การออพติไมซ์ประสิทธิภาพของ XVID MPEG-4 โค้ดเดอร์และดีโค้ดเดอร์บนระบบปฏิบัติการซิมเบียนสมาร์ทโฟน

PERFORMANCE OPTIMIZATION OF XVID MPEG-4 VIDEO CODEC ON SYMBIAN SMARTPHONE

ไพโรจน์ ศักดิ์ไพบูลย์ผล

PAIROJ PAKDEEPAIBOONPOL

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต

สาขาวิชาวิศวกรรมคอมพิวเตอร์

บัณฑิตวิทยาลัย

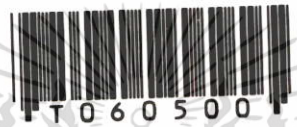
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

พ.ศ. 2548

ISBN 974-15-1082-6

การออพติไมซ์ประสิทธิภาพของ XVID MPEG-4 โค้ดเดอร์และดีโค้ดเดอร์บนระบบปฏิบัติการซิมเบียนสมาร์ทโฟน

PERFORMANCE OPTIMIZATION OF XVID MPEG-4 VIDEO CODEC ON SYMBIAN SMARTPHONE



ไพโรจน์ ภักดีไพบูลย์ผล

PAIROJ PAKDEEPAIBOONPOL

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต

สาขาวิชาวิศวกรรมคอมพิวเตอร์

บัณฑิตวิทยาลัย

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

พ.ศ. 2548

ISBN 974-15-1982-6

PERFORMANCE OPTIMIZATION OF XVID MPEG-4 VIDEO CODEC ON SYMBIAN SMARTPHONE

PAIROJ PAKDEEPAIBOONPOL

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENT FOR THE DEGREE OF

MASTER OF ENGINEERING IN TELECOMMUNICATIONS ENGINEERING

SCHOOL OF GRADUATE STUDIES

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

2005

ISBN 974-15-1982-6

หัวข้อวิทยานิพนธ์	การออพติไมซ์ประสิทธิภาพของ XVID MPEG-4 โค้ดเดอร์และดีโค้ดเดอร์บนระบบปฏิบัติการซิมเบียนสมาร์ทโฟน
นักศึกษา	นายไพโรจน์ ภักดีไพบูลย์ผล
รหัสประจำตัว	46061022
ปริญญา	วิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชา	วิศวกรรมคอมพิวเตอร์
พ.ศ.	2548
อาจารย์ผู้ควบคุมวิทยานิพนธ์	ผศ.ดร สุรินทร์ กิตติธรกุล

บทคัดย่อ

มัลติมีเดียแอพพลิเคชั่นบนสมาร์ทโฟนควรจะต้องสนใจเกี่ยวกับความเร็วและพลังงานที่ใช้ เนื่องจาก CPU ที่มีความเร็วต่ำและพลังงานแบตเตอร์รี่ที่จำกัด ในวิทยานิพนธ์นี้ เราจะเสนอวิธีการออพติไมเซชั่นทั้งความเร็วและพลังงานสำหรับระบบมัลติมีเดียแบบสมองกลฝังตัวบน ARM เช่น โทรศัพท์มือถือ, พอกเก็ต พีซี, ระบบมัลติมีเดียส่วนบุคคล เป็นต้น ในการทดลองใช้ MPEG-4 simple profile level 0 (SP@L0) video codec (โค้ดเดอร์และดีโค้ดเดอร์) บน ARM920T และใช้สอง QCIF วีดีโอมาตรฐานที่ 15 เฟรมต่อวินาที 24 กิโลบิตต่อวินาที ซึ่งผลที่ออกมาเมื่อใช้วิธีการที่เราเสนอกับดีโค้ดเดอร์สามารถเพิ่มความเร็วได้ 7.02 % และลดพลังงานที่ใช้ได้ลง 13 % และเมื่อใช้กับเอ็นโค้ดเดอร์เพิ่มความเร็วได้ 35.5% และลดพลังงานได้ 5% ในทุกระดับการออพติไมเซชั่น ARM C++ คอมไพเลอร์บนพื้นฐานที่มี 16 กิโลบิตแคชชุดคำสั่งและ 16 กิโลบิตแคชข้อมูลของ ARM920T

