การเพิ่มประสิทธิภาพ Garbage Collection ของ Java โดยฮาร์ดแวร์ด้วยวิธี Reference Counting

PERFORMANCE IMPROVEMENT OF JAVA GARBAGE COLLECTION USING HARDWARE WHICH EMPLOYS THE REFERENCE COUNTING METHOD

อุดม รานอก
UDOM RANOK

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์
บัณฑิตวิทยาลัย
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
พ.ศ. 2549

ISBN 974-15-2631-8

การเพิ่มประสิทธิภาพ Garbage Collection ของ Java โดยฮาร์ดแวร์ด้วยวิธี Reference Counting

PERFORMANCE IMPROVEMENT OF JAVA GARBAGE COLLECTION USING HARDWARE WHICH EMPLOYS THE REFERENCE COUNTING METHOD

อุดม รานอก

UDOM RANOK

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์
บัณฑิตวิทยาลัย
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
พ.ศ.2549

ISBN 974-15-2631-8

PERFORMANCE IMPROVEMENT OF JAVA GARBAGE COLLECTION USING HARDWARE WHICH EMPLOYS THE REFERENCE COUNTING METHOD

UDOM RANOK

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENT FOR THE DEGREE OF

MASTER OF ENGINEERING IN COMPUTER ENGINEERING

SCHOOL OF GRADUATE STUDIES

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

2006

ISBN 974-15-2631-8

หัวข้อวิทยานิพนธ์	การเพิ่มประสิทธิภาพ Garbage Collection ของ Java โดยฮาร์ดแวร์ด้วยวิธี Reference Counting
นักศึกษา	นายอุดม รานอก
รหัสนักศึกษา	44061609
ปริญญา	วิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชา	วิศวกรรมคอมพิวเตอร์
พ.ศ.	2549
อาจารย์ผู้ควบคุมวิทยานิพนธ์	รศ. ประทีป บัญญัตินพรัตน์
อาจารย์ผู้ควบคุมวิทยานิพนธ์ร่วม	รศ. บรรจง ปิยะธำรง

บทคัดย่อ

การโปรแกรมเชิงวัตถุ เช่น ภาษา C++ และ JAVA ทำงานกับพื้นที่หน่วยความจำส่วนฮีป ในการจัดการหน่วยความจำส่วนฮีป ระบบทำการจองพื้นที่หน่วยความจำสำหรับวัตถุใหม่ และคืนพื้นที่เมื่อไม่มีการใช้งาน ซึ่งในการคืนพื้นที่ระบบจะทำโดยอัตโนมัติด้วยกลไกของ Garbage Collector ปัจจุบัน Garbage Collection จะทำโดยวิธีการทางซอฟต์แวร์ ซึ่งทำให้ไม่สามารถเพิ่มความเร็วในการทำงานได้มากนัก แต่ปัจจุบันเทคโนโลยี VLSI ได้มีการพัฒนาขึ้นมากทำให้สามารถพัฒนาอัลกอริทึมด้วยวิธีการทางฮาร์ดแวร์แทนวิธีการทางซอฟต์แวร์ได้ ซึ่งถ้า Garbage Collector เปลี่ยนจากวิธีการทางซอฟต์แวร์เป็นวิธีการทางฮาร์ดแวร์ได้ จะทำให้สามรถเพิ่มประสิทธิภาพในการจัดการหน่วยความจำและเพิ่มความเร็วในการทำงานของระบบได้ งานวิจัยนี้เป็นนำเสนอการออกแบบสร้าง Reference Counting Garbage Collection ด้วยวิธีการทางฮาร์ดแวร์โดยใช้ภาษา VHDL ซึ่งเป็นงานวิจัยใหม่ที่ได้นำเสนอในครั้งแรก ในการออกแบบนี้จะนำระบบ Binary buddy และ Bitmap มาใช้ในการออกแบบเพราะง่ายต่อการสร้างด้วยวงจรรวม โดยฮาร์ดแวร์ที่ได้ออกแบบประกอบด้วย 2 ส่วน คือส่วนการจองพื้นที่สำหรับวัตถุใหม่และการกำหนดค่าตัวนับอ้างอิงเริ่มต้นและส่วน Garbage Collection ประกอบด้วยฮาร์ดแวร์สำหรับทำการปรับค่าและตรวจสอบตัวนับอ้างอิงและฮาร์ดแวร์สำหรับทำการคืนพื้นที่เมื่อตัวนับอ้างอิงมีค่าเป็นศูนย์ ซึ่งวิธีการที่นำเสนอนี้ สามารถเพิ่มความเร็วและประสิทธิภาพในการจัดการหน่วยความจำ สามารถลดค่าใช้จ่ายในการจัดการตัวนับอ้างอิง และสามารถลดการเกิด internal fragmentation จากวิธีซอฟต์แวร์ได้

Thesis Title	Performance Improvement of Java Garbage Collection using Hardware which Employs the Reference Counting Method
Student	Mr. Udom Ranok
Student ID.	44061609
Degree	Master of Engineering
Programme	Computer Engineering
Year	2005
Thesis Advisor	Assoc. Prof. Pratheep Bunyatnoparat
Co-Thesis Advisor	Assoc. Prof. Banjong Piyatumrong

ABSTRACT

Object-oriented programming, such as C++ and JAVA, normally uses heap memory. In order to manage heap memory, the system allocates memory space for a new object and frees the unused space automatically. Garbage collector is used to free memory space. To improve the garbage collection performance, VLSI technology is used for this purpose to develop a hardware garbage collector. This thesis proposes an implementation of Reference Counting Garbage Collection (RCGC) by using hardware algorithm based on VHDL that was the first presented research. The binary buddy system and bit-map techniques are chosen to design the proposed RCGC because they are easy to be implemented with combinational logic. The RCGC is composed with two hardware designs: (1) object address allocation and initialization of reference count part and (2) garbage collection part. The result shows that the proposed on RCGC implementation can improve the system performance in comparison with software approaches in case of internal fragments, cost of reference count management and the overall speed.

กิตติกรรมประกาศ

วิทยานิพนธ์ฉบับนี้สำเร็จได้อย่างดี ด้วยคำแนะนำ และคำปรึกษาจาก รศ.ประทีป บัญญัตินพรัตน์ ซึ่งเป็นอาจารย์ผู้ควบคุมวิทยานิพนธ์ และ รศ.บรรจง ปิยะธำรง ซึ่งเป็นอาจารย์ผู้ควบคุมวิทยานิพนธ์ร่วม ข้าพเจ้ารู้สึกซาบซึ้งในความอนุเคราะห์จากท่านอาจารย์ทั้งสองท่าน และขอขอบพระคุณเป็นอย่างสูง

ขอกราบพระคุณคณาจารย์ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง ทุกท่านที่ได้ประสิทธิ์ประสาทวิชาให้กับข้าพเจ้า

ขอขอบคุณเพื่อนๆ พี่ๆ น้องๆ ในภาควิชาวิศวกรรมวิศวกรรมคอมพิวเตอร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง ทุกคนที่ให้คำแนะนำต่างๆ และคอยให้กำลังใจเสมอมา

ขอขอบคุณเพื่อนอาจารย์ ในภาควิชาวิศวกรรมคอมพิวเตอร์ มหาวิทยาลัยธุรกิจบัณฑิตย์ ทุกคนที่ให้คำแนะนำต่างๆ และคอยให้กำลังใจเสมอมา

ขอขอบคุณบัณฑิตศึกษาและบัณฑิตวิทยาลัย คณะวิศวกรรมศาสตร์ที่ให้ความช่วยเหลือในเรื่องต่างๆ

สุดท้ายนี้ข้าพเจ้าขอกราบขอบพระคุณ บิดา มารดา และครอบครัวของข้าพเจ้าที่เป็นกำลังใจ และให้การสนับสนุนในทุกเรื่องๆ ทำให้ข้าพเจ้าสามารถทำวิทยานิพนธ์ฉบับนี้สำเร็จลุล่วงด้วยดี

คุณค่าและประโยชน์อันพึงมาจากวิทยานิพนธ์ฉบับนี้ ข้าพเจ้าขอมอบแด่ผู้มีพระคุณทุกท่าน

อุดม รานอก

สารบัญ

หน้า

สารบัญ (ต่อ)

หน้า

สารบัญ (ต่อ)

หน้า

สารบัญตาราง

สารบัญรูป

สารบัญรูป (ต่อ)

รูปที่ หน้า

บทที่ 1
บทนำ

1.1 ความเป็นมาและความสำคัญของปัญหา

ปัจจุบันเทคโนโลยีด้านอิเล็กทรอนิกส์และคอมพิวเตอร์ได้พัฒนาไปมาก เราสามารถผลิตชิพที่เป็นวงจรขนาดใหญ่ที่มีทรานซิสเตอร์จำนวนมากเป็นล้านๆตัว อยู่ในชิพขนาดเล็กได้ โดยเฉพาะเทคโนโลยีการผลิตซีพียู สามารถผลิตซีพียูที่มีความเร็ว มีประสิทธิภาพสูงมาใช้งาน ที่เห็นได้ชัดคือคอมพิวเตอร์ส่วนบุคคลที่ปัจจุบันมีความเร็วเพิ่มมากขึ้น ทำให้สามารถนำไปใช้งานได้หลากหลาย โดยเฉพาะงานซับซ้อนที่จำเป็นต้องใช้ซีพียูที่มีความเร็วและประสิทธิภาพสูง ส่วนในด้านภาษาการโปรแกรม มีการกำเนิดภาษาโปรแกรมเชิงวัตถุขึ้นมาใหม่ โดยมีการลดข้อจำกัดบางอย่างและกำหนดข้อจำกัดบางอย่าง เพื่อให้โปรแกรมเมอร์สามารถเขียนโปรแกรมได้ง่ายและรวดเร็วขึ้น เช่น ภาษาที่ใช้พัฒนาโปรแกรมบนเครื่องพีดีเอ โทรศัพท์มือถือ เป็นต้น

ในระบบคอมพิวเตอร์หนึ่งระบบประกอบไปด้วยส่วนฮาร์ดแวร์และซอฟต์แวร์ที่ต้องสามารถทำงานสอดประสานกันได้ดี ถ้าระบบไหนการทำงานอยู่ในส่วนที่เป็นฮาร์ดแวร์มาก จะทำให้ระบบนั้นทำงานได้เร็วและมีประสิทธิภาพ โดยเฉพาะในงานระบบคอมพิวเตอร์ฝังตัวที่ต้องการทำงานแบบเวลาจริง (real time) จำเป็นต้องลดการทำงานในส่วนที่เป็นซอฟต์แวร์ลงไปให้มากที่สุด

ปัจจุบันมีการทำวิจัยโดยนำองค์ประกอบบางอย่างที่จัดการโดยวิธีทางซอฟต์แวร์เปลี่ยนเป็นจัดการด้วยวิธีทางฮาร์ดแวร์ โดยเฉพาะภาษาโปรแกรมเชิงวัตถุ เช่นภาษาจาวา (JAVA) ที่การทำงานไม่ขึ้นอยู่กับแพลตฟอร์ม (platform) ปัจจุบันนิยมใช้กันมาก โดยเฉพาะการเขียนโปรแกรมควบคุมอุปกรณ์ขนาดเล็ก แต่มีปัญหาในเรื่องของการทำงานช้า เพราะต้องเสียเวลาในการจัดการหน่วยความจำแบบพลวัตโดยเฉพาะขั้นตอนในการคืนพื้นที่หน่วยความจำที่ไม่ได้ใช้งาน ที่เรียกว่า Garbage Collection ทำให้ความเร็วในการทำงานของระบบช้าอยู่ จากการศึกษาพบว่าโปรแกรมที่พัฒนาโดยภาษาจาวา ใช้เวลาในการจัดการหน่วยความจำประมาณ 23% ถึง 38% ของเวลาที่ใช้ในการประมวลผลทั้งหมด [1] ดังนั้นวิธีการแก้ปัญหาคือ ปรับปรุงประสิทธิภาพของหน่วยจัดการหน่วยความจำแบบพลวัต โดยทั่วไปหน่วยจัดการหน่วยความจำแบบพลวัตจะประกอบด้วย 2 ส่วน คือ การจองพื้นที่หน่วยความจำสำหรับวัตถุใหม่ และการจัดการคืนพื้นที่วัตถุที่ไม่ได้ใช้งาน ซึ่งทำโดยอัตโนมัติ ในการคืนพื้นที่วัตถุ ปกติแล้วพัฒนาโดยวิธีการทางซอฟต์แวร์ ซึ่งอัลกอริทึมที่ใช้มีหลายวิธีด้วยกัน โดยพื้นฐานแล้วมีอยู่ 2 วิธีคือ Reference counting collector และ Tracing collector [2] แต่ละวิธีมีทั้งข้อดีและข้อเสีย ในการเพิ่มประสิทธิภาพหน่วยจัดการหน่วยความจำแบบพลวัตนั้นนักวิจัยพยายามคิดค้นวิธีการใหม่ๆมาแก้ปัญหา ส่วนมากเป็นวิธีการทางซอฟต์แวร์ แต่ยังไม่สามารถเพิ่มประสิทธิภาพได้มากนัก ดังนั้นถ้าสามารถแก้ปัญหาด้วยวิธีการทางฮาร์ดแวร์ได้จะทำ

ให้ประสิทธิภาพในการจัดการหน่วยความจำของระบบดีขึ้นเพราะเป็นการทำงานในระดับฮาร์ดแวร์โดยตรง และมีผลต่อความเร็วที่เพิ่มขึ้นของระบบที่พัฒนาด้วยภาษาจาวาด้วย โดยเฉพาะในกรณีที่เป็นระบบคอมพิวเตอร์ฝังตัวที่ต้องการหน่วยความจำน้อย จึงจำเป็นต้องมีระบบจัดการหน่วยความจำที่ดีและมีประสิทธิภาพ

1.2 ความมุ่งหมายและวัตถุประสงค์ของการศึกษา

วิทยานิพนธ์ฉบับนี้มุ่งหวังเพื่อศึกษาและปรับปรุงการทำงานของหน่วยจัดการหน่วยความจำของจาวา ประกอบด้วย 2 ส่วน คือส่วนการจองพื้นที่หน่วยความจำสำหรับวัตถุใหม่และส่วนการคืนพื้นที่วัตถุที่ไม่ได้ใช้งาน (Garbage Collection) โดยเน้นไปที่อัลกอริทึม reference counting เพราะเหมาะกับงานในระบบคอมพิวเตอร์ฝังตัวซึ่งทำงานตามเวลาจริง โดยการศึกษานั้นเป็นการศึกษาถึงการทำงานในส่วนของเดิมที่เป็นซอฟต์แวร์ และปรับปรุงการทำงานให้ดีขึ้นด้วยวิธีการทางฮาร์ดแวร์ ดังนั้นในวิทยานิพนธ์นี้จึงนำเสนอวิธีการใหม่ด้วยฮาร์ดแวร์สำหรับจัดการหน่วยความจำแบบพลวัตโดยใช้วิธี reference counting เพื่อปรับปรุงประสิทธิภาพการทำงานของระบบให้ดีขึ้น

1.3 สมมุติฐานของการศึกษา

เนื่องจากภาษาการโปรแกรมเชิงวัตถุโดยเฉพาะภาษาจาวา มักเสียเวลาในการจัดการหน่วยความจำโดยเฉพาะขั้นตอนในการคืนพื้นที่วัตถุที่ไม่ได้ใช้งาน ทำให้ความเร็วในการทำงานของระบบช้าเมื่อเทียบกับการพัฒนาด้วยภาษาการโปรแกรมอื่น โดยเฉพาะวิธี reference counting garbage collection ซึ่งเหมาะกับการทำงานที่ใช้เวลาสั้นๆภายในเวลาของการประมวลผลโปรแกรม ทำให้เวลาต่อการตอบสนอง (response time) ดี เหมาะสมในการนำไปใช้งานที่ทำงานแบบเวลาจริง แต่มีปัญหาคือเกิด internal fragmentation มีค่าใช้จ่ายในการจัดการตัวนับอ้างอิง (reference count) และโดยรวมแล้วระบบยังทำงานช้า

ดังนั้น ถ้าปรับปรุงการทำงานในส่วนการจัดการหน่วยความจำโดยเฉพาะขั้นตอนการคืนพื้นที่วัตถุที่ไม่ได้ใช้งานจะทำให้ประสิทธิภาพการทำงานของระบบดีขึ้น ในวิทยานิพนธ์เล่มนี้ได้ทำการแก้ปัญหาโดยให้ส่วนการจัดการหน่วยความจำทำงานในระดับฮาร์ดแวร์ ซึ่งจะทำงานได้เร็วและประสิทธิภาพที่ดีกว่าโดยวิธีการทางฮาร์ดแวร์ใช้วิธีการระบบไบนารีบัดดี (binary buddy system) และเทคนิคบิตแมป(bit map) จัดการด้วยวงจรรวม (combinational logic) ซึ่งทำให้สามารถจองพื้นที่และคืนพื้นที่หน่วยความจำได้เร็วและทำงานด้วยเวลาที่คงที่ และงานวิจัยที่นำเสนอนี้เป็นงานวิจัยใหม่ที่นำเสนอในครั้งแรก

1.4 ขอบเขตของการศึกษา

ในวิทยานิพนธ์ฉบับนี้ได้เลือกวิธี reference counting มาทำการศึกษาและปรับปรุงแก้ไขด้วยวิธีทางฮาร์ดแวร์ พัฒนาด้วยภาษาวีเอชดีแอล (VHDL) และจำลองการทำงาน สำหรับการเปรียบเทียบความเร็วที่เพิ่มขึ้นนั้นจะเปรียบเทียบกับ KVM โดยทำการแก้ไข KVM ในส่วนการคืนพื้นที่หน่วยความจำให้มี อัลกอริทึมการทำงานเป็น reference counting และทำการคอมไพล์ใหม่ และนำผลที่ได้เปรียบเทียบกันเพื่อชี้ให้เห็นว่าวิธีการทางฮาร์ดแวร์ทำงานได้เร็วในระดับที่ยอมรับได้

1.5 ขั้นตอนของการศึกษา

วิทยานิพนธ์ฉบับนี้ได้แบ่งเนื้อหาออกเป็น 6 บทด้วยกันคือ

บทที่ 1 กล่าวถึงความเป็นมาของงานวิจัย ความมุ่งหมายและวัตถุประสงค์ สมมติฐานของการศึกษา ขอบเขตของการวิจัย และขั้นตอนการศึกษา

บทที่ 2 กล่าวถึงทฤษฎีพื้นฐานที่ใช้ในการวิจัย ได้แก่ การจัดการหน่วยความจำแบบพลวัตเบื้องต้น เทคนิคการจองหน่วยความจำ การจองหน่วยความจำด้วยฮาร์ดแวร์ระบบไบนารีบัดดีและและบิตแมป เทคนิคการคืนค่าวัตถุที่ไม่ได้ใช้งานจาวาเวอร์ชวลแมชชีน (java virtual machine) และ KVM

บทที่ 3 กล่าวถึงงานวิจัยที่เกี่ยวข้องที่ใช้เป็นแนวทางในการทำวิทยานิพนธ์

บทที่ 4 กล่าวถึงวิธีการออกแบบสร้างฮาร์ดแวร์สำหรับการจองพื้นที่หน่วยความจำสำหรับวัตถุใหม่และการคืนค่าวัตถุที่ไม่ได้ใช้งาน

บทที่ 5 กล่าวถึงการทดสอบ RCGC เริ่มตั้งแต่วิธีการสร้างฮาร์ดแวร์สำหรับการทำสอบเพื่อนำไปใช้ในการจำลองการทำงาน ผลการทดลองทั้งในส่วนการสังเคราะห์วงจรและการจำลองการทำงานและการประเมินประสิทธิภาพ โดยการเปรียบเทียบกับวิธีการทางซอฟต์แวร์

บทที่ 6 เป็นบทสรุปผลการวิจัยและข้อเสนอแนะ

บทที่ 2
ทฤษฎีพื้นฐานที่ใช้ในการทำวิจัย

ในบทนี้กล่าวถึงทฤษฎีพื้นฐานต่างๆ ที่เกี่ยวข้องในการวิจัย โดยเนื้อหาภายในบทประกอบด้วย การจัดการหน่วยความจำแบบพลวัตเบื้องต้น เทคนิคการจองหน่วยความจำ การจองหน่วยความจำด้วยฮาร์ดแวร์ระบบไบนารีบัดดิและและบิตแมป การคืนค่าวัตถุที่ไม่ได้ใช้งาน จาวาเวอร์ชวลแมชชีน และ KVM

2.1 การจัดการหน่วยความจำแบบพลวัตเบื้องต้น [3]

เป้าหมายของการออกแบบหน่วยจัดการหน่วยความจำแบบพลวัต คือเพื่อใช้พื้นที่หน่วยความจำอย่างมีประสิทธิภาพและใช้ทรัพยากรน้อยที่สุด โดยต้องสามารถค้นหาส่วนของหน่วยความจำที่ถูกใช้ไปและส่วนหน่วยความจำที่ว่างได้ โดยปกติแล้วหน่วยจองพื้นที่หน่วยความจำไม่สามารถนำพื้นที่หน่วยความจำส่วนที่ว่างนำมารวมกันได้ ทำได้เพียงหาพื้นที่ว่างที่เพียงพอต่อการใช้งานเท่านั้น [4] และการเก็บตำแหน่งและขนาดของบล็อกหน่วยความจำที่ว่าง จะเก็บในรูปแบบโครงสร้างข้อมูล ซึ่งอาจอยู่ในรูปของลิสต์ (list) ทรี (tree) หรือ บิตแมป

2.1.1 Fragmentation

fragmentation เป็นปัญหาที่สำคัญที่สุดของการจัดการหน่วยความจำ โดยปัญหานี้ทำให้การใช้งานหน่วยความจำเป็นไปอย่างไม่มีประสิทธิภาพ โดย fragmentation สามารถแบ่งได้ทั้งแบบภายใน (Internal Fragmentation) และภายนอก (External Fragmentation)

- **แบบภายใน** เกิดเมื่อบล็อกที่ต้องการจองมีขนาดน้อยกว่าขนาด บล็อกว่างที่สามารถจองได้ แต่จำเป็นต้องจองพื้นที่บล็อกนั้นใช้งาน ซึ่งทำให้พื้นที่ส่วนไม่ได้ใช้งานในบล็อกนั้นไม่สามารถใช้งานได้
- **แบบภายนอก** เกิดเมื่อบล็อกหน่วยความจำที่ว่างมีขนาดเล็กและจำนวนมาก แต่ไม่สามารถจองใช้งานได้ เนื่องจากหน่วยความจำที่ต้องการจองมีขนาดใหญ่กว่า

2.1.2 การรวมและการแยกบล็อกหน่วยความจำ

หน่วยจัดการหน่วยความจำอาจจำเป็นต้องแยกบล็อก (split) ออกเป็นบล็อกย่อยๆ เพื่อรองรับการจองบล็อกหน่วยความจำที่มีขนาดเล็ก และอาจต้องมีการรวมบล็อก (coalesces)โดยนำบล็อกที่มีขนาดเล็กมารวมให้มีขนาดใหญ่ เพื่อให้สามารถจองพื้นที่สำหรับบล็อกที่ขนาดใหญ่ได้ และเมื่อบล็อกที่นำมารวมกันไม่มีการใช้งานแล้ว หน่วยจัดการหน่วยความจำอาจนำบล็อกที่อยู่ใกล้

เดียงนำมารวมเป็นบล็อกว่าง เพื่อใช้ได้อีกในอนาคต โดยกระบวนกาแยกและรวมบล็อกอาจทำให้ปัญหา fragmentation เพิ่มขึ้นได้

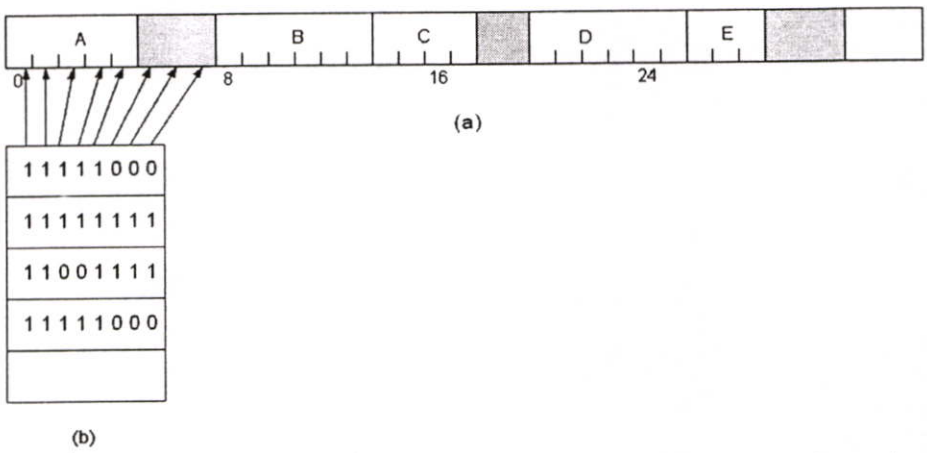
2.2 เทคนิคการจองหน่วยความจำ

ปัจจุบันมีหลายเทคนิคของการจองหน่วยความจำที่พยายามแก้ปัญหา fragmentation แม้ว่าจะมีบางเทคนิคที่สามารถลดการเกิด external fragmentation หรือบางเทคนิคแก้ปัญหา internal fragmentation แต่ก็ยังไม่มีเทคนิคใดที่สามารถแก้ปัญหาการเกิด fragmentation ได้อย่างมีประสิทธิภาพ โดยเทคนิคการจัดการจองหน่วยความจำเบื้องต้นแบ่งออกเป็น 3 ประเภท [5] ได้แก่

- เทคนิคบิตแมป (Bitmap)
- เทคนิคลิงค์ลิสต์ (Linked list)
- เทคนิคระบบบัดดิ (Buddy systems)

2.2.1 เทคนิคบิตแมป

พื้นฐานของเทคนิคบิตแมปคือ สถานะของแต่ละบล็อกในหน่วยความจำแทนด้วยบิตในบิตแมป ซึ่งบิต '0' แทนบล็อกที่ว่าง และบิต '1' แทนบล็อกที่ถูกจอง ในรูปที่ 2.1 แสดงส่วนของหน่วยความจำและบิตแมปที่สัมพันธ์



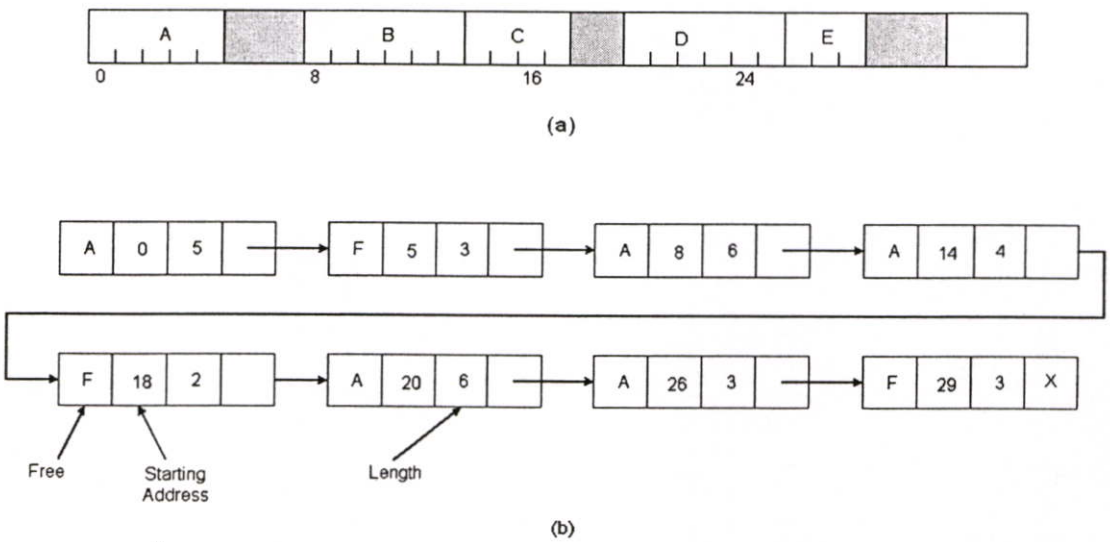
รูปที่ 2.1 แสดงความสัมพันธ์ระหว่าง (a) ส่วนหน่วยความจำ และ (b) บิตแมป

เทนนิคนี้มีความง่ายในการแยกบล็อก แต่ปัญหาหลักคือเมื่อนำโปรเซส K ไปประมวลผลในหน่วยความจำ หน่วยจัดการหน่วยความจำต้องค้นหาบิต '0' ติดกันในบิตแมป สำหรับรองรับโปรเซส K และในการค้นหาบิตแมปนี้มีผลทำให้ความเร็วของระบบลดลง ดังนั้นด้วยเหตุผลนี้บิตแมปจึงไม่นิยมใช้

2.2.2 เทคนิคลิงค์ลิสต์

เทคนิคการจัดการหน่วยความจำโดยใช้ลิงค์ลิสต์เป็นเทคนิคที่นิยมใช้มากในปัจจุบัน เนื่องจากมีความง่ายและยืดหยุ่นต่อการใช้งาน โดยสามารถแบ่งออกเป็นเทคนิคย่อยๆ ดังนี้

- **First Fit** จองหน่วยความจำบล็อกแรกที่เจอและใหญ่เพียงพอ
- **Next Fit** จองหน่วยความจำบล็อกแรกที่เจอและใหญ่เพียงพอ แต่จะแตกต่างจากเทคนิค first fit ตรงที่การค้นหาเริ่มต้นที่ตำแหน่งใดๆในลิสต์ก็ได้
- **Best Fit** ทำการค้นหาในลิสต์เพื่อหาขนาดบล็อกที่เล็กที่สุดที่ใหญ่เพียงพอ แล้วทำการจองบล็อกนั้น เทคนิคนี้ช้ากว่าแบบ first fit และ next fit เพราะว่าต้องทำการค้นหาทุกรายการในลิสต์
- **Worst Fit** การทำงานของเทคนิคนี้คล้ายๆกับแบบ best fit แต่แตกต่างตรงที่แบบ best fit เลือกบล็อกที่ขนาดเล็กที่สุด แต่แบบ worst fit เลือกบล็อกที่ขนาดใหญ่ที่สุด



รูปที่ 2.2 แสดงความสัมพันธ์ระหว่าง (a) ส่วนหน่วยความจำและ (b) ลิงค์ลิสต์

ในรูปที่ 2.2 แสดงความสัมพันธ์ระหว่างส่วนหน่วยความจำและลิงค์ลิสต์ที่แทนสถานะบล็อกในหน่วยความจำ ตัวอย่างเช่น เมื่อต้องการจอง 2 บล็อก ถ้าแบบ first fit จะจองบล็อกที่ 5 แบบ best-fit จะจองบล็อกที่ 18 และแบบ worst fit จะบล็อกที่ 8

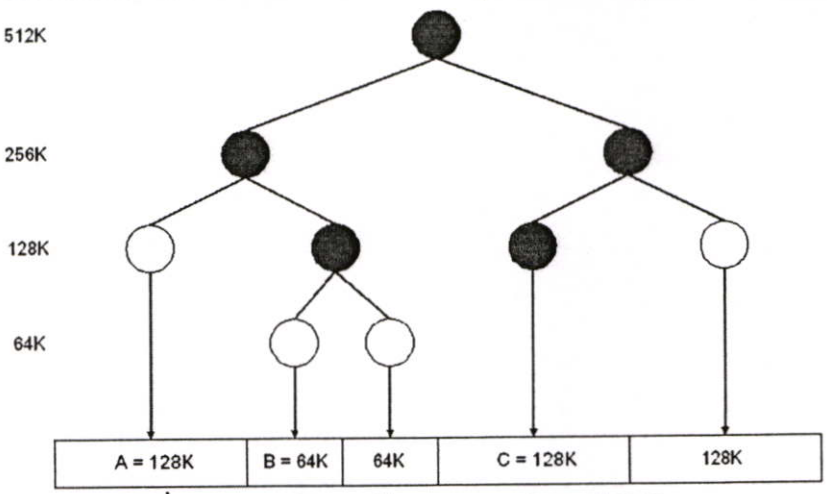
2.2.3 เทคนิคระบบบัดดิ

เทคนิคนี้ใช้โครงสร้างข้อมูลแบบทรี ดังนั้นการหาบล็อกหน่วยความที่ว่างจึงทำได้เร็ว ซึ่งช่วยแก้ปัญหาเรื่องเวลาที่ใช้ในการค้นหาและการจองบล็อกหน่วยความจำ และแม้ว่าระบบบัดดิจะ

ทำงานได้เร็ว แต่มีข้อเสียคือการใช้หน่วยความจำไม่ค่อยมีประสิทธิภาพ นั่นคือมีปัญหาทั้ง external fragmentation และ internal fragmentation เนื่องจากการจองบล็อกมีขนาดเป็นกำลังของสอง

2.2.3.1 ระบบไบนารีบัดดิ (Binary Buddy System)

ระบบไบนารีบัดดิ มีการทำงานดังนี้ เริ่มต้นด้วยมีบล็อกขนาด 2U และต้องการจองบล็อกขนาด S โดยถ้า 2U-1 < S <= 2U เป็นจริงแล้ว จะได้บล็อกขนาด 2U ออกมา แต่ถ้าไม่ใช่ให้ทำการแยกบล็อกออกเป็นสองส่วน และแต่ละส่วนที่ได้มีขนาด 2U-1 และถ้า U-2 < S <= 2U-1 เป็นจริงแล้ว สามารถจองบล็อกทรีย่อยในไบนารีบัดดิ แต่ถ้าไม่ใช่ทรีย่อย ต้องทำการแยกออกเป็นสองอีกครั้ง และกระบวนการนี้จะทำจนกระทั่งได้บล็อกเล็กที่สุดที่ขนาดใหญ่กว่าหรือเท่ากับ S [6]



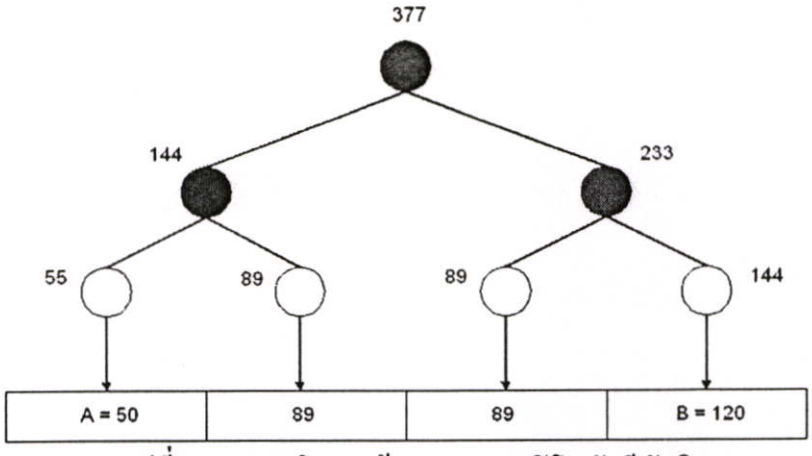
รูปที่ 2.3 แสดงโครงสร้างของระบบไบนารีบัดดิ

2.2.3.2 ระบบฟิโบนักชีบัดดิ (Fibonacci Buddy System)

ในระบบฟิโบนักชีบัดดิบล็อกถูกแยกเป็นจำนวนฟิโบนักชีโดยการทำงานจะคล้ายๆ กับแบบไบนารีบัดดิ โดยอนุกรมฟิโบนักชี กำหนดดังนี้

$$F_0 = 0, F_1 = 1, F_{n+2} = F_{n+1} + F_n (n >= 0)$$

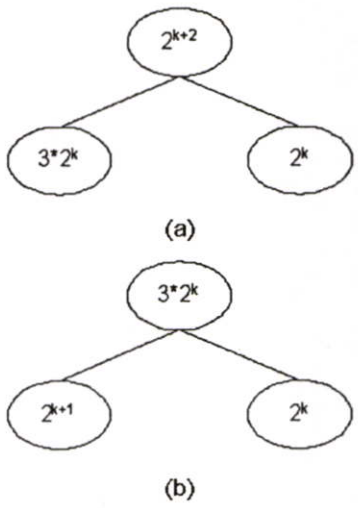
จากอนุกรมฟิโบนักชี สมาชิกที่ได้มีดังนี้ 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987 ... โดยในรูปที่ 2.4 แสดงโครงสร้างของระบบฟิโบนักชีอย่างง่าย



รูปที่ 2.4 แสดงโครงสร้างของระบบฟีโบนักชีบัดดิ

2.2.3.4 ระบบเวทบัดดิ (Weighted Buddy Sytem)

ในเทคนิคนี้ขนาดของบล็อกเป็น 2^k หรือ $3*2^k$ จากรูปที่ 2.5 (a) เมื่อบล็อกขนาด 2^{k+2} ถูกแยกจะได้บล็อกขนาด $3*2^k$ และ 2^k และบล็อกขนาด $3*2^k$ จากรูปที่ 2.5 (b) สามารถแยกได้เป็นบล็อกขนาด 2^k และ 2^{k+1} $[3*2^k \rightarrow (2^{k+1}, 2^k)]$

