาราง Cache table และหาค่ากฎที่มีความสำคัญสูงสุดเพื่อส่งค่าแอ็คชันจาก Rule table ต่อไป ตัวอย่างการค้นหาแพ็กเก็ต 3.5.12.10 โดยในที่นี้ตั้งสมมติฐานว่า ความกว้างของ เพจ เท่ากับ 2 บิต ทำการแบ่งแพ็กเก็ตเป็น segment เล็กๆ ประกอบด้วย 3, 5, 12, 10 นำค่า segment ไปอ่านค่า page counter จากตาราง Cache table ปรากฏดังนี้ 0, 0, 0, 0 พิจารณาหาค่าต่ำสุด เท่ากับ 0 ดังนั้นในการค้นหาสำหรับแพ็กเก็ต 3.5.12.10 จะเริ่มต้นที่เพจ 0 นำ Bitvector จากแต่ละตารางตามลำดับดังนี้ 00, 00, 00 และ 00 (Bitvector ที่แสดงในรูปที่ 3.3) มาทำการ intersection ด้วยตรรกะ AND ปรากฏว่ามีค่าเป็น 0 จึงทำการค้นหาในเพจต่อไป นำ Bitvector จากแต่ละตารางตามลำดับดังนี้ 01, 01, 01 และ 01 มีผลลัพธ์ที่เรียกว่า rule bitmap พบว่ามีเลข “1” ทำการเขียนหมายเลขเพจลงใน Cache table ตามที่ได้อ่านค่า page counter มาในข้างต้น ดังนั้นค่า page counter ในตามราง Cache table1 ในแอดเดรสที่ 3, ตาราง Cache table2 ในแอดเดรสที่ 5, ตาราง Cache table3 ในแอดเดรสที่ 12 และตาราง Cache table4 ในแอดเดรสที่ 10 เก็บค่าหมายเลขเพจ เท่ากับ 1 จากนั้นทำการหาตำแหน่งของบิตที่มีค่าเป็น “1” ของบิตที่มีความสำคัญสูงสุด พบว่าอยู่ตรงกับตำแหน่งของกฎหมายเลข 2 (rule id 2) จากนั้นนำค่า rule id ไปหาค่าแอ็คชันใน Rule table เพื่อส่งค่าแอ็คชัน ดังนั้นเมื่อมีแพ็กเก็ตที่ซ้ำกันนี้ผ่านเข้ามา การค้นหาเพจเริ่มต้นควรเริ่มต้นที่เพจ 1 ข้ามเพจ 0 ไปตามหลักของการใช้แคช

```
1. if ( ค่า page counter ของแต่ละ segment จาก cache table เท่ากัน ){// cache hit
2.     ให้เพจเริ่มต้นในการค้นหาเท่ากับเพจที่เคยก่อนหน้า
3. }else{// cache miss
4.     ให้เพจเริ่มต้นในการค้นเท่ากับเพจแรก (เพจ0)
5. }
6. for (i = เพจเริ่มต้น; i < PAGE; i++) {
7.     for (j = 0; j < D;  j++) {
8.         rule_bitmap &= bil_table[j][split[j].value][i];
9.     }
10.    if (rule_bitmap != 0) {
11.        ใส่ค่า page counter ใน cache table ของแต่ละ segment เก็บค่าเพจปัจจุบันที่ได้พบกฎ
12.        ทำการหาค่ากฎที่มีความสำคัญสูงที่สุด
13.        ส่งค่าแอคชันที่ตรงกับกฎดังกล่าวใน Rule table
14.    }
15. }
16. ส่งค่า default แอคชั่นกลับไป
```

รูปที่ 3.5 อัลกอริทึมของการค้นหา (lookup)

3.1.3 การปรับปรุงกฎ

การเพิ่มกฎ - การเพิ่มกฎเข้าไปใน Rule database นั้นสามารถทำการปรับปรุงกฎโดยการพิจารณาว่าหมายเลขกฎที่เพิ่มเข้าไปใหม่นั้นมี เพจ รองรับหรือไม่ หากมีเพจรองรับแล้วก็เพียงทำการเปลี่ยนค่า Bitvector ในแอดเดรสนั้นๆให้มีค่าเป็น "1" แต่หากกฎหมายเลขดังกล่าวยังไม่มี เพจรองรับเราต้องทำการ allocate เพจ ขึ้นมารองรับก่อนจากนั้นจึงทำการเปลี่ยนค่า Bitvector ตัวอย่าง ต้องการเพิ่มกฎลงไปในฐานข้อมูลกฎ

ตารางที่ 3.2 แสดงกฎที่ต้องการเพิ่ม

Rule id	Action	Value	Mask
4	Accept	12.12.12.12	15.15.15.15

ตารางที่ 3.3 แสดงรูปแบบ Prefix จากกฎที่ 4

Rule id	Range
4	12

จากรูปที่ 3.3 ปรากฏว่าไม่มี เพจ ที่รองรับกฎหมายเลข 4 ดังนั้นจำเป็นต้องทำการ allocate เพจ ใหม่ขึ้นมาได้ผลลัพธ์ดังรูปที่ 3.6

	page2		page1		page0	
	5	4	3	2	1	0
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	1	0	0
4	0	0	0	0	0	1
5	0	0	1	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	1	0
11	0	0	0	0	0	0
12	0	0	0	0	0	0
13	0	0	0	0	0	0
14	0	0	0	0	0	0
15	0	0	0	0	0	0

Bil table1

รูปที่ 3.6 แสดงการ allocate เพจเพื่อรองรับกฎข้อที่ 4

จากนั้นทำการเปลี่ยนค่า Bitvector ของแอดเดรสที่ 12 ให้ค่าบิตเป็น "1" ดังนั้น BIL table1 เก็บค่าดังรูป 3.7 ทำแบบเดียวกันกับทุกตาราง BIL table ตามรูปแบบ Prefix ในแต่ละ segment

	page2		page1		page0	
	5	4	3	2	1	0
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	1	0	0
4	0	0	0	0	0	1
5	0	0	1	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	1	0
11	0	0	0	0	0	0
12	0	1	0	0	0	0
13	0	0	0	0	0	0
14	0	0	0	0	0	0
15	0	0	0	0	0	0

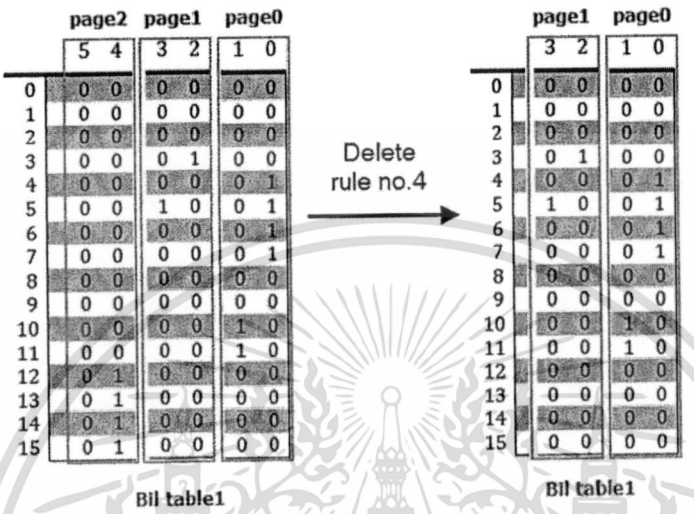
Bil table1

รูปที่ 3.7 แสดงการปรับค่า Bitvector กฎข้อที่ 4

```
1.  if ( ไม่มีเพจที่รองรับกฎที่ต้องการเพิ่มใหม่ ) {
2.       allocate เพจใหม่เพื่อรองรับกฎดังกล่าว
3.  }
4.  if ( rule_table ยังไม่เก็บข้อมูลกฎ ) {
5.      เก็บข้อมูลลงใน rule table
6.      prefix = convert_to_prefix(source_ip.address.s_addr, source_ip.addressmask.s_addr);
7.      for(i=0; i < D ;i++){
8.          value = prefix [i].value;
9           mask = prefix [i].mask;
10.         for(j=(value & mask);j<=((value & mask) + (~mask));j++){
11.             setBitvector(i,j,rule_id);
12.         }
13.     }
14. }else if(( rule_table มีการเก็บข้อมูลกฎอื่นอยู่แล้ว){
15.      return -1;
16. }
```

รูปที่ 3.8 อัลกอริทึมการเพิ่มกฎ

การลบกฎ - เมื่อมีการลบกฎออกจากฐานข้อมูลกฎ ทำการอ่านค่ากฎจากฐานข้อมูลกฎ แล้วเปลี่ยนให้อยู่ในรูปแบบของ Prefix ทำการเปลี่ยนค่า Bitvector ตำแหน่งที่ตรงกับกฎเท่ากับ "0" จากนั้นทำการตรวจสอบว่า เพจ ดังกล่าวมีค่า Bitvector ทั้งหมดเป็น "0" หรือไม่ หากใช่ทำการลบ (de-allocate) เพจ ดังกล่าวเพื่อเป็นการใช้เนื้อที่ให้คุ้มค่าที่สุด



รูปที่ 3.9 แสดงโครงสร้างข้อมูลที่ลบกฎข้อที่ 4

```
1.  if ( rule_table ไม่ว่าง ){
2.      prefix = convert_to_prefix(source_ip.address.s_addr, source_ip.addressmask.s_addr);
3.      for(i=0; i < D ;i++){
4.          value = prefix[i].value;
5.          mask = prefix[i].mask;
6.          for(j=(value & mask);j<=((value & mask) + (~mask));j++){
7.              resetBitvector(i,j,rule_id);
8.          }
9.      }
10.     rule_table[rule_id].action เก็บค่าว่าง;
11. }else if (( rule_table ว่างอยู่แล้ว){
12.     ไม่สามารถลบกฎได้
13.     return -1;
14. }
15. if( เพจดังกล่าวเป็นเพจสุดท้าย && Bitvector ของเพจดังกล่าว = = 0 ){
16.     ทำการ de-allocate เพจดังกล่าว
17. }
```