Thesis Title	Performance Optimization of XVID MPEG-4 Video Codec on Symbian Smartphone
Student	Mr.Pairoj Pakdeepaiboonpol
Student ID	46061022
Degree	Master of Engineering
Programme	Computer Engineering
Year	2005
Thesis Advisor	Asst.Prof.Dr Surin Kittitornkun

ABSTRACT

Multimedia applications on smart phones should concern about speed and power consumption because low speed CPU and limited battery. In this thesis, we present power/energy and speed optimization methodologies for ARM-based battery-powered embedded multimedia systems, e.g. mobile phones, pocket PCs, personal multimedia systems, etc. The experiments using MPEG-4 simple profile level 0 (SP@L0) video codec (coder and decoder) on ARM920T with two QCIF video sequences 15 fps, 24 kbps show that using the proposed techniques in decoder can gain speed up to 7.02 % and achieve lower energy/power consumption up to 13% and using it in coder can gain speed up to 35.5 % and achieve lower energy/power consumption up to 5% relative to all ARM C++ optimization levels despite the 16-KB instruction and 16-KB data caches of ARM 920T core.

กิตติกรรมประกาศ

คุณความดีอันใดที่บังเกิดจากวิทยานิพนธ์ฉบับนี้ ขอมอบแด่บิดาและมารดาของผู้วิจัย ผู้ที่คอยห่วงใย เข้าใจและให้การสนับสนุนในการศึกษามาโดยตลอด

วิทยานิพนธ์ฉบับนี้สำเร็จลุล่วงลงได้ด้วยดี โดยได้รับความกรุณาจาก ผศ.ดร สุรินทร์ กิตติธรกุล อาจารย์ที่ปรึกษา ที่ได้ช่วยเหลือในการให้คำแนะนำ ความรู้ทางทฤษฎีต่างๆที่ใช้ และชี้แนะแนวทางในการแก้ปัญหาต่างๆอย่างทุ่มเทรวมทั้งฝึกฝนผู้วิจัยให้มีความสามารถในการทำวิจัยและพัฒนาได้อย่างมีประสิทธิภาพ ผู้วิจัยรู้สึกซาบซึ้งในความอนุเคราะห์จากท่านและขอกราบขอบพระคุณเป็นอย่างสูง

ขอขอบคุณกรรมการสอบวิทยานิพนธ์ทุกท่านที่ได้กรุณาให้คำแนะนำในทุกๆเรื่อง ทั้งวิธีแก้ปัญหาที่เกิดขึ้นในการทำวิทยานิพนธ์และมุมมองในเชิงวิศวกรรมอื่นๆซึ่งช่วยให้ผู้วิจัยมีวิสัยทัศน์ที่กว้างไกลขึ้น

ขอขอบคุณบัณฑิตวิทยาลัย สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง ที่ให้การสนับสนุนการทำวิทยานิพนธ์นี้

ขอบคุณพี่ๆเพื่อนๆและน้องๆนักศึกษาทุกคนในห้องวิจัย รวมทั้งเพื่อนๆหลายๆคนในอินเตอร์เน็ตที่ช่วยเหลือให้คำแนะนำต่างๆและให้กำลังใจแก่ผู้วิจัยตลอดมา

ไพโรจน์ ภักดีไพบูลย์ผล

สารบัญ

หน้า

สารบัญ (ต่อ)

หน้า

สารบัญ (ต่อ)

หน้า

สารบัญตาราง

สารบัญรูป

สารบัญรูป (ต่อ)

สารบัญรูป (ต่อ)