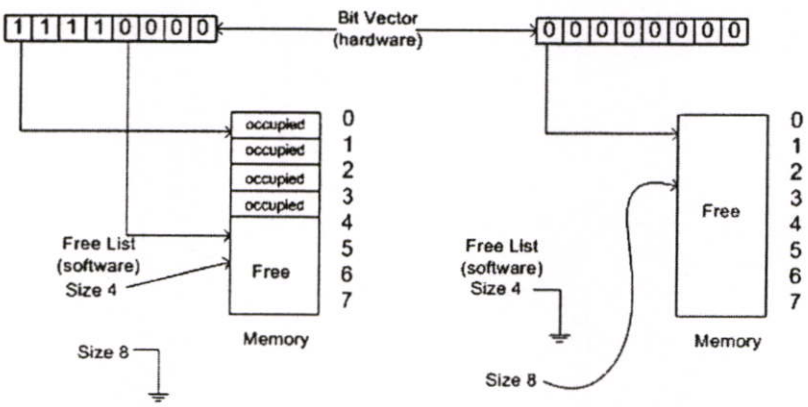


รูปที่ 2.5 แสดงโครงสร้างของระบบเวทบัดดิ

2.3 การจองหน่วยความจำด้วยฮาร์ดแวร์ระบบไบนารีบัดดิและบิตแมป

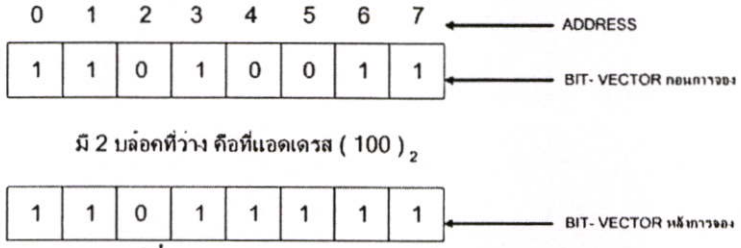
ในการเพิ่มประสิทธิภาพให้แก่หน่วยจัดการหน่วยความจำแบบพลวัต นักวิจัยพยายามคิดค้นวิธีการใหม่ๆมาแก้ปัญหา ส่วนมากเป็นวิธีการทางซอฟต์แวร์ซึ่งยังไม่ช่วยเพิ่มประสิทธิภาพได้มากนัก ใน [1] ได้ทำการออกแบบวิธิการจองพื้นที่สำหรับวัตถุใหม่ด้วยฮาร์ดแวร์ โดยใช้วิธีการระบบไบนารีบัดดิและบิตแมป ทำให้สามารถจองพื้นที่ได้เร็วขึ้นและสามารถทำงานด้วยเวลาที่คงที่ จากแนวคิดดังกล่าวสามารถช่วยเป็นแนวทางในการออกแบบ garbage collector ด้วยวิธีการทางฮาร์ดแวร์เพื่อเพิ่มประสิทธิภาพในการจัดการหน่วยความจำได้

ใน [7] เมื่อมีการร้องขอจองพื้นที่หน่วยความจำระบบไบนารีบัดดิจะจองบล็อกเป็นจำนวนกำลังของสองตามขนาดที่ต้องการ โดยแบ่งบล็อกออกเป็นสองส่วนทุกครั้งที่เป็นไปได้ จนกระทั่งได้บล็อกตามขนาดที่ต้องการ เมื่อบล็อกถูกแบ่งเป็นสองแล้ว ส่วนที่สองเรียกว่าบัดดิ และเมื่อบล็อกถูกทำให้ว่างส่วนบัดดิต้องทำให้ว่างด้วยเช่นกัน จากนั้นจะนำมารวมกันให้พื้นที่ใหญ่ขึ้นเพื่อสามารถจองได้อีกได้ในภายหลัง ในวิธีการทางซอฟต์แวร์กระบวนการในการแบ่งและรวมบล็อกบัดดิทำให้เสียค่าใช้จ่ายมาก ขณะวิธีการทางฮาร์ดแวร์ใช้เทคนิคบิตแมบมาช่วย ทำให้กำจัดกระบวนการแบ่งและรวมบล็อกออกไป โดยกระบวนแบ่งบล็อกไม่จำเป็นต้องมี เพราะว่าการจอง (Allocation) ใช้วงจรรวมที่จัดการแบบไบนารีทรี (Binary tree) เพื่อหาบล็อกว่าง และใช้บิตเวกเตอร์(Bit-vector) แทนบล็อกในหน่วยความจำ โดยบล็อกไหนถูกจองก็จะเซตบิตในบิตเวกเตอร์ และสำหรับการคืนพื้นที่ (Deallocation) ทำได้โดยรีเซตบิตที่สัมพันธ์กับบล็อกที่ต้องการในบิตเวกเตอร์ซึ่งสามารถกำจัดกระบวนการรวมบล็อกไปได้ ในรูปที่ 2.6 แสดงตัวอย่างของกระบวนการรวมบล็อกระหว่างวิธีการทางซอฟต์แวร์และฮาร์ดแวร์ และเนื่องจากวิธีการทางฮาร์ดแวร์กระทำโดยวงจรรวมอย่างเดียวจึงทำให้สามารถลดเวลาในการจัดการหน่วยความจำได้ [1]



รูปที่ 2.6 แสดงการรวมบล็อกระหว่างวิธีการทางซอฟต์แวร์และฮาร์ดแวร์

ในระบบไบนารีบัดดิขนาดของพื้นที่หน่วยความจำที่จองจะเป็นกำลังของสอง ซึ่งอาจทำให้เกิดพื้นที่ไม่ได้ใช้งาน ที่เรียกว่า internal fragmentation แต่ถ้าจัดการด้วยฮาร์ดแวร์ทำให้สามารถกำจัด internal fragmentation ไปได้ ในรูปที่ 2.7 แสดงตัวอย่างการจองบล็อกหน่วยความจำ และจากการจองพื้นที่ด้วยฮาร์ดแวร์ระบบไบนารีบัดดิและบิตแมบ ช่วยลดการเกิด internal fragmantation ได้มาก เพราะถึงแม้ระบบจะจองพื้นที่เป็นขนาดของกำลังของสอง เช่นต้องการ 5 บล็อก ระบบก็จะหาบล็อกที่ว่างขนาด 8 บล็อกซึ่งจองจริงๆเพียง 5 บล็อกเท่านั้น ส่วนที่เหลือ 3 บล็อกจะนำไปรวมเป็นบล็อกว่างที่สามารถใช้งานได้อีกในอนาคต



รูปที่ 2.7 ตัวอย่างการจองบล็อกหน่วยความจำ

2.4 การคืนพื้นที่วัตถุที่ไม่ได้ใช้งาน

ส่วนการคืนพื้นที่วัตถุที่ไม่ได้งาน (Garbage collection) ของจาวาจะทำงานโดยอัตโนมัติ ซึ่งช่วยให้ผู้เขียนโปรแกรมไม่ต้องเสียเวลาในการคืนพื้นที่หน่วยความจำ มีข้อดี ดังนี้

1. ผู้เขียนโปรแกรมไม่ต้องมาคอยจัดการคืนพื้นที่หน่วยความจำ เมื่อหน่วยความจำไม่เพียงพอต่อการใช้งาน
2. ช่วยให้โปรแกรมมีความมั่นคง เพราะว่าส่วนการคืนค่าวัตถุที่ไม่ได้ใช้งานเป็นส่วนสำคัญหลักของกลไกรักษาความปลอดภัยของจาวา ทำให้ผู้เขียนโปรแกรมไม่ต้องคืนค่าหน่วยความจำโดยตรง ทำให้ป้องกันปัญหาการคืนค่าพื้นที่หน่วยความจำที่ไม่ถูกต้องได้

ส่วนข้อเสีย คือเพิ่ม overhead ทำให้ประสิทธิภาพการทำงานของโปรแกรมลดลง โดยจาวา-เวอร์ชวลแมชชีนจะเก็บเส้นทาง (Trace) ของวัตถุที่ถูกอ้างอิงหรือใช้งานโดยโปรแกรมที่กำลังทำงานอยู่ปัจจุบัน และทำการคืนพื้นที่หน่วยความจำของวัตถุที่ไม่ได้ถูกอ้างอิง ซึ่งการทำงานนี้จะกินเวลาส่วนหนึ่งของซีพียู มีผลให้โปรแกรมทำงานช้า การแก้ปัญหาอาจให้ผู้เขียนโปรแกรมควบคุมการใช้งานซีพียูในการจัดการคืนค่าพื้นที่หน่วยความเอง หรืออาจพัฒนาขั้นตอนวิธีในการทำการคืนพื้นที่วัตถุที่ไม่ได้งานดีเพียงพอที่จะทำให้โปรแกรมทำงานได้อย่างมีประสิทธิภาพ

2.4.1 เทคนิคการคืนพื้นที่วัตถุที่ไม่ได้ใช้งาน

การคืนพื้นที่วัตถุที่ไม่ได้ใช้งานเป็นการคืนค่าพื้นที่หน่วยความจำส่วนฮีปที่วัตถุจองไว้แต่ไม่ได้ใช้งานเป็นเวลานาน การคืนพื้นที่นี้จะทำโดยอัตโนมัติ [2] ขณะที่ในหลายระบบ ผู้เขียนโปรแกรมต้องทำการคืนค่าพื้นที่หน่วยความจำส่วนฮีปเอง ด้วยคำสั่ง “free” ซึ่งฟังก์ชันการทำงานของหน่วยคืนพื้นที่วัตถุไม่ได้ใช้งาน ทำหน้าที่หาวัตถุที่ไม่ได้ใช้งานเป็นเวลานาน แล้วทำการคืนพื้นที่หน่วยความจำที่วัตถุนั้นจองอยู่ เพื่อให้มีพื้นที่ว่างสำหรับวัตถุอื่นใช้งานได้ โดยวัตถุจะถูก เรียกว่าเป็น “garbage” ก็ต่อเมื่อตัวชี้ไม่สามารถชี้ไปถึงในช่วงระหว่างโปรแกรมทำงาน และวัตถุที่ถูกอ้างถึงในช่วงระหว่างโปรแกรมกำลังทำงาน เรียกเป็น “live”

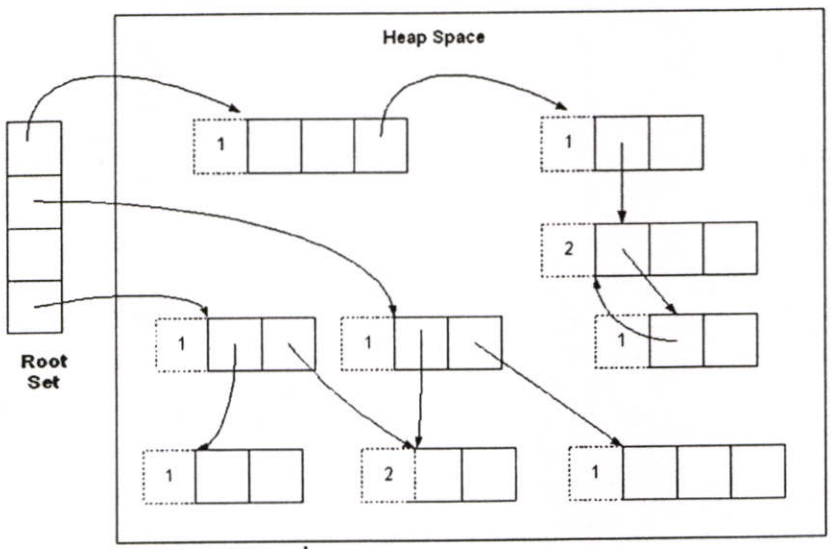
การคืนพื้นที่วัตถุที่ไม่ได้ใช้งานจะทำโดยอัตโนมัติ ซึ่งการทำงานพื้นฐาน ประกอบด้วย 2

ขั้นตอน ได้แก่ การจำแนกวัตถุ live ออกจากวัตถุที่เป็น garbage และการคืนพื้นที่ที่ถูกจองโดยวัตถุ garbage ซึ่งในทางปฏิบัติทั้งสองขั้นตอนอาจทำงานเป็นฟังก์ชันอยู่ภายในโปรแกรม และเทคนิคการคืนพื้นที่ที่ถูกจองโดยวัตถุ garbage จะทำงานได้ดีจะขึ้นอยู่กับเทคนิคการค้นหาวัตถุที่เป็น garbage

เนื่องจากการคืนพื้นที่วัตถุที่ไม่ได้ใช้งานมีขั้นตอนพื้นฐานอยู่ 2 ขั้นตอน [2] โดยขั้นตอนแรก คือการจำแนกวัตถุ live ออกจากวัตถุ garbage และขั้นตอนที่สองเป็นการคืนพื้นที่วัตถุที่จองโดยวัตถุ garbage และการทำงานทั้งสองขั้นตอนนี้ มี 2 วิธีพื้นฐานได้แก่ วิธี reference counting และวิธี tracing โดยวิธี reference counting มีตัวนับสำหรับทำการนับจำนวนตัวชี้ของแต่ละวัตถุ และตัวนับนี้ถูกใช้กำหนดว่าวัตถุไหนเป็นวัตถุ live ส่วนวิธี Tracing collector ทำการติดตามไปยังทุกวัตถุที่ถูกอ้างอิงโดยโปรแกรมที่กำลังทำงานอยู่ในปัจจุบัน โดยวัตถุไหนที่ติดตามไม่ถึงจะกำหนดให้เป็นวัตถุ garbage ซึ่งมีหลายวิธีด้วยกัน เช่นวิธี mark-sweep วิธี mark-compact และวิธี copying

2.4.2.1 วิธี Reference Counting

Reference counting เป็นเทคนิคการคืนพื้นที่วัตถุที่ไม่ได้ใช้งานที่ง่ายที่สุด โดยแต่ละวัตถุจะมีตัวนับการอ้างอิง (reference count) ถึงตัวเองอยู่ภายใน เมื่อวัตถุถูกสร้างในครั้งแรก ตัวนับการอ้างอิงถูกกำหนดเป็นหนึ่ง และเมื่อวัตถุอื่นๆ หรือ root ถูกกำหนดให้อ้างไปยังวัตถุนั้น ตัวนับการอ้างอิงของวัตถุนั้นก็จะเพิ่มขึ้นหนึ่ง และเมื่อมีวัตถุนั้นไม่มีการใช้งาน หรือมีการกำหนดค่าให้ใหม่ ตัวนับนั้นจะถูกลดค่าลงหนึ่ง แสดงดังรูปที่ 2.8



รูปที่ 2.8 แสดงตัวนับอ้างอิง

และเมื่อวัตถุใดๆมีค่าตัวนับอ้างอิงเป็นศูนย์ วัตถุนั้นจะถูกกำหนดเป็นวัตถุ garbage ซึ่งต้องทำการคืนพื้นที่ที่วัตถุนั้นจองไว้ และเมื่อวัตถุที่เป็น garbage นั้นอ้างถึงวัตถุใดต้องทำการลบ

ค่าตัวนับอ้างอิงลงหนึ่งด้วย ฉะนั้นการคืนพื้นที่ไม่ได้ใช้งานของวัตถุหนึ่ง อาจจะนำไปสู่การทำการคืนพื้นที่ไม่ได้ใช้งานของวัตถุอื่นด้วย

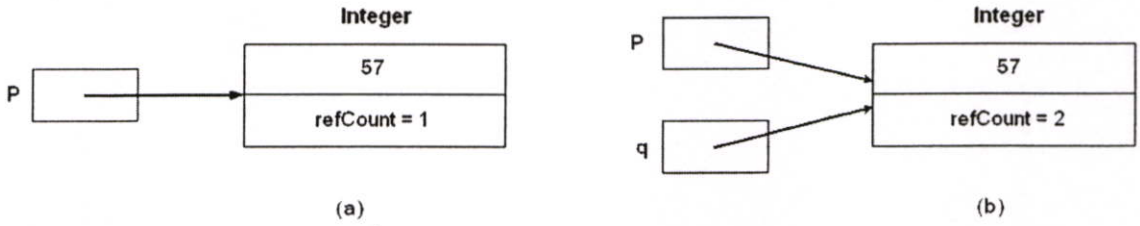
ประโยชน์ของเทคนิคนี้ คือสามารถทำงานในเวลาสั้นๆระหว่างที่โปรแกรมกำลังทำงานอยู่ได้ คุณสมบัติตรงนี้ทำให้เหมาะสมกับงานที่ทำงานในสภาพแวดล้อมเวลาจริงที่โปรแกรมไม่สามารถไปทำงานอย่างอื่นเป็นเวลานานได้ ส่วนข้อเสียของวิธีนี้คือไม่สามารถตรวจสอบกรณีที่วัตถุอ้างอิงกันเป็นวงกลมได้ การอ้างอิงกันเป็นวงกลม คือการมีวัตถุมากกว่า 2 วัตถุอ้างอิงถึงกันและกัน ตัวอย่างเช่น วัตถุแม่อ้างอิงไปถึงวัตถุลูก และวัตถุลูกก็อ้างอิงกลับไปยังวัตถุแม่ ซึ่งทั้งสองวัตถุนี้ไม่มีทางที่ตัวนับการอ้างอิงจะมีค่าเป็นศูนย์เลย ซึ่งถึงแม้ว่าวัตถุทั้งสองไม่สามารถถูกอ้างถึงโดย root ในระหว่างที่โปรแกรมกำลังทำงานอยู่ก็ตาม ส่วนข้อเสียอื่นคือทำให้เสียเวลาในการจัดการการเพิ่มค่าตัวนับอ้างอิงและลดค่าตัวนับอ้างอิงในแต่ละครั้ง และเนื่องจากมีข้อเสียทั้งสองกรณีดังที่ได้กล่าวมาจึงทำให้เทคนิค reference counting ไม่เป็นที่นิยม

แนวความคิดของขั้นตอนการคืนพื้นที่วัตถุไม่ได้ใช้งาน ประกอบด้วย 2 ขั้นตอนคือ การปรับค่าและตรวจสอบค่าตัวนับอ้างอิง (adjustment and checking) และการคืนพื้นที่ใช้งานของวัตถุเมื่อตัวนับอ้างอิงมีค่าเป็นศูนย์ โดยทั้งสองขั้นตอนนี้ทำงานภายในเวลาสั้นๆ (interleaved) ระหว่างการประมวลผลโปรแกรม

พิจารณาประโยคคำสั่งต่อไปนี้

```
Object p = new Integer(57);
```

คำสั่งนี้จะสร้างอินสแตนซ์ (Instance) ของคลาส integer และมีเพียงค่าเดียว คือตัวแปร p เท่านั้นที่ชี้ไปยังวัตถุ interger ดังนั้นตัวนับอ้างอิงจะมีค่าเป็น 1 แสดงดังรูปที่ 2.9 (a)



รูปที่ 2.9 แสดงวัตถุและตัวนับอ้างอิง

พิจารณาประโยคคำสั่งต่อไปนี้

```
Object p = new Integer (57);
Object q = p;
```

คำสั่งเหล่านี้จะสร้างอินสแตนซ์ของคลาส integer เพียงอินสแตนซ์เดียว แต่ทั้ง p และ q อ้างไปยังวัตถุเดียวกัน ดังนั้นตัวนับอ้างอิงของวัตถุ integer จะมีค่าเป็น 2 แสดงดังรูปที่ 2.9 (b) และทุกครั้งที่ตัวแปรอ้างอิงหนึ่งถูกกำหนดค่าให้กับตัวแปรอ้างอิงอื่น จำเป็นต้องอัพเดตตัวนับอ้างอิงด้วย สมมุติว่าทั้ง p และ q เป็นตัวแปรอ้างอิง (reference variable) กำหนดค่าดังนี้

```
p = q;
```

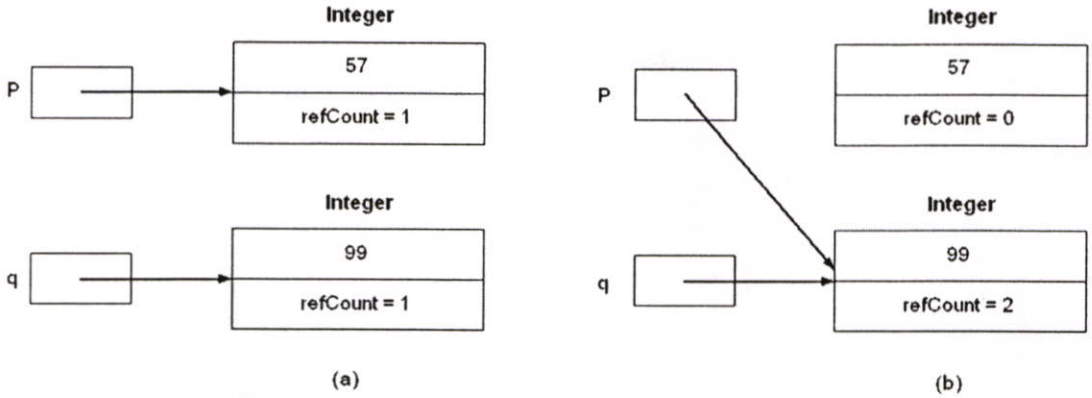
เขียนด้วยจาวาเวอร์ชวลแมชชีน ดังนี้

```
if(p != q)
{
    if (p != null)
            --p.refCount;
    p = q;
    if (p != null)
            ++p.refCount;
}
```

ให้ p และ q มีค่าเริ่มต้นดังนี้

```
Object  p = new Integer(57);
Object  q = new Integer(99);
```

จากรูปที่ 2.10 (a) มีวัตถุ integer ถูกสร้างสองวัตถุ และแต่ละวัตถุตัวนับอ้างอิงมีค่าเป็น 1 และกำหนดให้ p = q ผลที่ได้แสดงดังรูปที่ 2.10 (b) ซึ่งจะเห็นว่าทั้ง p และ q ชี้ไปยังวัตถุเดียวกัน เป็นผลทำให้ตัวนับอ้างอิงของวัตถุ Integer(99) มีค่าเป็น 2 และตัวนับอ้างอิงของวัตถุ Integer(57) มีค่าเป็นศูนย์ เพราะว่าตัวอ้างอิง p เปลี่ยนการชี้ไปยังวัตถุ Integer(57) แทน



รูปที่ 2.10 แสดงค่าตัวนับอ้างอิงก่อนและหลังกำหนด p = q

พิจารณาคำสั่งในจาวาเวอร์ชวลแมชชีนในการกำหนด p = q ดังนี้

```
if(p != q)
 {
  if(p != null)
       if(--p.refCount == 0)
          heap.release(p);
    p = q;
    if(p != null)
       ++p.refCount;
 }
```

จากชุดคำสั่งข้างต้นฟังก์ชัน release จะถูกเรียกใช้งานเมื่อตัวนับอ้างอิงของวัตถุมีค่าเป็นศูนย์

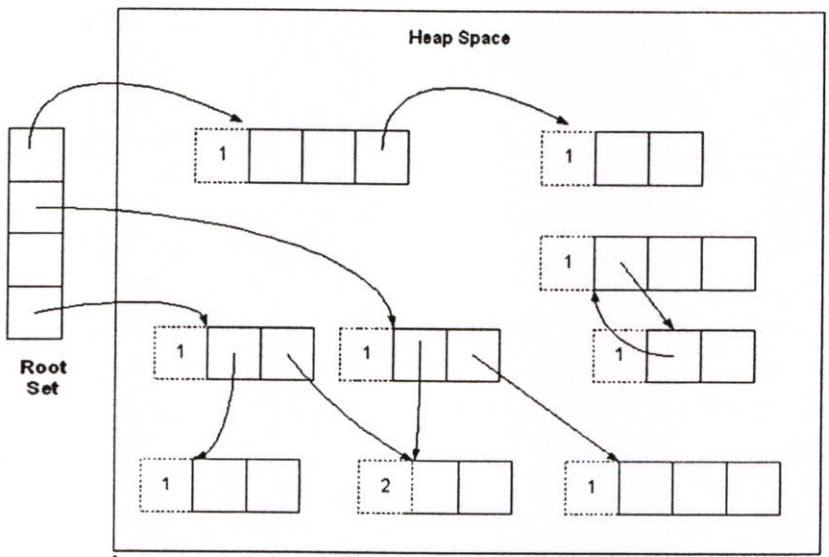
ประโยชน์ของการใช้ตัวนับอ้างอิง คือมีความง่ายในการใช้งาน และไม่ต้องรอให้หน่วยความจำมีไม่พอใช้งาน เราสามารถคืนค่าพื้นที่หน่วยความจำในทันทีเมื่อตัวนับอ้างอิงของวัตถุนั้นมีค่าเป็นศูนย์ และการคืนพื้นที่วัตถุที่ไม่ได้ใช้งานจะใช้เวลาสั้นๆในช่วงระหว่างที่โปรแกรมกำลังประมวลผล ซึ่งทำให้มีความง่ายในการจัดการและมั่นใจได้ว่าพื้นที่หน่วยความจำมีความเพียงพอต่อการใช้งาน และเหมาะกับการทำงานแบบเวลาจริง (real time) ทำให้สามารถรองรับงานที่ต้องการตอบสนองต่อเวลาจริงได้

ส่วนข้อเสียคือทำให้มีค่าใช้จ่ายเพิ่มมากขึ้น เพราะว่าทุกวัตถุมีฟิลด์ที่เก็บตัวนับอ้างอิงนั่นคือต้องมีการจองพื้นที่หน่วยความจำสำหรับเก็บตัวนับอ้างอิงเพิ่ม และทุกครั้งที่มีการอ้างอิงไปยังวัตถุอื่นจำเป็นต้องมีปรับตัวนับอ้างอิงทุกครั้ง ซึ่งทำให้มีค่าใช้จ่ายเพิ่มมากขึ้น และนอกจากนี้ยังมี

ปัญหาเกี่ยวกับพื้นที่ใช้เก็บตัวนับอ้างอิง ซึ่งจะเพิ่มมากขึ้นเมื่อมีการใช้งานวัตถุมากขึ้น โดยในบางระบบอาจกำหนดให้หน่วยความจำมีขนาดเป็นเวิร์ดในการเก็บตัวนับอ้างอิง นั่นคือตัวนับอ้างอิงจะมีขนาดเท่าใดก็ได้สามารถเก็บได้หมด ปัญหานี้อาจจะทำให้ทำงานไม่มีประสิทธิภาพ [2]

2.4.2.1.1 ปัญหาเรื่องการอ้างอิงกันเป็นวงกลม

ปัญหาที่ทำให้วิธี reference counting ทำงานไม่มีประสิทธิภาพคือการที่ไม่สามารถคืนค่าวัตถุที่อ้างอิงกันเป็นวงกลมได้ การอ้างอิงกันเป็นวงกลมคือ การที่ตัวชี้ในกลุ่มของวัตถุนั้นๆได้สร้างวงกลมขึ้น ทำให้ ตัวนับอ้างอิงของวัตถุนั้นไม่มีทางเป็นศูนย์



รูปที่ 2.11 แสดงการไม่สามารถคืนค่าวัตถุที่อ้างอิงเป็นวงกลม

ในรูปที่ 2.11 แสดงปัญหานี้ โดยให้พิจารณาคู่ของวัตถุด้านขวา ตัวชี้ของแต่ละวัตถุชี้กันเอง และแต่ละวัตถุมี ตัวนับอ้างอิงเป็นหนึ่งทั้งคู่ และไม่มีเส้นทางจาก root ไปยังวัตถุทั้งสอง นั่นคือทำให้โปรแกรมไม่สามารถเข้าถึงวัตถุทั้งสองผ่านทาง root ได้ เมื่อเป็นเช่นนี้ทำให้ไม่สามารถคืนพื้นที่ใช้งานของวัตถุทั้งสองนี้ได้

ในบางระบบใช้เทคนิคอื่นเข้ามาช่วย หรืออีกทางหนึ่งคือผู้เขียนโปรแกรมต้องเขียนโปรแกรมหลีกเลี่ยงการเกิดการอ้างอิงเป็นวงกลมของวัตถุ ทำให้สามารถแก้ปัญหาการอ้างอิงเป็นวงกลมได้ทางหนึ่ง แต่การทำแบบนี้อาจมีผลกระทบต่อโครงสร้างของโปรแกรมและอาจทำให้มีปัญหาเรื่องหน่วยความจำไม่เพียงพอตามมาด้วย เมื่อวัตถุ garbage ที่อ้างอิงกันเป็นวงกลมสะสมเพิ่มขึ้นเรื่อยๆ แต่ปัญหานี้อาจยอมรับได้ในกรณีที่ใช้ในงานตอบสนองตามเวลาจริง

2.4.2.1.2 ปัญหาเรื่องประสิทธิภาพการทำงาน

ปัญหาในเรื่องประสิทธิภาพของวิธี reference counting คือการใช้ทรัพยากรที่จำเป็นต้องใช้ที่เป็นไปตามจำนวนงานที่ทำในการทำงานโปรแกรมนั้นๆ คือเมื่อวัตถุถูกสร้างหรือไม่ได้ใช้งานแล้วตัวนับอ้างอิงของวัตถุต้องทำการปรับค่าและถ้าค่าตัวแปรอ้างอิงถูกเปลี่ยนจากตัวชี้หนึ่งไปยังตัวชี้อื่นตัวนับอ้างอิงทั้งสองวัตถุจะต้องทำการปรับ โดยวัตถุหนึ่งต้องเพิ่มค่า และอีกวัตถุหนึ่งต้องลดค่า และเมื่อลดค่าแล้วต้องทำการตรวจสอบค่าเป็นศูนย์ของตัวนับอ้างอิง

การปรับปรุงประสิทธิภาพ reference counting แนวทางหนึ่ง คือกำหนดฟิลด์สำหรับเก็บตัวนับอ้างอิงให้มีขนาดเล็ก ซึ่งอาจใช้เพียงบิตเดียวในการเก็บ แต่ก็มีเพียงส่วนน้อยเท่านั้นที่ตัวนับอ้างอิงไม่เป็นศูนย์ หรือถ้าไม่สามารถคืนค่าพื้นที่ใช้งาน ก็อาจใช้วิธี tracing collector แก้ปัญหาตรงนี้แทน

ปัญหาการใช้ทรัพยากรของวิธี reference counting คือการไม่สามารถคืนค่าพื้นที่ของวัตถุที่เป็น garbage ได้ โดยปกติแล้วเมื่อตัวนับอ้างอิงของวัตถุเป็นศูนย์ วัตถุนั้นต้องถูกคืนพื้นที่หน่วยความจำ โดยนำพื้นที่ว่างเหล่านี้มาเข้าอยู่ในลิสต์ว่าง (free list) เพื่อให้วัตถุอื่นจองพื้นที่ใช้งานได้ โดยฟิลด์ที่เก็บตัวชี้ต้องทำการทดสอบว่าพื้นที่นี้ว่างจริงหรือไม่ ซึ่งเป็นการยากที่จะทำให้คำสั่งสำหรับการคืนพื้นที่วัตถุที่ไม่ได้ใช้งานใช้จำนวนน้อยกว่า 10 คำสั่ง และปัญหาการใช้ทรัพยากรก็ขึ้นอยู่กับจำนวนของวัตถุที่ถูกจองโดยโปรแกรมที่กำลังทำงานอยู่ด้วย เมื่อนำปัญหาการใช้ทรัพยากรเหล่านี้ไปรวมกับการที่ไม่สามารถคืนพื้นที่วัตถุที่มีการอ้างอิงเป็นวงกลมจึงทำให้ไม่เป็นที่นิยมใช้ในปัจจุบัน โดยเทคนิคที่นิยมใช้ซึ่งมีประสิทธิภาพและยืดหยุ่นมากกว่าวิธี reference counting จะอธิบายในหัวข้อ 2.4.2.2

แต่อย่างไรก็ตามวิธี reference counting ยังคงมีประโยชน์ นั่นคือการคืนพื้นที่วัตถุไม่ได้ใช้งานจะทำในทันทีทันใดช่วยทำให้การใช้งานหน่วยความจำเป็นไปอย่างมีประสิทธิภาพ ส่วนในกรณีที่ไม่สามารถคืนค่าพื้นที่ของวัตถุที่อ้างอิงกันเป็นวงกลมได้นั้น ไม่เป็นปัญหาในบางภาษาโปรแกรม นั่นคือไม่ยอมให้มีการสร้างโครงสร้างข้อมูลที่จะทำให้วัตถุอ้างอิงเป็นวงกลมได้

2.4.2.2 วิธี Mark-Sweep Collection

ใช้เทคนิคการค้นหาวัตถุ live โดยเริ่มต้นจากโหนดราก และวัตถุที่เจอระหว่างการค้นหาก็ทำการ mark (การ mark คือการเซตแฟล็ดในวัตถุ หรือกำหนดบิตเป็นลอจิก '1' ในบิตแมป) และหลังจากทำการค้นหาวัตถุ live เรียบร้อยแล้ว วัตถุใดที่ไม่ได้ถูก mark ก็ให้คิดว่าเป็นวัตถุ garbage นั่นคือต้องทำการคืนค่าพื้นที่หน่วยความจำ เรียกว่าการ sweep สำหรับเทคนิคการคืนพื้นที่วัตถุที่ไม่ได้ใช้งาน ประกอบด้วย 2 ขั้นตอน คือ

1. จำแนกวัตถุ live ออกจากวัตถุ garbage โดยการค้นหาวัตถุ เริ่มจากโหนดรากก่อน แล้วเดินทางไปยังวัตถุต่างๆในหน่วยความจำในลักษณะกราฟ ซึ่งอาจเดินทางแบบเชิงลึก (depth-first) หรือแบบเชิงกว้าง (breadth-first) โดยวัตถุที่ถูกหาเจอจะทำการ mark เพื่อบอกว่าวัตถุนี้กำลังถูกใช้งานอยู่
2. การคืนค่าวัตถุที่ไม่ได้งาน เมื่อวัตถุ live ถูกจำแนกออกจากวัตถุ garbage ก็จะทำการเคลียร์หน่วยความจำ เรียกว่าการ sweep หน่วยความจำ นั่นคือจะทำการค้นหาวัตถุที่ไม่ถูก mark และทำการคืนพื้นที่หน่วยความจำที่วัตถุจองอยู่

เมื่อวัตถุคืนพื้นที่แล้วจะถูกลิงค์เข้าด้วยกันเป็นลิงค์ลิสต์ว่าง เพื่อให้ฟังก์ชันการจองพื้นที่เข้ามาใช้งาน สำหรับการจองพื้นที่ใช้งานมีปัญหา คือ

1. ยากในการกำหนดให้พื้นที่หน่วยความจำของวัตถุมีขนาดต่างกันโดยไม่เกิด fragmentation และวัตถุที่เป็น garbage ถูกคืนพื้นที่ใช้งานแล้วจะกระจัดกระจายปนกับวัตถุ live ดังนั้นเมื่อมีความจำเป็นต้องจองพื้นที่หน่วยความจำขนาดใหญ่ให้กับวัตถุ จึงเป็นการยากที่จะจองได้ เพราะพื้นที่หน่วยความจำที่ว่างอยู่กระจัดกระจายไปกับพื้นที่หน่วยความจำที่วัตถุ live จองอยู่ และไม่สามารถนำพื้นที่เหล่านี้มารวมกันให้มีขนาดใหญ่ได้ แต่สามารถแก้ได้โดยเก็บแอดเดรสของวัตถุที่ถูกคืนพื้นที่แล้วให้อยู่ในลิสต์ว่างที่มีขนาดต่างๆกันและนำพื้นที่ว่างมารวมกัน แต่ก็ยากในการจัดการพื้นที่ว่างส่วนที่เหลือ
2. ถ้าไม่มีคำสั่งทำการคืนพื้นที่หน่วยความจำ เนื่องจากมีทั้งพื้นที่หน่วยความจำของวัตถุ live และพื้นที่หน่วยความจำของวัตถุ garbage อยู่ในพื้นที่ส่วนฮีปนี้ จะทำให้มีพื้นที่หน่วยความจำส่วนฮีปเพิ่มขึ้น

2.4.2.3 วิธี Compacting collectors

เป็นวิธีที่นำมาแก้ปัญหา fragmentation ของหน่วยความจำส่วนฮีฟโดยวิธีนี้จะทำการปรับย้ายที่อยู่ของวัตถุในหน่วยความจำเพื่อลดปัญหา fragmentation วิธีการคือทำการเลื่อนวัตถุ live ไปในส่วนพื้นที่หน่วยความฮีฟส่วนล่างสุด เพื่อให้มีพื้นที่ว่างเพียงพอต่อการจองใช้งานของวัตถุใหม่ และวัตถุที่ถูกเลื่อนไปยังตำแหน่งใหม่ก็ทำการอัพเดตตัวอ้างอิงตามตำแหน่งที่ย้ายไปด้วย การอัพเดตตัวอ้างอิงวิธีง่ายสุดคือ สร้างตัวอ้างอิงซ้อนอีกชั้นหนึ่งเพื่อชี้ไปยังตำแหน่งใหม่ในหน่วยความจำส่วนฮีฟเมื่อวัตถุถูกคืนพื้นที่ใช้งาน