รูปที่ 3.10 อัลกอริทึมการลบกฎ

บทที่ 4

การทดลองและผลการทดลอง

ในบทนี้กล่าวถึงการทดลองและผลการทดลองของการวัดประสิทธิภาพการจัดประเภทแพ็กเก็ต เพื่อเปรียบเทียบประสิทธิภาพความเร็วในการค้นหา ความเร็วในการปรับปรุงกฎ และเนื้อที่ในการเก็บข้อมูล เปรียบเทียบกับการจัดประเภทแพ็กเก็ตตามกฎด้วยวิธี Bitmap Intersection Lookup

4.1 การออกแบบการทดลอง

การวัดประสิทธิภาพของการจัดประเภทแพ็กเก็ตในงานวิจัยฉบับนี้ ทำโดยการพัฒนาอัลกอริทึมด้วยตัวแปรภาษา GCC เวอร์ชัน 4.3 บนระบบปฏิบัติการ FreeBSD เวอร์ชัน 6.4 บนคอมพิวเตอร์ CPU Pentium 4 1.6 GHz ฮาร์ดดิสก์ 80 GB โดยการทดลองจะใช้ฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (ขนาด 32 บิต), WORD มีขนาด 32 บิต ตามสถาปัตยกรรมคอมพิวเตอร์ที่ใช้การทดลอง ทำการศึกษาประสิทธิภาพด้านต่างๆ ประกอบด้วย 1) ความเร็วในการค้นหา 2) ด้านความเร็วในการปรับปรุงกฎ และ 3)การใช้เนื้อที่ในการเก็บข้อมูล โดยสำหรับการวัดประสิทธิภาพด้านความเร็วในการค้นหาจะทำการจำลองข้อมูลแพ็กเก็ต โดยทำการจำลองข้อมูลแพ็กเก็ตเป็น 3 ลักษณะ ได้แก่ 1) Repeat first rule 2) Repeat last rule และ 3) Random rule โดยแต่ละลักษณะมีรายละเอียดดังนี้

1) Repeat first rule กำหนด Source IP Address ของแพ็กเก็ตให้ตรงกับกฎข้อแรกในฐานข้อมูลกฎเรียงติดกันทั้งหมด

2) Repeat last rule กำหนด Source IP Address ของแพ็กเก็ตให้ตรงกับกฎข้อสุดท้ายในฐานข้อมูลกฎเรียงติดกันทั้งหมด

3) Random rule กำหนด Source IP Address ของแพ็กเก็ตโดยการสุ่มกฎจากฐานข้อมูลโดยไม่ให้แพ็กเก็ตที่ตรงกับกฎเดียวกันเรียงติดกัน

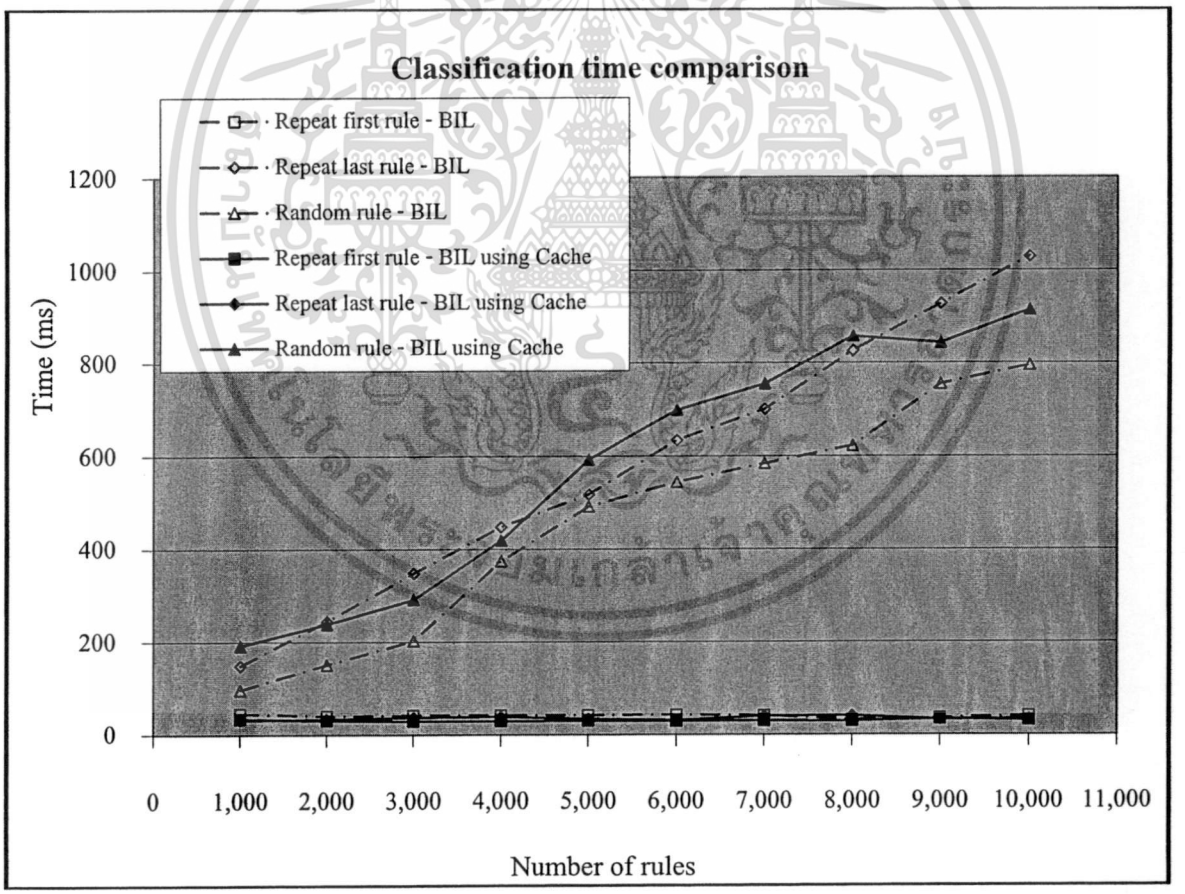
จากลักษณะที่กล่าวมานั้นเมื่อเปรียบเทียบกับงานวิจัยของ Bitmap Intersection Lookup นั้นลักษณะของ Repeat first rule จะตรงกับกรณี Best case ในการค้นหากฎ, Repeat last rule จะตรงกับกรณี Worst case และ Random rule จะตรงกับกรณี Average case การวัดประสิทธิภาพด้านความเร็วในการปรับปรุงกฎแบ่งออกเป็น 2 ส่วน คือ 1) การเพิ่มกฎ 2) การลบกฎ แบ่งออกเป็น 2 กรณีได้แก่การลบที่ทำให้เกิดการสร้างโครงสร้างใหม่(มีการ de-allocate เพจ) และการลบกฎที่ไม่ทำให้เกิดการสร้างโครงสร้างใหม่ (ไม่มีการ de-allocate เพจ)

4.2 การทดลองวัดประสิทธิภาพ

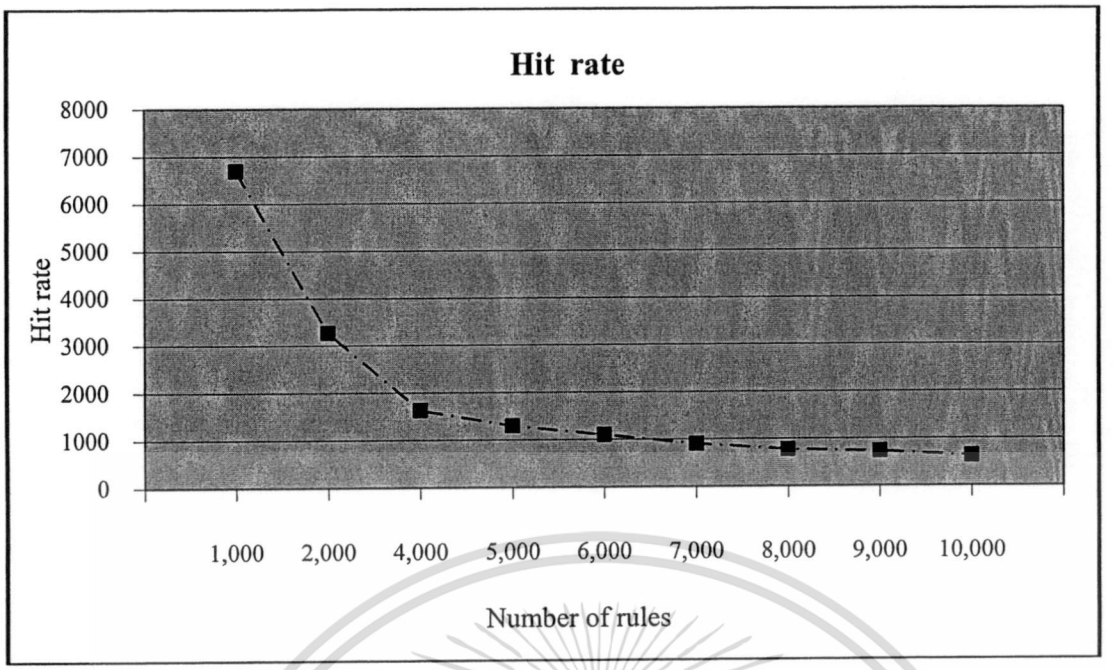
4.2.1 การเปรียบเทียบเวลาในการค้นหา

วิธีการทดลอง

1. กำหนดฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (ขนาด 32 บิต)
2. สร้างฐานข้อมูลโดยให้จำนวนกฎเท่ากับ 1,000, 2,000, 3,000 ... 10,000 ข้อ
3. สร้างข้อมูลแพ็กเก็ตจำนวน 100,000 แพ็กเก็ต โดยแบ่งเป็น 3 ลักษณะดังที่กล่าวไว้ข้างต้น ได้แก่ 1) Repeat first rule 2) Repeat last rule และ 3) Random rule
4. สร้างข้อมูลแพ็กเก็ตจำนวน 100,000 แพ็กเก็ต จากฐานข้อมูลกฎจำนวน 9,000 กฎ โดยมีลักษณะการซ้ำกันเป็นช่วง 10, 20 ถึง 100
5. สร้างข้อมูลแพ็กเก็ตจำนวน 100,000 แพ็กเก็ต จากฐานข้อมูลกฎจำนวน 9,000 กฎ โดยมีลักษณะการซ้ำกันคิดเป็นร้อยละ 10, 20, ... 100