บทที่ 1
บทนำ

1.1 ความเป็นมาและความสำคัญของปัญหา

จากเมื่อก่อนที่การสนทนากันทางโทรศัพท์ด้วยเสียง จนกระทั่งเทคโนโลยีต่างๆได้พัฒนาขึ้นจนกระทั่งปัจจุบันโทรศัพท์เคลื่อนที่ได้เข้ามามีบทบาทในชีวิตประจำวันมากขึ้น ดังจะเห็นได้จากวิวัฒนาการของโทรศัพท์เคลื่อนที่ที่ได้มีการพัฒนาไปมาก จากอดีตที่มีการทำงานด้วยระบบอนาล็อก (Analog) มาสู่ยุคปัจจุบันที่ทำงานด้วยระบบดิจิตอล (Digital) หรือยุคของ 2จี (2G), 2.5จี (2.5G) และยังมีแนวโน้มจะก้าวไปสู่ยุค 3จี (3G) ในเวลาอันใกล้นี้ จึงทำให้โทรศัพท์เคลื่อนที่ไม่ได้เป็นแค่เพียงอุปกรณ์ที่ใช้สำหรับการสื่อสารด้วยเสียงเพียงอย่างเดียวเท่านั้น แต่ยังสามารถใช้การสื่อสารด้วยภาพเพิ่มเติมเข้าไปด้วย ช่วยให้สามารถสื่อสารและทำความเข้าใจกับคู่สนทนาได้ดียิ่งขึ้น สามารถสนทนากันได้ยินเสียงและภาพพร้อมกันทั้งบนโทรศัพท์มือถือ, พีดีเอ และระบบมัลติมีเดียส่วนตัว ซึ่งเทคโนโลยี MPEG-4 วีดีโอดีโค้ดเดอร์และเอ็นโค้ดเดอร์ ได้รับความนิยมอย่างแพร่หลายในการบีบอัดข้อมูลในการสื่อสารแบบเรียลไทม์ (Real-time) เนื่องด้วยความสามารถในการบีบอัดข้อมูลที่มีความสามารถเพียงพอในการลดขนาดของภาพให้สามารถผ่านช่องทางที่มี bandwidth ที่จำกัดเช่นเครือข่ายไร้สายอย่างโทรศัพท์มือถือ แต่อย่างไรก็ตามวีดีโอ MPEG-4 บน RISC โพรเซสเซอร์ ยังต้องคำนวณมากและใช้พลังงานจากแบตเตอร์รี่มาก RISC โพรเซสเซอร์ในระบบพวกนี้ที่ได้รับความนิยมมากคือ ARM เนื่องด้วยกินพลังงานน้อยและขนาดเล็ก [10] โดยเฉพาะอย่างยิ่งในมือถือที่มีพลังงานในการคำนวณที่จำกัดและมีผลกระทบโดยตรงกับเวลาในการใช้งานแบตเตอร์รี่

ดังนั้นจึงมีความจำเป็นต้องออพติไมซ์ (Optimize) พลังงานและความเร็วของโคเดค (เอ็นโค้ดเดอร์และดีโค้ดเดอร์) MPEG-4 เป็นงานที่จำเป็น ในงานวิทยานิพนธ์ชิ้นนี้จะมุ่งเน้นการออพติไมเซชั่นเพื่อลดการใช้พลังงานและและเพิ่มความเร็วสำหรับวีดีโอ MPEG-4 simple profile ที่ทำงานบน ARM โพรเซสเซอร์

MPEG-4 วีดีโอโคเดคส่วนมากจะเป็นในทางการค้า แต่ก็มี opensource ของ MPEG-4 อยู่แต่ได้เลือกของ XviD เนื่องจากความเหมาะสมของขนาดโค้ดที่เล็กพอที่จะทำงานบนมือถือและความเร็วในการทำงาน จากนั้นทำการ port โค้ด MPEG-4 ของ XviD ไปทำงานบนมือถือระบบปฏิบัติการซิมเบียน (Symbian) และจะใช้โค้ดที่ port นี้ทดลองในงานวิจัยนี้

1.2 วัตถุประสงค์ของการศึกษา

1. ศึกษาหลักการของ MPEG-4 วีดีโอโคเดค
2. ศึกษา XviD MPEG-4 วีดีโอโคเดคที่ port บนมือถือ
3. ศึกษาหลักการทำงานของ ARM โปรเซสเซอร์ซีรี่ที่ใช้ในมือถือ
4. ปรับปรุงลดการใช้พลังงานและเพิ่มความเร็ว XviD MPEG-4 วีดีโอโคเดคที่ port บนมือถือ
5. เปรียบเทียบผลการลดการใช้พลังงานและเพิ่มความเร็วทั้งก่อนและหลังการปรับปรุง