2.4.2.4 วิธี Copying collectors

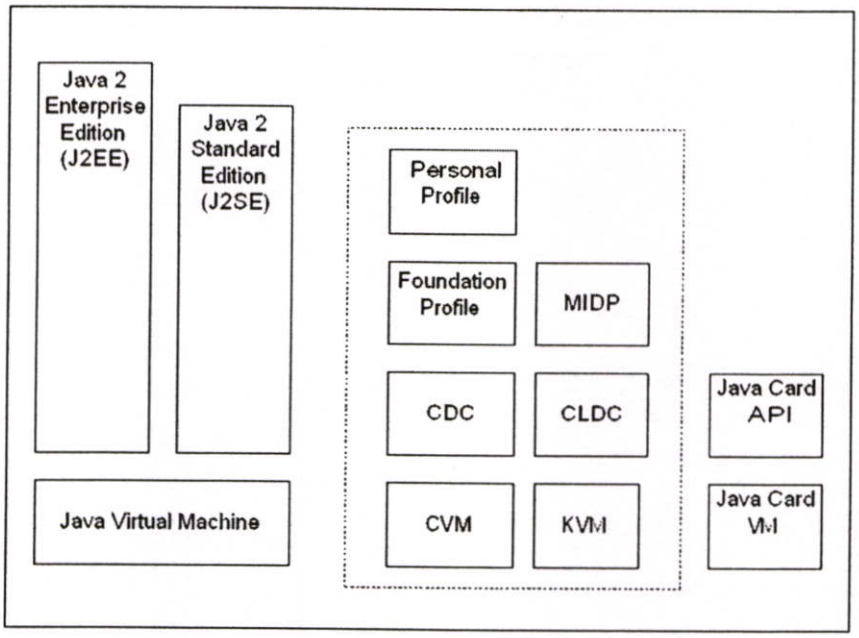
วิธีนี้ช่วยแก้ปัญหา fragmentation โดยจะคัดลอกวัตถุ live ทั้งหมดไปยังพื้นที่หน่วย-

ความจำส่วนใหม่ ซึ่งช่วยแก้ปัญหาพื้นที่หน่วยความจำว่างที่ไม่ต่อเนื่องกัน ประโยชน์ของวิธีนี้คือ วัตถุสามารถถูกคัดลอกทั้งหมดซึ่งครอบคลุมไปถึงโหนดรากและไม่มีการแยกขั้นตอนการ mark และ sweep

วิธีนี้เรียกว่า stop และ copy ซึ่งหน่วยความจำส่วนฮีปแบ่งออกเป็น 2 ส่วน โดยมีเพียงส่วนเดียวเท่านั้นที่ถูกใช้ตลอดเวลา วัตถุจะถูกจองจากพื้นที่ส่วนที่ 1 จนกระทั่งพื้นที่ในส่วนนี้ ไม่เพียงพอต่อการใช้งาน ณ จุดนี้โปรแกรมที่กำลังประมวลผลอยู่จะหยุดทำงาน (ขั้นตอน stop) และหน่วยความจำส่วนฮีปจะถูกเข้าถึงเพื่อค้นหาวัตถุ live และทำการคัดลอกไปยังพื้นที่หน่วยความจำฮีปส่วนที่ 2 (ขั้นตอน copy) เมื่อขั้นตอน stop และ copy ทำงานเสร็จ โปรแกรมก็กลับมาประมวลผลต่อ โดยทำงานกับหน่วยความจำฮีปส่วนที่ 2 จนกระทั่งพื้นที่ไม่เพียงพอใช้งาน ก็หยุดประมวลผลโปรแกรมอีกครั้ง และหน่วยความจำฮีปส่วนที่ 2 ก็จะถูกค้นหาวัตถุ live เพื่อทำการคัดลอกไปยังพื้นที่หน่วยความจำฮีปส่วนที่ 1 แล้วโปรแกรมก็จะประมวลผลกับหน่วยความจำฮีปส่วนที่ 1 ต่อไป จากการทำงานวิธีนี้ทำให้เกิดค่าใช้จ่ายสำหรับการจัดการหน่วยความจำที่แบ่งเป็น 2 ส่วน และสลับการทำงานไปมาระหว่างหน่วยความจำฮีปทั้ง 2 ส่วน

2.5 KVM

KVM คือ K virtual machine เป็นจาวาเวอร์ชวลแมชชีนขนาดเล็ก [8] ไดอะแกรมแสดงดังรูปที่ 2.12 ถือกำเนิดโดยบริษัทซันในปี 1998 โดย 'K' ความหมายคือ กิโล (Kilo เป็นหน่วยวัดการใช้หน่วยความจำในหน่วย กิโลไบต์) ขนาดหน่วยความจำที่ต้องการใช้งานน้อย และมีความยืดหยุ่นในการนำไปใช้กับอุปกรณ์ชนิดต่างๆได้ง่าย เหมาะสมในการนำไปใช้กับอุปกรณ์ขนาดเล็กที่ติดต่อกันบนเครือข่ายไร้สายและอุปกรณ์ระบบคอมพิวเตอร์ฝังตัว เช่นโทรศัพท์มือถือ เพจเจอร์ และพีดีเอ เป็นต้น



รูปที่ 2.12 แสดงไดอะแกรมของ Java 2 Platform Editions

ลักษณะของ KVM ประกอบด้วย

- หน่วยความจำสำหรับ Object code เพียง 40K เท่านั้น ซึ่งไม่รองรับชนิดข้อมูลขนาดใหญ่ อาร์เรย์หลายมิติ ฟังก์ชันด้านกราฟิกต่างๆ และ Java Native Interface
- ขนาดของเวอร์ชวลแมชชีนและคลาสไลบรารีน้อย ขนาด 60K-72K เท่านั้น
- ต้องการหน่วยความจำน้อยในการประมวลผลเพียง 128K เท่านั้น
- ออกแบบให้สามารถทำงานกับคอมพิวเตอร์ขนาด 16 บิตหรือ 32 บิต ที่ความเร็ว 25MHz ได้อย่างมีประสิทธิภาพ
- มีความยืดหยุ่นสูง ง่ายในการพอร์ทลงบนอุปกรณ์ต่างๆเพราะโค้ดเขียนด้วยภาษา C ประมาณ 25,000 บรรทัด
- ความเร็วในการทำงานประมาณ 30-80% ของ JDK 1.1
- ไม่มี Just-In-Time (JIT)

บทที่ 3
งานวิจัยที่เกี่ยวข้อง

ในบทนี้กล่าวถึงงานวิจัยที่เกี่ยวข้องในการทำวิจัยทั้งเป็นทฤษฎีความรู้พื้นฐานเทคนิคการคืนค่าวัตถุที่ไม่ได้ใช้งาน งานวิจัยเกี่ยวกับการสร้างฮาร์ดแวร์สำหรับจาวา งานวิจัยเกี่ยวกับการจองหน่วยความจำโดยวิธีระบบไบนารีบัดดีและเทคนิคบิตแมบด้วยวิธีฮาร์ดแวร์ ตลอดจนงานวิจัยอื่นที่เกี่ยวข้องกับการเพิ่มประสิทธิภาพในการจัดการหน่วยความจำแบบพลวัตโดยเฉพาะการคืนค่าวัตถุที่ไม่ได้ใช้งาน

3.1 งานวิจัยเรื่อง Uniprocessor Garbage Collection Techniques [2]

งานวิจัยนี้นำเสนออัลกอริธึมการคืนพื้นที่วัตถุที่ไม่ได้ใช้งานด้วยวิธีการทางซอฟต์แวร์ โดยได้นำเสนอรายละเอียดการทำงานแต่ละอัลกอริธึม ตลอดจนเปรียบเทียบข้อดีข้อเสีย และวิธีที่น่าสนใจคือ reference counting โดยสรุปได้ดังนี้

1. ง่ายในการสร้าง
2. เกิด internal fragmentation
3. ประสิทธิภาพต่ำเมื่อเทียบกับวิธีอื่นๆ เพราะมีการติดต่อกับหน่วยความจำบ่อยเกินไป
4. เกิดการอ้างอิงการเป็นวงกลม ทำให้ไม่สามารถคืนค่าพื้นที่ใช้งานของวัตถุเหล่านี้ได้
5. ต้องเพิ่มฟิลด์ตัวนับการอ้างอิงที่ header ของวัตถุทำให้ต้องใช้พื้นที่หน่วยความจำเพิ่มขึ้น
6. ต้องเสียเวลาในการปรับค่าตัวนับการอ้างอิง ทุกครั้งที่มีการเปลี่ยนการอ้างอิงวัตถุ
7. สามารถทำงานในช่วงเวลาสั้นๆระหว่างที่โปรแกรมกำลังทำงานอยู่ได้ ทำให้การตอบสนองต่อเวลาจริง (response time) ดีกว่าทุกวิธี

จากวิธี reference counting ที่ได้กล่าวมาจะเห็นว่าเหมาะสมในงานที่ต้องการตอบสนองต่อเวลาจริง นั่นคือใช้เป็นอัลกอริธึมของหน่วยจัดการหน่วยความจำแบบพลวัตในระบบคอมพิวเตอร์ฝังตัว

สรุปสิ่งที่ได้จากงานวิจัยนี้คือ จากข้อดีของวิธี reference counting เหมาะสมต่อการนำไปใช้จัดการคืนพื้นที่ของวัตถุที่ไม่ได้ใช้งานเป็นเวลานานในงานระบบคอมพิวเตอร์ฝังตัวได้ ขณะที่ข้อเสียบางข้อสามารถที่จะแก้ด้วยฮาร์ดแวร์อัลกอริธึมได้

3.2 งานวิจัยเรื่อง Real-Time Reference Counting [9]

งานวิจัยนี้นำเสนอปัญหาของการจัดการหน่วยความจำแบบพลวัตในสภาพแวดล้อมเวลาจริง และได้นำเสนอวิธี RT-Reference Counting เพื่อแก้ปัญหาเหล่านี้

งานวิจัยนี้ยังได้อธิบายการทำงานของวิธี reference counting ซึ่งเป็นวิธีที่สามารถทำงานในเวลาสั้นๆระหว่างที่โปรแกรมกำลังทำงานอยู่ โดยในงานที่ต้องการตอบสนองต่อเวลาจริงส่วนมากจะใช้วิธี copying garbage collection แต่ว่ามีข้อเสียคือมีค่าใช้จ่ายสูงและการเข้าจังหวะการทำงานยังมีความซับซ้อนมาก ดังนั้นวิธี reference counting จึงเหมาะสมต่องานที่ตอบสนองต่อเวลาจริงมากกว่า แต่ก็มีปัญหาคือ

1. เกิดปัญหา fragmentation
2. เมื่อวัตถุถูกคืนค่าพื้นที่ใช้งานวัตถุลูกต้องถูกทดสอบการคืนค่าพื้นที่ใช้งานด้วย ถ้าตัวนับการอ้างอิงของวัตถุลูกมีค่าเป็นศูนย์ วัตถุลูกต้องถูกคืนพื้นที่หน่วยความจำ ซึ่งต้องทำการทดสอบวัตถุลูกต่อเป็นทอดๆ ทำให้เสียเวลา
3. ประสิทธิภาพต่ำเมื่อเทียบกับวิธีอื่น
4. เกิดการอ้างอิงกันเป็นวงกลมของวัตถุ ทำให้ไม่สามารถคืนพื้นที่ใช้งานของวัตถุเหล่านี้ได้

สำหรับวิธี RT-Reference Counting ที่ได้นำเสนอในงานวิจัยนี้ สามารถแก้ปัญหาทั้ง 4 ข้อนี้ได้

3.3 งานวิจัยเรื่อง Designing a java microprocessor core using FPGA technology [10]

งานวิจัยนี้กล่าวถึงการออกแบบJAVAไมโครโปรเซสเซอร์ด้วยเอฟพีจีเอโดยใช้เทคโนโลยี VLSI ที่ปัจจุบันพัฒนาไปมาก ทำให้สามารถพัฒนาอัลกอริธึมด้วยวิธีการทางฮาร์ดแวร์แทนวิธีการทางซอฟต์แวร์ได้ การออกแบบใช้เทคนิคการออกแบบจากบนลงล่าง โดยเริ่มจากการออกแบบชุดคำสั่งและเส้นทางเดินข้อมูลระหว่างฮาร์ดแวร์ต่างๆ และนำสิ่งที่ออกแบบได้มาเขียนด้วยภาษาวีเอช-ดีแอล (VHDL) และจำลองการทำงานเพื่อตรวจสอบความถูกต้อง จากนั้นทำการสังเคราะห์จากวีเอชดีแอลเป็นวงจรด้วยเครื่องมือที่ใช้ในการสังเคราะห์โดยเฉพาะ และนำวงจรที่ได้ไปสร้างบนอุปกรณ์เอฟพีจีเอ(FPGA) ในงานวิจัยนี้ได้เลือกใช้เอฟพีจีเอ เนื่องจากว่าราคาค่าใช้จ่ายถูกและง่ายต่อการพัฒนางานเป็นต้นแบบ การประมวลผลคำสั่งประกอบด้วย ขั้นตอนการเฟชคำสั่ง (Fetch) การแปลความหมายคำสั่ง (Decode) การประมวลผลคำสั่ง (Excute)

โครงสร้างJAVAไมโครโปรเซสเซอร์แบ่งเป็นฮาร์ดแวร์ย่อยๆ ดังนี้

- I/O : สำหรับควบคุมการติดต่อกับภายนอกชิพ

- Instruction decoder : สำหรับแปลความหมายไบต์โค้ดและสร้างสัญญาณควบคุมภายในให้แก่หน่วยประมวลผล
- Execution Unit : ทำการประมวลผลข้อมูลตามคำสั่งไบต์โค้ดที่ได้แปลความหมาย
- ALU : ทำการประมวลผลทางด้านคณิตศาสตร์และตรรกะ
- Set of register : สำหรับเก็บข้อมูลระหว่างคำสั่งทำการประมวลผล
- Stack : เป็นกลไกหลักของการประมวลผลแบบสแตค (Stack)

สำหรับในงานวิจัยนี้ได้ออกแบบสถาปัตยกรรมเป็น 2 รูปแบบ คือแบบไปป์ไลน์ 2 สเตจ (2-state pipeline) และแบบไม่ใช่ไปป์ไลน์ เพื่อทำการเปรียบเทียบกัน โดยทำงานที่ความถี่สัญญาณนาฬิกา 21 MHz และไม่มีฮาร์ดแวร์ในส่วนที่ทำหน้าที่คืนพื้นที่หน่วยความจำของวัตถุที่ไม่ได้งาน (Garbage Collection) การจำลองการทำงานใช้เครื่องมือ Modelsim ของบริษัท MTI และผลการจำลองการทำงาน ดังนี้

- ผลการจำลองจากจาวาไมโครโปรเซสเซอร์ที่สถาปัตยกรรมแบบไปป์ไลน์ใช้จำนวน 4 รอบคำสั่งต่อคำสั่ง
- ผลการจำลองจากจาวาไมโครโปรเซสเซอร์ที่สถาปัตยกรรมแบบไม่ใช่ไปป์ไลนใช้จำนวน 4 ถึง 7 รอบคำสั่งต่อคำสั่ง

ผลจากการจำลองการทำงานแสดงให้เห็นว่าประสิทธิภาพเพิ่มขึ้น 25% เมื่อใช้สถาปัตยกรรมแบบไปป์ไลน์ สำหรับการสังเคราะห์วงจรและสร้างลงบนอุปกรณ์เอฟพีจีเอ ในงานวิจัยนี้ได้เลือกใช้เทคโนโลยีของ Altera Flex 10K และ Lucent ORCA เพื่อทำการเปรียบเทียบกัน ผลจากการวิเคราะห์ผลที่ได้ สรุปว่าแบบไปป์ไลนใช้พื้นที่และเวลาดีกว่าในทั้งสองอุปกรณ์ ขณะที่แบบไม่ใช่ไปป์ไลนใช้จำนวนฟลิปฟลอปมากกว่า

ในส่วนของการทดสอบการทำงานจริงกับฮาร์ดแวร์ งานวิจัยนี้ได้เลือกใช้บอร์ด Altera- UP1 Education จากบริษัท Altera เพื่อทดสอบการทำงาน สำหรับแนวทางการวิจัยในอนาคตจะทำการวิจัยในส่วนการจัดการหน่วยความจำของวัตถุที่ไม่ได้ใช้ และประสิทธิภาพการจัดการหน่วยความจำ

สรุปสิ่งที่ได้จากงานวิจัยนี้คือ ได้แนวทางการทำวิจัยเกี่ยวกับการสร้างฮาร์ดแวร์ด้วยเทคโนโลยีเอฟพีจีเอ และหัวข้อที่สนใจในการทำวิจัย คือการจัดการหน่วยความจำของวัตถุที่ไม่ได้ใช้งานเวลานานด้วยวิธีฮาร์ดแวร์

3.4 งานวิจัยเรื่อง A High-Performance Memory Allocator for Object-Oriented Systems [7]

งานวิจัยนี้นำเสนอการออกแบบฮาร์ดแวร์สำหรับการจองพื้นที่หน่วยความจำ โดยใช้การออกแบบวงจรรวมที่สามารถจองบล็อกหน่วยความจำใช้เวลาเพียง 1 รอบคำสั่ง เพื่อเพิ่มประสิทธิภาพ

การจัดการหน่วยความจำจากวิธีซอฟต์แวร์ที่มีปัญหาการเกิด internal fragmentation และทราฟฟิก (Traffic) ของหน่วยความจำมีมาก ซึ่งประโยชน์จากวิธีการทางฮาร์ดแวร์ช่วยเพิ่มความเร็วของการทำงาน และเพิ่มประสิทธิภาพการใช้งานหน่วยความจำ

ในงานวิจัยนี้ได้ใช้เทคนิคการจองหน่วยความจำบนพื้นฐานของระบบไบนารีโดยใช้เทคนิคบิตแมบแทนบล็อกที่จัดการแบบไบนารีทรี นั่นคือถ้าต้องการจองบล็อกพื้นที่หน่วยความจำ ก็กำหนดบิตที่สัมพันธ์กับบล็อกนั้นให้มีค่าเป็นลอจิก '1' และถ้าต้องการคืนบล็อกพื้นที่หน่วยความจำ ก็เพียงเคลียร์บิตที่สัมพันธ์กับบล็อกนั้น โดยการจองพื้นที่ด้วยฮาร์ดแวร์มีขั้นตอนดังนี้

- ตรวจสอบว่าพื้นที่หน่วยความจำมีเพียงพอสำหรับการจองวัตถุใหม่หรือไม่ โดยใช้ฮาร์ดแวร์ Or-Gate tree
- ถ้ามีพื้นที่เพียงพอ แล้วให้ทำการหาแอดเดรสเริ่มต้นแก่วัตถุใหม่ โดยใช้ฮาร์ดแวร์ And-Gate tree
- กำหนดบิตที่สัมพันธ์กับหน่วยความจำที่ว่างลงบนเวกเตอร์ โดยใช้ฮาร์ดแวร์ Bit-Flipper

จากนั้นนำฮาร์ดแวร์ทั้งสามมารวมเข้าด้วยกันเป็นฮาร์ดแวร์ complete binary tree (CBT) เพื่อทำการจองพื้นที่หน่วยความจำ สำหรับฮาร์ดแวร์ที่งานวิจัยนี้ได้นำเสนอ สามารถช่วยลดการเกิด internal fragmentation และสามารถเพิ่มประสิทธิภาพและความเร็วในการจัดการหน่วยความจำได้

สรุปสิ่งที่ได้จากงานวิจัยนี้คือ ได้วิธีการจองหน่วยความจำด้วยฮาร์ดแวร์และเทคนิคการพัฒนาอัลกอริธึมด้วยวิธีทางฮาร์ดแวร์ ซึ่งเป็นพื้นฐานที่สำคัญในการทำงานวิจัยในหัวข้อที่สนใจ

3.5 งานวิจัยเรื่อง Design of a Reusable Memory Management System [11]

งานวิจัยนี้นำเสนอการออกแบบและสร้างหน่วยจัดการหน่วยความจำ (Active Memory Manager Unit) ที่สามารถฝังรวมอยู่ภายในชิพซีพียู โดยพัฒนาด้วยภาษาวีเอชดีแอลและสร้างบนเอฟพีจีเอ และใช้เทคนิคการจัดการหน่วยความจำแบบ Modified buddy system และซีพียูที่ใช้ในการประมวลผลและการเชื่อมต่อกับหน่วยจัดการหน่วยความจำ เพื่อความยืดหยุ่นในการสร้างจะใช้ซีพียูแบบลดจำนวนชุดคำสั่ง (RISC)

การจัดการหน่วยความจำที่งานวิจัยนี้ได้นำเสนอ ประกอบด้วย 2 ขั้นตอน คือการจองพื้นที่หน่วยความจำ และการคืนพื้นที่หน่วยความจำ โดยใช้เทคนิคการจัดการหน่วยความจำด้วยวิธีการทางฮาร์ดแวร์ที่ได้นำเสนอในงานวิจัย "A High-Performance Memory Allocator for Object-Oriented Systems" [7] แต่ได้เพิ่มฮาร์ดแวร์สำหรับการคืนพื้นที่หน่วยความจำ โดยนำอัลกอริธึมของฮาร์ดแวร์ Bit-Flipper มาทำการปรับปรุงแก้ไข โดยเปลี่ยนจากการเซต (set) บิตที่สัมพันธ์กับบล็อกหน่วยความจำที่ว่างลงบนเวกเตอร์ มาเป็นการเคลียร์ (clear) บิตในเวกเตอร์ที่สัมพันธ์กับบล็อกหน่วยความจำแทน นอกจากนี้ยังได้ลงรายละเอียดการสร้างฮาร์ดแวร์และตัวอย่างโค้ดภาษาวีเอชดี

แอลด้วย และในการทดสอบการทำงานได้ใช้ OpenRISC 1000 เป็นซีพียูที่มีสถาปัตยกรรมแบบลดจำนวนชุดคำสั่ง มีขนาด 32 บิต และเชื่อมต่อเข้ากับหน่วยจัดการหน่วยความจำที่ได้นำเสนอ และการสังเคราะห์วงจรใช้เทคโนโลยีเอฟพีจีเอของ Xilink Virtex

งานวิจัยนี้แสดงให้เห็นว่าหน่วยการจัดการหน่วยความจำที่นำเสนอนี้ช่วยเพิ่มประสิทธิภาพการจองพื้นที่หน่วยความจำและการคืนพื้นที่หน่วยความจำเมื่อเทียบกับวิธีการทางซอฟต์แวร์

สรุปสิ่งที่ได้จากงานวิจัยนี้คือ เป็นแนวทางการสร้างฮาร์ดแวร์สำหรับการจองพื้นที่หน่วยความจำ และฮาร์ดแวร์สำหรับการคืนค่าพื้นที่หน่วยความจำของวัตถุที่ไม่ได้ใช้เวลานาน สำหรับงานวิจัยที่จะทำต่อไป

3.6 งานวิจัยเรื่อง An Introduction to DMMX (Dynamic Memory Management Extension) [12]

งานวิจัยนี้นำเสนอการออกแบบส่วนขยายของหน่วยจัดการหน่วยความจำแบบพลวัตได้ (Dynamic Memory Management Extension) ซึ่งสามารถจัดการหน่วยความจำแบบอัตโนมัติโดยฮาร์ดแวร์ โดยประสิทธิภาพของวิธีที่นำเสนอนี้สามารถคาดเดาเวลาในการทำการจองและการทำคืนค่าพื้นที่หน่วยความจำของวัตถุที่ไม่ได้ใช้งานได้ โดยในส่วนการจองทำผ่าน Modified buddy system ซึ่งสามารถใช้เวลาในการจองคงที่ ทำให้ความเร็วเพิ่มขึ้นและง่ายในการนำไปเป็นส่วนหนึ่งของซีพียู หรือเป็นฮาร์ดแวร์ของจาวาเวอร์ชวลแมชชีน หรือเป็นหน่วยประมวลผลสำหรับการจัดการหน่วยความจำโดยเฉพาะ

3.7 งานวิจัยเรื่อง Scable Hardware-algorithm for Mark-sweep Garbage Collection [13]

งานวิจัยนี้นำเสนอการออกแบบฮาร์ดแวร์สำหรับ Mark-sweep garbage collection โดยวิธีการที่นำเสนอนี้ฮาร์ดแวร์ Bit-sweeper สามารถตรวจหาวัตถุที่ไม่ได้ใช้งานเวลานานได้และทำการ sweep ในเวลาที่คงที่แน่นอน และในส่วนของขั้นตอนการ mark ใช้เพียงคำสั่งเดียวในการทำ ซึ่งเมื่อเทียบกับวิธีทางซอฟต์แวร์แล้วใช้จำนวนคำสั่งมากกว่า แต่อย่างไรก็ตามฮาร์ดแวร์ของขั้นตอนการ sweep มีความซับซ้อนกว่าฮาร์ดแวร์ของขั้นตอนการ mark โดยฮาร์ดแวร์ Bit-sweeper มีความซับซ้อนเป็น O(n) เมื่อ n เป็นขนาดของ bit-map

งานวิจัยนี้นำเสนอการออกแบบฟังก์ชันการ sweep ที่ทำในฮาร์ดแวร์สำหรับทำการคืนพื้นที่วัตถุไม่ได้ใช้งาน โดยจากเดิมวิธีซอฟต์แวร์ขั้นตอนการ mark ความซับซ้อนขึ้นอยู่กับจำนวนวัตถุที่กำลังทำงานอยู่ในเวลานั้น ขณะที่ขั้นตอนการ sweep ความซับซ้อนขึ้นอยู่กับจำนวนวัตถุทั้งหมดที่มีอยู่ในหน่วยความจำ ดังนั้นถ้าจำนวนวัตถุเพิ่มมากขึ้นจะทำให้เวลาในการ sweep นานไปด้วย ดังนั้น

ฮาร์ดแวร์ที่นำเสนอในงานวิจัยนี้ได้ใช้ bit-map จำนวน 3 เวกเตอร์ด้วยกันในการเก็บข้อมูลที่เกี่ยวข้องกับวัตถุ โดยเวกเตอร์ A-map เก็บสถานะการจองของหน่วยความจำส่วนฮีฟ เวกเตอร์ B-map เก็บขนาดบล็อกหน่วยความจำที่ทำการจอง และเวกเตอร์ M-map สำหรับใช้ในขั้นตอนการ mark โดยฮาร์ดแวร์ Complete Binary Tree (CBT) จะรับสัญญาณการจองและขนาดวัตถุที่ต้องการจองหน่วยความจำ แล้วทำการหาตำแหน่งหน่วยความจำที่ว่างที่สามารถจองใช้งานได้ พร้อมกับทำการเก็บสถานะที่จองได้ในเวกเตอร์ A และในการจองจะหาพื้นที่ว่างเป็น 2^n เมื่อ n เป็นจำนวนบิตที่แทนขนาดของบล็อกที่ต้องการจอง แต่เวลาในการจองจริงจะจองตามขนาดที่ต้องการ ตัวอย่างเช่น ต้องการจองหน่วยความขนาด 5 บล็อก ระบบจะหาพื้นที่หน่วยความจำที่ว่างขนาด 8 บล็อก (2^3) หลังจากหาพื้นที่ว่างได้แล้วระบบจะจองเพียง 5 บล็อกเท่านั้น และทุกครั้งที่วัตถุถูกสร้างและถูกคืนพื้นที่ใช้งาน ค่าในเวกเตอร์ B ต้องทำการอัพเดตโดยฮาร์ดแวร์ B-Unit และเวกเตอร์ M จะถูกใช้ในขั้นตอนการ mark ด้วยฮาร์ดแวร์ Bit-flipper เมื่อขั้นตอนการ mark เสร็จสิ้น ฮาร์ดแวร์ Bit-sweeper จะทำการ sweep ต่อไป

ผลจากการจำลองการทำงานฮาร์ดแวร์ที่ได้นำเสนอในงานวิจัยนี้ ขั้นตอนในการ sweep สามารถทำงานในเวลาที่คงที่แน่นอน โดยเมื่อเทียบกับวิธีซอฟต์แวร์แล้วขั้นตอนการ sweep ทำการค้นหาทุกวัตถุทั้งหมดที่มีอยู่ในหน่วยความจำ ดังนั้นวิธีฮาร์ดแวร์จะช่วยทำให้ประสิทธิภาพการทำงานดีขึ้น และสมมุติว่าหน่วยความจำทั้งระบบมีขนาด 64 เมกกะไบต์ และกำหนดให้ 1 บิต แทนบล็อกหน่วยความจำขนาด 16 ไบต์ ดังนั้นทั้ง 3 เวกเตอร์ใช้หน่วยความจำไป 1.5 เมกกะไบต์ คิดเป็น 2.3% ของหน่วยความจำทั้งหมด และความซับซ้อนของฮาร์ดแวร์ Bit-sweeper คิดเป็น O(n) เมื่อ n แทนขนาดบิตเวกเตอร์

สรุปสิ่งที่ได้จากงานวิจัยนี้คือ ได้แนวทางในการสร้างฮาร์ดแวร์สำหรับวิทยานิพนธ์ฉบับนี้

3.8 สรุปผลงานวิจัยอื่นที่นำมาใช้ในวิทยานิพนธ์

จากงานวิจัยที่ได้กล่าวมาทั้งหมด สามารถสรุปแนวทางการนำไปใช้สำหรับการทำวิทยานิพนธ์ดังนี้

- การจองพื้นที่หน่วยความจำด้วยระบบไบนารีบัดดีและบิตแมป ได้จากงานวิจัยหัวข้อ 3.4 3.5 3.6 และ3.7
- การสร้างอัลกอริธึมด้วยวิธีการทางฮาร์ดแวร์ ได้จากงานวิจัยหัวข้อ 3.3 3.4 3.5 3.6 และ 3.7
- ข้อดีและข้อเสียของวิธี Reference Counting ได้จากงานวิจัยหัวข้อ 3.1
- วิธี Reference Counting เหมาะกับงานระบบคอมพิวเตอร์ฝังตัว ได้จากงานวิจัยหัวข้อ 3.2

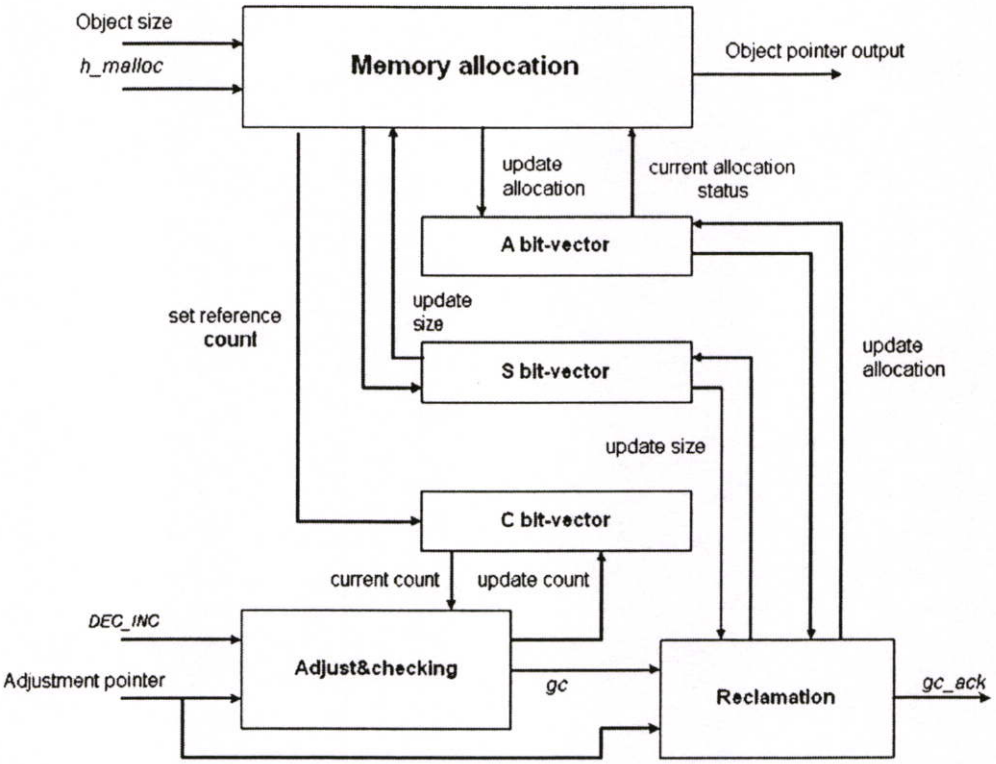
บทที่ 4
การออกแบบสร้างฮาร์ดแวร์ RCGC

บทนี้กล่าวถึงรายละเอียดของการออกแบบสร้างฮาร์ดแวร์Reference Counting Garbage Collection (RCGC) ซึ่งประกอบด้วย ฮาร์ดแวร์สำหรับการจองพื้นที่หน่วยความจำ (Memory allocation) ฮาร์ดแวร์สำหรับการปรับค่าและทดสอบตัวนับการอ้างอิง (Adjust & checking) และฮาร์ดแวร์สำหรับการคืนพื้นที่วัตถุที่ไม่ได้ใช้งาน (Reclamation)

4.1 โครงสร้างฮาร์ดแวร์ของ RCGC

วิธีการทางฮาร์ดแวร์ที่ใช้ในงานวิจัยนี้ อ้างอิงจากวิธีการทางซอฟต์แวร์ [2][14][15] ประกอบด้วย 3 ขั้นตอน คือขั้นตอนการจองพื้นที่หน่วยความจำสำหรับวัตถุใหม่และกำหนดค่าตัวนับอ้างอิง (reference count) เริ่มต้น ขั้นตอนการปรับค่าและตรวจสอบตัวนับการอ้างอิง และขั้นตอนการคืนพื้นที่หน่วยความจำเมื่อตัวนับอ้างอิงมีค่าเป็นศูนย์ โดยโครงสร้างของ Reference Counting Garbage Collection (RCGC) แสดงดังรูปที่ 4.1 ใช้บิตเวกเตอร์แทนสถานะต่างๆที่จำเป็น ประกอบด้วยจำนวน 3 บิตเวกเตอร์ ได้แก่ บิตเวกเตอร์ A bit-vector ใช้แทนสถานะการจองหน่วยความจำ นั่นคือแต่ละบิตในเวกเตอร์แทนพื้นที่หน่วยความจำขนาดหนึ่งบล็อกโดยถ้าเซตเป็นลอจิก '1' แสดงว่าบล็อกนั้นถูกจอง และถ้าบล็อกไหนว่างจะเซตเป็นลอจิก '0' บิตเวกเตอร์ S bit-vector เก็บขนาดของวัตถุที่มีการจองแต่ละครั้ง และบิตเวกเตอร์ C bit-vector สำหรับเก็บตัวนับการอ้างอิงของแต่ละวัตถุ และเมื่อมีการร้องขอการจองพื้นที่สำหรับวัตถุใหม่ ระบบจะส่งสัญญาณ *h_malloc* พร้อมกับขนาดของวัตถุที่ต้องการจองไปยังฮาร์ดแวร์ Complete Binary Tree (CBT) เพื่อทำการจองโดยการหาแอดเดรสที่เหมาะสมให้พร้อมกับอัพเดตสถานะการจองที่ A bit-vector และกำหนดค่าตัวนับอ้างอิงเริ่มต้นใน C bit-vector

สำหรับการเก็บขนาดของวัตถุต้องเก็บทุกครั้งที่มีการจองพื้นที่โดยฮาร์ดแวร์ที่เรียกว่า B-unit ซึ่งเก็บลงใน S bit-vector โดยข้อมูลที่เก็บนี้เป็นข้อมูลที่สำคัญมากเพราะเป็นข้อมูลที่ใช้บอกว่าจำนวนบิตที่เซตใน A bit-vector มีจำนวนกี่บิต และทุกครั้งเมื่อมีการเปลี่ยนตัวชี้ระหว่างสองวัตถุ ฮาร์ดแวร์ Adjust&Checking ต้องทำงานสองครั้ง ด้วยสัญญาณ *DEC_INC* พร้อมกับรับแอดเดรสของวัตถุเข้ามาด้วย ครั้งแรกสำหรับการลดค่าตัวนับการอ้างอิงที่สัมพันธ์กับวัตถุที่มีการอ้างอิง และครั้งสองสำหรับการเพิ่มค่าตัวนับการอ้างอิงที่สัมพันธ์กับวัตถุที่ถูกยกเลิกการอ้างอิงแล้วทำการอัพเดต C bit-vector และเมื่อปรับค่า ตัวนับอ้างอิง แล้วมีค่าเป็นศูนย์ต้องส่งสัญญาณ *gc* ให้ฮาร์ดแวร์ Reclamation ทำงาน เพื่อทำการคืนพื้นที่ด้วยการเคลียร์บิตใน A bit-vector และเคลียร์ค่าขนาดของวัตถุใน S bit-vector



รูปที่ 4.1 โครงสร้างฮาร์ดแวร์ของ RCGC

และขั้นตอนการทำงานของฮาร์ดแวร์ RCGC แสดงดังตารางที่ 4.1

ตารางที่ 4.1 แสดงขั้นตอนการทำงานในแต่ละฟังก์ชันของฮาร์ดแวร์ RCGC

Hardware	Allocation	Adjust&checking	Reclamation
Complete Binary Tree (CBT)	**A1**		
Allocation bit-vector (A bit-vector)	**A2**		**C2**
Count bit-vector (C bit-vector)	**A3**	**B1**	
Size encoder (B-unit)	**A4**		
Size bit-vector (S bit-vector)	**A5**		**C3**
Adjust&checking		**B2**	
Reclamation			**C1**

จากขั้นตอนการทำงานดังตารางที่ 4.1 ถ้าต้องการจองพื้นที่ขนาด 5 บล็อกสัญญาณ *h_mallc* จะเป็นลอจิก '1' และส่งไปยังฮาร์ดแวร์ CBT (ขั้นตอน A1) จากนั้น CBT จะรับขนาดของวัตถุและหาแอดเดรสสำหรับวัตถุใหม่ พร้อมกับทำการอัพเดตสถานะการจองบน A bit-vector (ขั้นตอน A2) และทำการกำหนดตัวนับอ้างอิงเริ่มต้นใน C bit-vector ในตำแหน่งที่สัมพันธ์กับ A bit-vector (ขั้น