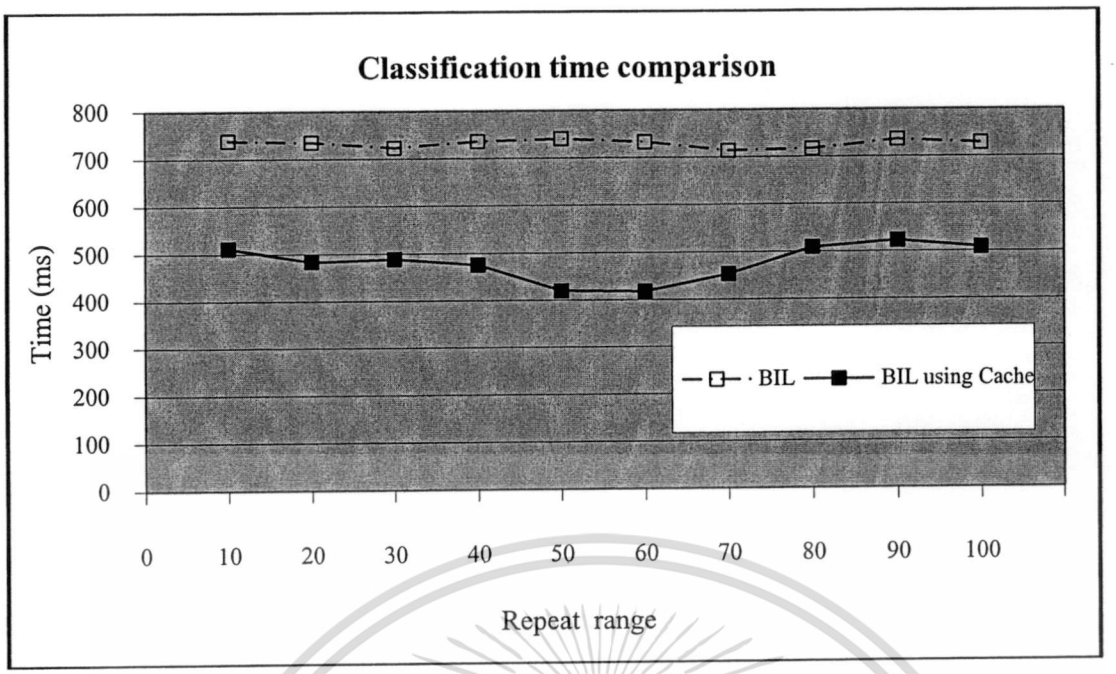


รูปที่ 4.1 แสดงผลเปรียบเทียบเวลาในการค้นหาระหว่าง BIL และ BIL using Cache



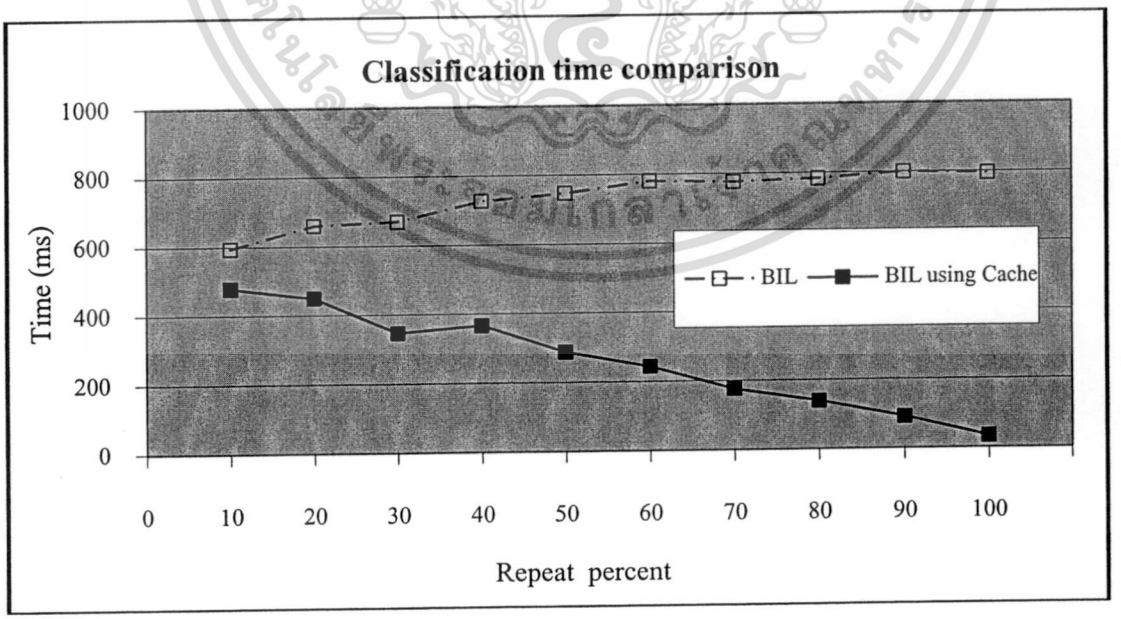
รูปที่ 4.2 แสดงการเกิดกรณี Hit rate

รูปที่ 4.1 แสดงผลการเปรียบเทียบเวลาในการค้นหาข้อมูลแพ็กเก็ตเปรียบเทียบระหว่าง BIL และ BIL using Cache โดยที่รูปที่ 4.1 เปรียบเทียบการค้นหาของแพ็กเก็ตทั้ง 3 ลักษณะ เห็นได้ว่าการเวลาในการค้นหาของ BIL ในลักษณะ Repeat first rule หรือที่ตรงกับกรณี Best case ของ BIL นั้น มีประสิทธิภาพในการค้นหาดีที่สุด ตามด้วยการการค้นหาของ BIL using Cache ในลักษณะ Repeat first rule และ Repeat last rule และ การค้นหาในลักษณะ Random rule ของ BIL using Cache ซึ่งใกล้เคียงกับลักษณะ Repeat last rule ของ BIL หรือ กรณี Worst case ของ BIL นั้นเอง การค้นหาของ BIL using Cache จะมีประสิทธิภาพดีในการค้นหาของแพ็กเก็ตในลักษณะแบบ Repeat ไม่ว่าจะเป็นการ Repeat ในตำแหน่งกฎข้อใดในฐานข้อมูลก็ตาม โดยมี Time complexity เท่ากับ $O(D+D)$ นอกจากนี้ผลการทดลองยังแสดงให้เห็นอย่างชัดเจนว่าจำนวนกฎมีผลต่อเวลาในการค้นหาเนื่องจากยิ่งค่ากฎเพิ่มขึ้นทำให้จำนวน PAGE เพิ่มขึ้น เวลาในการค้นหาขึ้นตามไปด้วย รูปที่ 4.2 แสดงการเกิดกรณี Hit rate แสดงให้เห็นว่าจำนวนของกฎส่งผลต่อ Hit rate ยิ่งจำนวนกฎสูงทำให้ Hit rate ต่ำลง



รูปที่ 4.3 แสดงผลเปรียบเทียบเวลาในการค้นหาของแพ็กเก็ตในลักษณะ Repeat เป็นช่วง

รูปที่ 4.3 แสดงผลการเปรียบเทียบเวลาในการค้นหาของแพ็กเก็ตที่มีลักษณะการซ้ำกันเป็นช่วงสลับกับการสุ่มกฎ โดยแบ่งออกเป็นช่วง 10, 20, 30 ... 100 โดยกำหนดจำนวนแพ็กเก็ตเท่ากับ 100,000 แพ็กเก็ต บนฐานข้อมูลกฎจำนวน 9,000 กฎ ถือว่ามีลักษณะการซ้ำกันเป็นร้อยละ 50 ของแพ็กเก็ตทั้งหมด



รูปที่ 4.4 แสดงผลเปรียบเทียบเวลาในการค้นหาของแพ็กเก็ตในลักษณะ Repeat เป็นร้อยละ

ตารางที่ 4.1 เปรียบเทียบเวลาในการค้นหา และ Hit rate

ร้อยละ (แพ็กเก็ตซ้ำกัน)	Hit rate	เวลาในการค้นหา (ms.)	
		BIL	BIL using Cache
10	10,656	594.3832	478.6839
20	20,590	660.3568	449.9026
30	30,477	670.0786	347.0518
40	40,431	727.7846	367.2060
50	50,382	748.3530	287.3269
60	60,298	780.9996	243.8100
70	70,221	776.3622	176.3965
80	80,140	783.3874	139.1944
90	90,071	800.5088	91.3929
100	100,000	795.6508	33.5197

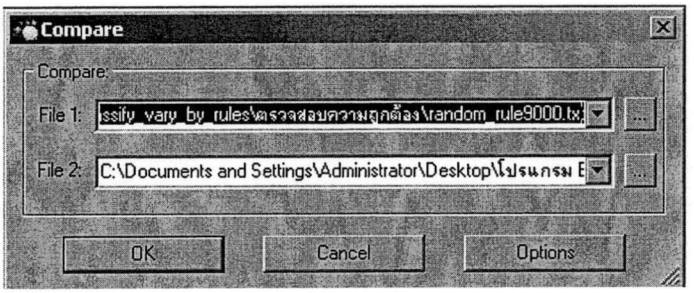
จากรูปที่ 4.4 และตารางที่ 4.1 พบว่าการซ้ำกันของแพ็กเก็ต หรือ Hit rate ยิ่งมีค่าสูงจะทำให้ประสิทธิภาพของการค้นหาของอัลกอริทึมการจัดประเภทแพ็กเก็ตตามกฎด้วยวิธีบิตแมปอินเตอร์เซคชันลุกอัพโดยใช้แคช ยิ่งมีประสิทธิภาพ ตั้งแต่ซ้ำกันร้อยละ 10 หรือ Hit rate 10% ก็มีประสิทธิภาพสูงกว่าวิธีการเดิม