1.3 สมมุติฐานของการศึกษา

เพื่อให้บรรลุวัตถุประสงค์ของการศึกษา ได้นำเสนอหลักการตามทฤษฎีที่เกี่ยวข้องเพื่อทำให้ลดการใช้พลังงานและเพิ่มความเร็วในการทำงานของ MPEG-4 วีดีโอโคเดค เพิ่มความเร็วด้วยวิธีเช่นการลด function call การลด branch prediction การทำ loop unrolling ซึ่งช่วยให้ pipe line ทำงานมีประสิทธิภาพสูงขึ้นทำให้ทำงานเร็วขึ้นและลดการใช้พลังงานด้วยวิธีเช่น Minimizing Buffer Allocation และ Read before Write ซึ่งเป็นการลด cache miss ลงทำให้การเข้าถึงหน่วยความจำลดลงด้วย ซึ่งในมัลติมีเดียแอพพลิเคชั่นจะใช้พลังงาน 50-80% กับการเข้าถึงหน่วยความจำ ดังนั้นด้วยวิธีการนี้ก็เท่ากับลดการพลังงานด้วยนั้นเอง

1.4 ขอบเขตของงานวิจัย

วิทยานิพนธ์ชิ้นนี้ได้ศึกษา MPEG-4 วีดีโอโคเดคแล้วทำการ port ไปทำงานบนมือถือระบบปฏิบัติการซิมเบียน (Symbain) และศึกษาหลักการทำงานของ ARM โปรเซสเซอร์ในซีรี่ที่ใช้บนมือถือเช่น ARM920T, ARM922T เป็นต้น แล้วปรับปรุงประสิทธิภาพทั้งเพิ่มความเร็วในการทำงานและการลดพลังงานโดยมุ่งเน้นการลดการเข้าถึงหน่วยความจำด้วยวิธีการที่สอดคล้องกับการทำงานของ ARM โปรเซสเซอร์ จากนั้นทำการวัดประสิทธิภาพเปรียบเทียบผลก่อนและหลังการปรับปรุง

1.5 ขั้นตอนการศึกษา

1. ศึกษาหลักการของ MPEG-4 วีดีโอโคเดค
2. ทำการศึกษาหลักการของสถาปัตยกรรม ARM โปรเซสเซอร์เพราะ ARM ได้รับความนิยมอย่างกว้างขวางใช้ในอุปกรณ์มือถือ
3. ศึกษางานวิจัยที่คล้ายคลึงหรือสอดคล้องกับงานวิจัยนี้
4. ทำการทดลองการออพติไมเซชั่นของเราโดยเปรียบเทียบก่อนและหลังการทดลองโดยใช้ ARM Developer Suite ในการซิมมูเลชั่นการทำงานของ ARM920T โปรเซสเซอร์
5. สรุปและวิเคราะห์ผลการทดลอง

1.6 ข้อตกลงเบื้องต้น

งานวิจัยนี้คือ การออพติไมซ์ประสิทธิภาพของ MPEG-4 Codec บนระบบปฏิบัติการซิมเบียนสมาร์ตโฟน และต้องยอมรับก่อนว่าโปรเซสเซอร์ส่วนมากที่ใช้ในอุปกรณ์โทรศัพท์มือถือจะใช้ ARM โปรเซสเซอร์ ซึ่งวิธีการของเราจะเป็นการออพติไมเซชั่นประสิทธิภาพบนพื้นฐานสถาปัตยกรรม ARM

1.7 ข้อจำกัดของการศึกษา

ในการจะวัดประสิทธิภาพของ MPEG-4 วีดีโอโคเดคบนอุปกรณ์มือถือ ทั้งในด้านความเร็วและในด้านการใช้พลังงานทำได้ยาก ดังนั้นจะใช้ tool ที่ชื่อว่า ARM developer suite จำลองการทำงานของ ARM โปรเซสเซอร์ แล้ววัดค่าต่างๆที่สำคัญ