ตอน A3) จากนั้นฮาร์ดแวร์ B-unit จะทำการเก็บขนาดของวัตถุบน S bit-vector ในตำแหน่งที่สัมพันธ์กับ A bit-vector (ขั้นตอน A4) เมื่อข้อมูลถูกเก็บใน S bit-vector เรียบร้อยแล้ว แสดงว่าเสร็จสิ้นกระบวนการจองพื้นที่สำหรับวัตถุใหม่ (ขั้นตอน A5)

สำหรับฮาร์ดแวร์การปรับค่าและตรวจสอบตัวนับการอ้างอิง จะทำงานเมื่อมีการเปลี่ยนตัวชี้ระหว่างสองวัตถุ โดยรับแอดเดรสของวัตถุเข้ามาเพื่อไปอ่านค่าตัวนับการอ้างอิงใน C bit-vector (ขั้นตอน B1) จากนั้นทำการปรับค่าตัวนับอ้างอิงตามสัญญาณอินพุท *DEC_INC* นั่นคือเมื่อเป็นลอจิก '0' ต้องทำการเพิ่มค่าตัวนับอ้างอิงและถ้าเป็นลอจิก '1' ต้องทำการลดค่าตัวนับอ้างอิง และหลังจากปรับค่าแล้วมีค่าเป็นศูนย์ต้องสร้างสัญญาณ *gc* ส่งให้ฮาร์ดแวร์สำหรับการคืนพื้นที่ทำงานต่อไป (ขั้นตอน B2)

สุดท้ายฮาร์ดแวร์สำหรับการคืนพื้นที่ จะทำงานเมื่อได้รับสัญญาณ gc จากฮาร์ดแวร์สำหรับการปรับค่าและตรวจสอบตัวนับการอ้างอิง (ขั้นตอน C1) โดยทำการคืนพื้นที่ด้วยการเคลียร์บิตใน A bit-vector (ขั้นตอน C2) และเคลียร์ค่าขนาดของวัตถุใน S bit-vector (ขั้นตอน C3) เสร็จสิ้นการทำงาน

4.2 การออกแบบฮาร์ดแวร์สำหรับการจองพื้นที่หน่วยความจำ (Memory allocation)

ในหัวข้อนี้เป็นการลงรายละเอียดการออกแบบฮาร์ดแวร์สำหรับการจองพื้นที่หน่วยความจำโดยใช้พื้นฐานการออกแบบระบบไบนารีบัดดี [6] และ อัลกอริธึมที่ใช้ในการจองพื้นที่สำหรับวัตถุใหม่ด้วยฮาร์ดแวร์พัฒนามาจากงานวิจัยใน [7][4] ประกอบด้วยด้วย 5 ขั้นตอนคือ

(ก) ตรวจสอบว่าพื้นที่หน่วยความจำมีเพียงพอสำหรับการจองวัตถุใหม่หรือไม่

(ข) ถ้ามีพื้นที่เพียงพอ ให้ทำการหาแอดเดรสเริ่มต้นให้แก่วัตถุใหม่

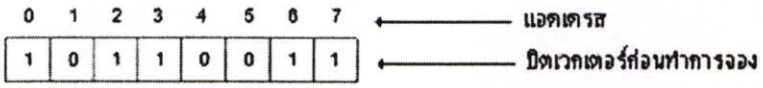
(ค) กำหนดบิตที่สัมพันธ์กับหน่วยความจำที่ว่างลงในเวกเตอร์ A bit-vector

(ง) เก็บขนาดของวัตถุที่จองลงในเวกเตอร์ S bit-vector

(จ) กำหนดค่าเริ่มต้นให้กับตัวนับการอ้างอิงในเวกเต อร์ C bit-vector

ในรูปที่ 4.2 เป็นตัวอย่างแสดงการทำงานของฟังก์ชัน(ก) ถึง (ค) โดยกำหนดให้แต่ละบิตในบิตเวกเตอร์แทนบล็อกที่น้อยที่สุดของหน่วยความจำที่สามารถจองได้ ณ เวลานั้น ในการใช้บิตเวกเตอร์แทนบล็อกหน่วยความจำทั้งหมดนั้น ต้องใช้บิตเวกเตอร์จำนวนเป็นหมื่นบิต และถ้าใช้เรจิสเตอร์เก็บบิตเวกเตอร์ทั้งหมดจะทำให้ค่าใช้จ่ายด้านฮาร์ดแวร์ (hardware cost) สูงมาก วิธีการแก้โดยแบ่งบิตเวกเตอร์ออกเป็นส่วนๆและโหลดเฉพาะส่วนที่ต้องการลงในเรจิสเตอร์ที่ต้องการใช้งานในเวลานั้นเท่านั้น [7] และเมื่อต้องการใช้งานบิตเวกเตอร์ แต่ไม่อยู่ในกลุ่มที่กำลังใช้งานอยู่ ต้องทำการโหลดส่วนที่ต้องการใช้งานเข้ามาแทนที่ การทำเช่นนี้จะทำให้ค่าใช้จ่ายด้านฮาร์ดแวร์ต่ำ และจากการทดสอบของ Chang & Grhringer [7] แสดงให้เห็นว่าเรจิสเตอร์ควรจะใหญ่เพียงพอสำหรับการ

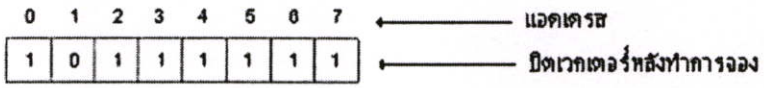
แทนหน่วยความจำขนาด 256K นั่นคือถ้ามีการร้องขอหน่วยความจำไม่เกิน 256K ให้ทำโดยวิธีการทางฮาร์ดแวร์ แต่ถ้าร้องขอหน่วยความจำมากกว่านี้ให้ทำโดยวิธีการทางซอฟต์แวร์



ก) มีพื้นที่เพียงพอสำหรับจองพื้นที่หน่วยความจำ 2 บล็อก

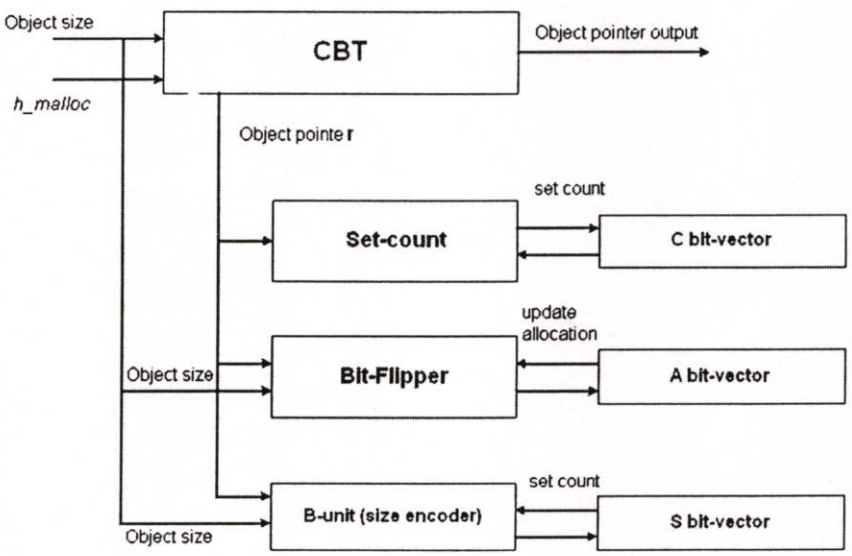
ข) แอดเดรสเริ่มต้นคือ 100_2

ค) เซตบิตที่แอดเดรส 100_2 และ 101_2



รูปที่ 4.2 แสดงตัวอย่างของจองพื้นที่หน่วยความจำ 2 บล็อกจากบิตเวกเตอร์ขนาด 8 บิต

จากฟังก์ชันการทำงาน (ก) ถึง (จ) สามารถนำไปสร้างเป็นฮาร์ดแวร์สำหรับการจองพื้นที่หน่วยความจำสำหรับวัตถุใหม่ แสดงดังรูปที่ 4.3 โดยฟังก์ชันการทำงาน (ก) และ (ข) สร้างเป็นฮาร์ดแวร์ Complete Binary Tree (CBT) ซึ่งจะอธิบายในหัวข้อ 4.2.1 ฟังก์ชันการทำงาน (ค) สร้างเป็นฮาร์ดแวร์ Bit-Flipper จะอธิบายในหัวข้อ 4.2.2 ฟังก์ชันการทำงาน (ง) สร้างเป็นฮาร์ดแวร์ B-unit จะอธิบายในหัวข้อ 4.2.3 และฟังก์ชันการทำงาน (จ) สร้างเป็นฮาร์ดแวร์ Set-count ซึ่งจะอธิบายในหัวข้อ 4.2.4



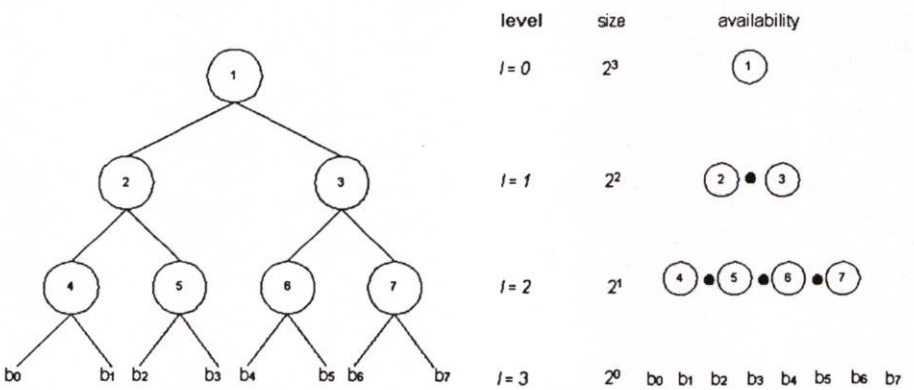
รูปที่ 4.3 แสดงโครงสร้างฮาร์ดแวร์ Memory allocation

4.2.1 ฮาร์ดแวร์ CBT

ในหัวข้อนี้จะอธิบายถึงการสร้างฮาร์ดแวร์ CBT ซึ่งเป็นฮาร์ดแวร์สำหรับหาแอดเดรสเริ่มต้นสำหรับวัตถุใหม่ โดยเริ่มจากการหาว่ามีพื้นที่หน่วยความจำที่เพียงพอต่อการจองใช้งานหรือไม่ และถ้ามีพื้นที่ว่างให้ทำการหาแอดเดรสเริ่มต้น

4.2.1.1 ฮาร์ดแวร์ Or-Gate-Tree

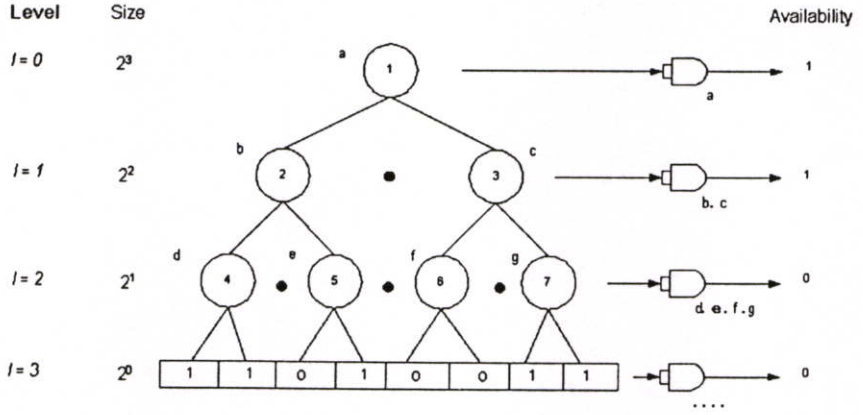
การตรวจสอบพื้นที่หน่วยความจำสำหรับการจองวัตถุใหม่ ตามฟังก์ชัน (ก) จะใช้ฮาร์ดแวร์ Or-Gate Tree เริ่มต้นด้วยหน่วยความจำขนาด 2^k และบล็อกหน่วยความจำที่น้อยที่สุดขนาด 2^b ดังนั้นจำนวนบล็อกหน่วยความจำทั้งหมด เท่ากับ 2^N บล็อก(เมื่อ $N = k - b$) และเนื่องจากขนาดบล็อกมีขนาดแตกต่างกัน ถ้าขนาดบล็อกไม่ได้ระบุ งานวิจัยนี้จะใช้ขนาดบล็อกที่น้อยที่สุด และใช้บิตเวกเตอร์ A bit-vector แทนสถานะการจองบล็อกในหน่วยความจำ โดยกำหนดบิต '1' แทนบล็อกที่มีการจองใช้งาน และบิต '0' แทนบล็อกที่ว่าง และเพื่อความง่ายในการหาตำแหน่งของบล็อกที่ว่างได้นำบิตเวกเตอร์มาทำให้อยู่ในรูปแบบของ complete binary tree (CBT) นั่นคือใบของทรี (leaf tree) จะเป็นบิตในบิตเวกเตอร์ โดยในแต่ละโหนดของ CBT สร้างด้วยออร์เกตขนาด 2 อินพุท โดยเอาท์พุทของออร์เกตจะแสดงสถานะการจองของทรีย่อย (subtree) นั้นๆ ถ้าเอาท์พุทเป็นลอจิก '0' แสดงว่าสถานะการจองบล็อกณ ตอนนั้นว่าง และถ้าเอาท์พุทเป็นลอจิก '1' แสดงว่าสถานะการจอง บล็อกณ ตอนนั้นไม่ว่าง และถ้ามีหน่วยความจำขนาด 2^N บล็อกจำนวนออร์เกตที่ต้องการทั้งหมดเป็น 2^N-1 และระดับ (level) ของทรีเริ่มจาก 0 ถึง N นั่นคือโหนดราก (root) อยู่ที่ระดับ 0 และสุดท้ายใบของทรีอยู่ที่ระดับ N และถ้ามีหน่วยความขนาด 8 บล็อก(2^3) ระดับ N คือระดับที่ 3 สำหรับตัวอย่างที่แสดงในรูปที่ 4.4 เป็น CBT สำหรับบิตเวกเตอร์ขนาด 8 บิต โดยแต่ละโหนดที่ระดับ l จะมีบิตที่แทนสถานะการจองของบล็อกจากทั้งหมด 2^N-1 บล็อก(บิตนี้ได้จากเอาท์พุทของออร์เกตขนาด 2 อินพุท)



○ : แทนออร์เกตขนาด 2 อินพุทสำหรับหาตำแหน่งบิตที่เป็นลอจิก '0' ติดกัน

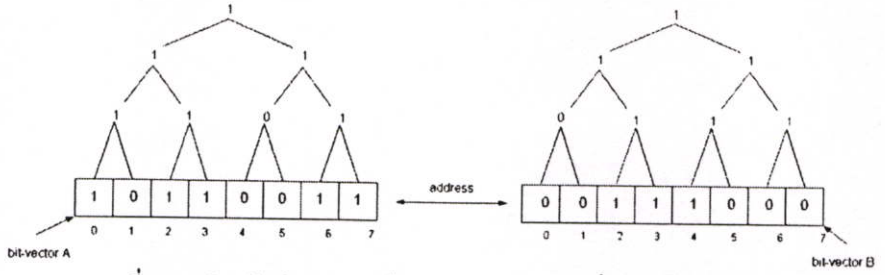
รูปที่ 4.4 แสดง CBT ของบิตเวกเตอร์ขนาด 8 บิต

จากตัวอย่างในรูปที่ 4.4 โหนด 2 และโหนด 3 อยู่ระดับที่ 1 โดยแต่ละโหนดแทนหน่วยความจำขนาด 2^2 บล็อกโดยโหนด 2 มีค่าเป็นศูนย์เมื่อใบของทรีที่ b0 b1 b2 และ b3 เป็นศูนย์ทั้งหมด และ availability มีไว้สำหรับบอกว่ามีบล็อกว่างเพียงพอที่สามารถใช้งานได้หรือไม่ โดยถ้าเป็น '0' แสดงว่ามีบล็อกว่างที่สามารถใช้งานได้ และเป็น '1' แสดงว่าไม่มีบล็อกว่างที่สามารถใช้งานได้ โดยค่าของ availability เกิดจากการนำเอาเอาท์พุทของโหนดมาทำฟังก์ชัน and ถ้ามีโหนดใดโหนดหนึ่งที่ระดับ *l* มีค่าเป็นศูนย์ ค่า availability ก็จะมีค่าเป็นศูนย์ นั่นคือถ้า availability เป็นศูนย์ที่ระดับ *l* หมายความว่าหน่วยความจำขนาด 2^{N-l} มีบล็อกหน่วยความจำว่างติดกันที่สามารถใช้งานได้ ดังนั้นถ้ามีการร้องขอจองหน่วยความจำขนาด 2^{N-l} แล้ว availability ที่ระดับ *l* จะถูกตรวจสอบเป็นอันดับแรก จากตัวอย่างในรูปที่ 4.5 CBT ที่ระดับ 2 มีโหนด b และ c แทนหน่วยความจำขนาด 2^2 และมีค่า availability เป็น '1'



รูปที่ 4.5 การสร้าง Or-Gate Tree

ฮาร์ดแวร์ Or-Gate tree ที่ทำงานตามฟังก์ชัน (ก) ในหัวข้อที่ 4.2 ทำหน้าที่ในการกำหนดว่ามีพื้นที่หน่วยความจำเพียงพอตามขนาดที่ร้องขอหรือไม่ ถ้ามีเพียงพอก็จะส่งแอดเดรสเริ่มต้นของพื้นที่หน่วยความจำนั้นกลับไป ในรูปที่ 4.6 แสดงการหาแอดเดรสเริ่มต้นของพื้นที่ว่างขนาด 2 บล็อกในบิตเวกเตอร์ที่แตกต่างกัน ประกอบด้วยบิตเวกเตอร์ A และบิตเวกเตอร์ B โดยทั้งสองบิตเวกเตอร์ให้ค่า availability เป็นศูนย์เหมือนกันที่ระดับ 2 นั่นหมายความว่าทั้งสองเวกเตอร์มีพื้นที่ว่างขนาด 2 บล็อกเหมือนกันแต่มีแอดเดรสเริ่มต้นของบล็อกว่างต่างกัน โดยบิตเวกเตอร์ A แอดเดรสของบล็อกว่างอยู่ที่แอดเดรส 100_2 และบิตเวกเตอร์ B แอดเดรสของ บล็อกว่างอยู่ที่แอดเดรส 000_2



แอดเดรสที่ 100 (สำหรับบิตเวกเตอร์ A) แอดเดรสที่ 000 (สำหรับบิตเวกเตอร์ B)

รูปที่ 4.6 แสดงตัวอย่างของการจองพื้นที่หน่วยความจำด้วย Or-Gate Tree ใน 2 บิตเวกเตอร์ที่แตกต่างกัน

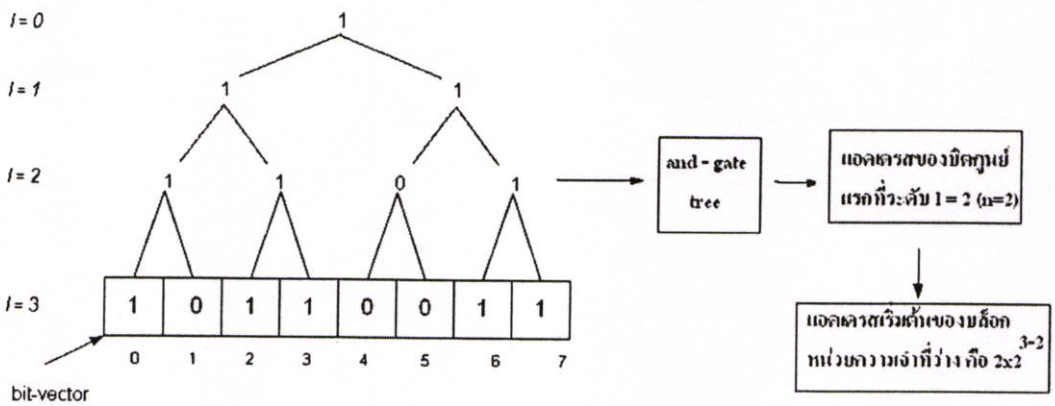
4.2.1.2 ฮาร์ดแวร์ And-Gate-Tree

ในหัวข้อนี้เป็นการออกแบบฮาร์ดแวร์สำหรับกระทำฟังก์ชัน (ข) ในหัวข้อที่ 4.2 ถ้ามีการร้องขอจองพื้นที่หน่วยความขนาด 2^{N-l} แล้วต้องทำการหาแอดเดรสเริ่มต้นของบล็อกว่างที่ระดับ l และในการออกแบบนั้นจะใช้บิตเวกเตอร์แทนบล็อกว่างโดยมีสองแนวทางให้เลือกใช้ แนวทางแรกเก็บบิตเวกเตอร์ในหน่วยความจำ [16] และแนวทางที่สองเก็บบิตเวกเตอร์ไว้ในชิพเรจิสเตอร์ (shift register) [17] โดยแบบที่ใช้ชิพเรจิสเตอร์ การอัพเดตไบนารีทรีต้องใช้ทั้งกระบวนการเลื่อนบิตแบบวงกลม (circular shift) และเลื่อนบิตแบบลอจิก (logical shift) และใช้การค้นหาแบบ first-fit หาตำแหน่งของบล็อกหน่วยความจำที่ว่าง นั่นคือถ้าเจอบล็อกที่ว่างติดกันเพียงพอกับขนาดที่ร้องขอก็จะจองพื้นที่ตรงนั้น ถึงแม้ว่าจะมีพื้นที่ว่างเพียงพอหรือมากกว่าในลำดับบิตเวกเตอร์ถัดไปก็ตาม ดังนั้นเวลาที่ใช้ในการค้นหาบล็อกที่ว่างจึงขึ้นอยู่กับจำนวนบิตเวกเตอร์ทั้งหมดที่มีอยู่

สำหรับในงานวิจัยนี้จะไม่ใช้ชิฟเรจิสเตอร์ (shift register) แต่จะใช้บิตเวกเตอร์ โดยทุกบิตในไบนารีทรีจะเก็บอยู่ใน Or-Gate Tree ซึ่งจะถูกอัพเดตโดยอัตโนมัติด้วยวงจรรวม (combinational logic) และการอัพเดตบิตเวกเตอร์จะทำงานเร็วกว่าชิฟเรจิสเตอร์ นอกจากนี้ Or-Gate Tree ยังสามารถสร้างได้ง่าย และค่าใช้จ่ายต่ำอีกด้วย

แต่ละโหนดในระดับ l ของ Or-Gate Tree จะแทนสถานะการจองบล็อกหน่วยความจำขนาด 2^{N-l} นั่นหมายความว่า ถ้ามีการร้องขอจองบล็อกขนาด 2^{N-l} แล้วจะต้องเจอศูนย์แรกในทรีระดับ l และถ้าใช้ชิฟเรจิสเตอร์ [8] ค้นหาศูนย์แรกในบิตเวกเตอร์จะใช้เวลาในการค้นหาเท่ากับ 2^N รอบสัญญาณนาฬิกา เมื่อหน่วยความจำมีขนาด 2^N บล็อกสำหรับในหัวข้อถัดไปจะนำเสนออัลกอริธึมในการค้นหาแบบไบนารีทรี ซึ่งเหมาะสมกับอัลกอริธึมที่เป็นฮาร์ดแวร์ โดยอัลกอริธึมใหม่นี้สามารถทำการค้นหาแบบ first-fit ในเวลาที่คงที่ โดยการกระทำฟังก์ชัน (ข) ในหัวข้อที่ 4.2 เพื่อหาแอดเดรสเริ่มต้นของบล็อกที่ว่าง

ในหัวข้อที่ 4.2.1.1 ได้อธิบายถึงการใช้ Or-Gate Tree คำนวณค่า availability ของบล็อกว่างที่อยู่ในทรีระดับ *l* โดยเอาท์พุทที่ได้จากทรีระดับ *l* จะเป็นอินพุทของทรีระดับที่ *l* +1 ซึ่งสร้างจากแอนด์เกต โดย And-Gate Tree จะแพร่กระจายข้อมูลไปที่โหนดราก (root) โดยการทำเช่นนี้เป็นการค้นหาแอดเดรสเริ่มต้นของบล็อกว่างที่ทรีระดับ *l* แบบย้อนกลับ (backtracking) สมมุติให้ And-Gate Tree มีแอดเดรสของบิตศูนย์แรกเป็น n แล้วแอดเดรสเริ่มต้นของหน่วยความจำที่ต้องการจองในบิตเวกเตอร์มีค่าเป็น n x 2^{N-l} ตัวอย่างในรูปที่ 4.7 แสดงการหาแอดเดรสเริ่มต้นของบล็อกว่างในบิตเวกเตอร์ที่อยู่ในรูป 4.4

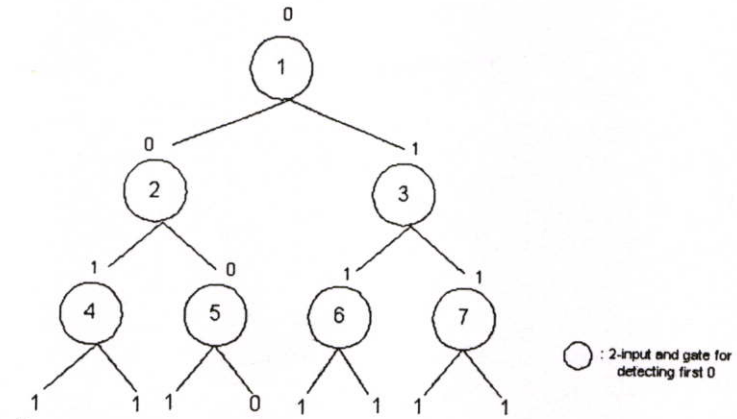


รูปที่ 4.7 แสดงการหาแอดเดรสเริ่มต้นของบล็อกว่างในบิตเวกเตอร์

ในหัวข้อถัดไปจะอธิบายถึงการออกแบบฮาร์ดแวร์ And-Gate Tree โดยละเอียด

4.2.1.2.1 อัลกอริธึมการค้นหาแบบย้อนกลับ (backtracking)

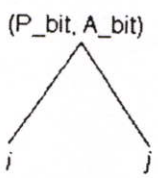
การค้นหาแบบธรรมดา สามารถใช้หาตำแหน่งของบิตศูนย์แรกในบิตเวกเตอร์ที่ระดับ *l* ของ And-Gate Tree ได้ อย่างไรก็ตามจำเป็นต้องมีการทำงานแบบย้อนกลับ (backtracking) ด้วย ในรูปที่ 4.8 แสดงการค้นหาของบิตศูนย์แรก โดยกำหนดให้มีบล็อกหน่วยความจำว่างอยู่ที่ทรีระดับ 2 การค้นหาจะทำผ่านโหนด 1 โหนด 2 และโหนด 4 ตามลำดับ และถ้าโหนด 4 ไม่ว่าง การค้นหาจะต้องย้อนกลับไปที่โหนด 2 ก่อนจะไปทดสอบโหนด 5 ต่อไป



รูปที่ 4.8 แสดงการค้นหาของบิตศูนย์แรกเมื่อบล็อกว่างอยู่ที่ทรีระดับ 2

ในการใช้ไบนารีทรีหาแอดเดรสของบล็อกที่ถูกจองจะกำหนดโดยการนำ A-bit เมื่อเชื่อมต่อกันและทำการค้นหาจากรากของทรีไปยังใบ ในรูปที่ 4.9 แสดงโหนด N ของไบนารีทรี โดยจะมีสองโหนดลูก ซึ่งค่า i และ j จะสัมพันธ์กับโหนดลูกทั้งสอง และค่าทั้งสองนี้จะเป็นตัวกำหนด P_bit และสมการบูลีนสำหรับกำหนดค่า i*j ค่า P_bit และค่า A_bit แสดงดังนี้

- A_bit (แอดเดรสของบิต) เป็นค่าที่ได้จากโหนดลูกทางด้านซ้ายของ P_bit ยกเว้นเสียแต่ว่าโหนดนั้นเป็นใบ ค่าของ A_bit จะไม่ถูกกำหนด
- P_bit (บิตสำหรับใช้แพร่กระจายข้อมูล) จะเก็บผลที่ได้จากการ and กันระหว่างโหนดลูกทั้งสองของ P_bit แต่ถ้าโหนดนั้นเป็นใบจะเก็บสถานะการจองของโหนด (เป็น '0' แทนสถานะว่าง และ '1' แทนสถานะที่ถูกจองแล้ว)



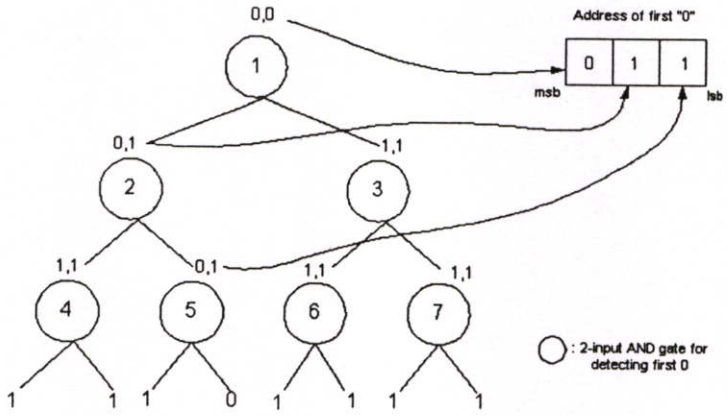
P_bit = *i* * *j*
A_bit = *i*

i	*j*	*i*J*	*address of first 0*
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	no first 0

รูปที่ 4.9 แสดงโหนด N ของไบนารีทรี

จากรูปที่ 4.9 แอดเดรสของบิตศูนย์แรกจะมีค่าเท่ากับ i ยกเว้นกรณี i = 1 และ j = 1 (ลูกทั้งของ P_bit มีค่าเป็น 1 ทั้งคู่) จะไม่กำหนดแอดเดรสของบิตแรก ดังนั้นกรณีนี้จะไม่สามารถใช้หาแอดเดรสของบิตศูนย์แรกได้ และถ้า P_bit เป็น '0' (กรณีที่ i เป็น '0') แล้ว A_bit จะแทน

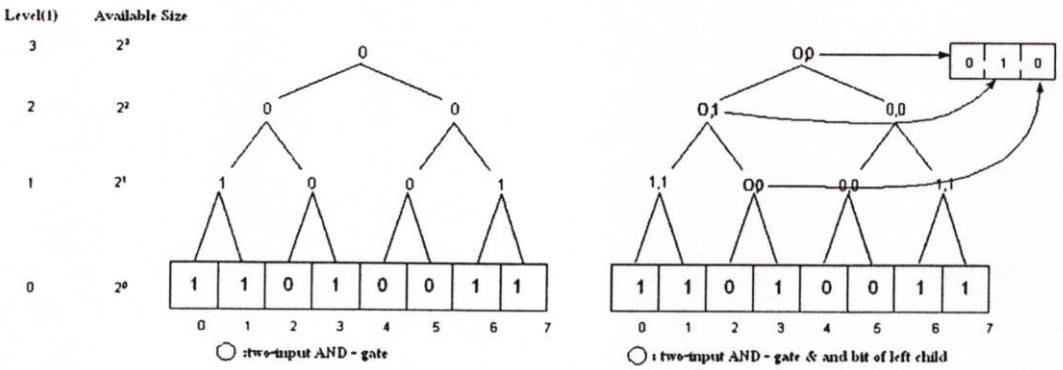
แอดเดรสของบิตศูนย์แรก แต่ถ้า P_bit เป็น '1' แล้ว A_bit จะไม่เก็บแอดเดรส และจากรูปที่ 4.9 แต่ละโหนดจะมีบิตที่เรียกว่า propagation bit หรือ P_bit จะเป็นตัวกำหนดการแพร่ข้อมูลจากใบไปยังรากของทรี ซึ่ง P_bit เป็นเอาท์พุทของแอนด์เกตในแต่ละโหนด และในการค้นหาบิตศูนย์แรกค่าของ P_bit ในระดับ *l* ต้องมีการย้อนกลับ และเพื่อหลีกเลี่ยงการทำเช่นนี้จะใช้ A_bit (address bit) เข้ามาช่วย ตัวอย่างดังรูปที่ 4.10



รูปที่ 4.10 แสดงการค้นหาของบิตศูนย์แรกและในไบนารีทรีจาก 8 บิตเวกเตอร์

ค่าของ A_bit จะคัดลอกมาจากโหนดลูกทางด้านซ้ายของ P_bit ซึ่งเป็นบิตที่มีไว้สำหรับการค้นหาไปยังโหนดลูก ตัวอย่างเช่น บิตที่ 2 ของโหนดราก หรือโหนด 1 เป็น '0' แสดงว่าในขั้นตอนถัดไปโหนดลูกด้านซ้าย คือโหนด 2 จะถูกทดสอบ และถ้าบิตที่ 2 ของโหนด 2 เป็น '1' แสดงว่าโหนดลูกด้านขวาของโหนด 2 คือโหนด 5 จะถูกทดสอบ ดังนั้นเป็นการค้นหาโดยตรงไปยังโหนดลูกที่มีพื้นที่ว่างเพียงพอสำหรับการจองใช้งาน หรืออีกนัยก็คือ A_bit เป็นบิตที่ใช้กำหนดแอดเดรสเริ่มต้นของบิตศูนย์แรกที่พบในบิตเวกเตอร์โดยค้นหาจากโหนดรากไปยังโหนดใบ โดยแอดเดรสของบล็อกหน่วยความจำที่ถูกจองเกิดจากการนำเอา A_bit ในแต่ละโหนด ตามเส้นทางเดินจากโหนดรากไปยังใบของทรี มารวมเข้าด้วยกัน และในระหว่างที่มีการเดินทางไปยังโหนดลูกค่าของ A_bit ในแต่ละโหนดที่มีการเดินทางผ่านจะต้องทำการแลทช์ไว้ก่อน จนกว่าจะเดินทางไปยังใบของทรี แล้วค่อยนำเอาค่า A_bit ที่แลทช์ได้มารวมเข้าด้วยกันเป็นแอดเดรสของบิตศูนย์แรกในบิตเวกเตอร์ และจากรูปที่ 4.10 เป็นตัวอย่างการหาบิตศูนย์แรกในบิตเวกเตอร์ โดยเริ่มจากโหนดราก ตรวจสอบว่าบิตที่ 2 หรือ A_bit มีค่าเป็น '0' หรือ '1' ถ้ามีค่า '0' ก็ไปยังโหนดลูกทางซ้าย ถ้ามีค่า '1'ก็ไปโหนดลูกทางด้านขวา จากรูป โหนดรากบิตที่ 2 มีค่า '0' แสดงว่าเดินทางไปที่โหนดลูกด้านซ้าย หรือโหนด 2 นั่นเอง ก่อนเดินทางไปยังโหนด 2 ค่า A_bit ของโหนดราก ต้องทำการแลทช์ไว้ก่อน จากนั้นก็ทำการทำสอบโหนด 2 เนื่องจากบิต A_bit มีค่าเป็น 1 แสดงว่าให้ไปที่โหนด 5 พร้อมกับแลทช์ค่า '1' ไว้ด้วย และสุดท้ายทำการทดสอบโหนด 5 โดยบิต A_bit ของโหนด 5 มีค่า

เป็น 1 แสดงว่าต้องไปทางลูกด้านขวา นั่นคือใบ ซึ่งเป็นบิตศูนย์แรกนั่นเอง จากนั้นทำการคำนวณหาค่าแอดเดรสของบิตศูนย์แรก โดยนำเอาค่า A_bit ที่ได้ทำการแลทช์ไว้นำมารวมกัน โดยเรียงจากโหนดรากเป็นบิตนัยสำคัญสูงสุด (msb) และโหนด 5 เป็นบิตนัยสำคัญต่ำสุด (lsb) ดังนั้นแอดเดรสของบิตศูนย์แรก คือ 011_2 สำหรับรูปที่ 4.11 เป็นอีกตัวอย่างของการหาแอดรเดรสของบิตศูนย์แรก โดยบิตศูนย์แรกในบิตเวกเตอร์อยู่แอดเดรสที่ 2 ดังนั้นเส้นทางการเดินจะเริ่มจากโหนดราก โหนด 2 และสุดท้ายโหนด 5 ดังนั้นแอดเดรสของบิตศูนย์แรกจึงมีค่าเป็น 010_2