จากผลการทดลองวัดประสิทธิภาพด้านความเร็วในการค้นหาพบว่าเมื่อแพ็กเก็ตที่เข้ามามีลักษณะที่ซ้ำกันเรียงติดกัน จะทำให้การค้นหาเกิด Hit rate สูงจะใช้เวลาในการค้นหาเร็ว ดังนั้น BIL using Cache มีประสิทธิภาพสูงขึ้นในกรณีดังกล่าวเป็นไปตามสมมติฐานที่ตั้งไว้

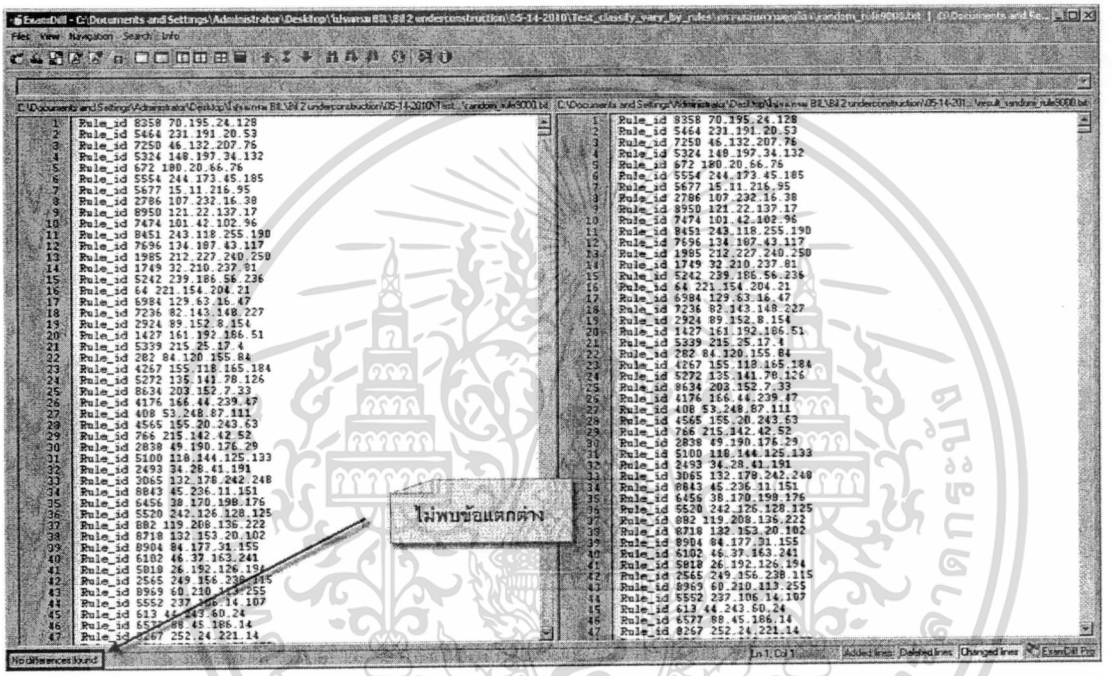
4.2.2 พิสูจน์ความถูกต้องในการค้นหากฎ

วิธีการทดลอง

1. กำหนดฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (ขนาด 32 บิต)
2. สร้างฐานข้อมูลโดยให้จำนวนกฎเท่ากับ 1,000, 2,000, 3,000 ... 10,000 ข้อ
3. สร้างข้อมูลแพ็กเก็ตจำนวน 100,000 แพ็กเก็ต และ เก็บลงใน Text ไฟล์อีกที่หนึ่ง โดยมีรูปแบบดังนี้ Rule_id หมายเลขกฎ Source IP Address เช่น Rule_id 8358 70.195.24.128
4. ในการค้นหาให้สร้างไฟล์ที่ชื่อว่า result.txt เพื่อใช้ในการตรวจสอบโดยมีรูปแบบเหมือนกับที่เก็บใน Text ไฟล์ ในข้อที่ 3
5. ใช้โปรแกรม ExamDiff ตรวจสอบทั้ง 2 ไฟล์ว่ามีข้อแตกต่างกันหรือไหม



รูปที่ 4.5 เลือกไฟล์ที่ต้องการตรวจสอบ



รูปที่ 4.6 แสดงผลการเปรียบระหว่างไฟล์ทั้งสองไฟล์

จากรูปที่ 4.6 พบว่าทั้ง 2 ไฟล์ไม่พบข้อแตกต่างของทั้ง 2 ไฟล์ แสดงว่าผลลัพธ์ในการค้นหาแพ็กเก็ตทั้งหมด 100,000 แพ็กเก็ตถูกต้อง

4.2.3 การเปรียบเทียบเวลาในการปรับปรุงกฎ

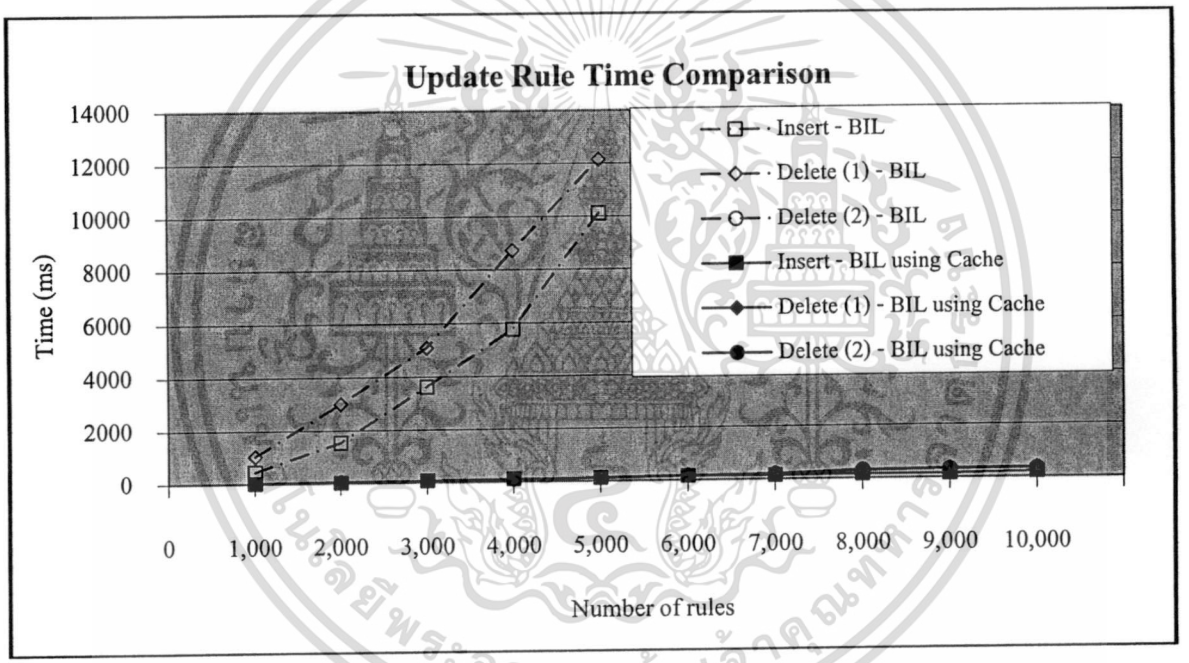
การปรับปรุงกฎประกอบด้วย 1) การเพิ่มกฎ 2) การลบกฎ กรณีการเพิ่มของ BIL นั้นมีการกำหนดจำนวนกฎที่สามารถรองรับได้ไว้อย่างชัดเจน หากจำเป็นต้องการเพิ่มกฎที่เกินจากค่าที่กำหนดไว้นั้น จำเป็นต้องสร้างโครงสร้างใหม่ทุกครั้ง กรณีการลบกฎแบ่งออกเป็น 2 กรณี คือ 1) การลบกฎจากมากไปน้อยเรียงจากข้อสุดท้ายไปยังข้อแรก 2) การลบกฎแบบสุ่มโดยให้แต่ละยังคงเก็บค่า Bitvector ที่ไม่เท่ากับ 0 (ทำให้ไม่สามารถ de-allocate เพจนั้นๆได้)

วิธีการทดลองการเพิ่มกฎ

1. กำหนดฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (ขนาด 32 บิต)
2. สร้างฐานข้อมูลโดยให้จำนวนกฎเท่ากับ 1,000, 2,000, 3,000 … 10,000 ข้อ
3. กำหนดให้ BIL table เบื้องต้นสามารถรองรับกฎ จำนวน 32 กฎ หรือ 1 เพจเหมือนใน BIL using Cache

วิธีการทดลองการลบกฎ

1. กำหนดฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (ขนาด 32 บิต)
2. สร้างฐานข้อมูลโดยให้จำนวนกฎเท่ากับ 2,000, 3,000 … 10,000 ข้อ
3. กำหนดให้ลบกฎจากมากไปน้อยโดยเริ่มจากกฎข้อสุดท้าย
4. กำหนดให้ลบกฎโดยการสุ่มโดยไม่จำเป็นต้องสร้างโครงสร้างใหม่