1.8 รายละเอียดของวิทยานิพนธ์

วิทยานิพนธ์ฉบับนี้ เป็นการนำเสนอวิธีการออพติไมเซชั่นเพื่อเพิ่มประสิทธิภาพของ MPEG-4 Codec ในแง่ของเพิ่มความเร็วและลดพลังงาน เพื่อให้สามารถทำงานบนมือถือได้ดีมากขึ้น โดยรายละเอียดต่างๆภายในวิทยานิพนธ์นี้ได้จัดแบ่งในส่วนเนื้อหาออกเป็น 5 บท ซึ่งแต่ละบทมีหัวข้อและเนื้อหาดังต่อไปนี้

บทที่ 1 “บทนำ” อธิบายถึงวัตถุประสงค์ สมมุติฐานของการศึกษา ขอบเขตและขั้นตอนการศึกษา รวมไปถึงรายละเอียดเนื้อหาโดยสรุปของแต่ละบท

บทที่ 2 “หลักการและงานวิจัยที่เกี่ยวข้อง” อธิบายการทำงานของ MPEG-4 video codec simple profile simple level นอกจากนี้ยังได้อธิบายถึงการทำงานของ ARM920T โพรเซสเซอร์ และยังอธิบายถึงระบบปฏิบัติการซิมเบียน (Symbian OS)

บทที่ 3 “วิธีการออพติไมเซชั่นเพื่อความเร็วและลดพลังงาน” บทนี้จะเสนอหลักการออพติไมเซชั่นเพิ่มประสิทธิภาพของ MPEG-4 video codec บน Symbian OS ทั้งในแง่ของการลดการบริโภคพลังงานและการเพิ่มความเร็วในการทำงานของ codec อธิบายถึงหลักการและเหตุผลที่เป็นเช่นนี้

บทที่ 4 “การทดลองและผลการทดลอง” กล่าวถึงการเปรียบเทียบประสิทธิภาพของ MPEG-4 video codec บน Symbian OS ทั้งก่อนและหลังการเพิ่มประสิทธิภาพในแง่ของพลังงานที่ใช้และความเร็วในการทำงานของ codec

บทที่ 5 “บทสรุป” เป็นการสรุปและวิจารณ์ผลที่ได้จากการวิเคราะห์และการทดลอง พร้อมทั้งปัญหาที่เกิดขึ้น ตลอดจนข้อเสนอแนะสำหรับนำงานวิจัยในวิทยานิพนธ์นี้ไปพัฒนาใช้ประโยชน์ต่อไป

บทที่ 2
หลักการและงานวิจัยที่เกี่ยวข้อง

2.1 งานวิจัยที่เกี่ยวข้อง

ส่วนนี้จะเสนองานวิจัยที่คล้ายคลึงหรือเกี่ยวข้องกับงานวิจัยที่เราพัฒนา ซึ่งในวิทยานิพนธ์นี้สนใจงานวิจัยทางด้านการเพิ่มความเร็วและงานวิจัยเกี่ยวกับการลดการใช้พลังงาน ทั้งบนแพลตฟอร์มพีซีหรือระบบสมองกลฝังตัว

2.1.1 งานวิจัยเกี่ยวกับการเพิ่มความเร็วในการทำงาน

[3] เสนอ ADdress OPTimisation (ADOPT) ที่ทำการออพติไมซ์ด้วยการเปลี่ยนแปลง source code โดยอิสระจาก instruction-set architecture ที่มุ่งที่จะออพติไมซ์ addressing code และ related control flow วิธีออพติไมเซชั่นนี้มี 2 ขั้นตอนใหญ่ๆ ขั้นแรกลดจำนวน address arithmetic related operation และจำนวนครั้งที่ operation เหล่านี้ถูกใช้ โดยทดลองออพติไมซ์ MPEG-4 video decoder บน Pentium III และ TriMedia TM 1000 ใช้ video sequence 3 ชุด ประกอบด้วย foreman, mother and daughter และ calendar and mobile ผลการทดลองเมื่อเทียบก่อนกับหลังการออพติไมเซชั่นได้ speed up 40-60%

[10] อธิบายเกี่ยวกับรายละเอียดในการทำให้ MPEG-4 video decoder บน AMR7TDMI ให้ใช้งานแบบ real time ได้ โดยได้ใช้ออพติไมซ์ที่ระดับ Algorithmic level และ Assembly code level