รูปที่ 4.11 แสดงการค้นหาของบิตศูนย์แรก

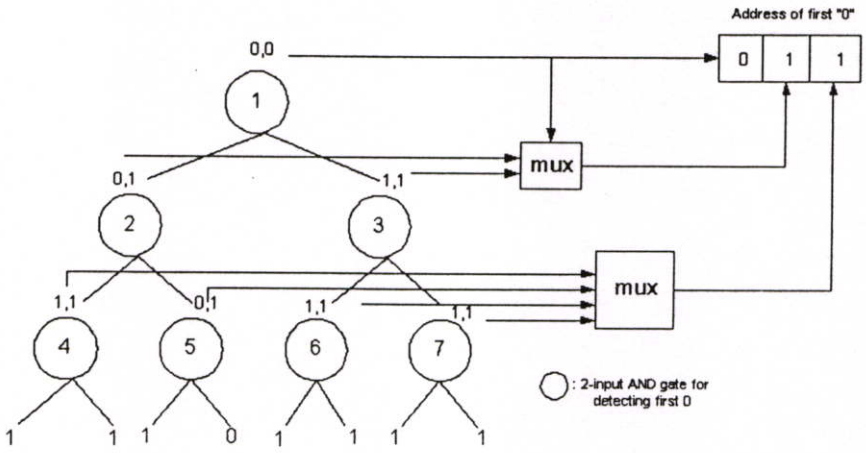
ดังนั้นถ้าฮาร์ดแวร์ CBT จัดรูปแบบ ทรีสำหรับบิตเวกเตอร์ขนาด 2^l แล้ว CBT จะมีระดับของทรีทั้งหมด l ระดับ ดังนั้นเส้นทางการเดินเพื่อให้ได้แอดเดรสของบิตศูนย์แรกจะต้องผ่านจำนวนโหนดทั้งหมด l โหนด สำหรับในหัวข้อถัดไปจะกล่าวถึงการเปลี่ยนจากอัลกอริธึมเป็นฮาร์ดแวร์

4.2.1.2.2 การเปลี่ยนจากอัลกอริธึมเป็นฮาร์ดแวร์

ในหัวข้อนี้จะกล่าวถึงวิธีการเปลี่ยนอัลกอริธึมที่ได้อธิบายในหัวข้อที่ผ่านมา เป็นฮาร์ดแวร์เพื่อสามารถนำไปสร้างจริงได้ จากที่กล่าวมาแต่ละระดับของทรี ยกเว้นโหนดรากจะมีจำนวนโหนดมากกว่า 1 โหนดและมีจำนวน A_bit มากกว่า 1 และสมมุติว่าทรีในรูปที่ 4.4 จองพื้นที่หน่วยความจำขนาด 2^N บล็อกโดยในแต่ละระดับ l ของทรีจะมีมัลติเพล็กเซอร์สำหรับกำหนดแอดเดรสของบิตศูนย์แรก โดยอินพุทของมัลติเพล็กเซอร์จะมาจาก A_bit ของโหนดในแต่ละระดับของทรี ดังนั้นมัลติเพล็กเซอร์ของทรีระดับที่ l จะเป็นมัลติเพล็กเซอร์ขนาด 2^{N-l} อินพุทและ 1 เอาท์-พุท และต้องใช้จำนวนมัลติเพล็กเซอร์ทั้งหมด N-1 ตัวโดยทรีระดับที่ N และระดับที่ 0 ไม่ต้องมีมัล-ติเพล็กเซอร์ ตัวอย่างเช่น หน่วยความจำขนาด 8 บิตเวกเตอร์ (2^3, N = 3) จะมีจำนวนมัลติเพล็กเซอร์ทั้งหมด 2 มัลติเพล็กเซอร์ โดยมัลติเพล็กเซอร์ระดับที่ l = 1 จะเป็นมัลติเพล็กเซอร์ขนาด 2^{3-1} อินพุท

และ 1 เอาท์พุท และมัลติเพล็กเซอร์ระดับที่ $l = 2$ จะเป็นมัลติเพล็กเซอร์ขนาด 2^{3-2} อินพุท และ 1 เอาท์พุท และเอาท์พุทที่ได้จากมัลติเพล็กเซอร์จะนำมารวมกันเป็นบิตแอดเดรสของบิตศูนย์แรกในบิตเวกเตอร์ และบิตที่ใช้เลือกเอาท์พุทของมัลติเพล็กเซอร์มาจากบิตแอดเดรสของบิตศูนย์แรกนั้นเอง นั่นคือมัลติเพล็กเซอร์ที่ทรีระดับ 2 บิตเลือกเอาท์พุทจะเป็นบิตแอดเดรสบิตที่ 2 และมัลติเพล็กเซอร์ที่ทรีระดับ 1 บิตเลือกเอาท์พุทจะเป็นบิตแอดเดรสบิตที่ 2 และบิตที่ 1 ตามลำดับ

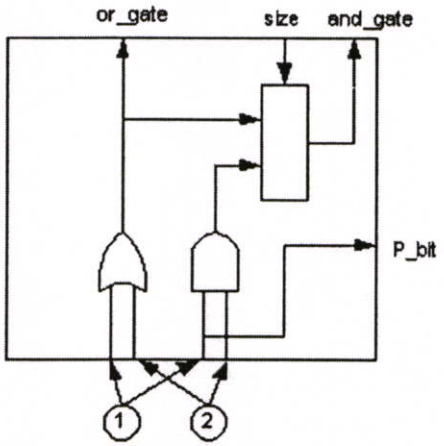
ในการแปลงจากอัลกอริธึมเป็นฮาร์ดแวร์นั้นเพื่อประหยัดค่าใช้จ่ายทางด้านฮาร์ดแวร์ จะไม่มีฮาร์ดแวร์สำหรับเก็บ A_bit แต่จะใช้ค่าของโหนดลูกทางด้านซ้ายแทนเป็น A_bit ซึ่งจะทำให้แต่ละโหนดใช้เพียง 20 เกทเท่านั้น และจากรูปที่ 4.12 เป็นฮาร์ดแวร์ของ And-Gate Tree ที่ได้จากรูปที่ 4.10 ประกอบด้วยมัลติเพล็กเซอร์ 2 อินพุท และมัลติเพล็กเซอร์ 4 อินพุท โดยบิตศูนย์แรกอยู่ที่แอดเดรสที่ 3 ในบิตเวกเตอร์ ดังนั้นเส้นทางการค้นหาจะเริ่มจากโหนดราก โหนด 2 และโหนด 5 โดยมัลติเพล็กเซอร์ 2 อินพุทจะเลือกเอาท์พุทจากโหนด 2 และมัลติเพล็กเซอร์ 4 อินพุทจะเลือกเอาท์พุทจากโหนด 5 ทำให้ได้แอดเดรสของบิตศูนย์แรกเป็น 011_2



รูปที่ 4.12 แสดงเปลี่ยน And-Gate Tree ไปเป็นฮาร์ดแวร์

4.2.1.3 การรวมฮาร์ดแวร์ And-Gate Tree และฮาร์ดแวร์ Or-Gate Tree

ในหัวข้อนี้จะเป็นการนำฮาร์ดแวร์ And-Gate Tree และฮาร์ดแวร์ Or-Gate Tree มาเชื่อมเข้าด้วยกัน จากรูปที่ 4.4 ทำการตรวจสอบดูว่าจำนวนบล็อกหน่วยความจำมีเพียงพอสำหรับการจองพื้นที่ใหม่หรือไม่ ทำโดยการตรวจสอบจากค่า availability ของบล็อกซึ่งได้จากฮาร์ดแวร์ Or-Gate Tree ขณะการหาแอดเดรสของบล็อกว่างจะทำผ่าน And-Gate Tree และเพื่อลดค่าใช้จ่ายและความซับซ้อนใน And-Gate Tree เราจะนำออร์เกตและแอนด์เกตมารวมกันเป็นโหนด Or-And Gate แสดงดังรูปที่ 4.13




```
PROCESS ()
BEGIN
   out_or <= or_a OR or_b;
    IF (level = '1') THEN
      out_and <= or_a OR or_b;
    ELSE
      out_and <= and_a AND and_b;
    END IF;
    out_to_mux <= and_a;
END PROCESS;
```

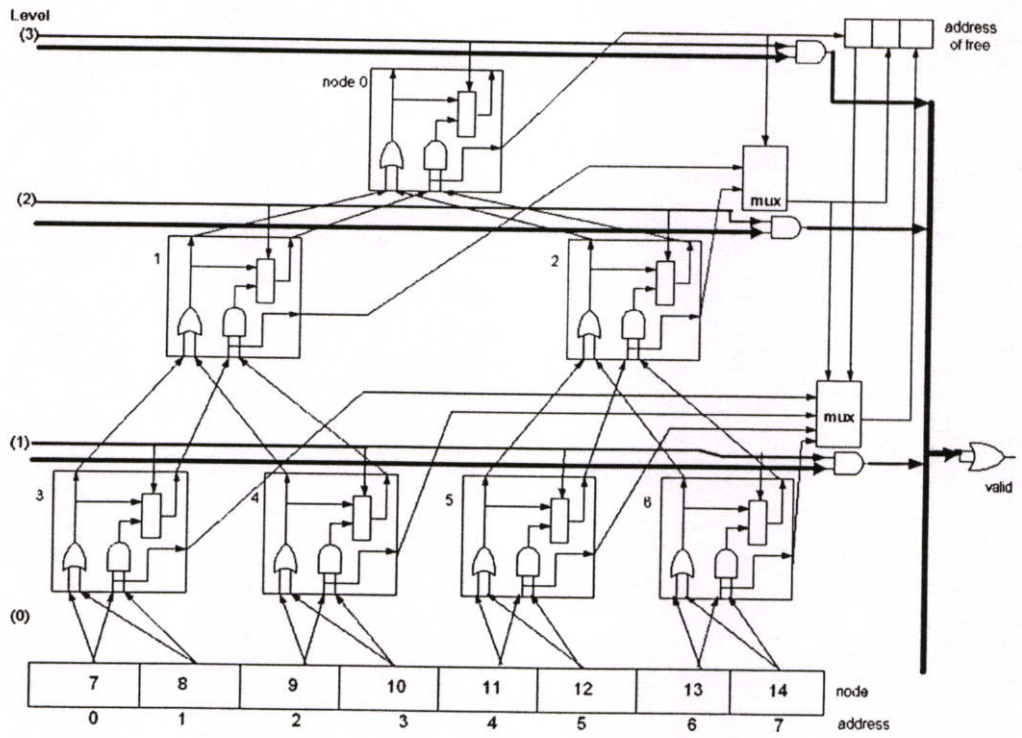
(b)

รูปที่ 4.13 แสดงตัวอย่างของโหนด Or-And Gate

จากรูปที่ 4.13 (a) แต่ละโหนดใน Or-Gate Tree ประกอบด้วยออร์เกต แอนด์เกต และมัลติเพล็กเซอร์ โดยมัลติเพล็กเซอร์จะส่งค่าเอาท์พุทจากออร์เกต หรือแอนด์เกตไปยัง And-Gate Tree ของโหนดที่อยู่ด้านบนขึ้นไป และการส่งค่านี้จะทำเฉพาะเมื่อมี size มาเลือกเท่านั้น โดย size จะกำหนดโดยค่าระดับของทรีที่ต้องการจองนั่นเอง ส่วน P_bit ใช้สำหรับการทำงานแบบย้อนกลับโดยจะส่งไปยังมัลติเพล็กเซอร์ เพื่อหาแอดเดรสของบล็อกหน่วยความจำที่ว่าง และในแต่ละระดับของทรีก็จะนำค่าของออร์เกตมากระทำการ and กันเพื่อกำหนดค่า availability ของขนาดหน่วยความจำในแต่ระดับของทรี และในรูปที่ 4.13 (b) เป็นโค้ดวีเอชดีแอลของโหนด Or-And Gate Tree ที่ใช้ในการสร้างฮาร์ดแวร์

4.2.2.4 ฮาร์ดแวร์ Or-And Gate Tree แบบสมบูรณ์ (CBT)

หลังจากนำเอาออร์เกตและแอนด์เกตมารวมกันเป็น Or-And Gate ที่ทำงานอยู่ภายในโหนดเดียวกันแล้ว ในหัวข้อนี้จะเป็นการนำเอา Or-And Gate มาสร้างเป็น CBT แบบสมบูรณ์ โดยในรูปที่ 4.14 เป็นตัวอย่างของฮาร์ดแวร์ CBT ขนาด 8 บิตเวกเตอร์แบบสมบูรณ์

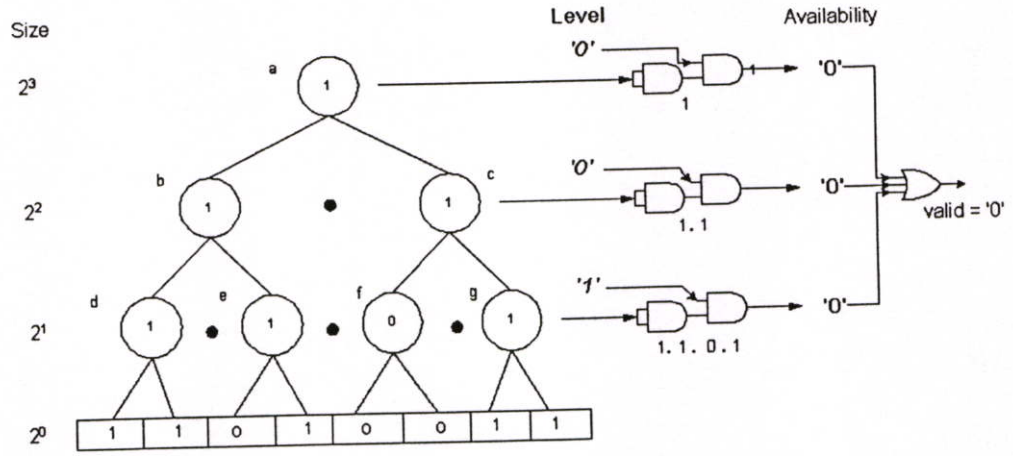


รูปที่ 4.14 แสดงฮาร์ดแวร์ Or-And Gate Tree ขนาด 8 บิตเวกเตอร์แบบสมบูรณ์ (CBT)

จากรูปที่ 4.14 เป็นฮาร์ดแวร์ CBT ที่เกิดจากการนำเอาโหนด Or-And Gate ในรูปที่ 4.13 เชื่อมต่อเข้าด้วยกันในรูปแบบทรีเหมือนในรูปที่ 4.12 ซึ่งจำนวนโหนดมีทั้งหมด 14 โหนด โดยโหนด 0 ถึงโหนด 6 จะเป็นโหนด Or-And Gate ขณะที่โหนด 7 ถึงโหนด 14 ใช้บิตในบิตเวกเตอร์เป็นโหนด นั่นคือโหนด 7 จะตรงกับบิต 0 โหนด 8 จะตรงกับบิต 1 และโหนด 14 จะตรงกับบิต 7 ตามลำดับ และในแต่ละระดับของทรี จะนำเอาท์พุทที่ได้จากออร์เกตในแต่ละโหนดมาทำการ and กันกับค่าระดับ (level) ซึ่งจะได้ค่า availability ของแต่ละระดับทรีออกมา โดยค่าระดับของทรีนี้จะเป็นตัวกำหนดขนาดของบล็อกหน่วยความจำที่สามารถจองได้ และ ณ เวลาหนึ่งๆ จะมีค่าเป็นลอจิก '1' เพียงระดับเดียวเท่านั้น และเพื่อความง่ายในการสร้างจะไม่ใช้ค่าระดับที่ 0 เช่น สมมุติบิตเวกเตอร์ขนาด 8 บิต และต้องการจองขนาด 1 บล็อกค่าระดับจะเป็น "000" สำหรับการจองขนาด 2 บล็อกค่าระดับจะเป็น "001" สำหรับการจองขนาด 4 บล็อกค่าระดับจะเป็น "010" และสำหรับการจองขนาด 8 บล็อกค่าระดับจะเป็น "100" ดังนั้น ถ้ามีบิตเวกเตอร์ขนาด N บิต จำนวนระดับที่ใช้จะเท่ากับ N-1 โดยเริ่มจากระดับที่ 1 ถึงระดับที่ N-1 และเนื่องจากใช้ระบบการจองหน่วยความจำแบบไบนารีบัดดี ซึ่งได้อธิบายไว้แล้วในบทที่ 2 การจองพื้นที่หน่วยความจำจะต้องจองเป็นกำลังของ 2 เช่น จองขนาด 1 บล็อกจองขนาด 2 บล็อกจองขนาด 4 บล็อกและจองขนาด 8 บล็อก และถึงแม้ว่าจะมีการร้องขอการจองหน่วยความจำไม่เป็นกำลังของ 2 ระบบก็จะจองพื้นที่เป็นอย่างน้อยกำลัง 2 เช่น ถ้าต้องการจองพื้นที่หน่วยความจำ 3 บล็อกจะเห็นว่าจำนวน 3 บล็อกที่ต้องการ

จองนั้นอยู่ในช่วง 2 บล็อกและ 4 บล็อกดังนั้นระบบก็จะจองให้อย่างน้อย 4 บล็อกและค่าระดับที่ใช้จะเป็น “010” (บิตเรียงจากระดับ 3 ถึงระดับ 1)

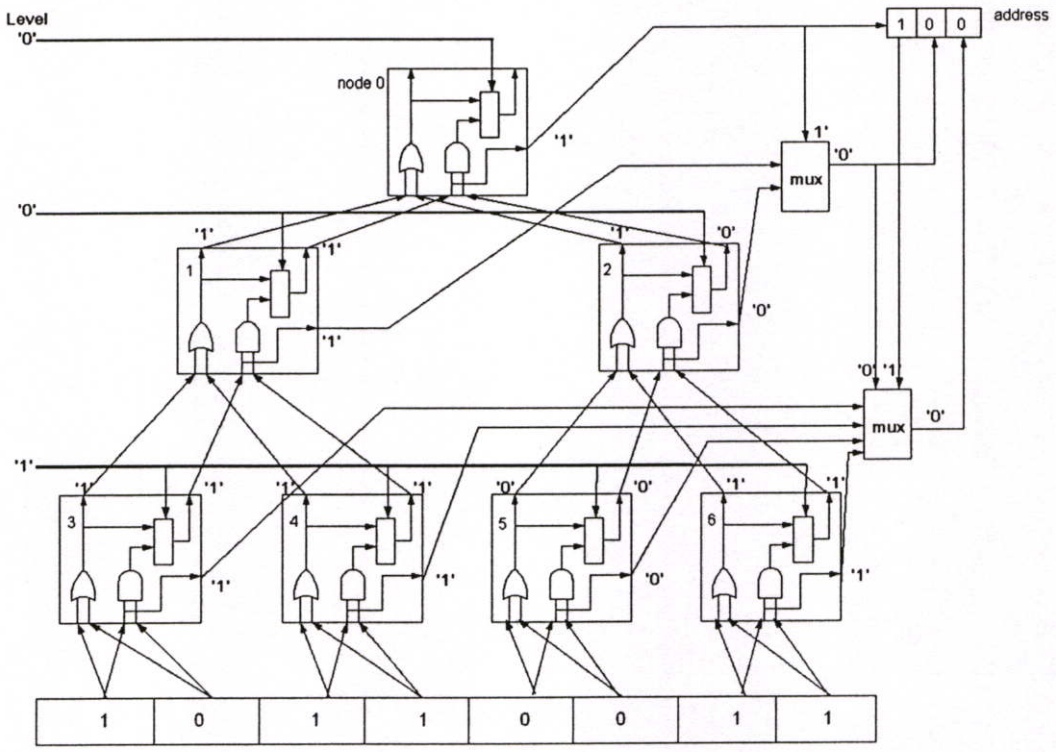
ระบบจะนำเอาค่า availability ที่ได้ในแต่ละระดับมาทำการ or กัน ถ้าเอาท์พุทที่ได้เป็นลอจิก ‘0’ แสดงว่าระบบมีพื้นที่บล็อกหน่วยความจำว่างเพียงพอตามที่ต้องการ และถ้าเป็นลอจิก ‘1’ แสดงว่ามีพื้นที่บล็อกว่างไม่เพียงพอ เช่น ในรูปที่ 4.15 สมมุติกำหนดบิตเวกเตอร์เป็น 11010011_2 และต้องการจองพื้นที่หน่วยความจำจำนวน 2 บล็อกดังนั้นค่าระดับที่ใช้เลือกขนาดบล็อกจะเป็น 001_2 ซึ่งจะทำให้ค่า availability ของทรีในระดับที่ 3 มีค่าเป็น ‘0’ ระดับที่ 2 มีค่าเป็น ‘0’ และระดับที่ 1 มีค่าเป็น ‘0’ จากนั้นนำเข้าสู่ออร์เกตมีผลทำให้ค่า valid ที่ได้มีค่าเป็น ‘0’ นั่นหมายความว่ามีพื้นที่ว่างจำนวน 2 บล็อก



รูปที่ 4.15 แสดงการหาค่า valid สำหรับการจองพื้นที่หน่วยความจำขนาด 2 บล็อก

ส่วนการหาแอดเดรสเริ่มต้นของบล็อกหน่วยความจำที่ว่างนั้น จากรูปที่ 4.12 จะนำเอาค่า P_bit ของโหนด 0 ถึงโหนด 6 มาเข้ามัลติเพล็กเซอร์ โดยค่าแอดเดรสบิตที่ 2 มาจากค่า P_bit ของโหนด 0 และค่าแอดเดรสบิตที่ 1 มาจากมัลติเพล็กเซอร์ขนาด 2 อินพุท ซึ่งอินพุทมาจากค่า P_bit ของโหนด 1 และโหนด 2 ตามลำดับ และค่าแอดเดรสบิตที่ 0 มาจากมัลติเพล็กเซอร์ขนาด 4 อินพุท ซึ่งอินพุทมาจากค่า P_bit ของโหนด 3 โหนด 4 โหนด 5 และโหนด 6 ตามลำดับ และในการเลือกเอาท์พุทของมัลติเพล็กเซอร์ค่าที่ใช้เลือกจะมาจากค่าแอดเดรส โดยมัลติเพล็กเซอร์ขนาด 2 อินพุทจะใช้ค่าแอดเดรสบิตที่ 2 และมัลติเพล็กเซอร์ขนาด 4 อินพุทจะใช้ค่าแอดเดรสบิตที่ 2 และบิตที่ 1 เช่น ในรูปที่ 4.16 สมมุติกำหนดบิตเวกเตอร์เป็น 11010011_2 และต้องการจองพื้นที่หน่วยความจำจำนวน 2 บล็อกดังนั้นค่าระดับที่ใช้เลือกขนาดบล็อกจะเป็น 001_2 และค่า P_bit ของโหนด 3 โหนด 4 โหนด 5 และโหนด 6 เท่ากับ “1101” ค่า P_bit ของโหนด 1 และโหนด 2 เท่ากับ “10” และค่า P_bit ของโหนด 0 เท่ากับ ‘1’ ดังนั้นค่าแอดเดรสของบล็อกว่างบิตที่ 2 เท่ากับ ‘1’ (ได้มาจาก

P_bit ของโหนด 0) ค่าแอดเดรสบิตที่ 1 เท่ากับ '0' (ได้มาจากเอาท์พุทของมัลติเพล็กเซอร์ขนาด 2 อินพุท โดยอินพุทมาจาก P_bit ของโหนด 1 และโหนด 2 และตัวเลือกเอาท์พุทจะมาจากบิตที่ 2 ของแอดเดรส) และค่าแอดเดรสบิตที่ 0 เท่ากับ '0' (ได้มาจากเอาท์พุทของมัลติเพล็กเซอร์ขนาด 4 อินพุท โดยอินพุทมาจาก P_bit โหนด 3 โหนด 4 โหนด 5 และโหนด 6 และตัวเลือกเอาท์พุทจะมาจากบิตที่ 2 และบิตที่ 1 ของแอดเดรส) ซึ่งจะทำให้ได้แอดเดรสของบล็อกหน่วยความจำที่ว่าง 2 บล็อกติดกันมีค่าเท่ากับ 100_2 ดังนั้นถ้าบิตเวกเตอร์ขนาด 2^N บิต จำนวนโหนด or-and gate ที่ใช้เท่ากับ 2^N -1 และจำนวนมัลติเพล็กเซอร์ที่ใช้ทั้งหมดเท่ากับ N-1 โดยมีมัลติเพล็กเซอร์ตั้งแต่ 2 อินพุท จนถึง 2^{N-1} อินพุท



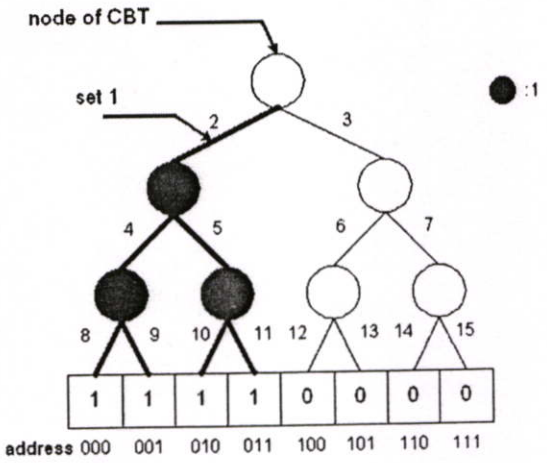
รูปที่ 4.16 แสดงการหาแอดเดรสของบล็อกความจำที่ว่างติดกันขนาด 2 บล็อก

4.2.2 ฮาร์ดแวร์ Bit-Flipper

ตามฟังก์ชันการทำงาน (ค) ในหัวข้อ 4.2 เมื่อ CBT ทำการจองบล็อกพื้นที่หน่วยความจำจะต้องมีการเก็บสถานะการจองโดยการเซตบิตที่สัมพันธ์กับบล็อกนั้นในบิตเวกเตอร์ที่เก็บสถานะการจอง และถ้าเป็นการคืนพื้นที่บล็อกหน่วยความจำที่จองไว้ ทำได้ง่ายโดยเพียงเคลียร์บิตที่สัมพันธ์กับบล็อกนั้น ดังนั้นเราจะสร้างฮาร์ดแวร์ Bit-Flipper เพื่อทำหน้าที่เปลี่ยนบิตในบิตเวกเตอร์จากลอจิก 0 เป็นลอจิก 1 สำหรับการจองบล็อกพื้นที่หน่วยความจำ โดยอินพุทของ Bit-Flipper ประกอบด้วยแอดเดรสเริ่มต้นของบิตที่ต้องการจองและจำนวนของบิตที่ต้องการจองและในการ

เซตบิตจะเซตเฉพาะบิตที่ต้องการตามการจองเท่านั้น ไม่ต้องเซตทั้งหมดตามขนาดบัดดี โดยในหัวข้อนี้จะเป็นการออกแบบฮาร์ดแวร์ที่สามารถเซตบิตตามจำนวนบิตที่ต้องการลงบนบิตเวกเตอร์ผ่านวงจรรวม

การทำงานของฮาร์ดแวร์ Bit-Flipper จะเป็นการแพร่กระจายข้อมูลจากโหนดรากไปยังใบของทรี ซึ่งไม่เหมือนกับ CBT ที่ได้อธิบายในหัวข้อก่อนหน้านี้ ซึ่งการทำงานจะแพร่กระจายจากใบขึ้นไปยังโหนดรากของทรี ที่เป็นเช่นนี้เพราะว่าต้องการเซตบิตที่ใบของทรี สำหรับตัวอย่างในรูปที่ 4.17 เป็นการเซตบิตบนบิตเวกเตอร์ขนาด 8 บิต และ CBT มีขนาด 7 โหนด โดยในแต่ละโหนดจะมี 1 อินพุท และ 2 เอาท์พุท และมีการกระจายสัญญาณจากโหนดแม่ไปยังโหนดลูกทางด้านซ้ายและทางด้านขวาตามคำสั่งที่ให้ทำ เช่น ถ้าเราต้องการจองพื้นที่ขนาด 100_2 บล็อกโดยเริ่มต้นที่แอดเดรส 000_2 โดยจะแพร่กระจาย 1 จากโหนดลูกทางด้านซ้ายของโหนดรากไปยังใบ (บิตเวกเตอร์) โดยเส้นสีดำเข้มในรูปที่ 4.17 แสดงเส้นทางการแพร่กระจายข้อมูล



รูปที่ 4.17 แสดงตัวอย่างการใช้ CBT เซตบิตบนบิตเวกเตอร์

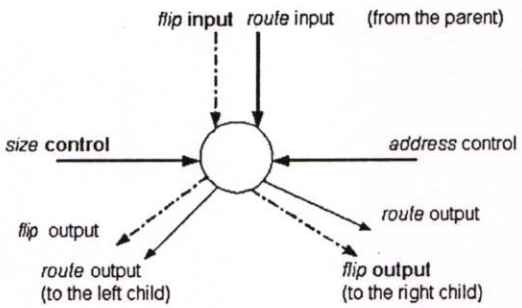
ในการออกแบบดังที่กล่าวมาจะง่ายในการออกแบบ แต่มีข้อจำกัดดังนี้

- ในการเซตบิตบนบิตเวกเตอร์จะเซตเป็นจำนวน 2^K ของบิต เมื่อ K มีค่าตั้งแต่ 1, 2, 3, …,N ซึ่งไม่สามารถเซตบิตตามจำนวนที่ต้องการได้
- ต้องมีวงจรลอจิกเพิ่มเติมสำหรับการกำหนดว่าทรีย่อยไหนที่ต้องการจอง โดยรับอินพุทที่เป็นแอดเดรสเริ่มต้นและจำนวนของบิตที่ต้องการเซตบนบิตเวกเตอร์

เพื่อลดข้อจำกัดเหล่านี้ จำเป็นต้องมีการสร้างโหนดขึ้นมาใหม่ที่มีความฉลาดในการแพร่กระจายข้อมูลใน CBT โดยในการออกแบบโหนดจะอธิบายในหัวข้อถัดไป

4.2.2.1 การแพร่กระจายข้อมูลการจองในทรี

ในการออกแบบโหนดให้มีความฉลาดในการแพร่กระจายข้อมูลนั้น โหนดจะต้องมีอินพุทอย่างน้อย 2 สัญญาณสำหรับเชื่อมระหว่างโหนดแม่และโหนดลูกเข้าด้วยกัน โดยสัญญาณแรก เป็นสัญญาณที่ใช้เซตบิตทุกบิตในทรีย่อย เมื่อมีค่าเป็นลอจิก '1' เรียกว่าสัญญาณ *flip* และถ้าสัญญาณ *flip* ไม่เซตหรือมีค่าเป็นลอจิก '0' ก็จะไปพิจารณาสัญญาณที่มีชื่อเรียกว่า *route* ซึ่งเป็นสัญญาณที่ใช้เซตบิตบางบิตในทรีย่อย เมื่อมีค่าเป็นลอจิก '1' และถ้าทั้งสัญญาณ *flip* และ *route* ไม่มีสัญญาณใดเป็นลอจิก '1' ก็จะไม่มีการเซตบิตในทรีย่อย นอกจากทั้งสองสัญญาณที่กล่าวมาแล้ว ยังต้องการอีก 2 สัญญาณเพื่อควบคุมแต่ละระดับของทรี โดยการเก็บสัญญาณจะเก็บในรูปของเรจิสเตอร์จำนวน 2 เรจิสเตอร์ โดยรีจิเตอร์แรกสำหรับเก็บจำนวนบิตที่ต้องการเซต เรียกว่า *size control* และอีกเรจิสเตอร์สำหรับเก็บแอดเดรสเริ่มต้นของบิตที่ต้องการเซตบนบิตเวกเตอร์ เรียกว่า *address control* ดังนั้นแต่ละโหนดจะมี 4 อินพุท และ 4 เอาท์พุท ประกอบด้วย สัญญาณ *flip* และสัญญาณ *route* ของลูกทางด้านซ้าย และสัญญาณ *flip* และสัญญาณ *route* ของลูกทางด้านขวา โดยไดอะแกรมแสดงอินพุทเอาท์พุทของโหนด แสดงดังรูปที่ 4.17



รูปที่ 4.18 แสดงอินพุทและเอาท์พุทของโหนดในฮาร์ดแวร์ Bit-Flipper

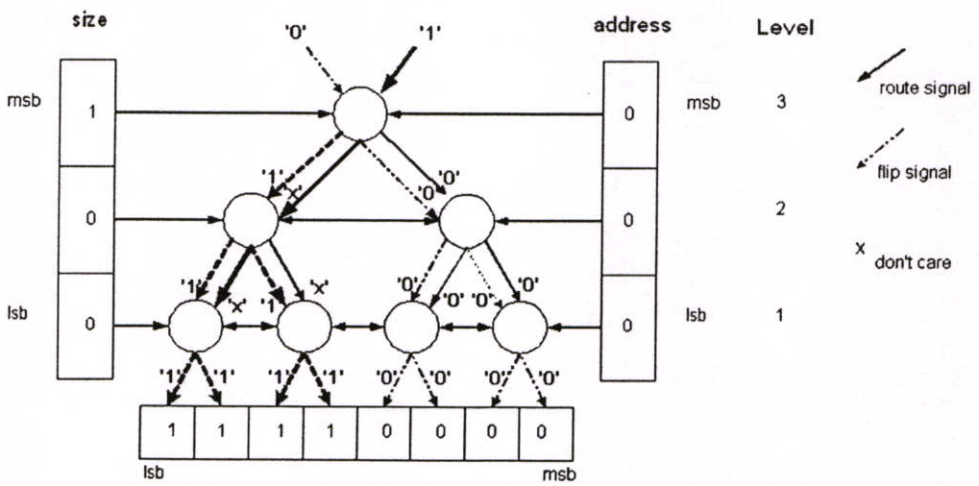
ในการกำหนดการทำงานของโหนด จะให้สัญญาณอินพุท *flip* มีลำดับความสำคัญสูงสุด โดยเมื่อ *flip* = '1' จะมีการแพร่กระจายข้อมูลไปยังใบของทรีย่อยๆ นั่นคือจะมีการเซตตามลำดับของทรีย่อยไปจนกระทั่งถึงใบของทรี ขณะที่สัญญาณอินพุท *route* จะกำหนดให้มีความสำคัญลำดับถัดไป โดยเมื่อ *route* = '0' เอาท์พุททั้ง 4 จะมีค่าเป็นลอจิก '0' อย่างไรก็ตาม ถ้า *route* = '1' ทรีย่อยอาจจะถูกเซตหรือไม่เซตก็ได้ ทั้งนี้ขึ้นอยู่กับสัญญาณ *address control* และสัญญาณ *size control* โดยถ้า *size control* = '1' แล้วเอาท์พุท *flip* ของโหนดจะมีค่าเป็นลอจิก '1' นั่นคือจะเริ่มมีการกระจายสัญญาณ *flip* ไปยังทรีย่อย สำหรับตารางความจริงของสัญญาณทั้งหมดของโหนดแสดงในตารางที่ 4.2

ตารางที่ 4.2 ค่าตารางความจริงของโหนดในฮาร์ดแวร์ bit-flipper

flip input	*route* input	*size control*	*address control*	*flip* output (left)	*route* output (left)	*flip* output (right)	*route* output (right)
1	x	x	x	1	x	1	x
0	0	x	x	0	0	0	0
0	1	0	0	0	1	0	0
0	1	0	1	0	0	0	1
0	1	1	0	1	x	0	0
0	1	1	1	0	0	1	x

ตัวอย่างการสร้างฮาร์ดแวร์ CBT สำหรับเซตบิตบนบิตเวกเตอร์ขนาด 8 บิต แสดงดังรูปที่ 4.19 โดยทรีที่ใช้สร้างจะนำมาจากรูปที่ 4.17 โดยสรุปได้ 3 ข้อ ดังนี้

1) เมื่อต้องการเซตบิตบนบิตเวกเตอร์ จะกำหนดให้สัญญาณอินพุท *flip* ของโหนดรากเป็นลอจิก '0'
2) และสัญญาณอินพุท *route* ของโหนดรากก็จะเซตเป็นลอจิก '1' ด้วย ถ้าต้องการเซตบิตบนบิตเวกเตอร์ แต่ถ้าเป็นลอจิก '0' ก็หมายความว่าจะไม่มีการเซตบิตบนบิตเวกเตอร์
3) ในโหนดของทรีระดับต่ำสุด (โหนด 4 โหนด 5 โหนด 6 และโหนด 7) จะมีเฉพาะสัญญาณเอาท์พุท *flip* เท่านั้น ขณะที่สัญญาณเอาท์พุท *route* ของโหนดจะไม่มีในทรีระดับต่ำสุด



รูปที่ 4.19 แสดงการเซตบิตบนบิตเวกเตอร์

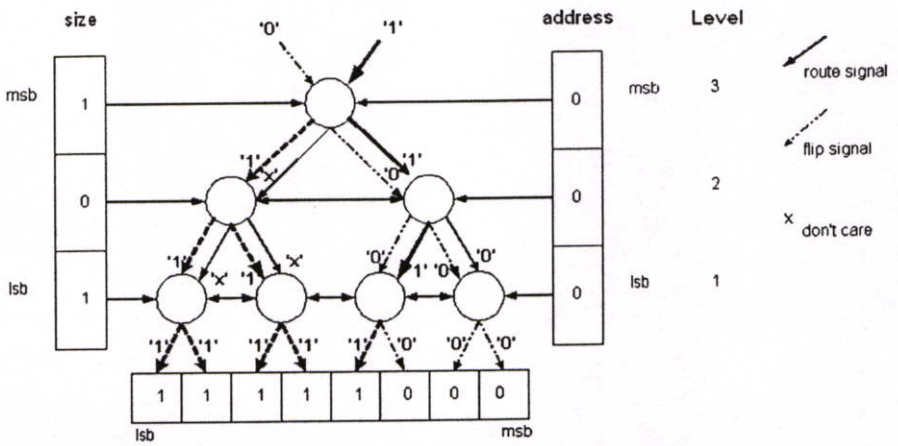
4.2.2.2 การเซตบิตบนบิตเวกเตอร์จำนวน n บิต