รูปที่ 4.7 แสดงผลเปรียบเทียบเวลาในการปรับปรุงกฎ

รูปที่ 4.7 ในกรณีการเพิ่มกฎ ผลการทดลองเป็นไปตามสมมติฐานว่า BIL ใช้เวลาในการเพิ่มกฎสูงกว่าเมื่อเริ่มต้นจากโครงสร้างขนาดเดียวกันกับ BIL using Cache เนื่องจาก BIL จำเป็นต้องสร้างโครงสร้างใหม่เมื่อมีการเพิ่มกฎที่เกินจากค่าที่จำนวนกฎกำหนดไว้ ทำให้ใช้เวลามากกว่า ส่วนในกรณีของการลบกฎ Delete (1) คือ การลบกฎที่มีกรณีของการ de-allocate เพจ, Delete (2) คือ การลบกฎที่ไม่มีการ de-allocate เพจตามที่ได้ที่ตั้งสมมติฐานว่าการลบที่มีการ de-allocate เพจนั้นจะใช้เวลาในการลบกฎสูงกว่า ผลการทดลองแสดงให้เห็นว่า การลบกฎจากตำแหน่งกฎในข้อสุดท้ายไปกฎข้อแรกที่มีการ de-allocate เพจนั้น BIL ใช้เวลาในการลบกฎสูงกว่า แต่สำหรับกรณีการลบกฎที่ไม่มี de-allocate ของ BIL กลับใช้เวลาในการปรับปรุงใกล้เคียงกับ BIL using Cache

4.2.4 การเปรียบเทียบการใช้เนื้อที่ในการเก็บข้อมูล

วิธีการทดลองการลบกฎ

1. กำหนดฐานข้อมูลกฎจำนวน 1 ฟิลด์ เป็น Source IP Address (ขนาด 32 บิต)
2. สร้างฐานข้อมูลโดยให้จำนวนกฎเท่ากับ 1,000, 2,000, ... 10,000 ข้อ
3. บันทึกเนื้อที่ที่ใช้ในการเก็บข้อมูล

ตารางที่ 4.2 การใช้เนื้อที่ของ BIL และ BIL using Cache

จำนวนกฎ	การใช้พื้นที่ (BIL table)		
	BIL	BIL using Cache	
	BIL table	BIL table	Cache table
1,000 กฎ	131,072 Bytes.	131,072 Bytes.	2,048 Bytes.
2,000 กฎ	258,048 Bytes.	258,048 Bytes.	2,048 Bytes.
3,000 กฎ	385,024 Bytes.	385,024 Bytes.	2,048 Bytes.
4,000 กฎ	512,000 Bytes.	512,000 Bytes.	2,048 Bytes.
5,000 กฎ	643,072 Bytes.	643,072 Bytes.	2,048 Bytes.
6,000 กฎ	770,048 Bytes.	770,048 Bytes.	2,048 Bytes.
7,000 กฎ	897,024 Bytes.	897,024 Bytes.	2,048 Bytes.
8,000 กฎ	1,024,000 Bytes.	1,024,000 Bytes.	2,048 Bytes.
9,000 กฎ	1,155,072 Bytes.	1,155,072 Bytes.	2,048 Bytes.
10,000 กฎ	1,282,048 Bytes.	1,282,048 Bytes.	2,048 Bytes.

จากตารางที่ 4.2 เปรียบเทียบการใช้เนื้อที่ของวิธีการเดิมและวิธีการได้นำเสนอใหม่ ส่วนของโครงสร้างข้อมูล BIL table จากสมมติฐานที่ตั้งไว้ว่าวิธีอัลกอริทึมการจัดประเภทแพ็กเก็ตตามกฎด้วยวิธีบิตแมปอินเตอร์เซคชันลุกอัพโดยใช้แคชจะใช้เนื้อที่สูงกว่าวิธีการเดิม จากผลการทดลองพบว่าใช้เนื้อที่เท่ากันในส่วนของ BIL table เนื่องจากการพัฒนาโปรแกรมไม่สามารถเก็บข้อมูล Bitvector เรียงต่อกันเท่ากับจำนวนกฎลงในตัวแปรเดียวกันได้หมด ดังนั้นจำเป็นต้องแบ่งเป็นชุดของ Bitvector เหมือนการเก็บลงในเพจ ทำให้การใช้เนื้อที่ในการเก็บข้อมูล BIL table เท่ากัน แต่ในอัลกอริทึมการจัดประเภทแพ็กเก็ตตามกฎด้วยวิธีบิตแมปอินเตอร์เซคชันลุกอัพโดยใช้แคชจะเพิ่มส่วนของการเก็บข้อมูล Cache table ทำให้ใช้เนื้อที่ทั้งหมดมากกว่าการจัดประเภทแพ็กเก็ตวิธีเดิมเป็นไปตามสมมติฐาน

บทที่ 5
สรุปผลการวิจัยและข้อเสนอแนะ

5.1 สรุปผลการวิจัย

งานวิจัยนี้ได้นำเสนออัลกอริทึมการจัดประเภทแพ็กเก็ตตามกฎด้วยวิธีบิตแมปอินเตอร์เซคชันลุกอัพโดยใช้แคช ก่อนอื่นเพื่อไม่ให้เกิดความสับสนขอกล่าวว่าหลักการใช้แคชในที่นี้มีความเร็วในการเข้าถึงข้อมูลในแคชเท่ากับการเข้าในถึง Bitvector ของตาราง BIL table เนื่องจากใช้หน่วยความจำชนิดเดียวกัน จากการศึกษาและทดลองวัดประสิทธิภาพประกอบด้วยประเด็นดังนี้

ด้านความเร็วในการค้นหา สำหรับกรณีแพ็กเก็ตที่ผ่านเข้ามามีลักษณะเหมือนกันเรียงติดกัน (Repeat rule) ทำให้เกิด Hit rate สูง มีประสิทธิภาพสูงกว่าการจัดประเภทแพ็กเก็ตแบบเดิม แต่หากกรณีที่แพ็กเก็ตไม่ซ้ำกันเลยจะทำให้เกิด Miss rate สูง ประสิทธิภาพลดลงเล็กน้อย

ด้านความเร็วในการปรับปรุงกฎมีประสิทธิภาพสูง ใช้เวลาในการปรับปรุงกฎ (เพิ่มกฎ, ลบกฎ) น้อยลงกว่า BIL แต่หากในกรณีของการลบกฎที่ไม่มีการสร้างโครงสร้างใหม่ เวลาที่ใช้ในการปรับกฎจะไม่แตกต่างกัน

ด้านประสิทธิภาพในการใช้เนื้อที่ในการเก็บข้อมูล พบว่าใช้เนื้อที่ในการเก็บข้อมูลมากกว่าวิธีการเดิมเล็กน้อยเพิ่มในส่วนของการเก็บข้อมูล Cache table โดยมีเนื้อที่ในการเก็บ BIL table เท่าเดิม

ดังนั้นจึงสามารถสรุปได้ว่าอัลกอริทึมการจัดประเภทแพ็กเก็ตตามกฎด้วยวิธีบิตแมปอินเตอร์เซคชันลุกอัพโดยใช้แคชมีประสิทธิภาพสูงขึ้น เมื่อเทียบกับการจัดประเภทแพ็กเก็ตตามกฎด้วยวิธี Bitmap Intersection Lookup ในด้านของความเร็วในการค้นหาและความเร็วในการปรับปรุงกฎ การประยุกต์หลักการใช้แคชสามารถช่วยเพิ่มประสิทธิภาพให้กับการจัดประเภทแพ็กเก็ตได้ผล

5.2 ข้อเสนอแนะและงานที่จะทำเพิ่มเติม

งานวิจัยที่ได้นำเสนอสามารถพิสูจน์ว่าการประยุกต์หลักการใช้แคชสามารถนำมาเพิ่มประสิทธิภาพให้กับการจัดประเภทแพ็กเก็ตได้ แต่ในงานวิจัยฉบับนี้มีการใช้เนื้อที่ในการเก็บข้อมูลของแคชคงที่ ทำให้การเกิด Hit rate ไม่สม่ำเสมอ หากมีจำนวนกฎสูงการเกิด Hit rate จะลดลงตามผลการทดลองในตารางที่ 4.1 ดังนั้นหากต้องการเพิ่มประสิทธิภาพให้กับการใช้แคชของงานวิจัยนี้ ควรทำการวิจัยในส่วนของการใช้พื้นที่เก็บข้อมูลของแคชที่เหมาะสมกว่าในรายงานฉบับนี้ คาดว่าจะสามารถช่วยเพิ่มประสิทธิภาพของการค้นหาให้สูงขึ้นด้วย เนื่องจากจะ

ทำให้เกิด Hit rate สูงขึ้นก่อนที่ทำการทดลองควรมีการวิเคราะห์ทฤษฎีตามแนวคิดในภาคผนวก ค. และพัฒนาอัลกอริทึมดังกล่าวเป็น ไฟร์วอลล์ หรือ เราเตอร์ เพื่อทดลองทำงานบนเครือข่ายจริง เพื่อเป็นการพิสูจน์ว่าอัลกอริทึมดังกล่าวสามารถนำไปใช้บนเครือข่ายจริงได้

บรรณานุกรม

[1] Akharin Khunkitti and Nuttachot Promrit, **"Bitmap Intersection Lookup (BIL): A Packet Classification's Algorithm With Rules Updating,"** Proceedings of International Conference on Control Automation and System 2005, Gyeonggi-Do, Korea, 2-5 June 2006.

[2] J. van Lunteren and A.P.J. Engbersen, **"Multi-field Packet Classification Using Ternary CAM,"** IEE Electronics Letters, Vol. 38, No 1, Jan 2002.

[3] P. Gupta and N. McKeown, **"Algorithms for Packet Classification,"** IEEE Netw, 2001.

[4] P. Gupta and N. McKeown, **"Packet Classification on Multiple Fields,"** In Proc. ACM Sigcomm, 1999.