ในการออพติไมเซชั่น Algorithmic level จะทำในภาษา C เป็นการเปลี่ยน algorithm เพื่อลดการคำนวณ เทคนิคที่ใช้ในการออพติไมซ์เช่น Loop unrolling, Loop distribution และ Loop interchange ในการออพติไมเซชั่น Assembly code level จะใช้วิธีการเช่น การลดจำนวน function parameter, การลดจำนวน memory access โดยใช้ LDM และ STM การลดจำนวน function parameter มีผลเนื่องจากใน ARM7TDMI มี register 4 ตัวในการส่ง parameter ไปใน function ถ้ามี parameter มากกว่า 4 ตัวต้องมีการใส่ stack เก็บไว้ ซึ่งก็จะมี overhead ดั้งนั้นการลดจำนวน parameter ให้น้อยกว่าเท่ากับ 4 จึงดีกว่า และ การลดจำนวน memory access โดยใช้ LDM และ STM เนื่องจาก LDM และ STM เป็นการ load และ store ข้อมูลที่ล่ะหลาย word เมื่อนำมาใช้แทนคำสั่ง Load และ Store จึงลด memory access ลง ผลลัพธ์ที่ได้จากการออพไมเซชั่นจากแต่ก่อนเดิม decode วีดีโอ foreman 15 fps ใช้ 100 Mhz ลดเหลือ 16.0 MHz

2.1.2 งานวิจัยเกี่ยวกับการลดการบริโภคพลังงาน

ระบบและแอพพลิเคชั่นที่เกี่ยวกับมัลติมีเดียจะใช้พลังงานมากจากการเข้าถึงหน่วยความจำ ดังนั้นในวิธีการลดพลังงานจึงเป็นสิ่งที่ท้าทายอย่างยิ่ง ในงานวิจัย

[1] เสนออัลกอริทึมการออพติไมต์พลังงานของวีดีโอดีโค้ดเดอร์ที่ทำงานบนระบบฝังตัวสมองกล (embedded) วิธีการที่ใช้จะออพติไมต์ส่วนการเก็บข้อมูล (data storage) และการจัดการกับการส่งข้อมูล (transfer organization) ในงานวิจัยนี้เลือกสองสถาปัตยกรรม ARM7 RISC โพรเซสเซอร์และ Custom standard cell ซึ่งวิธีนี้สามารถลดทั้งสัญญาณนาฬิกา (Clock cycles) และการเข้าถึงหน่อยความจำ (Memory) ข้อมูลคำสั่ง (Instruction) แต่วิธีนี้จะแทบไม่มีผลเมื่อใช้บนสถาปัตยกรรม ARM ทีมีแคช (Cache) เช่น ARM920T

[4] และ [9] นำเสนอวิธีการและเครื่องมือ (Tool) ซึ่งเป็นการออพติไมต์บนแพลตฟอร์มพีซีระดับสูงเรียกว่า DTSE (Data Transfer and Storage Exploration) เมื่อนำมาใช้กับ MPEG-4 วีดีโอดีโค้ดเดอร์ทำให้ได้ประสิทธิ์ภาพที่สูงขึ้น แต่งานวิจัยนี้ทดลองบนแพลตฟอร์มพีซีอย่างเดียว ซึ่งอาจจะใช้ไม่ได้ดีบน ARM โพรเซสเซอร์