จากฮาร์ดแวร์ในรูป 4.19 กระทำการเซตบิตในลักษณะ 2^K บิต เมื่อ k = 1, 2, 3,...,N ซึ่งไม่สามารถทำการเซตบิตตามจำนวนบิตที่ต้องการได้ เช่นต้องการเซต 5 บิต จะไม่สามารถทำได้ ดังนั้นจึงจำเป็นต้องพัฒนาแก้ไขตารางความจริงของโหนดใหม่เพื่อให้สามารถทำงานตามที่ต้องการได้ วิธีการแก้ คือแก้ที่ *size control* (เป็นสัญญาณกำหนดจำนวนบิตที่จะเซตบนบิตเวกเตอร์) โดยที่ฟังก์ชันการทำงานอื่นยังเหมือนเดิม ถ้าเราต้องการเซตบิตจำนวน 101_2 บิต เราควรจะแบ่งการเซตบิตออกเป็น 2 ทรีย่อยคือเซตจำนวน 100_2 บิตและและเซตจำนวน 001_2 บิต โดยเราจะปรับกฎให้ 2 ทรีย่อยสามารถทำงานไปพร้อมๆ กันได้ โดยพิจารณาทรีย่อย 100_2 ก่อน จะกำหนดให้สัญญาณเอาท์พุท *route* มีค่าเป็นลอจิก '1'

ตารางที่ 4.3 ค่าตารางความจริงของโหนดเมื่อเซต n บิต

flip input	*route* input	*size control*	*address control*	*flip* output (left)	*route* output (left)	*flip* output (right)	*route* output (right)
1	x	x	x	1	x	1	x
0	0	x	x	0	0	0	0
0	1	0	0	0	1	0	0
0	1	0	1	0	0	0	1
0	1	1	0	1	x	0	1
0	1	1	1	0	0	1	x

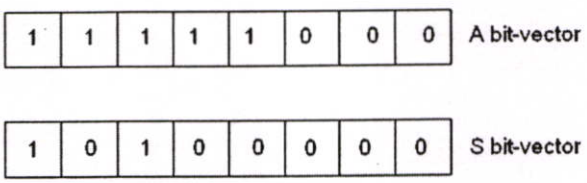
ตารางที่ 4.3 แสดงค่าตารางความจริงของโหนดหลังจากผ่านการปรับปรุงแล้ว จากนั้นจะนำฟังก์ชันการทำงานของโหนดไปสร้างเป็นฮาร์ดแวร์ CBT ที่สามารถเซตบิตตามจำนวนบิตที่ต้องการได้ รูปที่ 4.20 แสดงตัวอย่างฮาร์ดแวร์ CBT เซตบิตบนบิตเวกเตอร์จำนวน 5 บิต เริ่มจากแอดเดรส 0 ถึง 4



รูปที่ 4.20 แสดงตัวอย่างของการเซตบิต n = 5 บิต

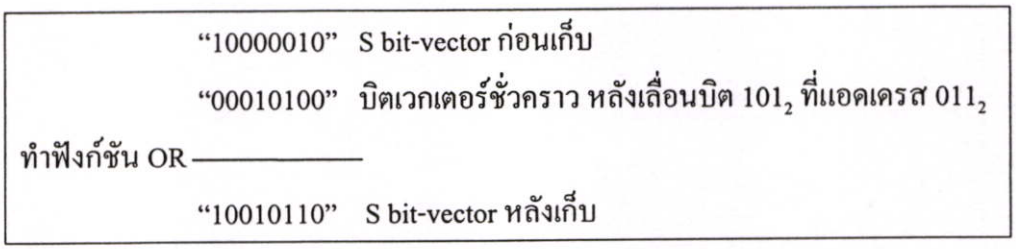
4.2.3 ฮาร์ดแวร์ B-unit

ตามฟังก์ชันการทำงาน (ง) ในหัวข้อ 4.2 เมื่อ CBT ทำการจองบล็อกพื้นที่สำหรับวัตถุใหม่และเก็บสถานะการจองในบิตเวกเตอร์ A bit-vector แล้ว จะต้องมีการเก็บขนาดของบล็อกที่ได้ทำการจองไว้ด้วย ซึ่งทำโดยฮาร์ดแวร์ B-nit โดยการทำงานนั้นจะเก็บขนาดของบล็อกลงบนบิต-เวกเตอร์ที่เรียกว่า S bit-vector ในตำแหน่งที่สัมพันธ์กับบิตเวกเตอร์ A bit-vector ตัวอย่างดังรูปที่ 4.21แสดงการเก็บขนาดของวัตถุบน S bit-vector ที่แอดเดรส 000_2 ซึ่งสัมพันธ์กับแอดเดรสเริ่มต้นของวัตถุขนาด 5 บล็อกบน A bit-vector ที่แอดเดรส 000_2



รูปที่ 4.21 แสดงตัวอย่างค่าขนาดของวัตถุบน S bit-vector

สำหรับวิธีการที่นำขนาดของวัตถุไปเก็บในบิตเวกเตอร์ จะใช้การเลื่อนบิตไปยังตำแหน่งที่ต้องการ ดังรูปที่ 4.22 สมมุติต้องการนำขนาดของวัตถุจำนวนบล็อกเท่ากับ 5 (101_2) ไปเก็บใน S bit-vector ที่มีค่าเดิมเท่ากับ 10000010_2 ในแอดเดรสที่ 3 วิธีการคือจะนำ 5 (101_2) เลื่อนใส่เข้าไปในในเวกเตอร์ชั่วคราว 00000000_2 ในแอดเดรสที่ 3 และหลังจากเลื่อนบิตแล้วค่าในเวกเตอร์ชั่วคราวที่ได้เท่ากับ 00010100_2 จากนั้นนำบิตเวกเตอร์ที่ได้มาทำการ or กับ C bit-vector จะได้เอาท์พุทเป็น 10010110_2



รูปที่ 4.22 แสดงตัวอย่างการเก็บขนาดวัตถุบน S bit-vector

4.2.4 ฮาร์ดแวร์ Set-count

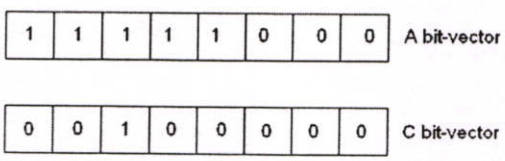
เนื่องจากในงานวิจัยนี้ใช้วิธีการจัดการหน่วยความจำที่ไม่ได้ใช้งานด้วยวิธี Reference counting ซึ่งต้องการฟิลด์ที่เรียกว่า ตัวนับการอ้างอิง สำหรับไว้นับจำนวนการอ้างอิงที่วัตถุอื่นอ้างอิงถึงตัวเอง โดยวัตถุที่อยู่ในฮีปจะต้องมีตัวนับอ้างอิงเป็นของตัวเอง และทุกครั้งที่วัตถุใหม่ถูกสร้าง

จะมีการกำหนดให้ตัวนับการอ้างอิงมีค่าเริ่มต้นเป็นหนึ่ง ดังนั้นในหัวข้อนี้จะพูดถึงการสร้างฮาร์ดแวร์สำหรับกำหนดค่าตัวนับการอ้างอิงเริ่มต้น จากการศึกษาวัดจำนวนตัวนับการอ้างอิงที่ใช้ต่อวัตถุใน KVM [18] พบว่ามากกว่า 99 เปอร์เซ็นต์มีจำนวนตัวนับการอ้างอิงเป็น 6 แสดงดังตารางที่ 4.4 ดังนั้นจำนวนบิตที่ใช้ในการจัดการตัวนับการการอ้างอิงใช้เพียง 3 บิตก็เพียงพอ

ตารางที่ 4.4 จำนวนเปอร์เซนต์ของ ตัวนับการอ้างอิงในวัตถุ

Application	Reference Count					
	1	2	3	4	5	6
Calculator	0.3	64.89	88.75	99.96	99.99	99.99
Dragon	32.01	48.28	57.98	59.12	63.19	99.79
Kvideo	10.44	42.64	65.64	93.62	98.92	99.57
Manyballs	55.56	55.66	55.75	83.54	83.57	99.98
Missiles	48.68	58.65	68.91	92.66	96.18	98.24
Scheduler	2.59	50.29	90.07	95.78	99.50	99.99
StarCruiser	30.01	61.63	93.88	99.97	99.99	99.99

ทุกครั้งที่มีการจองพื้นที่สำหรับวัตถุใหม่ฮาร์ดแวร์ CBT ก็จะทำการเก็บสถานะการจองบนบิตเวกเตอร์ A bit-vector และพร้อมกับต้องมีการกำหนดตัวนับการอ้างอิงเริ่มต้น โดยจะเก็บบนบิต-เวกเตอร์ C bit-vector ในตำแหน่งที่สัมพันธ์กับบิตเวกเตอร์ A bit-vector ตัวอย่างดังรูปที่ 4.23 แสดงค่า ตัวนับการอ้างอิง เริ่มต้นบน C-bit vector ที่แอดเดรส 000_2 ซึ่งสัมพันธ์กับแอดเดรสเริ่มต้นของวัตถุขนาด 5 บล็อกบน A bit-vector ที่แอดเดรส 000_2



รูปที่ 4.23 แสดงตัวอย่างค่าตัวนับการอ้างอิงเริ่มต้นบน C bit-vector

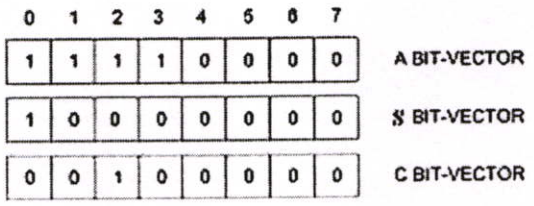
สำหรับวิธีการที่นำตัวนับการอ้างอิงไปเก็บในบิตเวกเตอร์ จะใช้การเลื่อนบิตไปยังตำแหน่งที่ต้องการ ดังรูปที่ 4.24 สมมุติต้องการนำค่า ตัวนับการอ้างอิง เท่ากับ 5 (101_2) ไปเก็บใน C bit-vector ที่มีค่าเดิมเท่ากับ 11100011_2 วิธีการคือจะนำ 5 (101_2) เลื่อนใส่เข้าไปในในเวกเตอร์ชั่วคราว 00000000_2 ในแอดเดรสที่ 3 ก่อนและหลังจากเลื่อนบิตแล้วค่าในเวกเตอร์ชั่วคราวที่ได้เท่า

กับ 00010100_2 จากนั้นนำบิตเวกเตอร์ที่ได้มามาร์ก (mark) กับ C bit-vector จะได้เอาท์พุทเป็น 11110111_2

	"11100011" C bit-vector ก่อนเก็บ
	"00010100" บิตเวกเตอร์ชั่วคราว หลังเลื่อนบิต 101_2 ที่แอดเดรส 011_2
ทำฟังก์ชัน XOR	————————
	"11110111" C bit-vector หลังเก็บ

รูปที่ 4.24 แสดงการเก็บตัวนับการอ้างอิงบนบิตเวกเตอร์

และในรูปที่ 4.25 แสดงข้อมูลบนบิตเวกเตอร์ทั้งสามหลังจากมีการจองพื้นที่สำหรับวัตถุใหม่ที่แอดเดรสศูนย์ และขนาดของวัตถุที่ต้องการจองขนาด 4 บล็อก



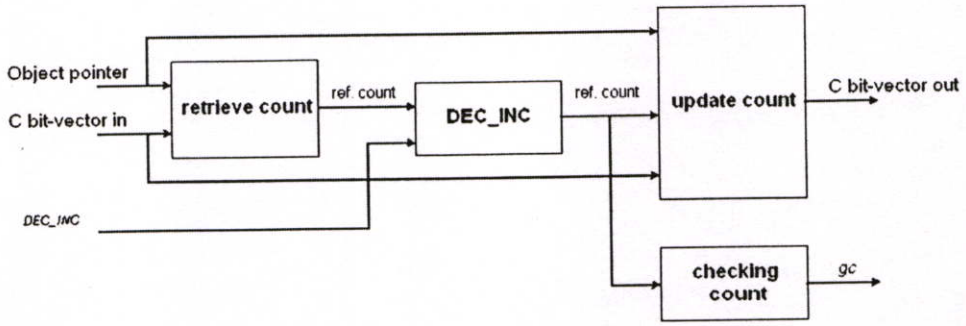
รูปที่ 4.25 แสดงการเก็บข้อมูลบนบิตเวกเตอร์

จากการจองพื้นที่ด้วยฮาร์ดแวร์ระบบไบนารีบัดดีและบิตแมบนี้จะสามารถช่วยลดการเกิด internal fragmentation ได้ เพราะระบบจะจองพื้นที่ตามจำนวนบล็อกที่ต้องการจริงๆ ถึงแม้จะจองพื้นที่เป็นขนาดกำลังของสอง เช่นต้องการ 5 บล็อกระบบก็จะหาบล็อกที่ว่างถึง 8 บล็อกซึ่งจองจริงๆเพียง 5 บล็อกเท่านั้น ส่วนที่เหลือ 3 บล็อกก็จะนำไปรวมเป็นบล็อกว่างที่สามารถใช้งานได้อีกในอนาคต

4.3 การออกแบบฮาร์ดแวร์การปรับค่าและตรวจสอบตัวนับอ้างอิง (Adjust&checking)

เมื่อมีการเปลี่ยนตัวชี้ระหว่างวัตถุ ซึ่งอาจจะเป็นการลดค่าตัวนับการอ้างอิงหรือเป็นการเพิ่มค่า [2][19] ฮาร์ดแวร์สำหรับการปรับค่าและตรวจสอบตัวนับการอ้างอิง จะใช้ สัญญาณ *DEC_INC* เป็นตัวกำหนดว่าเมื่อไรจะเพิ่มและเมื่อไรจะลดค่า ดังแสดงในรูปที่ 4.26 ซึ่งถ้า *DEC_INC* เป็นลอจิก '0' เป็นการเพิ่มค่า และถ้า *DEC_INC* เป็นลอจิก '1' เป็นการลดค่า พร้อมกับส่งแอดเดรสของวัตถุมาที่ขาสัญญาณ object_pointer และเมื่อทำการปรับค่าแล้วก็จะตรวจสอบตัวนับการอ้างอิงว่าเป็น

ศูนย์ด้วยฮาร์ดแวร์ checking count ถ้าตัวนับการอ้างอิงมีค่าเป็นศูนย์ แสดงว่าต้องทำการคืนพื้นที่หน่วยความ และจะสร้างสัญญาณ *gc* ส่งให้ฮาร์ดแวร์สำหรับคืนพื้นที่ทำงานต่อไป



รูปที่ 4.26 โครงสร้างการทำงานของฮาร์ดแวร์การปรับค่าและตรวจสอบตัวนับการอ้างอิง

ฮาร์ดแวร์การปรับค่าและตรวจสอบตัวนับการอ้างอิง ภายในประกอบด้วย 4 ฮาร์ดแวร์ย่อย ดังนี้

- Retrieve count สำหรับอ่านค่าตัวนับการอ้างอิงจากบิตเวกเตอร์ C bit-vector จะใช้หลักการเลื่อนกลุ่มบิตของตัวนับการอ้างอิงที่ต้องการไปทางซ้ายสุด โดยให้อยู่ในตำแหน่งบิตที่ 0 ถึงบิตที่ 2 จากนั้นจึงนำกลุ่มบิตไปใช้งาน
- DEC_INC สำหรับทำการปรับค่าตัวนับการอ้างอิง ตามสัญญาณอินพุท *DEC_INC* ถ้า DEC_INC มีค่าเป็นลอจิก '0' ให้เป็นการเพิ่มค่า และมีค่าเป็นลอจิก '1' เป็นการลดค่าโดย โค้ดภาษาวีเอชดีแอลที่ใช้สร้างฮาร์ดแวร์ DEC_INC แสดงดังรูปที่ 4.27

```
process(DEC_INC,Ref_counter)
begin
    if DEC_INC = '0' then
    counter <= Ref_counter + 1;
    else
    counter <= Ref_counter - 1;
    end if;
end process;
```

รูปที่ 4.27 แสดงโค้ดภาษาวีเอชดีแอลที่ใช้สร้างฮาร์ดแวร์ DEC_INC

- Update count สำหรับทำการอัพเดตตัวนับการอ้างอิงที่ได้ทำการปรับค่าแล้วลงบน C bit-vector อีกครั้ง ใช้หลักการเลื่อนบิตไปทางขวา ซึ่งการทำงานจะตรงข้ามกับฮาร์ดแวร์ Retrieve count
- Checking count สำหรับการตรวจสอบค่าตัวนับการอ้างอิงว่ามีค่าเป็นศูนย์หรือไม่ ถ้ามีค่าเป็นศูนย์ต้องทำการคืนพื้นที่หน่วยความจำที่ได้จองไว้ โดยการสร้างสัญญาณ *gc* ส่งไปยังฮาร์ดแวร์สำหรับคืนพื้นที่ต่อไป โดยการตรวจสอบค่าตัวนับการอ้างอิงจะใช้วงจรเปรียบเทียบกับค่าศูนย์ และโค้ดภาษาวีเอชดีแอลที่ใช้สร้างฮาร์ดแวร์ Checking count แสดงดังรูป 4.28

```
comp:   process (a)
                    begin
                        if a = "000" then
                           z <= '1';
                        else z <= '0';
                        end if;
        end process comp;
```

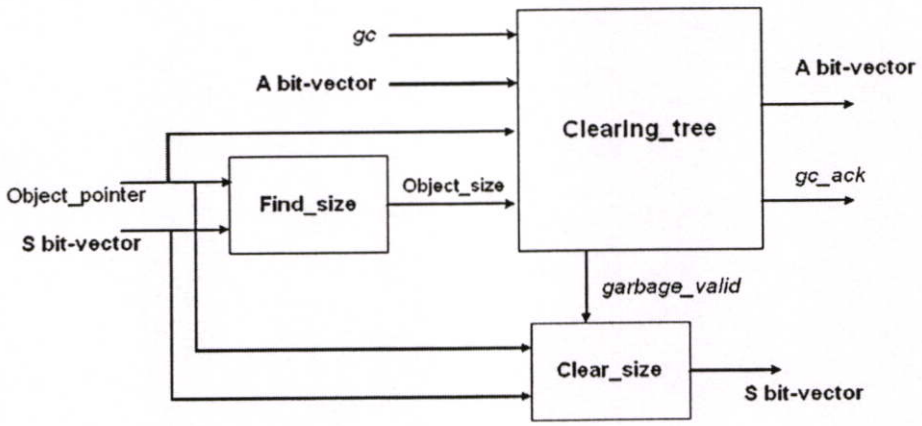
รูปที่ 4.28 แสดงโค้ดภาษาวีเอชดีแอลที่ใช้สร้างฮาร์ดแวร์ Checking_count

กระบวนการปรับค่าตัวนับอ้างอิงทำโดยวิธีการทางฮาร์ดแวร์สามารถลดค่าใช้จ่าย และเวลาในการจัดการได้ และแต่ละวัตถุก็ใช้เพียง 3 บิตเท่านั้นในการเก็บ ทำให้ไม่ต้องใช้พื้นที่สำหรับจัดเก็บตัวนับอ้างอิงมาก นอกจากนี้กระบวนการปรับค่าและตรวจสอบตัวนับอ้างอิงยังไม่ต้องใช้รอบไซเคิลการทำงานของหน่วยประมาณกลาง เพราะสามารถทำงานได้เองด้วยวงจรรวม (combinational logic) ทำให้ความเร็วในการทำงานเพิ่มขึ้นและประสิทธิภาพของระบบดีขึ้น

4.4 การออกแบบฮาร์ดแวร์การคืนพื้นที่ (Reclamation)

เมื่อฮาร์ดแวร์ปรับค่าและตรวจสอบตัวนับอ้างอิง ตรวจสอบค่าตัวนับอ้างอิงแล้วมีค่าเป็นศูนย์ก็จะส่งสัญญาณ *gc* ไปยังฮาร์ดแวร์สำหรับการคืนพื้นที่ โดยโครงสร้างฮาร์ดแวร์ แสดงดังรูปที่ 4.29 แนวคิดของการสร้างฮาร์ดแวร์มาจากฮาร์ดแวร์การจองพื้นที่หน่วยความจำในหัวข้อ 4.2 โดยการทำงานจะเป็นตรงข้าม คือเป็นการเคลียร์บิตเวกเตอร์ในตำแหน่งที่ต้องการ นั่นคือตำแหน่งที่วัตถุที่ต้องการคืนค่าจองใช้งานอยู่นั่นเอง

การทำงานเริ่มจากค้นหาขนาดของวัตถุที่ต้องการคืนค่าที่เก็บอยู่ในบิตเวกเตอร์ S bit-vector และนำขนาดวัตถุที่ได้ส่งไปยังฮาร์ดแวร์ Clearing_tree เพื่อทำการเคลียร์บิตใน A bit-vector พร้อมกับทำการเคลียร์ค่าขนาดของวัตถุที่เก็บอยู่ในบิตเวกเตอร์ S bit-vector ในตำแหน่งเดิม



รูปที่ 4.29 โครงสร้างการทำงานของฮาร์ดแวร์การคืนพื้นที่

4.4.1 ฮาร์ดแวร์ Find_size

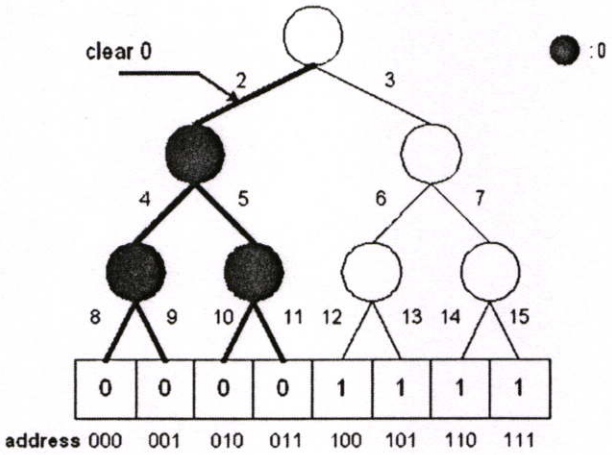
สำหรับอ่านค่าขนาดของวัตถุจากบิตเวกเตอร์ S bit-vector จะใช้หลักการเลื่อนกลุ่มบิตของขนาดวัตถุที่ต้องการไปทางซ้ายสุด โดยให้อยู่ในตำแหน่งบิตที่ 0 ถึงบิตที่ 2 จากนั้นจึงนำกลุ่มบิตไปใช้งาน

4.4.2 ฮาร์ดแวร์ Clearing_tree

ฮาร์ดแวร์ Clearing_tree แนวความคิดมาจากฮาร์ดแวร์ Bit-Flipper ในหัวข้อ 4.2.2 ซึ่งเอาท์พุทที่ได้จาก Bit-Flipper จะเป็นการเซตบิตลงบนเวกเตอร์ ขณะที่ฮาร์ดแวร์ Clearing_tree จะทำการเคลียร์บิตบนเวกเตอร์ (A bit-vector) ตามขนาดของวัตถุและตำแหน่งแอดเดรสของวัตถุที่ถูกคืนพื้นที่ ดังนั้นในหัวข้อนี้จะกล่าวถึงการสร้างฮาร์ดแวร์ Clearing_tree ทำหน้าที่เปลี่ยนบิตในบิต-เวกเตอร์ จากลอจิก '1' เป็นลอจิก '0' ผ่านวงจรรวม โดยอินพุทของ Clearing_tree ประกอบด้วยแอดเดรสเริ่มต้นและจำนวนของบิตที่ต้องการเคลียร์ค่าและในการเคลียร์บิตจะทำเฉพาะบิตที่ต้องการเคลียร์เท่านั้น

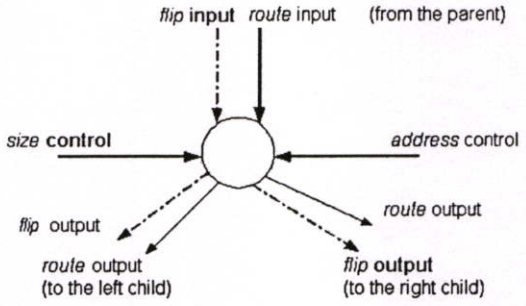
การทำงานของฮาร์ดแวร์ Clearing_tree จะเป็นการแพร่กระจายข้อมูลจากโหนดรากไปยังใบของ ทรี เหมือนฮาร์ดแวร์ bit-flipper สำหรับตัวอย่างในรูปที่ 4.30 เป็นการเคลียร์บิตบนบิตเวกเตอร์ขนาด 8 บิต และ CBT มีขนาด 7 โหนด โดยในแต่ละโหนดจะมี 1 อินพุท และ 2 เอาท์พุท และมีการ

กระจายสัญญาณจากโหนดแม่ไปยังโหนดลูกทางด้านซ้ายและทางด้านขวาตามคำสั่งที่ให้ทำ เช่นถ้าเราต้องการเคลียร์พื้นที่ขนาด 100_2 บล็อกโดยเริ่มต้นที่แอดเดรส 000_2 โดยจะแพร่กระจาย 0 จากโหนดลูกทางด้านซ้ายของโหนดรากไปยังใบ (บิตเวกเตอร์) โดยเส้นสีดำเข้มในรูปแสดงเส้นทางการแพร่กระจายข้อมูล



รูปที่ 4.30 แสดงตัวอย่างการใช้ CBT เคลียร์บิตบนบิตเวกเตอร์

ในการออกแบบโหนดให้มีความฉลาดในการแพร่กระจายข้อมูลนั้น จะเลียนแบบโหนดในรูปที่ 4.18 แต่การทำงานของแต่ละสัญญาณจะตรงกันข้าม นั่นคือสัญญาณ *flip* ใช้เคลียร์บิตทุกบิตในทรีย่อย เมื่อมีค่าเป็นลอจิก '0' และถ้ามีค่าเป็นลอจิก '1' ก็จะไปพิจารณาสัญญาณ *route* ซึ่งใช้เคลียร์บิตบางบิตในทรีย่อย เมื่อมีค่าเป็นลอจิก '0' และถ้าทั้งสัญญาณ *flip* และ *route* ไม่มีสัญญาณใดเป็นลอจิก '0' ก็จะไม่มีการเคลียร์บิตในทรีย่อย ไดอะแกรมของโหนดแสดงดังรูปที่ 4.31



รูปที่ 4.31 แสดงอินพุทและเอาท์พุทของโหนดในฮาร์ดแวร์ Clearing_tree

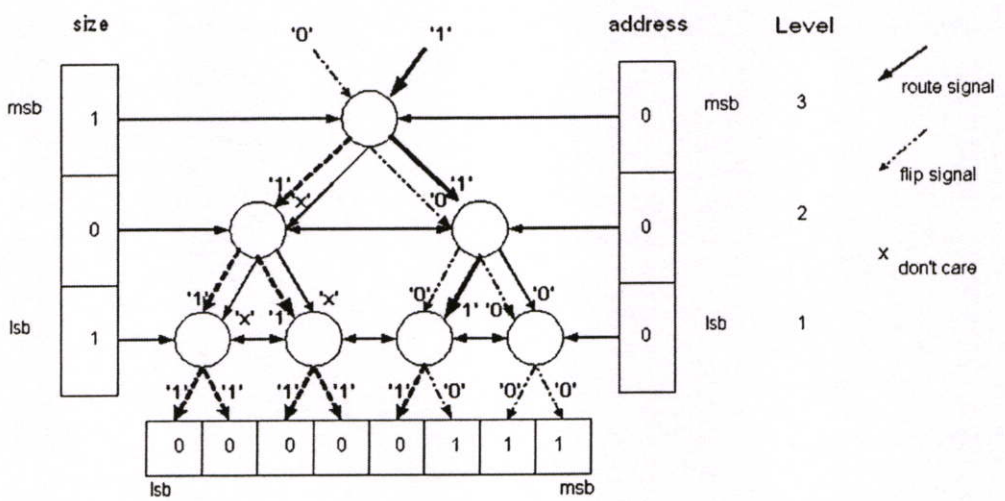
ในการกำหนดการทำงานของโหนด จะให้สัญญาณอินพุท *flip* มีลำดับความสำคัญสูงสุด โดยเมื่อ *flip* = '0' จะมีการแพร่กระจายข้อมูลไปยังใบของทรีย่อยๆ นั่นคือจะมีการเคลียร์ตามลำดับของทรี

ย่อยไปจนกระทั่งถึงใบของทรี ขณะที่สัญญาณอินพุท *route* จะกำหนดให้มีความสำคัญลำดับถัดไป โดยเมื่อ *route* = '1' เอาท์พุททั้ง 4 จะมีค่าเป็นลอจิก '1' และถ้า *route* = '0' ทรีย่อยอาจจะเคลียร์หรือไม่เคลียร์ ทั้งนี้ขึ้นอยู่กับสัญญาณ *address control* และสัญญาณ *size control* โดยถ้า *size control* = '1' แล้วเอาท์พุท *flip* ของโหนดจะมีค่าเป็นลอจิก '0' นั่นคือจะเริ่มมีการกระจายสัญญาณ *flip* ไปยังทรีย่อย สำหรับการเคลียร์บิตตามจำนวนบิตที่ต้องการ เช่นต้องการเคลียร์ 5 บิต ก็จะใช้หลักการเดียวกันในหัวข้อ 4.2.2 และตารางความจริงของสัญญาณทั้งหมดของโหนด แสดงดังตารางที่ 4.5

ตารางที่ 4.5 ค่าตารางความจริงของโหนดในฮาร์ดแวร์คืนพื้นที่

flip input	*route* input	*size control*	*address control*	*flip* output (left)	*route* output (left)	*flip* output (right)	*route* output (right)
0	x	x	x	0	x	0	x
1	1	x	x	1	1	1	1
1	0	0	0	1	0	1	1
1	0	0	1	1	1	1	0
1	0	1	0	0	x	1	0
1	0	1	1	1	1	0	x

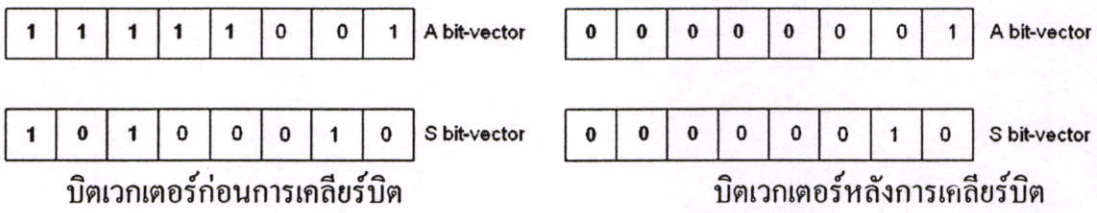
และรูปที่ 4.32 แสดงตัวอย่างฮาร์ดแวร์ CBT เคลียร์บิตบนบิตเวกเตอร์จำนวน 5 บิต โดยเริ่มจากแอดเดรส 0 ถึงแอดเดรส 4



รูปที่ 4.32 แสดงตัวอย่างของการเคลียร์บิต n = 5 บิต

4.4.3 ฮาร์ดแวร์ Clear_size

นอกจากนี้ฮาร์ดแวร์คืนพื้นที่ยังต้องทำการเคลียร์ค่าขนาดของวัตถุที่เก็บใน S bit-vector อีกด้วย ซึ่งทำโดยฮาร์ดแวร์ Clear_size โดยจะทำการเคลียร์ขนาดของวัตถุบนบิตเวกเตอร์ S bit-vector ในตำแหน่งที่สัมพันธ์กับบิตเวกเตอร์ A bit-vector ตัวอย่างดังรูปที่ 4.33 เมื่อมีการเคลียร์ค่าใน A bit-vector เริ่มต้นแอดเดรส 000_2จำนวน 5 บล็อกจะต้องทำการเคลียร์ขนาดของวัตถุที่แอดเดรสแดรส 000_2ใน S bit-vector ด้วย



รูปที่ 4.33 แสดงตัวอย่างการเคลียร์บิตบน S bit-vector

สำหรับวิธีการเคลียร์ขนาดของวัตถุในบิตเวกเตอร์ S bit-vector จะใช้การเลื่อนบิตไปยังตำแหน่งที่ต้องการ ดังรูปที่ 4.34 สมมุติต้องการขนาดวัตถุที่แอดเดรส 000_2 ใน S bit-vector ที่มีค่าเดิมเท่ากับ 10100010_2 วิธีการคือจะนำ 000_2 เลื่อนใส่เข้าไปในในเวกเตอร์ชั่วคราว 11111111_2 ในแอดเดรส 000_2 และหลังจากเลื่อนบิตแล้วค่าในเวกเตอร์ชั่วคราวที่ได้เท่ากับ 00011111_2 จากนั้นนำบิตเวกเตอร์ที่ได้มา AND กับ S bit-vector จะได้เอาท์พุทเป็น 00000010_2

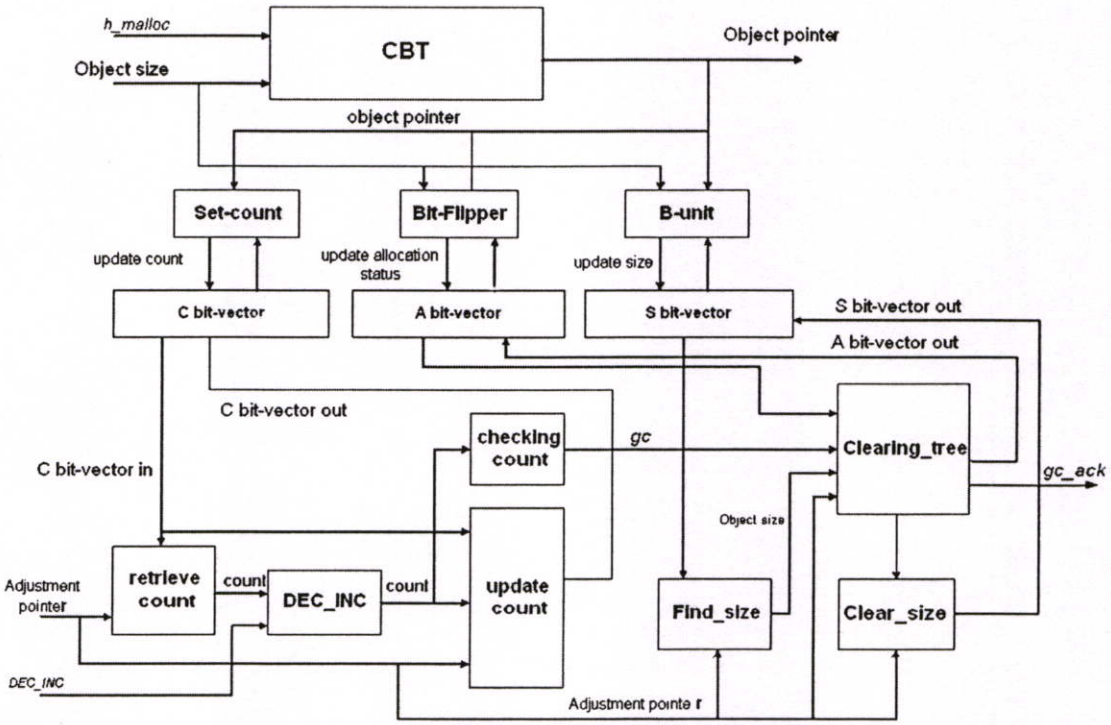
"10100010" S bit-vector ก่อนเคลียร์บิต
"00011111" บิตเวกเตอร์ชั่วคราว หลังเลื่อนบิต 000_2 ที่แอดเดรส 000_2
ทำฟังก์ชัน AND ————————
"00000010" S bit-vector หลังเคลียร์บิต

รูปที่ 4.34 แสดงตัวอย่างการเคลียร์ค่าบน S bit-vector

จากการทำการคืนพื้นที่โดยใช้วิธีการฮาร์ดแวร์นี้ ทำให้ไม่ต้องใช้วงรอบการทำงานของหน่วยประมาณกลางเลย เพราะสามารถทำงานได้เองอัตโนมัติขึ้นอยู่กับค่าตัวนับอ้างอิงว่ามีค่าศูนย์หรือไม่ ดังนั้นจึงทำให้ประสิทธิภาพของระบบดีขึ้นกว่าวิธีซอฟต์แวร์

4.5 โครงสร้างฮาร์ดแวร์ RCGC แบบสมบูรณ์

หลังจากได้สร้างฮาร์ดแวร์ในแต่ละฟังก์ชันการทำงานอย่างละเอียด ในหัวข้อนี้เป็นการนำเอาฮาร์ดแวร์ย่อยที่ได้สร้างไว้นำมารวมเข้าด้วยกันเป็นฮาร์ดแวร์ RCGC แบบสมบูรณ์ แสดงดังรูปที่ 4.35



รูปที่ 4.35 แสดงโครงสร้างฮาร์ดแวร์ RCGC แบบสมบูรณ์

บทที่ 5

การทดสอบ RCGC

ในบทนี้กล่าวถึงการทดสอบฮาร์ดแวร์ RCGC ซึ่งประกอบด้วย การทดสอบการทำงานในด้านความเร็ว การทดสอบหาความล่าช้าของการแพร่กระจายวงจร (propagation delay) การทดสอบค่าใช้จ่ายด้านฮาร์ดแวร์ (estimation hardware cost) และการทดสอบวัดประสิทธิภาพที่เพิ่มขึ้น (potential performance gain) การวิเคราะห์ทรัพยากรที่ใช้ไปในเชิงคณิตศาสตร์ และสรุปผลการทดสอบ