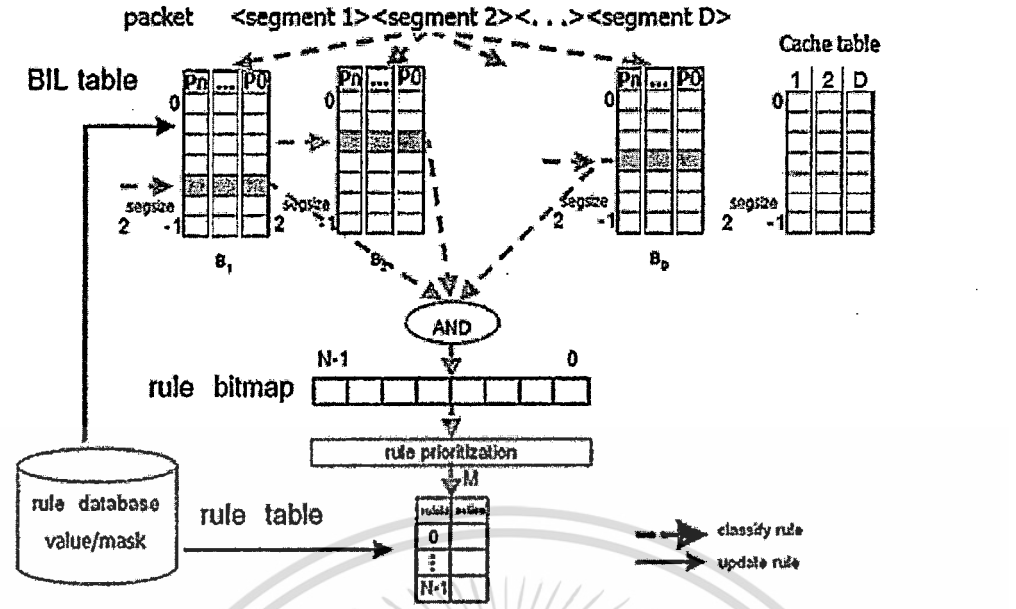
[5] P. Gupta and N. McKeown, **"Packet Classification using Hierarchical Intelligent Cuttings,"** Proc. Hot Interconnects VII, August 99, Stanford. Reference in IEEE Micro. Vol 20, No 1, pp. 34-41, Jan/Feb 2000.

[6] V. Srinivasan, S. Suri, and G. Varghese, **"Packet Classification using Tuple Space Search,"** In Proc.ACM Sigcomm, 1999.

ภาคผนวก

ภาคผนวก ก.

โปรแกรมที่ใช้ในการจำลองการทำงานของอัลกอริทึมการจัดประเภทแพ็กเก็ตตามกฎด้วยวิธีบิตแมปอินเตอร์เซคชันลุกอัพโดยใช้แคช


packet
<segment 1><segment 2><. . .><segment D>
BIL table
Cache table
1 2 D
segsize
AND
N-1
0
rule bitmap
rule prioritization
M
rule database
value/mask
rule table
classify rule
update rule

รูปที่ ก.1 แสดงสถาปัตยกรรมของวิธีบิตแมปอินเตอร์เซคชันลุกอัพโดยใช้แคช

สถาปัตยกรรมของโปรแกรม Linear Search ประกอบด้วยส่วนต่างๆ ดังต่อไปนี้

1. BIL table

เป็น Array 3 มิติ มีชนิดข้อมูลเป็น u_int32_t สำหรับเก็บ Bitvector ประกาศเป็น u_int32_t ***bil_table ให้อยู่ในรูปแบบของ pointer เพื่อสามารถใช้คำสั่งในการ realloc() ได้ โดยมีลักษณะดังนี้ bil_table[D][Address][Page]

กำหนดให้

D คือ จำนวนตาราง BIL table

Address คือ จำนวนแอดเดรสของแต่ละ BIL table เริ่มตั้งแต่ 0 ถึง $2^{Segmentsize} - 1$

Page คือ จำนวนเพจ โดยมีความกว้างเท่ากับ WORD (จำนวนบิตที่สามารถประมวลผลได้ในแต่ละครั้งในที่ให้เท่ากับ 32 บิต) ดังนั้น Page = N/WORD ปัดเศษขึ้น

2. Cache table

เป็น Array 2 มิติ มีชนิดข้อมูลเป็น u_int16_t สำหรับเก็บ page counter ประกาศเป็น u_int16_t cache_table[D][Address] เนื่องจาก u_int16_t สามารถรองรับค่าได้ตั้งแต่ 0 – 65535 คิดว่าเพียงพอต่อการเก็บ page counter

กำหนดให้

D คือ จำนวนตาราง Cache table ซึ่งจะตรงกับจำนวน BIL table

Address คือ จำนวนแอดเดรสของแต่ละ BIL table เริ่มตั้งแต่ 0 ถึง $2^{Segmentsize} - 1$

3. Rule bitmap

เป็นผลลัพธ์ที่ได้จากการนำ Bitvector มา intersection ด้วยตรรกะ AND โดยอัลกอริทึมจะเลือกกฎที่มีลำดับความสำคัญสูงสุดเป็นผลลัพธ์

4. Rule table

เป็น Array 1 มิติ มีชนิดข้อมูลเป็น struct โดยประกาศชนิดตัวแปร rule สำหรับเก็บค่าแอ็คชันของกฎ และคำสั่งการเพิ่มกฎที่อ่านมาจากฐานข้อมูล โดยประกาศเป็น

```
typedef struct  {
    long int rule_id;
    int action;       // ACTION_DROP / ACTION_ACCEPT / ACTION_NONE
    ip_addr ip;
} rule;
rule_table[N]
```

กำหนดให้

rule_id คือ หมายเลขกฎ

action คือ เก็บค่าแอ็คชันของกฎ

ip คือ เก็บเงื่อนไขของกฎ (งานวิจัยเก่าเก็บคำสั่งที่ใช้เพิ่มกฎ เพื่อให้สามารถลบเฉพาะ Bitvector ที่เพิ่มเข้าไป)

N คือ จำนวนกฎ

5. Rule database

เป็นฐานข้อมูลกฎเก็บในรูปแบบของ rule.db เพื่อใช้ในการสร้าง BIL table

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////  CLASSIFY FUNCTION  //////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
int classify(ip_addr ip){
        prefix *split;
        split = divide_segment(ip.address.s_addr);
        Result_ruleID = -1;
        int i=0, j=0, start_page ;
        int found_page = 0;
        int sum, mask;
        u_int32_t tmpbil;
        // หาว่า cache hit หรือ cache miss
        if(all_equal(split)){
                // cache hit
                hitrate+=1 ;
                start_page = cache_table[0][split[0].value] ;
        }else{
                // cache miss
                missrate+=1;
                start_page = 0;
        }
        for (i = start_page; i < P; i++) {
                tmpbil = 0xFFFFFFFF;
                for (j = 0; j < NUMBER_OF_BIL_TABLE;  j++)
                        tmpbil &= bil_table[j][split[j].value][i];
                if (tmpbil != 0) {
                        found_page = i;
                        sum = 0;
                        mask = 1;
                         while ((tmpbil & mask) == 0) {
                                sum += 1;
                                tmpbil >>= 1;
                        }
```

```
                    Result_ruleID = (i * (WORD_BIT)) +  sum;
                    // ใส่ค่า page counter ใน cache table
                    for (j = 0; j < NUMBER_OF_BIL_TABLE;  j++) {
                            cache_table[j][split[j].value] = found_page;
                    }
                    return Result_ruleID;
                } //end if
        } //end for
        return Result_ruleID; //แพ็กเก็ตที่เข้ามาไม่ตรงกับกฎเลย
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////  INSERT RULE  FUNCTION  ////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
int insert_rule(unsigned long int rule_id, ip_addr source_ip, char *action){
        prefix *split;
        int i,j;
        u_int32_t value, mask, inverse8bit = 0xFF;
        if((rule_id >= RuleCounter) &&  (rule_id < RuleCounter+1)){
                int k, befor;
                befor = P;
                P += 1;
                for(i=0;i<NUMBER_OF_BIL_TABLE;i++){
                   for(j=0;j<A;j++)
                      bil_table[i][j] = (u_int32_t*) realloc(bil_table[i][j], sizeof(u_int32_t)*P);
                }
                rule_table = (rule*) realloc(rule_table,sizeof(rule)*(WORD_BIT*P));
                for(i=RuleCounter;i<(WORD_BIT*P);i++){
                        rule_table[i].rule_id = i;
                        rule_table[i].action = ACTION_NONE;
                }
                RuleCounter += 32;
```

```
        // ค่าเริ่มต้นของ page counter ต้องเท่ากับ 0 เสมอ
        free(cache_table);
        cache_table = (u_int16_t**) malloc(sizeof(u_int16_t*)*NUMBER_OF_BIL_TABLE);
        for(i=0;i<NUMBER_OF_BIL_TABLE;i++){
                cache_table[i] = (u_int16_t*) malloc(sizeof(u_int16_t)*A);
        }
        }else if(rule_id > RuleCounter+1){
                printf("\n Rule id so long !!\n\n");
                return -1;
        }
        if((rule_id < RuleCounter) && (rule_table[rule_id].action == ACTION_NONE)){
           rule_table[rule_id].rule_id = rule_id;
           rule_table[rule_id].action = parse_action(action);
           split = convert_to_prefix(source_ip.address.s_addr, source_ip.addressmask.s_addr);
           for(i=0;i<NUMBER_OF_BIL_TABLE;i++){
                value = split[i].value;
                mask = split[i].mask;
                for(j=(value & mask);j<=((value & mask) + (~mask&inverse8bit));j++){
                    setBitvector(i,j,rule_id);//printf("\nsetBitvector %d , %d ,
%d",i,j,rule_id);
                }
           }
        }else if(rule_table[rule_id].action != ACTION_NONE){
                printf("\n\nCan not insert duplicate rule id %d\n\n",rule_id);
                return -1;
        }
}
```