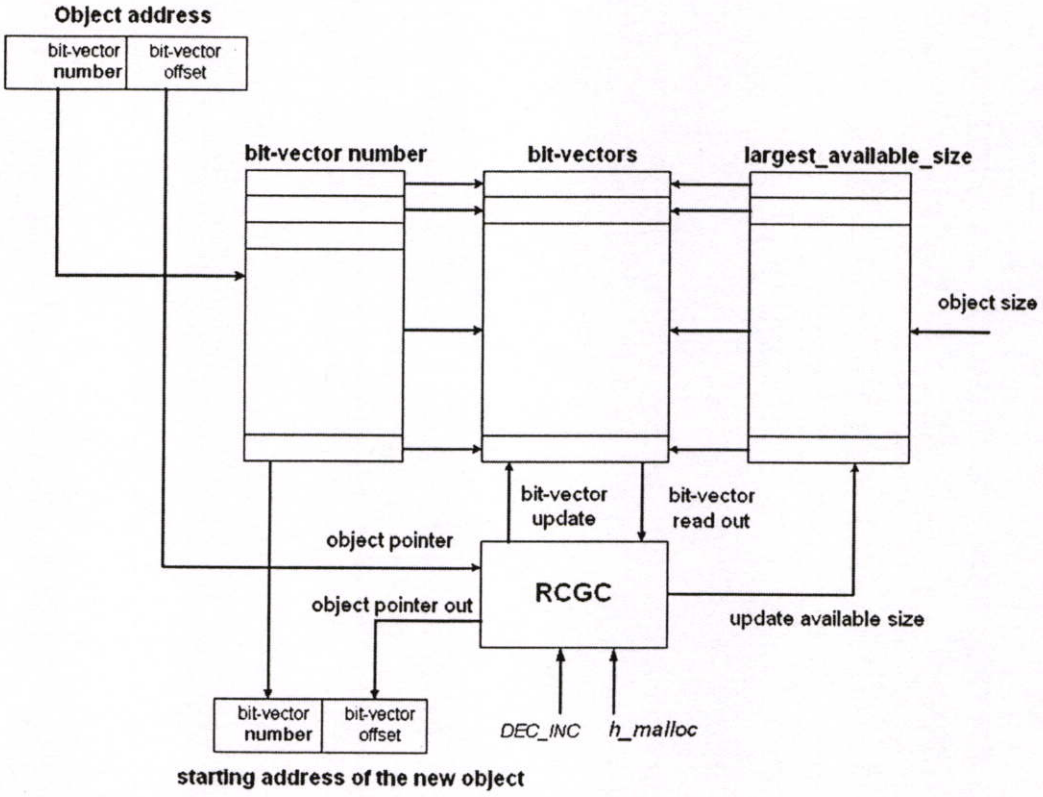
5.1 การทดสอบความเร็ว

เนื่องจากฮาร์ดแวร์ RCGC ที่ได้สร้างในบทที่ 4 ทำงานด้วยวงจรรวม แต่การเก็บสถานะต่างๆจะเก็บลงบนบิตเวกเตอร์ ซึ่งประกอบด้วย 3 บิตเวกเตอร์ได้แก่ A bit-vector S bit-vector และ C bit-vector ดังนั้นก่อนทดสอบการทำงานในด้านความเร็วจึงจำเป็นต้องสร้างฮาร์ดแวร์เพิ่มเติม สำหรับเก็บบิตเวกเตอร์ทั้ง 3 และเพื่อความง่ายในการจัดการจะนำทั้ง 3 บิตเวกเตอร์มารวมกันเป็นเวกเตอร์เดียวแล้วเก็บในหน่วยความจำที่เรียกว่า เซกเมนต์ (Segment) และเมื่อต้องการอ่านบิตเวกเตอร์ไปใช้งานจึงทำการแยกออกเป็น 3 บิตเวกเตอร์ในภายหลัง และเนื่องจากขนาดความซับซ้อนของฮาร์ดแวร์ CBT เป็น O(n) เมื่อ n เป็นจำนวนบิตเวกเตอร์ ถ้าแทนหน่วยความจำทั้งหมดในบิตเวกเตอร์เดียวจะทำให้ฮาร์ดแวร์ CBT มีขนาดใหญ่มาก ดังนั้นต้องทำการแบ่งบิตเวกเตอร์ออกเป็นเวกเตอร์ย่อยๆแล้วเก็บลงในเซกเมนต์ วิธีการนี้จะคล้ายๆ กับแนวคิดของการใช้ TLB (Translation Look-aside Buffer) ในหน่วยความจำเสมือน และนอกจากเซกเมนต์ที่ใช้เก็บบิตเวกเตอร์แล้ว ยังต้องมีอีก 2 เซกเมนต์ สำหรับเก็บจำนวน บล็อกว่างในแต่ละบิตเวกเตอร์ โดยเซกเมนต์ทั้งหมดมีดังนี้

- **Bit-vector number** เป็นเซกเมนต์เก็บแอดเดรสของบิตเวกเตอร์ที่เก็บอยู่ในเซก-เมนต์ Bit-vectors
- **Bit-vectors** เป็นเซกเมนต์เก็บบิตเวกเตอร์ย่อย
- **Largest_available_size** เป็นเซกเมนต์เก็บจำนวนบล็อกว่างที่สามารถจองใช้งานได้ในแต่ละบิตเวกเตอร์ที่เก็บอยู่ในเซกเมนต์ bit-vectors

โดยในรูปที่ 5.1 แสดงโครงสร้างฮาร์ดแวร์สำหรับทดสอบ RCGC ที่ประกอบด้วยเซกเมนต์ทั้งสาม และในการทดสอบการทำงาน กำหนดให้แต่ละบิตเวกเตอร์ที่เก็บในเซกเมนต์ Bit-vectors มีขนาด 192 บิต ประกอบด้วย A bit-vector จำนวน 64 บิต S bit-vector จำนวน 64 บิต และ C bit-vector จำนวน 64 บิต และกำหนดให้ขนาดบล็อกละ 16 ไบต์ ดังนั้นถ้าต้องการทดสอบกับหน่วยความจำขนาด 64KB จะต้องกำหนดขนาดเซกเมนต์ทั้ง 3 ดังนี้

- **Bit-vector number** ขนาด 64 X 6 บิต
- **Bit-vectors** ขนาด 64 x 192 บิต
- **Largest_available_size** ขนาด 64 x 6 บิต

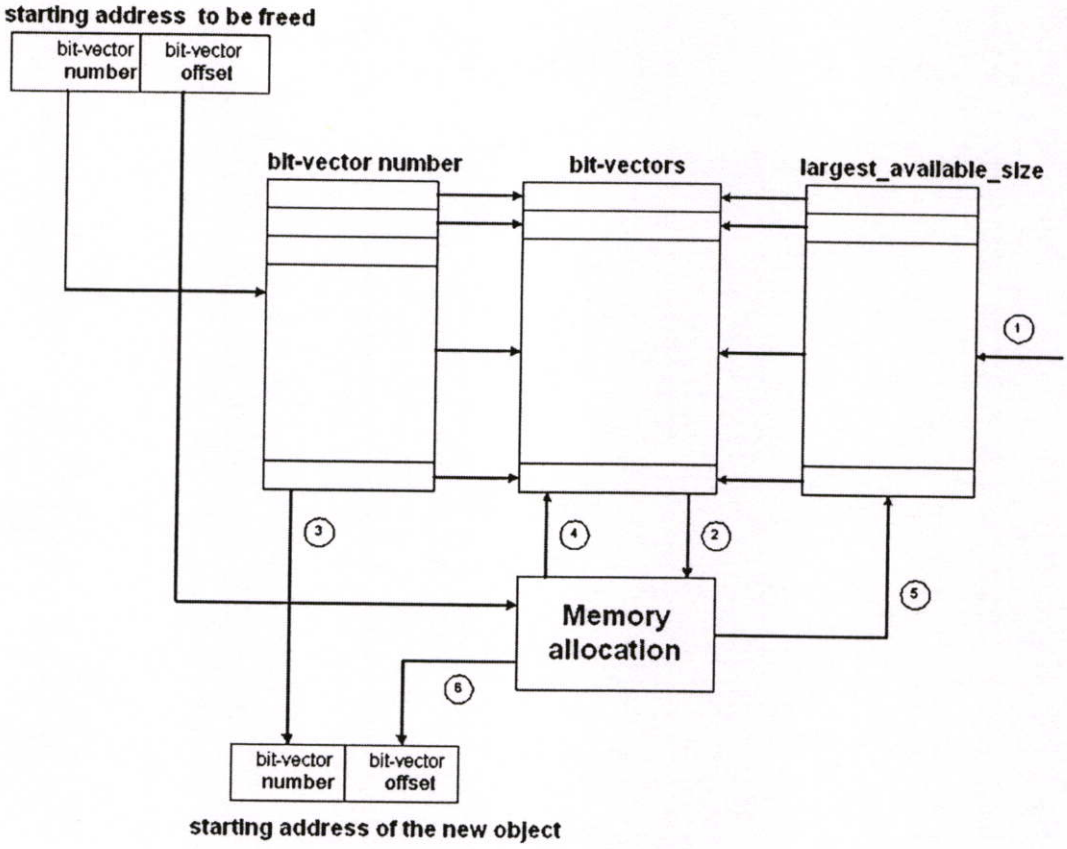


รูปที่ 5.1 แสดงโครงสร้างฮาร์ดแวร์สำหรับทดสอบ RCGC

การทดสอบการทำงานเพื่อความง่ายในการนำผลไปเปรียบเทียบกับวิธีการทางซอฟต์แวร์จึงทำการแบ่งออกเป็น 3 ฮาร์ดแวร์ตามฟังก์ชันการทำงานได้แก่ ฮาร์ดแวร์ทดสอบการจองพื้นที่หน่วยความจำ ฮาร์ดแวร์ทดสอบการปรับค่าและทดสอบตัวนับการอ้างอิง และฮาร์ดแวร์ทดสอบการคืนพื้นที่

5.1.1 ฮาร์ดแวร์ทดสอบการจองพื้นที่หน่วยความจำ

เป็นฮาร์ดแวร์ที่ใช้ทดสอบฟังก์ชันการจองพื้นที่หน่วยความจำและการกำหนดค่าตัวนับอ้างอิงเริ่มต้น โดยฮาร์ดแวร์ที่ใช้ทดสอบ แสดงดังรูปที่ 5.2



รูปที่ 5.2 ฮาร์ดแวร์ทดสอบการจองพื้นที่หน่วยความจำ

ขั้นตอนการจองพื้นที่หน่วยความจำ แสดงดังรูปที่ 5.3

ขั้นตอนการจองพื้นที่หน่วยความจำ	
1	Allocation request
2	Bit-vector read out
3	Bit-vector number read out
4	Bit-vector update
5	Update largest_available_size
6	Offset address of newly creates object

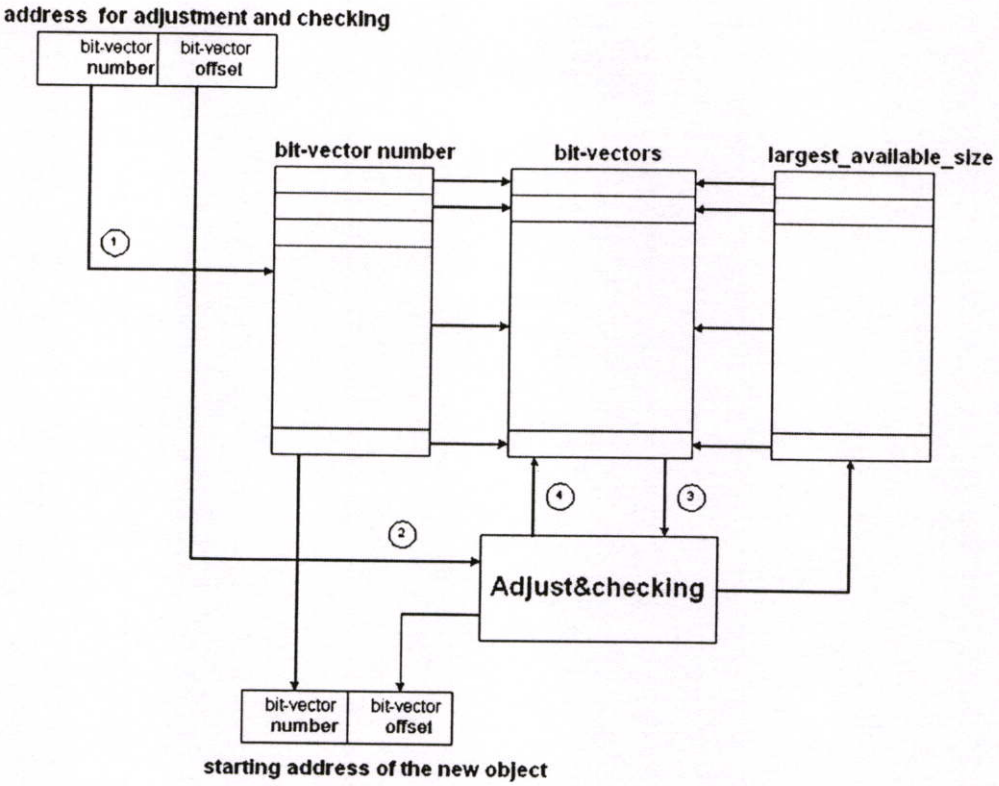
รูปที่ 5.3 แสดงขั้นตอนการทำงานของฮาร์ดแวร์ทดสอบการจองพื้นที่หน่วยความจำ

จากรูป 5.3 อธิบายการทำงานดังนี้

1) ขั้นตอน **Allocation request** เมื่อมีการร้องขอการจองพื้นที่หน่วยความจำ ขนาดของวัตถุที่ต้องการจองจะเปรียบเทียบกับขนาดของวัตถุที่เก็บอยู่ในเซกเมนต์ Largest_available_size โดยเปรียบเทียบทุกข้อมูลในแบบขนานพร้อมกันไป ซึ่งการทำงานแบบนี้จะคล้ายๆ กับการเปรียบเทียบ tag ใน fully associated cache โดยบิตเวกเตอร์ที่มีจำนวนบล็อกว่างมากกว่าหรือเท่ากับจำนวนบล็อกที่ต้องการจะถูกอ่าน
2) ขั้นตอน **Bit-vector read out** นำบิตเวกเตอร์ที่อ่านได้ส่งไปยังฮาร์ดแวร์ Memory allocation เพื่อทำการจองพื้นที่หน่วยความจำสำหรับวัตถุใหม่บนบิตเวกเตอร์ และหลังจากฮาร์ดแวร์ Memory allocation ทำการเซตสถานะการจอง เก็บขนาดของวัตถุ และกำหนดตัวนับการอ้างอิงเริ่มต้นสำหรับวัตถุใหม่บนบิตเวกเตอร์แล้ว ก็ทำการอัพเดตบิตเวกตอร์โดยนำไปเก็บไว้ในเซกเมนต์ bit-vectorsในตำแหน่งเดิม
3) ขั้นตอน **Bit-vector number read out** อ่านแอดเดรสของบิตเวกเตอร์ที่เก็บอยู่ในเซกเมนต์ Bit-vectors
4) ขั้นตอน **Bit-vector update** ทำการอัพเดตจำนวนบล็อกที่ว่างที่สามารถใช้งานได้ในเซก-เมนต์ Largest_available_size ในตำแหน่งเดิม
5) ขั้นตอน **Update largest_available_size** ทำการอัพเดตจำนวนบล็อกที่ว่างที่สามารถใช้งานได้ในเซกเมนต์ largest_available_size ที่ตำแหน่งเดิม
6) ขั้นตอน **Offset address of newly creates object** นำแอดเดรสที่ได้จากการจองโดยฮาร์ดแวร์ Memory allocation รวมกับแอดเดรสที่ได้ในขั้นตอนที่ 3 จะทำให้ได้แอดเดรสสำหรับวัตถุใหม่ออกมา

5.1.2 ฮาร์ดแวร์ทดสอบการปรับค่าและตรวจสอบตัวนับอ้างอิง

เป็นฮาร์ดแวร์ที่ใช้ทดสอบการปรับค่าและตรวจสอบตัวนับการอ้างอิง โดยฮาร์ดแวร์ที่ใช้ทดสอบ แสดงดังรูปที่ 5.4



รูปที่ 5.4 ฮาร์ดแวร์ทดสอบการปรับค่าและตรวจสอบตัวนับอ้างอิง

ขั้นตอนการจองพื้นที่หน่วยความจำ แสดงดังรูปที่ 5.5

ขั้นตอนการปรับค่าและตรวจสอบตัวนับอ้างอิง	
1	Select bit-vector
2	Starting bit-vector-offset address
3	Bit-vector read out
4	Bit-vector update

รูปที่ 5.5 แสดงขั้นตอนการทำงานของฮาร์ดแวร์ทดสอบการปรับค่าและตรวจสอบตัวนับอ้างอิง

จากรูป 5.5 อธิบายการทำงานดังนี้

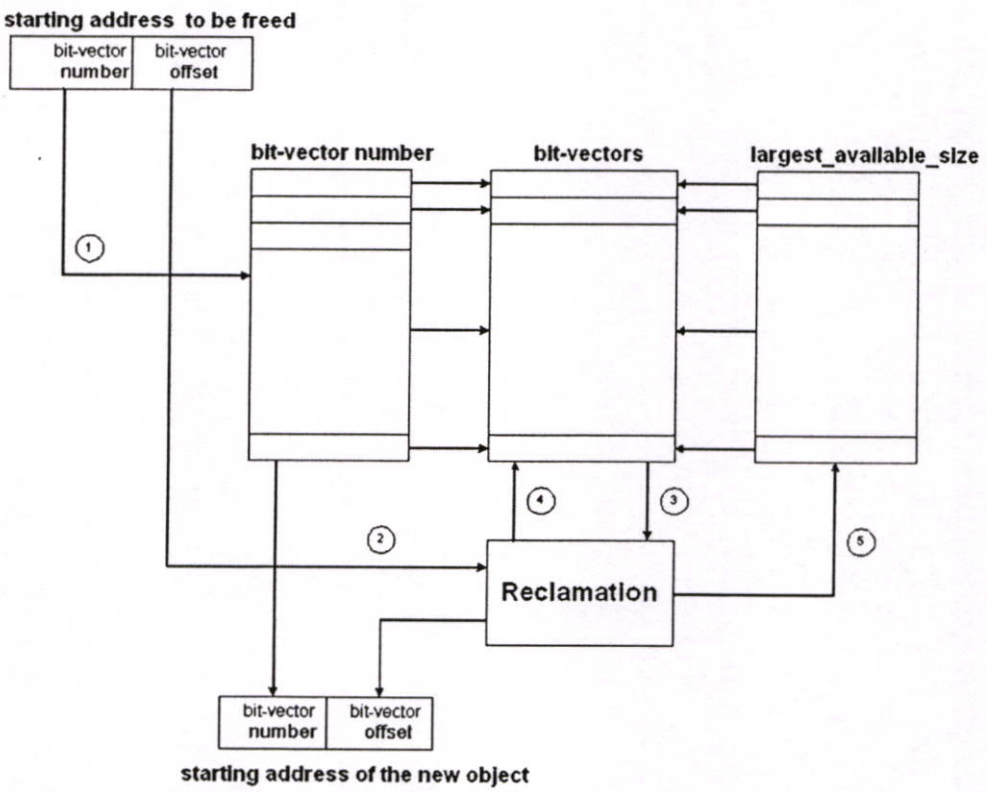
1) ขั้นตอน **Select bit-vector** เมื่อมีการเปลี่ยนตัวชี้ระหว่าง 2 วัตถุ จะต้องมีการอัพเดตตัวนับอ้างอิง โดยแอดเดรสของวัตถุจะทำการแบ่งออกเป็น 2 ส่วน ส่วนแรกเป็นตัวชี้แอดเดรส

ภายในเซกเมนต์ bit-vector number เพื่อหาแอดเดรสของบิตเวกเตอร์ที่เก็บอยู่ในเซกเมนต์ bit-vectors

2) ขั้นตอน **Starting bit-vector-offset address** แอดเดรสส่วนที่ 2 จากขั้นตอนที่ 1 เป็นแอดเดรสที่ส่งเข้าไปในฮาร์ดแวร์ Adjust&checking เพื่อทำการอ่านค่าตัวนับอ้างอิงในบิตเวกเตอร์
3) ขั้นตอน **Bit-vector read out** ทำการอ่านบิตเวกเตอร์ C bit-vector จากเซกเมนต์ bit-vectors เพื่อส่งให้ฮาร์ดแวร์ Adjust&checking ทำการปรับค่าตัวนับการอ้างอิงและถ้าตัวนับการอ้างอิงมีค่าเป็นศูนย์ก็จะสร้างสัญญาณ *gc* ส่งให้ฮาร์ดแวร์ Reclamation ทำงานต่อไป
4) ขั้นตอน **Bit-vector update** ทำการอัพเดตบิตเวกเตอร์โดยนำไปเก็บไว้ในเซกเมนต์ bit-vectorsในตำแหน่งเดิม

5.1.3 ฮาร์ดแวร์ทดสอบการคืนพื้นที่

เป็นฮาร์ดแวร์ที่ใช้ทดสอบการปรับค่าและตรวจสอบตัวนับการอ้างอิง โดยฮาร์ดแวร์ที่ใช้ทดสอบ แสดงดังรูปที่ 5.6



รูปที่ 5.6 ฮาร์ดแวร์ทดสอบการคืนพื้นที่

ขั้นตอนการจองพื้นที่หน่วยความจำ แสดงดังรูปที่ 5.5

ขั้นตอนการคืนพื้นที่	
1	Select bit-vector
2	Starting bit-vector-offset address
3	Bit-vector read out
4	Bit-vector update
5	Update largest_available_size

รูปที่ 5.7 แสดงขั้นตอนการทำงานของฮาร์ดแวร์สำหรับทดสอบการคืนพื้นที่

จากรูป 5.7 อธิบายการทำงานดังนี้

1) ขั้นตอน **Select bit-vector** เมื่อมีสัญญาณ *gc* มาจากฮาร์ดแวร์ Adjust&checking จะต้องทำการคืนพื้นที่ที่วัตถุจองใช้งานโดยการเคลียร์บิตใน A bit-vector และเคลียร์บิตใน S bit-vector ซึ่งแอดเดรสของวัตถุจะทำการแบ่งออกเป็น 2 ส่วน ส่วนแรกเป็นตัวชี้ แอดเดรสภายในเซกเมนต์ bit-vector number เพื่อหาแอดเดรสของบิตเวกเตอร์ที่เก็บอยู่ในเซกเมนต์ bit-vectors
2) ขั้นตอน **Starting bit-vector-offset address** เป็นแอดเดรสส่วนที่ 2 จากขั้นตอนที่ 1 ส่งเข้าไปในฮาร์ดแวร์ Reclamation เพื่อทำการอ่านค่าสถานะการจองและขนาดของวัตถุในบิตเวกเตอร์
3) ขั้นตอน **Bit-vector read out** ทำการอ่านบิตเวกเตอร์ A bit-vector และ S bit-vector จากเซกเมนต์ bit-vectors เพื่อส่งให้ฮาร์ดแวร์ Reclamation ทำการเคลียร์บิต
4) ขั้นตอน **Bit-vector update** ทำการอัพเดตบิตเวกตอร์โดยนำไปเก็บไว้ในเซกเมนต์ bit-vectors ในตำแหน่งเดิม
5) ขั้นตอน **Update largest_available_size** ทำการอัพเดตจำนวนบล็อกที่ว่างที่สามารถใช้งานได้ในเซกเมนต์ Largest_available_size ในตำแหน่งเดิม

5.1.4 ผลการทดสอบ

หลังจากสร้างฮาร์ดแวร์สำหรับการทดสอบความเร็วของทั้ง 3 ฟังก์ชันของฮาร์ดแวร์ RCGC แล้ว จะต้องเขียนโค้ด VHDL เพิ่มในส่วนที่เพิ่มเติมฮาร์ดแวร์เข้าไป จากนั้นนำมาจำลองการทำงานใช้โปรแกรม Model Sim SE-EE 5.4E จากบริษัท MTI และผลที่ได้จะเป็นจำนวน clock ที่

ใช้ไป ดังนั้นการวัดความเร็วจึงใช้การนับ clock ที่เกิดขึ้นในแต่ละฮาร์ดแวร์ โดยผลการจำลองแสดงดังตารางที่ 5.1

ตารางที่ 5.1 แสดงผลการจำลองการทำงาน

ฟังก์ชัน	จำนวน clock
การจองพื้นที่	14
การปรับค่าและตรวจสอบตัวนับอ้างอิง	5
การคืนพื้นที่	7

5.2 การทดสอบความล่าช้าของการแพร่กระจายวงจร

ในหัวข้อนี้เป็นการทดสอบความล่าช้าของการแพร่กระจายวงจร (propagation delay) ของฮาร์ดแวร์ RCGC โดยใช้วิธีการศึกษาความล่าช้าของการแพร่กระจายภายในวงจร และในการศึกษานี้สมมุติว่าใช้ Very High-Speed CMOS (VHCMOS) เป็นโครงสร้างของระบบ โดย VHCMOS มีความล่าช้าของการแพร่กระจายวงจรที่ 5.2 ns และกินพลังงาน 0.025 มิลลิวัตต์ [20]

สมมุติว่าฮาร์ดแวร์ RCGC ใช้หน่วยความจำขนาด 128 KB โดยกำหนดแต่ละบล็อกมีขนาด 16 ไบต์และความยาวของบิตเวกเตอร์ที่ได้เป็น 8,192 บิต สามารถจัดให้อยู่ในรูปไบนารีได้ทั้งหมด 13 ระดับ ดังนั้นเวลาหน่วงที่เกิดในไบนารีทรีคือ 135.2 ns (2 เกต/โหนด) ฉะนั้นอัตราสัญญาณนาฬิกาของระบบควรจะกำหนดอยู่ที่ 135.2 ns หรือ 7.39 MHz โดยในตารางที่ 5.2 แสดงจำนวน clock ที่ใช้ในการทำการจองพื้นที่หน่วยความจำ การปรับค่าและตรวจสอบตัวนับการอ้างอิง และการคืนพื้นที่วัตถุที่ไม่ได้ใช้งาน

ตารางที่ 5.2 แสดงเวลาหน่วงการแพร่กระจายวงจรของ RCGC

ฟังก์ชัน	จำนวน clock (สมมุติที่ 7.39 MHz)
การจองพื้นที่	14
การปรับค่าและตรวจสอบตัวนับอ้างอิง	5
การคืนพื้นที่	7

5.3 การทดสอบค่าใช้จ่ายด้านฮาร์ดแวร์

สำหรับการทดสอบในด้านค่าใช้จ่ายของฮาร์ดแวร์ที่เกิดขึ้น จะนำฮาร์ดแวร์ RCGC ที่ได้จากบทที่ 4 ไปทำการสังเคาะห์วงจร และจากนั้นจึงนำผลที่ได้มาประเมินค่าใช้จ่ายต่างๆในด้านฮาร์ดแวร์

5.3.1 การสังเคราะห์วงจร

การสังเคราะห์วงจรจะใช้โปรแกรม Xilinx Foundation series 3.1i จากบริษัท Xilinx ทำการสังเคราะห์เพื่อดูค่าความเร็ว จำนวนเกตที่ใช้ และความล่าช้าของวงจรที่เกิดขึ้น โดยในวิทยานิพนธ์นี้ได้เลือกใช้เทคโนโลยีของ Virtex-V800BG560 ซึ่งผลการสังเคราะห์วงจรแสดงตารางที่ 6.1 โดยผลที่ได้เป็นการแสดงประสิทธิภาพที่ได้จากฮาร์ดแวร์ RCGC ประกอบด้วยจำนวน Slice จำนวนเกตที่ใช้ และความล่าช้าของวงจร (delay) ที่เกิดขึ้นเพื่อดูความเร็วภายในวงจรโดยแบ่งพิจารณาตามขนาดของบิตเวกเตอร์ (8, 16, 32, 64) และองค์ประกอบย่อยของ Memory allocation Adjust&checking และ Reclamation และตารางที่ 5.3 แสดงผลของการสังเคราะห์ฮาร์ดแวร์ในแต่ละองค์ประกอบย่อยบน Virtex-V800BG560

ตารางที่ 5.3 แสดงผลลัพธ์ขององค์ประกอบย่อยที่ได้จากการสร้างบน Virtex-V800BG560

Component	Size (bit)	Num. Slice	Gate Count	Max Net Delay (ns)	Max Delay (ns)
Memory Allocation	8	42	501	5.774	21.884
	16	77	908	7.662	31.704
	32	163	1,933	8.098	42.583
	64	312	3,708	8.233	57.504
Adjust& Checking	8	36	437	3.549	17.655
	16	74	887	7.688	30.944
	32	150	1,787	8.497	34.982
	64	323	3,857	12.938	39.753
Reclamation	8	26	300	3.356	20.037
	16	63	738	8.549	31.91
	32	139	1,668	9.618	32.326
	64	287	3,414	13.165	43.771

และตารางที่ 5.4 แสดงผลของการสังเคราะห์ฮาร์ดแวร์ RCGC บน Virtex-V800BG560

ตารางที่ 5.4 แสดงผลลัพธ์ของฮาร์ดแวร์ที่ได้จากการสร้างบน Virtex-V800BG560

	Size 8	Size 16	Size 32	Size 64
Number of Slice	112(1%)	241(2%)	506(5%)	987(10%)
Gate count	1,322	2,875	6,048	11,747
Max Clock Freguency (Mhz)	105.742	103.008	101.061	95.483
Max Net Delay (ns)	9.606	9.975	10.197	10.473

จากผลการสังเคราะห์วงจรจะเห็นว่าฮาร์ดแวร์ที่ได้ออกแบบสามารถทำงานได้เร็วด้วยเวลาที่ลดลงเล็กน้อยแม้จำนวนบิตเวกเตอร์จะเพิ่มมากขึ้นก็ตาม

5.3.2 คำนวณค่าใช้จ่ายด้านฮาร์ดแวร์

สำหรับค่าใช้จ่ายด้านฮาร์ดแวร์จะมีผลต่อประสิทธิภาพการทำงานของ RCGC ซึ่งเป็นหนึ่งในปัจจัยหลักที่สามารถกำหนดความน่าเชื่อถือของ RCGC ได้ สมมุติว่า RCGC ใช้ หน่วยความจำขนาด 128 KB โดยกำหนดแต่ละบล็อกมีขนาด 16 ไบต์ ดังนั้นจำนวนหน่วยความจำที่ต้องการสำหรับสร้างบิตเวกเตอร์ทั้งสาม เท่ากับ

$$\underset{\text{bit-vectors}}{\text{mem}} = \frac{3(128) \times 10^3}{16 \times 8} = 3\,\text{KB}$$

ดังนั้น overhead ของหน่วยความจำที่ใช้สร้างบิตเวกเตอร์ ที่ประกอบด้วย A-bit vector S-bit vector และ C -bit vector คิดเป็น 2.34%

สำหรับค่าใช้จ่ายด้านฮาร์ดแวร์ที่ใช้ในการสร้างวงจรจะพิจารณาจากจำนวนเกตที่ใช้ในแต่ละฮาร์ดแวร์ย่อยจากตารางที่ 5.3 โดยสามารถสรุปได้ในตารางที่ 5.5

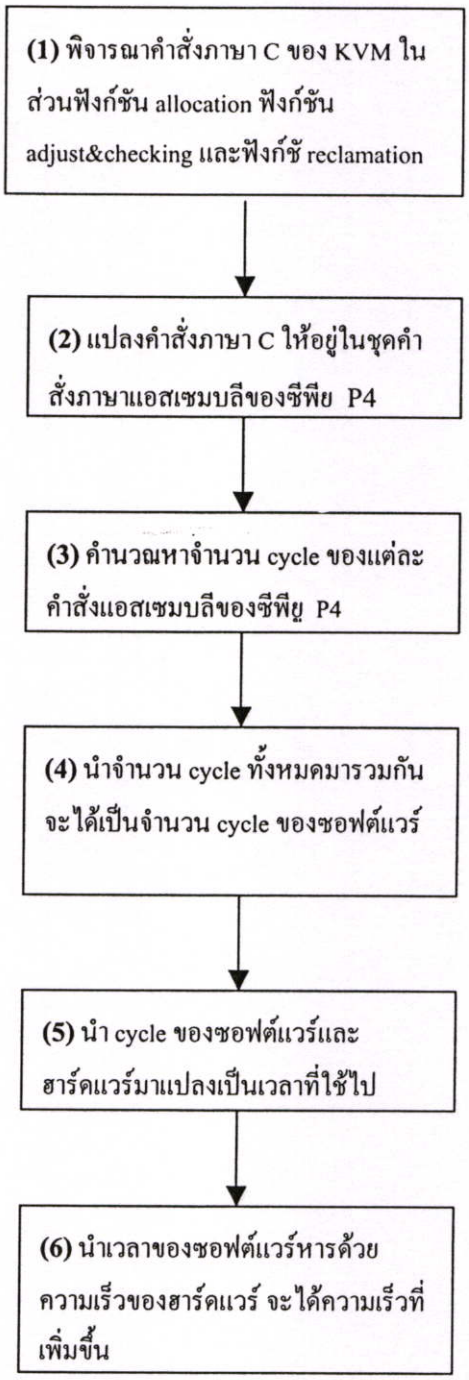
ตารางที่ 5.5 จำนวนเกตที่ใช้สร้างฮาร์ดแวร์ย่อยของ RCGC

ฟังก์ชัน	จำนวนเกตเมื่อ N=64 บิต
การจองพื้นที่หน่วยความจำ	3,708 เกต
การปรับค่าและตรวจสอบตัวนับอ้างอิง	3,857 เกต
การคืนพื้นที่	3,414 เกต
รวม	10,979 เกต

กำหนดให้ RCGC ใช้หน่วยความจำขนาด 1KB แต่ละบล็อกมีขนาด 16 ไบต์และความยาวของบิต-เวกเตอร์เป็น 64 บิตและ สมมุติใช้ DRAM เป็นหน่วยความจำของระบบ ซึ่งแต่ละบิตของ DRAM ใช้ทรานซิสเตอร์จำนวน 1 ตัว ดังนั้น DRAM จะใช้ทรานซิสเตอร์จำนวน 8,192 ตัว ขณะที่ฮาร์ดแวร์ RCGC ใช้ทรานซิสเตอร์จำนวน 21,958 ตัว (2 ทรานซิสเตอร์ต่อ 1 เกต) ดังนั้นจำนวนทรานซิสเตอร์ที่ ใช้ทั้งหมดเป็นจำนวน 30,150 ตัว (RCGC + DRAM) นั่นคือมีจำนวนทรานซิสเตอร์เพิ่มขึ้น 2.6 เท่าจาก DRAM

5.4 การทดสอบวัดประสิทธิภาพที่เพิ่มขึ้น

การวัดประสิทธิภาพการทำงานด้านความเร็วจะเทียบกับวิธีทางการซอฟต์แวร์ โดยเลือกใช้ KVM 1.0.4 [8] เป็นซอฟต์แวร์สำหรับการทดสอบ เพราะมีขนาดเล็ก ซึ่งเหมาะกับงานในสภาพแวดล้อมระบบฝังตัว แต่เนื่องจาก KVMใช้อัลกอริทึม mark-sweep ดังนั้นจึงต้องทำการแก้ไขโค้ดโปรแกรมและคอมไพล์ใหม่ให้อยู่ในอัลกอริทึม reference counting และแพลตฟอร์มที่เลือกใช้ทดสอบได้เลือกใช้ ซีพียูเพนเทียม 4 ทำงานที่ความ 2.0 GHz (เป็นซีพียูขนาด 32 บิตรุ่นสุดท้ายของอินเทล) เนื่องจาก KVM เป็นจาวาเวอร์ชวลแมชชีนขนาด 32 บิต โดยวิธีการเปรียบเทียบแสดงดังรูปที่ 5.8



รูปที่ 5.8 วิธีการเปรียบเทียบความเร็วระหว่างวิธีการทางซอฟต์แวร์และฮาร์ดแวร์

จากรูปที่ 5.8 การเปรียบเทียบความเร็วระหว่างวิธีการทางซอฟต์แวร์และฮาร์ดแวร์ มีขั้นตอนดังนี้

1) เนื่องจาก KVM พัฒนาด้วยภาษา C โดยในส่วนฟังก์ชันการจองไม่ต้องพัฒนาใหม่ แต่ฟังก์ชันการปรับค่าและตรวจสอบตัวนับอ้างอิง และฟังก์ชันการคืนพื้นที่จะต้องทำการแก้ไขให้อยู่ในอัลกอริทึม reference counting และคอมไพล์ใหม่

2) ทำการแปลงคำสั่งภาษา C ให้อยู่ในรูปแบบคำสั่งภาษาแอสเซมบลีเพนเทียม 4 ด้วยคอมไพล์ด้วยคอมไพเลอร์ ***gcc*** ด้วยคำสั่ง ***gcc – S –O2 – march=pentium4*** ซึ่งจะได้ไฟล์นามสกุล .S ออกมา ซึ่งภายในบรรจุโค้ดภาษาแอสเซมบลี โดยชุดคำสั่งที่คอมไพเลอร์สร้างออกมาจะเป็นชุดคำสั่ง x86 เพนเทียม 4 ขนาด 32 บิต โดยไวยกรณ์เป็นไปตามมาตรฐาน AT&T (เนื่องจากใช้คอมไพเลอร์ gcc) และต้องทำการเปลี่ยนไวยกรณ์ให้อยู่ในรูปแบบของ Intel

3) นำคำสั่งแอสเซมบลีใน clock ทั้งหมดที่ใช้ไปในแต่ละฟังก์ชันการทำงาน โดยผลที่ได้แสดงดังตารางที่ 5.6