```
///////////////////////////////////////////////////////////////////// //////////////////////////////////////////////////
//////////////////////////////////////////// DELETE RULE  FUNCTION  ///////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////
int delete_rule(unsigned long int rule_id){
        int i, j;
        u_int32_t tmpPagefocus =0, sum = 0x0;
        if((rule_table[rule_id].action != ACTION_NONE) && (rule_id < RuleCounter)){
                rule_table[rule_id].action = ACTION_NONE;
                tmpPagefocus = rule_id / (WORD_BIT);
                split = convert_to_prefix(rule_table[rule_id].ip.address.s_addr,
                                        rule_table[rule_id].ip.addressmask.s_addr);
                for(i=0;i<NUMBER_OF_BIL_TABLE;i++){
                        for(j=0;j<A;j++){
                                resetBitvector(i, j, rule_id);
                                sum |= bil_table[i][j][tmpPagefocus];
                        }
                }
                if((sum == 0) && ((tmpPagefocus+1) == P)){
                // ตรวจสอบว่าจะลบหน้าเปล่าทิ้งหรือไม่
                        P -= 1;
                        for(i=0;i<NUMBER_OF_BIL_TABLE;i++){
                                for(j=0;j<A;j++){
                                        bil_table[i][j] = (u_int32_t*) realloc(bil_table[i][j],
                                                        sizeof(u_int32_t)*P);
                                }
                        }
                        RuleCounter-=32;
                        rule_table = (rule*) realloc(rule_table,sizeof(rule)*(WORD_BIT*P));
                }
                // ค่าเริ่มต้นของ page counter ต้องเท่ากับ 0 เสมอ
                free(cache_table);
                cache_table = (u_int16_t**) malloc(sizeof(u_int16_t*)*NUMBER_OF_BIL_TABLE);
```

```
            for(i=0;i<NUMBER_OF_BIL_TABLE;i++)
                cache_table[i] = (u_int16_t*) malloc(sizeof(u_int16_t)*A);
            return 0;
        }else{
            // ไม่สามารถลบกฎได้
            printf("\n\nRule_id %d not found \n\n",rule_id);
            return -1;
        }
}
```

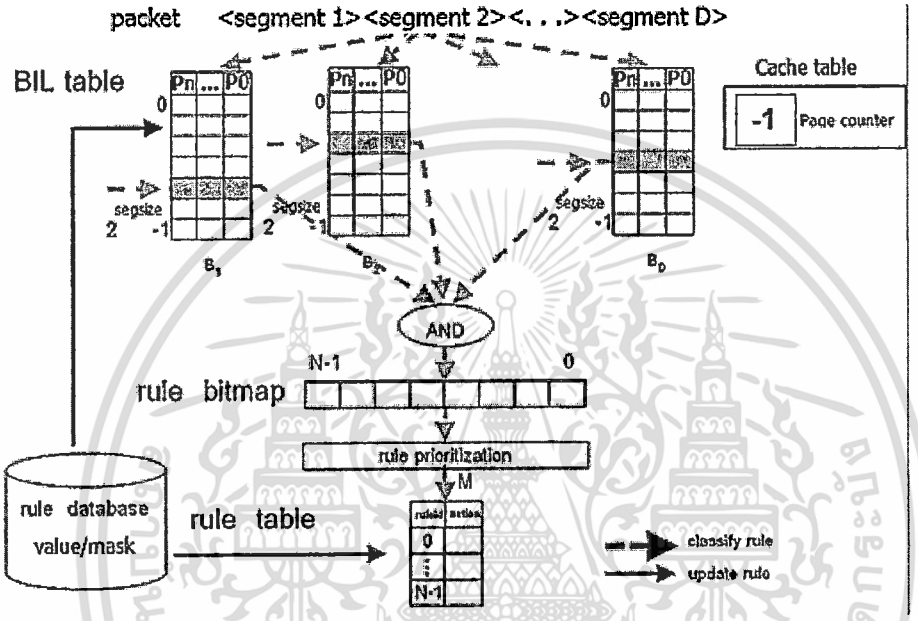
ภาคผนวก ช.

งานวิจัยที่พบข้อผิดพลาด

3) xxx.xxx.25.xxx ตรงกับกฎในเพจที่ 40

4) xxx.xxx.xxx.110 ตรงกับกฎในเพจที่ 50

ใน Cache table จึงมีค่า page counter ของแต่ละ segment ตามลำดับดังนี้ 20, 30, 40 และ 50 เมื่อพิจารณาเงื่อนไขโดยการหาค่าน้อยที่สุดของ page counter ได้ผลลัพธ์เท่ากับ 20 การค้นหาจึงเริ่มค้นที่เพจ 20 เมื่อค้นหาจนถึงเพจสุดท้ายไม่พบกฎ เกิดข้อผิดพลาดเนื่องแพ็กเก็ต 172.20.25.110 ควรพบกฎในเพจที่ 10


packet
<segment 1><segment 2><. . .><segment D>
BIL table
Cache table
-1
Page counter
AND
N-1
0
rule bitmap
rule prioritization
M
rule database
value/mask
rule table
classify rule
update rule

รูปที่ ข.2 แสดงสถาปัตยกรรมของวิธีบิตแมปอินเตอร์เซคชันลุกอัพโดยใช้แคช แบบ Multi-thread

สถาปัตยกรรมของวิธีการ Bitmap Intersection Lookup using Cache หรือ BIL using Cache ประกอบด้วย BIL table จำนวน D ตาราง โดยแต่ละตารางประกอบด้วย เพจ เรียงต่อกัน มีความกว้างเท่ากับ Psize หรือ WORD ดังนั้นจำนวนของ เพจ ใน BIL table จึงเท่ากับ (Rule/Psize) ปัดเศษขึ้น เช่น กฎจำนวน 100 กฎ, Psize ขนาด 32 ดังนั้นใน BIL table มีจำนวน เพจ (n) เท่ากับ 100/32 = 3.125 ปัดเศษขึ้นเป็น 4 ในแต่ละ เพจ ประกอบด้วยแอดเดรสจำนวน $2^{Segsize}$ เริ่มตั้งแต่ 0 ถึง $2^{Segsize} - 1$ แต่ละแอดเดรสประกอบด้วยบิตเรียงต่อกันเท่ากับความกว้างของ เพจ เรียกว่า Bitvector บิตที่เรียงกันจะเรียงตรงกันตำแหน่งของกฎตั้งแต่ (Pcurrent*Psize) ถึง (Pnext*Psize) - 1 เช่น เพจ 0 ประกอบด้วยบิตที่ตรงกับตำแหน่ง (0*32) = 0 ถึง (1*32)-1 = 31 BIL table ถูกสร้างขึ้นจากฐานข้อมูลกฎ ซึ่งฐานข้อมูลกฎ (Rule database) เก็บอยู่ในรูปแบบของ value/mask หรือ ช่วง (range) จากนั้นทำการแปลงค่าให้อยู่ในรูปแบบของ Prefix เพื่อใช้ในการแปลงค่า Bitvector ใน BIL table ต่างๆ และจัดเก็บแอ็คชันของกฎแต่ละข้อลงใน Rule table และมี Cache table เพื่อ

ทำการเก็บค่าของเพจที่ทำการค้นหาพบล่าสุด เพื่อใช้ในการค้นหารอบต่อไป โดยจะมีค่าเริ่มต้นเท่ากับ "-1"

การค้นหาทำการแบ่งเฮดเดอร์ของแพ็กเก็ตจำนวน D segments นำแต่ละ Segment ไปค้นหาใน Cache lookup และ BIL lookup ลักษณะการทำงานแบบ Multi-thread โดยที่การค้นหาใน BIL table เริ่มทำการหาจาก เพจ0 จนถึงเพจที่ Cache table - 1 ส่วนการค้นหาใน Cache table จะเริ่มค้นหาที่เพจที่ได้ทำการค้นหาพบกฎล่าสุดจนถึงเพจสุดท้าย หากผลการค้นหาพบกฎใน Cache table เรียกว่า Cache hit หรือ หากไม่พบกฎ เรียกว่า Cache miss ยังมีผลลัพธ์ที่ได้จากค้นหาใน BIL table จากนั้นนำผลลัพธ์ที่ได้ (rule bitmap) หาค่ากฎ และส่งค่าแอ็คชันต่อไป

อัลกอริทึมของการค้นแบบ Multi-thread นี้พบข้อผิดพลาดเนื่องจากสถาปัตยกรรมนั้นมีโอกาสที่แพ็กเก็ตหนึ่งสามารถตรงกับกฎได้มากกว่า 1 ข้อ ด้วยกรณีนี้มีความเป็นไปได้ว่าหากแพ็กเก็ตที่ทำการค้นตรงกับกฎของทั้ง 2 ช่วงของการค้นหา เช่น กำหนดให้ Page counter เก็บค่า 20, PAGE เท่ากับ 30 การค้นหาแพ็กเก็ตสมมติให้แพ็กเก็ตดังกล่าวตรงกับกฎในเพจที่ 9 และ 21 ในการค้นหา Cache lookup เริ่มค้นหาตั้งแต่เพจ 20 – 29 , BIL lookup เริ่มค้นหาตั้งแต่เพจ 0 – 19 ดังนั้นเมื่อทำการค้นหาพบกฎในช่วงการค้นหาของ Cache lookup ก่อนคือพบกฎที่ตรงกันในเพจที่ 21 ส่งผลให้ผลลัพธ์ในการค้นหาเท่ากับหมายเลขกฎในเพจที่ 21 ทั้งที่ความเป็นจริงแล้วความสำคัญของเพจที่ 9 มีความสำคัญกว่า ถือเป็นข้อผิดพลาดในการค้นหา

ภาคผนวก ค.

การวิเคราะห์ความสัมพันธ์ (Order of Magnitude)

การวิเคราะห์เปรียบเทียบเวลาในการปรับปรุงกฎ

การเพิ่มกฎ