ตารางที่ 5.6 แสดงจำนวน clock ที่ใช้ในแต่ละฟังก์ชันการทำงาน

ฟังก์ชัน	วิธีซอฟต์แวร์
การจองพื้นที่	102
การปรับค่าและตรวจสอบตัวนับอ้างอิง	54
การคืนพื้นที่	73

4) ทำการแปลงจากจำนวนสัญญาณนาฬิกาที่ใช้ไปเป็นหน่วยความเร็ว (ns) โดยพิจารณาทำงานที่ความถี่ 2.0 GHz โดยผลการเปรียบเทียบความเร็วระหว่างสองวิธี แสดงดังตารางที่ 5.7

ตารางที่ 5.7 แสดงการเปรียบเทียบความเร็วระหว่างวิธีฮาร์ดแวร์และวิธีซอฟต์แวร์

ฟังก์ชัน	วิธีฮาร์ดแวร์	วิธีซอฟต์แวร์
การจองพื้นที่	7 ns	51 ns
การปรับค่าและตรวจสอบตัวนับอ้างอิง	2.5 ns	27 ns
การคืนพื้นที่	3.5 ns	36.5 ns

5) ทำการหารความเร็วระหว่างวิธีซอฟต์แวร์และวิธีฮาร์ดแวร์จะได้ความเร็วที่เพิ่มขึ้นของฮาร์ดแวร์ที่นำเสนอในวิทยานิพนธ์นี้ ดังนี้

- ฮาร์ดแวร์สำหรับการจองพื้นที่ ความเร็วเพิ่มขึ้น **7.28** เท่า หรือเพิ่มขึ้น **86.27%**
- ฮาร์ดแวร์สำหรับการปรับค่าและตรวจสอบตัวนับอ้างอิง ความเร็วเพิ่มขึ้น เพิ่มขึ้น **10.8** เท่า หรือเพิ่มขึ้น **90.74%**
- ฮาร์ดแวร์สำหรับการคืนพื้นที่ ความเร็วเพิ่มขึ้น **10.42** เท่า หรือเพิ่มขึ้น **90.41%**

5.5 การวิเคราะห์ทรัพยากรที่ใช้ไปในเชิงคณิตศาสตร์

ในหัวข้อนี้เป็นการวิเคราะห์ทรัพยากรที่ใช้ไปในเชิงคณิตศาสตร์ โดยจะวิเคราะห์ในด้านความซับซ้อน (Complexity) ของทรัพยากรที่ใช้ไปในรูปของบิกโอ (Big O) แบ่งออกเป็น 2 กรณีได้แก่ กรณีแรกเป็นการวิเคราะห์ความซับซ้อนด้าน Space และกรณีที่สองเป็นการวิเคราะห์ ความซับซ้อนด้าน Time

5.5.1 การวิเคราะห์ความซับซ้อนด้าน space

การวิเคราะห์ความซับซ้อนด้าน space เป็นการวิเคราะห์ความซับซ้อนขนาดของฮาร์ดแวร์ที่ใช้ไปเมื่อบิตเวกเตอร์เพิ่มมากขึ้น จะพิจารณาจากจำนวนทรานซิสเตอร์ที่ใช้ไปเมื่อจำนวนบิตเวกเตอร์มากขึ้น การวิเคราะห์จะเปรียบเทียบระหว่างวิธีการทางฮาร์ดแวร์และวิธีการทางซอฟต์แวร์ โดยวิธีการทางฮาร์ดแวร์ความซับซ้อนที่ได้แสดงดังตารางที่ 5.8

ตารางที่ 5.8 แสดงความซับซ้อนด้าน space ของ RCGC

ฟังก์ชัน	ความซับซ้อน
การจองพื้นที่	N-1
การปรับค่าและตรวจสอบตัวนับอ้างอิง	N-1
การคืนพื้นที่	N-1

จากตารางที่ 5.8 จะแทนความซับซ้อนของทั้งสามฟังก์ชันในรูปแบบบิกโอ ดังนี้

จากสมการ $O(f1(n) + f2(n)) = O(\max\{g1(n), g2(n)\})$

ดังนั้น $O(RCGC) = O(\max\{N-1, N-1, N-1\})$

$= O(N)$

เมื่อ N คือจำนวนบิตเวกเตอร์

นั่นหมายความว่าเมื่อจำนวนบิตเวกเตอร์มากขึ้นฮาร์ดแวร์ที่ได้ก็จะมากขึ้นด้วยในแบบสมการลิเนียร์ (linear)

ขณะที่วิธีการทางซอฟต์แวร์ กำหนดให้หน่วยความจำขนาด 1 บล็อก เท่ากับ 16 ไบต์ และสมมุติให้ DRAM เป็นหน่วยความจำของระบบ ซึ่งแต่ละบิตของ DRAM ใช้ทรานซิสเตอร์จำนวน 1 ทรานซิสเตอร์ ดังนั้นเมื่อจำนวนบิตเวกเตอร์เพิ่มมากขึ้น (จำนวนบล็อกเพิ่มมากขึ้น) จะทำให้จำนวนทรานซิสเตอร์เพิ่มมากขึ้นในแบบสมการลิเนียร์ หรือแทนความซับซ้อนในรูปบิกโอได้เป็น O(N) เมื่อ N เป็นจำนวนบิตเวกเตอร์ หรือจำนวนบล็อกหน่วยความจำ

จากผลการวิเคราะห์ทั้งวิธีการทางฮาร์ดแวร์และวิธีการทางซอฟต์แวร์จะได้ความซับซ้อน O(N) เหมือนกัน แต่จากการวิเคราะห์ค่าจ่ายใช้จ่ายด้านฮาร์ดแวร์ในหัวข้อ 5.3.2 จะเห็นว่าวิธีการทางฮาร์ดแวร์จะมีค่าใช้จ่ายเพิ่มขึ้น 2.34% ดังนั้นสรุปได้ว่าวิธีการทางฮาร์ดแวร์มีความซับซ้อนด้าน space มากกว่าวิธีการซอฟต์แวร์

5.5.2 การวิเคราะห์ความซับซ้อนด้าน time

การวิเคราะห์ความซับซ้อนด้าน time เป็นการวิเคราะห์ความซับซ้อนที่มีผลต่อความเร็วในการทำงาน เมื่อจำนวนบิตเวกเตอร์เพิ่มมากขึ้นจะแบ่งออกเป็น 2 ส่วน คือการวิเคราะห์ความซับซ้อนของวิธีการทางฮาร์ดแวร์ที่มีผลต่อความเร็วในการทำงาน และการวิเคราะห์ความซับซ้อนของวิธีการทางซอฟต์แวร์ที่มีผลต่อความเร็วในการทำงานแล้วนำผลมาเปรียบเทียบกัน

เนื่องจากโครงสร้างฮาร์ดแวร์ของ RCGC เป็นไบนารีทรีแบบสมบูรณ์ (Complete binary tree) ดังนั้นความซับซ้อนด้าน time ของวิธีการทางฮาร์ดแวร์เป็น O(logN) เมื่อ N เป็นจำนวนบิตเวกเตอร์ ขณะที่ความซับซ้อนของวิธีการทางซอฟต์แวร์ แสดงดังตารางที่ 5.9

ตารางที่ 5.9 แสดงความซับซ้อนด้าน time ของวิธีการทางซอฟต์แวร์

ฟังก์ชัน	ความซับซ้อน
การจองพื้นที่	N^2+N
การปรับค่าและตรวจสอบตัวนับอ้างอิง	N
การคืนพื้นที่	$(1/2)N^2+(1/2)N$

จากตารางที่ 5.9 จะแทนความซับซ้อนของทั้งสามฟังก์ชันในรูปแบบบิกโอ ดังนี้

จากสมการ $O(f1(n) + f2(n)) = O(\max\{g1(n), g2(n)\})$

ดังนั้น $O(\text{วิธีการทางซอฟต์แวร์}) = O(\max\{ N^2+N, N, (1/2)N^2+(1/2)N\})$

$= O(N^2)$

เมื่อ N คือจำนวนบิตเวกเตอร์ หรือจำนวนบล็อกหน่วยความจำ

นั่นหมายความว่า เมื่อจำนวนบิตเวกเตอร์มากขึ้นอัตราการเติบโตของความซับซ้อนจะเป็นแบบกำลังสอง นั่นคือยิ่งซับซ้อนมากความเร็วในการทำงานก็จะช้าลง

จากผลการวิเคราะห์ทั้งวิธีการทางฮาร์ดแวร์และวิธีการซอฟต์แวร์จะเห็นว่าวิธีการทางฮาร์ดแวร์มีความซับซ้อนน้อยกว่า ดังนั้นสรุปได้ว่าวิธีการทางฮาร์ดแวร์ทำงานได้ความเร็วกว่าวิธีการทางซอฟต์แวร์

5.6 สรุปผลการทดสอบ

ผลจากการทดสอบฮาร์ดแวร์ RCGC สามารถสรุปได้ดังนี้ การสังเคราะห์วงจรฮาร์ดแวร์ที่ออกแบบได้สามารถทำงานได้เร็วด้วยเวลาที่ลดลงเล็กน้อย แม้จำนวนบิตเวกเตอร์เพิ่มมากขึ้นก็ตาม การทดสอบการทำงานในด้านความเร็ว ฟังก์ชันการจองพื้นที่ใช้สัญญาณนาฬิกา14 ลูก ฟังก์ชันการปรับค่าและตรวจสอบตัวนับอ้างอิงใช้สัญญาณนาฬิกา 5 ลูก และฟังก์ชันการคืนพื้นที่ใช้สัญญาณนาฬิกา 7 ลูก การทดสอบหาความล่าช้าของการแพร่กระจายวงจรอัตราสัญญาณนาฬิกาของระบบที่ได้จากการสังเคราะห์วงจรอยู่ที่ 135.2 ns หรือ7.39 MHz การทดสอบค่าใช้จ่ายด้านฮาร์ดแวร์ค่า overhead ของหน่วยความจำที่ใช้สร้างบิตเวกเตอร์ทั้งสาม คิดเป็น 2.34% ของหน่วยความทั้งหมด การทดสอบวัดประสิทธิภาพที่เพิ่มขึ้นของฮาร์ดแวร์ RCGC (เฉลี่ยทั้ง 3 ฮาร์ดแวร์) ความเร็วเพิ่มขึ้นประมาณ 89% และการวิเคราะห์ทรัพยากรที่ใช้ไปในเชิงคณิตศาสตร์ โดยวิธีการทางฮาร์ดแวร์มีความซับซ้อนด้าน space มากกว่าวิธีการทางซอฟต์แวร์ และวิธีการทางฮาร์ดแวร์มีความซับซ้อนด้าน time น้อยกว่าวิธีการทางซอฟต์แวร์ ดังนั้นวิธีการทางฮาร์ดแวร์จึงทำงานได้ความเร็วกว่า ที่เป็นเช่นนี้เพราะว่าโครงสร้างของวิธีการทางฮาร์ดแวร์เป็นแบบไบนารีทรีแบบสมบูรณ์

บทที่ 6
สรุปผลการวิจัย และข้อเสนอแนะ

วิทยานิพนธ์ฉบับนี้เป็นการศึกษาและปรับปรุงการทำงานของหน่วยจัดการหน่วยความจำของจาวาซึ่งประกอบด้วย 2 ส่วนคือส่วนการจองพื้นที่หน่วยความจำสำหรับวัตถุใหม่และส่วนการคืนพื้นที่วัตถุที่ไม่ได้ใช้งานโดยจะเน้นไปที่อัลกอริทึม reference counting เพราะเหมาะกับงานในระบบคอมพิวเตอร์ฝังตัว (embedded system) ซึ่งทำงานตามเวลาจริง โดยการศึกษานั้นจะศึกษาถึงการทำงานในส่วนของเดิมที่เป็นซอฟต์แวร์และปรับปรุงการทำงานให้ดีขึ้นด้วยวิธีการทางฮาร์ดแวร์

ดังนั้นในวิทยานิพนธ์นี้จึงนำเสนอวิธีการใหม่ซึ่งได้นำเสนอในครั้งแรก ด้วยวิธีการทางฮาร์ดแวร์สำหรับจัดการหน่วยความจำของจาวาโดยใช้วิธี reference counting เพื่อปรับปรุงประสิทธิภาพการทำงานของระบบให้ดีขึ้นโดยวิธีการทางฮาร์ดแวร์นี้จะใช้วิธีการระบบไบนารีบัดดีและเทคนิคบิตแมป จัดการด้วยวงจรรวม พัฒนาด้วยภาษาวีเอชดีแอล และการเปรียบเทียบความเร็วที่เพิ่มจะเทียบกับ KVM ซึ่งทำการแก้ไขและคอมไพล์ใหม่ในส่วนการคืนพื้นที่หน่วยความจำให้มีอัลกอริธึมเป็น reference counting และนำผลที่ได้เปรียบเทียบกันเพื่อชี้ให้เห็นว่าวิธีการทางฮาร์ดแวร์ทำงานได้เร็วกว่า

ผลจากการจำลองการทำงานและการประเมินประสิทธิภาพแสดงให้เห็นว่าวิธีการที่นำเสนอนี้สามารถเพิ่มประสิทธิภาพในด้านความเร็วที่เพิ่มขึ้นโดยเพิ่มขึ้น 89% ลดการเกิด internal fragmentation ลดค่าใช้จ่ายในการจัดการตัวนับอ้างอิงที่เป็นปัญหาจากวิธีการซอฟต์แวร์ได้ และในด้านการประเมินความซับซ้อนของขนาดฮาร์ดแวร์ที่ใช้ไป วิธีการทางฮาร์ดแวร์จะมากกว่าวิธีการทางซอฟต์แวร์เพราะว่ามี overhead เพิ่มขึ้นจากการจองพื้นที่หน่วยความจำสำหรับเก็บบิตเวกเตอร์ทั้ง 3 ขณะที่การประเมินความซับซ้อนด้าน time ที่มีผลต่อความเร็วในการทำงานวิธีการทางฮาร์ดแวร์มีความซับซ้อนน้อยกว่า ทำให้การทำงานทำได้เร็วกว่าวิธีการทางซอฟต์แวร์ที่เป็นเช่นนี้เพราะว่าโครงสร้างของฮาร์ดแวร์ RCGC เป็นแบบไบนารีทรีแบบสมบูรณ์ และปัญหาหนึ่งที่จำเป็นต้องมีการพัฒนาต่อไป คือการอ้างอิงกันเป็นวงกลมของสองวัตถุ ซึ่งถ้าสามารถแก้ได้โดยวิธีฮาร์ดแวร์จะทำให้ประสิทธิภาพของวิธี reference counting เพิ่มมากขึ้น

สำหรับแนวทางในการพัฒนาต่อ สามารถนำไปสร้างเป็นฮาร์ดแวร์สำหรับจัดการ garbage collection ในระบบคอมพิวเตอร์ฝังตัว หรือสร้างเป็นองค์ประกอบหนึ่งภายในจาวาโปรเซสเซอร์

บรรณานุกรม

[1] B. Zorn, "**Custo-malloac: efficient synthesized memory allocators**," Technical Report CU-CS-602-92, Computer Science Department, University of Colorado, July 1992.

[2] P. R. Wilson, "**Uniprocessor Garbage Collection Techniques**, "1992 SIGPLAN Intl. Workshop on Memory Management, pp. 1-42, Sept. 1992.

[3] F. Karabiber, A. SertbaS and A.H. Zaiam, "**Dynamic Memory Allocator Algorithms Simulation And Performance Analysis**," Journal of electrical & electronics engineering, Vol.5, No.2,1435-1441,March 2005.

[4] P. R. Wilson, M. S. Johnstone, M. Neely and D. Boles, "**Dynamic Storage Allocation: A Survey and Critical Review**," International Workshop on Memory Management, Kinross, Scotland, UK, September 1995.

[5] A. S. Tanenbaum, A. S. Woodhull, **Operating Systems Design and Implementation**, Prentice Hall, Portland, 0-13-638677-6, 1997.

[6] J. L. Peterson, T. A. Norman, "**Buddy systems**," Communication of the ACM, Vol. 20, 421-431, June 1977.

[7] J.M. Chang, E. F. Grhringer, " **A High Performance Memory Allocator for Object-Oriented-Systems**," IEEE Trans. Computers, vol. 45, no.3, pp. 357-366, March 1996.

[8] **CLDC and the K Virtual Machine (KVM)**. [Online] Available : http://java.sun.com/products/cldc/.

[9] T. Ritzau, "**Real-Time Reference Counting**", The Embedded System Show, London, May 24-25, 2000.

[10] A. Kim, J.M. Chang, "**Designing A Java Microprocessor Core Using FPGA Technology**", Proceedings of 1998 IEEE International ASIC Conference, Rochester, NY, Sep. 13-16, 1998

[11] S. K. Agun, J. M. Chang, "**Design of a Reusable Memory Management System**," Proceedings of the 14th IEEE International ASIC/SOC Conference, Washington, D.C., Sep. 12-15, 2001.

[12] J. M. Chang, W. Srisa-an, and J. M. Chang, "**An Introduction to DMMX (Dynamic Memory Management Extension)**, " Workshop notes of ICCD workshop on Hardware Support for Objects and Microarchitectures for Java, Austin, Texas, pp. 11-14, Oct. 10,

1999.

[13] W. Srisa-an, C. D. Lo and J. M. Chang, "**Scalable Hardware algorithm for Mark-sweep Garbage Collection**," Proceedings of Euromicro Conference on Digital System Design, pages 274-279, Maastricht, Netherlands, September 2000.

[14] B. Venners. **Java's Garbage Collected Heap**. [Online] Available : www.java-world.com.

[15] B. Venners, **Inside the Java Virtual Machine**, McGraw-Hill, 1999.

[16] S. Isoda, E. Goto, and I. Kimura, "**AnEfficient Bit Table Technique for Dynamic Storage Allocation of 2n Word Blocks**," Comm. ACM, vol 7, pp.589-592, Sept. 1971.

[17] E.V. Puttkamer, "**A Simple Hardware Buddy System Memory Allocator**," IEEE Trans. Computers, vol.24, no.10, pp.953-957, Oct. 1975.

[18] W. Srisa-an, C. D. Lo and J. M. Chang, "**Active Memory Processor: A Hardware Garbage Collector for Real-Time Java Embedded Devices**", IEEE Trans. Mobile Computing, vol 2. no. 2, pp. 89-92, Appil-June. 2003.

[19] R. Jones, R. Lins, **Garbage Collection: Algorithms for automatic Dynamic Memory Management**, John Wiley and Sons, 1996.

[20] W. H. Wolf, **Modern VLSI Designs: Systems on Silicon**, Prentice Hall, 1998.

ภาคผนวก

ภาคผนวก ก.

ผลงานวิจัยที่ได้รับการตีพิมพ์เผยแพร่

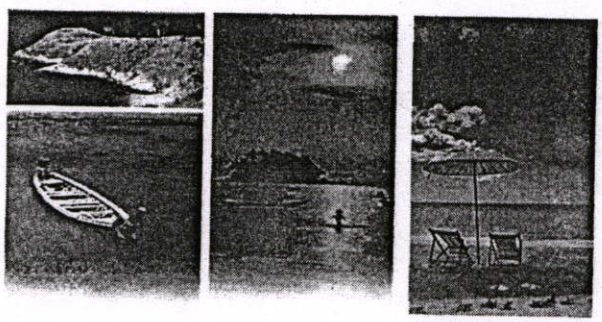
1. P. Bunyatnoparat, U. Ranok, "**Improvement of The Reference Counting Garbage Collection using Hardware Algorithm**," The 4th Information and Computer Engineering Postgraduate Workshop 2004 (ICEP 2004), pp. 138-142, Phuket, Thailand, January 22-23, 2004.

PROCEEDINGS

The 4th Information and Computer Engineering Postgraduate Workshop 2004

ICEP 2004
22nd - 23rd January 2004

Phuket, Thailand



Prof Jun Marai, KEIO University, Japan

Organised and Sponsored by
Department of Computer Engineering,
Faculty of Engineering,
Prince of Songkla University, Thailand
and Prof Jun Marai, KEIO University, Japan

In cooperation with
IEEE ComSoc, Thailand Chapter,
ECTI (Electrical Engineering/Electronics,
Computer, Telecommunications and Information
Technology Association of Thailand)

ISBN 974-644-518-9

Improvement of The Reference Counting Garbage Collection using Hardware Algorithm

P. Bunyatnoparat and U. Ranok

Department of Computer Engineering
King Mongkut's Institute of Technology Ladkrabang BKK 10520
Email: kbprathe@kmitl.ac.th, t0013282@hotmail.com

Abstract

Object-oriented programming, such as C++ and JAVA, normally uses heap memory. In order to manage heap memory, the system should allocate memory space for a new object and should free unused space automatically. Garbage collector is used to free memory space. For improving the system performance, VLSI technology is introduced to develop hardware garbage collector. This paper presents the implementation of Reference Counting Garbage Collection (RCGC) by using hardware algorithm based on VHDL. The binary buddy system and bit-map techniques are chosen to design the proposed RCGC because they are easy by implemented with combinational logic. The RCGC is composed of two hardware designs, objects address allocation and initialization of reference count part and garbage collection part. The result shows that the proposed RCGC can reduce cost for the reference count and also reduce the number of internal fragments.

1. Introduction

Object-oriented programming languages such as C++ and Java can provide high memory intensity in the heap memory. [1] shows that Java/C++ applications can spend from 23% to 38% of execution time performing dynamic memory management[1]. One of the ways to improve the system performance is using the dynamic memory management. Usually, dynamic memory management unit is divided into 2 main sections, a memory allocation unit and a garbage collection unit. Presently, garbage collections implemented by software approaches with basic algorithms including reference counting collector and mark-sweep collector [2]. In a reference counting system, each object has an associated count of the references (pointer). Each time a reference to the object is created, the value of the pointed to object's count is incremented. When an existing reference to an object is eliminated, the count is decremented. The memory occupied by an object may be reclaimed when the value of the object's count equals zero. In terms of abstract two-phase garbage collection, the adjustment and checking of reference counts are implemented in the first phase, and the reclamation phase occurs when reference counts hit zero. The advantages of reference counting is the most of its operation interleaved with the running program's own execution ,to response time as well, but the problems are internal fragmentation, overhead for management the reference count in each time. In a mark-sweep garbage collection are named for the two phases that implement the abstract garbage collection algorithm, one for distinguish the live objects from the garbage by tracing to mark the live object and two for reclaim the garbage. Once the live objects have been made distinguishable from the garbage objects, memory is swept, that is, exhaustively examined, to find all of the unmarked (garbage) objects and reclaim their space. One of the advantages is elimination of the reference cycle objects but the disadvantages is that the marking complexity depends on the number of active objects while the sweeping complexity depends on the overall number of objects. For improving the system performance, researcher attempts to make enhancement through the software approaches, but the system performance can not be highly improved. Over the several years, there have been several approaches to implement garbage collection function in the hardware. One of the approaches is proposed by [3] introduced a new hardware implementation for allocation the objects using binary buddy system and bit-map techniques. This hardware design takes advantage of the speed of a pure of combinational logic which can be done in constant-time.

This paper presents the design of Reference Counting Garbage Collection (KCGC) using hardware algorithm implemented by VHDL. The proposed scheme takes an advantage of the binary buddy system and bit-map techniques, because they are easy to form the base of binary tree and they can be implemented efficiently in combinational logic [3]. This scheme is a new hardware design and introduced in this paper in first time. We get the concept from the previous software approaches [2][10]. This hardware algorithm can improve the system performance from software approaches [2][5][6][7][10] in

case of internal fragmentation, cost of reference count management and size of memory heap.

The remainder of this paper is organized as follows. Section 2 provides the detailed memory allocation, memory block based on binary buddy system. Section 3 shows the architecture of RCGC. Section 4 shows the implementation hardware algorithm. Section 5 shows the results of our design. The last section is the conclusion.

2. Memory Allocation based on Binary Buddy System

In [3], when a memory block of a given size is allocated, buddy system locates the block whose size is power of two and large enough to hold the requested size. The block is split in half as many times as possible, until it researches the block size which can hold the requested size. The two any half of spited block are known as buddies. When any block is freed and its body is free, the buddy system naturally coalesces the blocks, making available a larger size of memory. In software realization, the operations of splitting and coalescing memory dominate the cost of the buddy system. A hardware-maintained bit-map approaches eliminate the need for splitting and coalescing operations. Splitting becomes unnecessary because allocation is done using combinational logic. The bit-vector (a portion of a bit-map) forms the base of this binary tree. Deallocation is indicated by resetting bits in the allocation bit-vector, eliminating the need for explicit coalescing. Figure 1 demonstrates an example for block deallocation. Hardware approach can be made realizes in pure combinational logic, the time needs for memory management is greatly reduced.

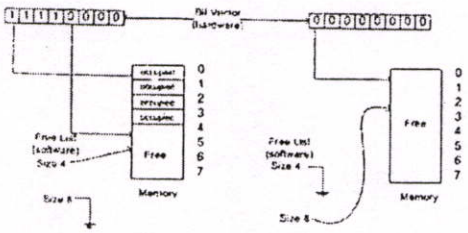


Figure 1 An example of memory deallocation

The binary buddy system always allocate memory in size of powers of two, so they may leave unused space known as internal fragments. Hardware approach eliminates internal fragments by allocating/deallocating the exact required size. This approach is known as modified buddy system. Although the buddy system may allocate a block that is larger than the requested size, the logic that finds a free block can be augmented by the "bit-flipper" in [3] to relinquish the unused portion at the end of the block. Figure 2 shows an example of memory block allocation.

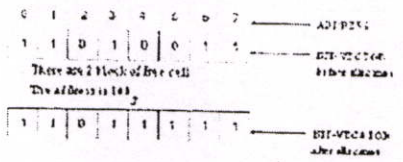


Figure 2 An example for block allocation

3. Architecture of Reference Counting Garbage Collection (RCGC)

Hardware algorithm of RCGC refer to software algorithm [2][6][7] consisting of three phase, the first phase is allocating memory heap for new objects and setting a reference count to one for the first time. The second phase is adjustment and checking of a reference count and the last phase is reclamating the garbage when reference counts hit zero. Figure 3 demonstrates an architecture of RCGC that based on [9]. Inside the RCGC, three bit-vector are used to keep all of the object relevant information such as allocation status of heap, the size information of occupied blocks, and the size information of reference count. The allocation status is kept on the *Allocation bit-vector (A bit-vector)*, the A bit-vector is set to logic '1' which represents a memory block allocated and set to logic '0' which represents the memory block deallocated. When *h_malloc* is called, the size information is received by the *Complete Binary Tree (CBT)*. This dedicated hardware unit is responsible for location the first free memory chunk that can satisfy the request using the modified buddy system. Besides location the memory chunk, the CBT also has to send out the address of that newly allocated memory and updates the status of that memory block from free to allocated and also reference count is set to logic '1' on C bit-vector. The size information of occupied blocks is kept on the *Size bit-vector (S bit-vector)*. Each time an object is created or reclaimed, the S bit-vector is instantly updated by hardware, *B-unit (size encoder)*. The adjustment and checking unit is only used during the adjustment and checking reference count phase of the reference counting garbage collection cycle, it is actived by *INC_DEC* signal and also receive object pointer relevant with reference object and unreferenced object of user program. The Reclamation unit is only used after that checking reference count is equal to zero, if successfull, it will sent *gc_ack* signal out.

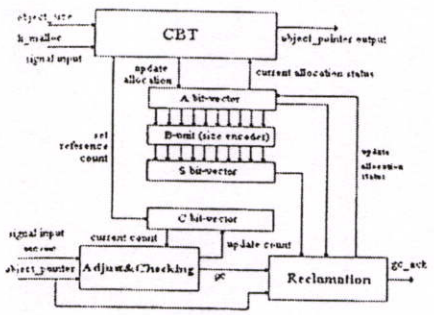


Figure 3. Architecture of Reference Counting Garbage Collector

4. Implementation

4.1. Hardware of memory allocation (Allocator)

Hardware of the memory allocation hardware for a new object refered to [3][9] consists of three phase.

a) determine, there is enough memory space for allocation request, by *Or-Gate tree hardware.*

b) find the beginning address of that memory chunk, by *And-Gate tree hardware.*

c) flip the bits corresponding to the memory chunk in the bit-vector, by *Bit-Flipper hardware.*

These hardware is incorporated as CBT hardware for allocating memory. This approach has created a new hardware for setting reference count to be one. This hardware keeps reference count on C bit-vector in position relevant with A bit-vector. The studies have shown that the reference count use three bit, being able to hold the reference count for at least 99.99% of objects [11]. Thus this approach will use three bits representing reference count. Figure 4 demonstrates an example of keeping reference count of value 5 on bit-vector The value of C bit-vector before keeping reference count is equal to 11100011_2. shifting the value 5 (101_2) three times into temp vector (000100100_2), the available output is 11110111_2.

```
"11100011"   C bit-vector before
"00010100"   bit-vector temp after shifting
             size of 101  at address 011
XOR__________
"11110111"   C bit-vector after
```

Figure 4. Keeping reference count on bit-vector

Figure 5 shows an example of the object information after allocation requested for a new object at address zero and object size of 4.

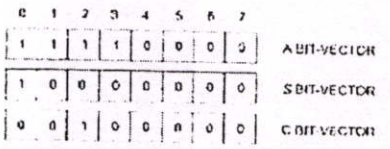


Figure 5. Keeping information on bit-vector

Hardware allocator using binary buddy system and bit-map techniques can reduce internal fragmentation. For example, if the buddy system allocates eight blocks for a 5-block requested, the hardware can mark exactly five of the eight bits(one bit per block). This allows using the unused three blocks in the future.

4.2. Hardware of adjustment and checking reference count (Adjust&Checking)

Each time a reference to the object is created, e.g., when a pointer is copied from one place to another place by an assignment, the *INC_DEC* will active high for incrementing the reference count and this hardware will also receive object pointer indexed for C bit-vector. When an existing reference to an object is eliminated, the *INC_DEC* will show active low signal for decrementing the reference count and will receive object pointer too. If reference count is equal zero, this hardware will sent *gc* signal to active the reclamation hardware. Figure 6 shows an architecture of adjustment and checking hardware.

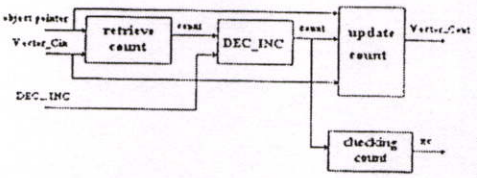


Figure 6. Architecture of Adjustment and checking hardware

The operation of adjustment and checking reference count is done by hardware approach, it can reduce the cost and the time for reference count management. Thus can improve the system performance as well.

4.3. Hardware of reclamation

When reference count is equal to zero, the hardware of reclamation will active for free memory space. This hardware that shows in figure 7 is developed from

bit_flipper in [3][9]. The output bits from bit-flipper will set in bit-vector. While hardware of reclamation will clear bits in A bit-vector with size of object and the address of object are reclamated and bits related in C bit-vecotor are also cleared. This hardware does not use an execute cycle time of program because the hardware is done automatically. Therefore, this approach can give better performance than software approach.

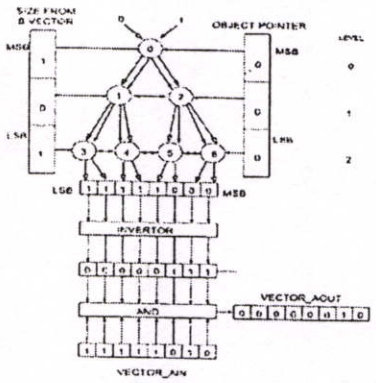


Figure 7. Architecture of reclamation hardware

5. Results

The Modelsim SE-EE 5.4E from MTI is used for a VHDL simulator and the Foundation series 3.1i from Xilinx is used for a synthesizer. The Virtex-V800BG560 was initially chosen as the target device. Table 1 shows the result of three component in terms of the number of slice, gate count and delays. Table 2 shows the comparisons number of the number of slice, gate count, maximum speed and delay time in design between Mark-Sweep Garbage Collector [12] and RCGC.

Gives results of the hardware designs show that the Allocator delay time and gate count more than others. Because Alocator need many resources for determining space, finding the first addresses of available memory blocks, flipping bits in bit-vector and setting reference count to one at the first time. While the response time is better than that of Mark-Sweep Garbage Collector because the RCGC can be interleaved with the program, so the RCGC should be suitable to apply in real time system. The cost for all three bit-maps in a 64 MB memory is about 1.5 MB. We assume that each bit represents memory of 16 bytes. This means that there is about 2.3% overhead

Table1. Component implementation results on Virtex-V800GB560-6

Component	Size(bit)	Num. Slice	Gate Count	Max Net Dealy(ns)	Max Delay(ns)
Allocator	8	42	501	5.774	21.634
	16	77	900	7.662	31.704
	32	163	1,933	8.233	42.583
	64	312	3,708	8.098	57.504
Adjust& checking	8	36	437	3.549	17.655
	16	74	867	7.669	30.944
	32	150	1,787	8.497	34.982
	64	329	3,657	12.936	39.753
Reclamation	8	26	300	3.356	20.037
	16	63	733	8.543	32.326
	32	139	1,568	9.618	31.91
	64	287	3,414	13.165	43.771

Table 2. Hardware implementation results on Virtex-V800GB560-6

	Size 8		Size 16		Size 32		Size 64	
	Mark-Sweep	Ref. Count	Mark-Sweep	Ref. Count	Mark-Sweep	Ref. Count	Mark-Sweep	Ref. Count
Number of Slice	60 (1%)	112 (1%)	108 (1%)	241 (2%)	229 (2%)	306 (3%)	430 (4%)	887 (8%)
Gate count	729	1,322	1,262	2,875	2,595	5,048	5,112	11,747
Max Clock Frequency (MHz)	164.328	105.742	103.806	132.937	102.87	101.061	77.8	82.143
Max Net Delay (ns)	8.458	9.826	8.827	9.974	8.62	10.197	9.717	12.473

6. Conclusion

This paper presents the design of Reference Counting Garbage Collection (RCGC) using hardware algorithm implemented by VHDL. The proposed scheme takes advantage of the binary buddy system and bit-map techniques because they can easily form the base of binary tree and they can be implemented in combinational logic efficiently. This hardware can improve the system performance from software approaches in case of internal fragments, cost of reference count management, and size of memory heap reference count. For a future research, our design will be implemented with Java processor.

References

[1] B. Zorn, "Custo-malloac: Efficient Synthesized Memory Allocators," Technical Report CU-CS-602-92, Computer Science Department, University of Colorado, July 1992.
[2] Paul R. Wilson, "Uniprocessor Garbage Collection Techniques," SIGPLAN Intl. Workshop on Memory Management, pp. 1-42, Sept. 1992.
[3] J. M. Chang and E. F. Gehringer, "A High-Performance

Memory Allocator for Object-Oriented-Systems," IEEE Trans. Computers, vol. 45, no. 3, pp. 357-366, March 1996.
[4] P. Wilson, M. Johnstone, M Neely and D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review," Proc. 1995 Int'l workshop on Memory Management, Scotland, UK, Sept. 27-29, 1995.
[5] T. Ritzau, "Real-Time Reference counting, " The Embedded system show . London, May 24-25, 2000.
[6] B. Venners, Java's Garbage Collected Heap, www.java-world.com.
[7] B. Venners, Inside the Java Virtual Machine, McGraw-Hill, 1999.
[8] J. M. Chang, W. Srisa-an, and J. M. Chang, "An Introduction to DMMX (Dynamic Memory Management Extension), " Workshop notes of ICCD workshop on Hardware Support for Objects and Microarchitectures for Java, Austin, Texas, pp. 11-14, Oct. 10, 1999.
[9] S. Kagan Agun and J. Morris Chang, "Design of a Reusable Memory Management System," Proceedings of 14th IEEE International ASIC/SOC Conference, Washington, D.C., Sep. 12-15, 2001.
[10] R. Jones and R. Lins, Garbage Collection: Algorithms for automatic Dynamic Memory Management, John Wiley and Sons, 1996.
[11] L .Fox, Measuring Maximum Reference Count, *http://www.cs.wustl.edu/~doc/RandD/ITR/Experiments/MaxRefCount/*.
[12] W. Srisa-an, C.-T. D. Lo and J. M. Chang, "Scalable Hardware algorithm for Mark-sweep Garbage Collection," IEEE, pp. 274-281, 2000.

ประวัติผู้เขียน

ชื่อ-นามสกุล	นายอุดม รานอก
วัน เดือน ปีเกิด	13 มิถุนายน พ.ศ.2518 ที่จังหวัดนครราชสีมา
ที่อยู่	129 หมู่ที่ 3 ต.วังโพธิ์ อ. บ้านเหลื่อม จ. นครราชสีมา 30350
ประวัติการศึกษา	2542 ปริญญาตรีวิศวกรรมศาสตรบัณฑิต สาขาวิศวกรรมคอมพิวเตอร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
ความชำนาญเฉพาะด้าน	ไมโครโปรเซสเซอร์และไมโครคอนโทรเลอร์
ประสบการณ์ทำงาน	2543 ทำงานในตำแหน่งอาจารย์ประจำภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ มหาวิทยาลัยธุรกิจบัณฑิตย์
ปัจจุบัน	อาจารย์ประจำและเลขานุการภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ มหาวิทยาลัยธุรกิจบัณฑิตย์