```
1.  if ( ไม่มีเพจที่รองรับกฎที่ต้องการเพิ่มใหม่ ) {
2.      allocate เพจใหม่เพื่อรองรับกฎดังกล่าว
3.  }
4.  if ( rule_table ยังไม่เก็บข้อมูลกฎ ) {
5.      เก็บข้อมูลลงใน rule table
6.      prefix = convert_to_prefix(source_ip.address.s_addr, source_ip.addressmask.s_addr);
7.      for(i=0; i < D ;i++){
8.          value = prefix [i].value;
9           mask = prefix [i].mask;
10.         for(j=(value & mask);j<=((value & mask) + (~mask));j++){
11.             setBitvector(i,j,rule_id);
12.         }
13.     }
14. }else if(( rule_table มีการเก็บข้อมูลกฎอื่นอยู่แล้ว){
15.     return -1;
16. }
```

รูปที่ ค.2 อัลกอริทึมการเพิ่มกฎ

บรรทัดที่ 1-3 ทำการตรวจสอบว่ากฎที่ทำการเพิ่มใหม่นั้นมีเพจรองรับแล้วหรือไม่ หากไม่มีจะทำการสร้าง (allocate) เพจใหม่ขึ้นมารองรับ บรรทัดที่ 4-13 เป็นขั้นตอนในการเพิ่มกฎเข้าไปโครงสร้างของ BIL using Cache ประกอบด้วย บรรทัดที่ 6 ทำการกำหนดค่าของกฎให้อยู่ในรูปแบบ prefix บรรทัดที่ 7-13 ทำการ Set Bitvector ใน BIL table ต่างๆ ให้ Bitvector ที่ตรงกับค่าของกฎให้มีค่าเท่ากับ “1”

วิเคราะห์ทฤษฎีของการเพิ่มกฎของอัลกอริทึมการจัดประเภทแพ็กเก็ตตามกฎด้วยวิธีบิตแมปอินเตอร์เซคชันลุกอัพโดยใช้แคช

กรณี Best case ของการเพิ่มกฎ เท่ากับ $O(D \bullet 2^{SegmentSize})$

กรณี Worst case ของการเพิ่มกฎ เท่ากับ $O((2 \bullet W - 2) \bullet D \bullet 2^{SegmentSize})$

เมื่อนำไปเปรียบเทียบกับวิธีการเดิม

กรณี Best case ของการเพิ่มกฎ เท่ากับ $O\left(\frac{D \cdot W}{PMX} \bullet 2^{PMX}\right)$

กรณี Worst case ของการเพิ่มกฎ เท่ากับ $O\left((2 \bullet W - 2) \bullet \frac{D \cdot W}{PMX} \bullet 2^{PMX}\right)$

จากการวิเคราะห์ทฤษฎีที่ได้กล่าว อาจเห็นว่าการวิเคราะห์ประสิทธิภาพส่วนของการเพิ่มกฎนั้น อาจเท่ากัน แต่การเพิ่มกฎของวิธีการแบบเดิมนั้นเมื่อนำมาเทียบกับอัลกอริทึมของวิธีการที่นำเสนอใหม่นั้นจะอยู่ในบรรทัดที่ 4-13 ซึ่งเป็นเพียงส่วนหนึ่งของอัลกอริทึมของวิธีการนำเสนอใหม่ หากมีการเพิ่มกฎที่เกินจากที่โครงสร้างสามารถรองรับได้ การเพิ่มกฎของวิธีแบบเดิมจะใช้เวลาในส่วนของการสร้างโครงสร้างใหม่ ดังนั้นการเพิ่มกฎด้วยวิธีการใหม่จะมีประสิทธิภาพสูงกว่า

การลบกฎ

```
1.  if ( rule_table กฎหมายเลขดังกล่าวไม่ว่าง ){
2.      prefix = convert_to_prefix(source_ip.address.s_addr, source_ip.addressmask.s_addr);
3.      for(i=0;i<D;i++){
4.          value = prefix[i].value;
5.          mask = prefix[i].mask;
6.          for(j=(value & mask);j<=((value & mask) + (~mask));j++){
7.              resetBitvector(i,j,rule_id);
8.          }
9.      }
10.     rule_table[rule_id].action เก็บค่าว่าง;
11. }else if (( rule_table กฎหมายเลขดังกล่าวไม่ว่าง){
12.     ไม่สามารถลบกฎได้
13.     return -1;
14. }
15. if ( เพจดังกล่าวเป็นเพจสุดท้าย && Bitvector ของเพจดังกล่าว = = 0 ){
16.     ทำการ de-allocate เพจดังกล่าว
17. }
```

รูปที่ ค.3 อัลกอริทึมการลบกฎ

บรรทัดที่ 1-11 เป็นขั้นตอนในการลบกฎของโครงสร้างของ BIL using Cache ประกอบด้วย บรรทัดที่ 2 ทำการกำหนดค่าของกฎให้อยู่ในรูปแบบ prefix บรรทัดที่ 3-9 ทำการ reset Bitvector ใน BIL table ต่างๆ เพื่อให้ Bitvector ที่ตรงกับตำแหน่งของกฎให้เก็บค่าเป็น “0” จากนั้น บรรทัดที่ 15-17 ทำการตรวจสอบว่าเพจดังกล่าวเป็นเพจสุดท้ายและ Bitvector ทั้งหมดมีค่าเท่ากับ “0” หากจริงจะทำการลบ (de-allocate) เพจดังกล่าว เพื่อเป็นการใช้เนื้อที่ได้อย่างมีประสิทธิภาพสูงสุด

จากการวิเคราะห์ทฤษฎีที่ได้กล่าวพบว่า การใช้เนื้อที่ในการเก็บข้อมูลของ BIL table ของวิธีการจัดประเภทแพ็กเก็ตตามกฎด้วยวิธีบิตแมปอินเตอร์เซคชันลุกอัพโดยใช้แคช ใช้เนื้อที่มากกว่าวิธีการแบบเดิม

ภาคผนวก ง.

ผลงานวิจัยที่ได้รับการตีพิมพ์เผยแพร่

1. Thapana Choorat, Akharin Khunkitti, " **A Packet Classification's Algorithm Bitmap Intersection Lookup using Cache** ", 2010 2rd International Conference on Computer and Automation Engineering, 26-28 February, 2010, Singapore.

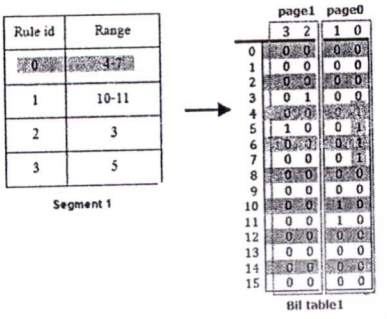


Figure 5. BIL table 1.

B. Classify algorithm.

Algorithm has parallel processing: 1) lookup in BIL table, 2) lookup in Cache table.

- Lookup in BIL table - at first, divide packet to small segment, next take that bivector on a page at address to be intersected with logic AND, that result is rule bitmap, and then the algorithm priority chooses the most important rule as the arrows stated in Fig. 3

```
1. Divide header of packet to small segment.
   // at line 2 has Parallel processing
     focus on Lookup in Cache table.
2. for (i=0 ; i<D ; i++){
3.       for (j=0 ; j<P ; j++){
4.           Take bitvector in page to intersect by
             logic AND.
5.       }
6.       if(result != 0){
7.           Calculate rule number of first 1 bit of
             bitvector and return action value
             from rule table.
8.       }
9. }
```

Figure 6. Lookup in BIL table.

Best Case, $Prob_{found}(N) = 1$

Worst Case, $Prob_{found}(N) = \frac{N}{WORD} + 1$

- Lookup in cache table – method same lookup in BIL table but it focus on last classify page at first, called "cache hit" if packet matching rule or called "cache miss" if don't matching rule.

```
1. for (i=0 ; i<D ; i++){
2.       Take bitvector in page to intersect by
         logic AND.
3.       if(rule bitmap != 0){ // Cache hit
4.           Calculate rule number of first 1 bit of
             bitvector and return action value
             from rule table.
5.       }
6. } // Cache miss
```

Figure 7. Lookup in Cache table.

Cache hit - Best Case, $Prob_{found}(N) = 1$

Cache hit - Worst Case, $Prob_{found}(N) = 1$

Classify is parallel processing:

Worst Case, $Prob_{found}(N) = \max(\frac{N}{WORD} + 1, 1)$

so $Prob_{found}(N) = \frac{N}{WORD} + 1$

C. Update algorithm.

Classifier has frequency update; thus, architecture has update frequency too. This method must not reset architecture, include 2 methods:

- Add rule – use to add new rule to architecture.
- Delete rule – use to delete rule from architecture.

Add rule - first, allocate continue page for each BIL table for rule don't have allocated page, then set bitvector in new page.

Example:

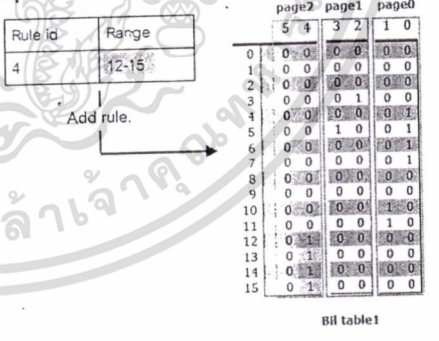


Figure 8. Add rule.

ประวัติผู้เขียน

นายฐาปนา ชูรัตน์ เกิดเมื่อวันที่ 25 มีนาคม พ.ศ. 2528 ที่จังหวัดนครศรีธรรมราช สำเร็จการศึกษาระดับปริญญาตรี สาขาระบบสารสนเทศเพื่อการจัดการ คณะเทคโนโลยีสารสนเทศ มหาวิทยาลัยวลัยลักษณ์ ในปีการศึกษา 2549 และเข้าศึกษาต่อในระดับปริญญาโท หลักสูตรวิทยาศาสตรมหาบัณฑิต สาขาเทคโนโลยีสารสนเทศ แขนงวิชาวิทยาการสารสนเทศ คณะเทคโนโลยีสารสนเทศ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง ในปีการศึกษา 2550