2.2 MPEG-4 simple level simple profile

2.2.1 บทนำ

การบีบอัดสัญญาณวิดีโอซึ่งเราจะว่ากันต่อไปนี้ จริง ๆ แล้วก็เป็นการกล่าวถึงทั้งการบีบอัดและขยายสัญญาณ เพราะเมื่อมีการบีบอัดสัญญาณก็ต้องมีการขยายกลับสัญญาณ จึงจะกลับมาแสดงผลเป็นภาพหรือเสียงที่เหมือน (หรือจะผิดเพี้ยนไปบ้าง) เพื่อนำกลับมาใช้งานได้ ดังนั้นเมื่อกล่าวถึงการบีบอัด (Compression) จึงหมายถึงทั้งการอัดและขยายสัญญาณควบคู่ไปด้วยกัน ก่อนจะเข้าเรื่องของวิดีโอ ขอให้ท่านย้อนกลับไปพิจารณา CD เพลงทั่วๆ ไปกันก่อน (ไม่ใช่แผ่น CD แบบ MP-3) แผ่น CD ที่ให้เสียงคุณภาพดีหมายถึงแผ่นที่มีการบันทึก (เข้ารหัสแล้วบันทึก) รายละเอียดของคลื่นเสียงได้มากที่สุดเพื่อเวลานำมาเล่น(ขยายสัญญาณ) กลับจะมีการถอดรหัสดิจิทัลแล้วแปลงออกมา เป็นคลื่นสัญญาณไฟฟ้า นำมาขยายเป็นคลื่นเสียงที่ไพเราะเพราะพริ้งเหมือนจริงทุกประการ เมื่อเราลองพิจารณาสัญญาณดิจิทัลที่บันทึกลงบนแผ่น โดยก่อนที่จะมีการบันทึก 0 – 1 – 0 – 1 ลงบนแผ่น CD นั้น จะมีการชักตัวอย่างสัญญาณจากคลื่นเสียงที่ต้องการบันทึก ในอัตรา 44,100 ครั้งต่อวินาที และในทุก ๆ ครั้งของการชักตัวอย่าง ก็จะทำการแปลงระดับสัญญาณที่ชักได้ (เข้ารหัส / Coder) ออกมาเป็นค่าตัวเลขค่าหนึ่ง ซึ่งจะถูกนำไปแปลงให้เป็นสัญญาณดิจิทัลขนาด 16 บิตเพื่อใช้แทนระดับความแรงของคลื่นเสียง แล้วจึงมีการบันสัญญาณดิจิทัล หรือสัญญาณ 0 – 1 – 0 – 1 ลงแผ่น CD เพื่อจะสามารถนำไปถอดรหัสแปลงเป็นสัญญาณเสียงในภายหลัง

ที่นี้เราลองมาคำนวณกันเล่น ๆ ว่าในแต่ละเวลา (วิ) นาทีของเสียงเพลงอันไพเราะเพราะพริ้งที่เราเล่น (ถอดรหัส/Decoder) ออกมาจากแผ่น CD นั้น จะกินเนื้อที่จัดเก็บข้อมูลบนแผ่น CD มากแค่ไหน

44,100 x 16 บิต = 705,600 บิตต่อวินาที (bps)

แต่เพราะว่า 16 บิต = 2 ไบต์

ดังนั้น 44,100 x 2 ไบต์ = 88,200 ไบต์ต่อวินาที

เมื่อหารด้วย 1Kbyte(1024) = 86.1 กิโลไบต์/วินาที/ข้าง (Channel)

เพราะว่าระบบสเตอริโอมีการบันทึกสัญญาณเสียงสองข้าง (Channel –A และ Channel – B)

ดังนั้น ในหนึ่งนาทีที่เราบันทึกสัญญาณเสียงลงบนแผ่น CD จะใช้เนื้อที่ในการบันทึกสัญญาณเท่ากับ

86.1K x 2 x 60 = 10,332 K ไบต์/นาที

= 10.1 M ไบต์/นาที

จะเห็นว่า เราต้องการใช้พื้นที่ในการเก็บข้อมูลประมาณ 10 MB ต่อนาที สำหรับการเก็บสัญญาณเสียงในระบบสเตอริโอลงบนแผ่น CD ดังนั้นแผ่น CD ทั่ว ๆ ไปซึ่งมีความจุประมาณ 650 – 700 MB ก็จะสามารถบันทึกเสียงเพลงได้เพียงประมาณ 60 นาทีหรือเพียง 1 ชั่วโมงเท่านั้นเอง

ถึงตอนนี้หากท่านคิดว่าการบันทึกสัญญาณเสียงต้องใช้พื้นที่ในการเก็บข้อมูลหรือการเข้ารหัสเพื่อเก็บข้อมูลของภาพแต่ละภาพดูบ้างว่า เราจะทำได้อย่างไร ?

เมื่อย้อนกลับไปพิจารณาภาพจากจอโทรทัศน์หรือจอมอนิเตอร์ภาพนิ่ง เมื่อขยายภาพมากขึ้น ๆ จะเห็นได้ว่าแต่ละภาพจะประกอบไปด้วยจุดเล็ก ๆ ที่แต่ละจุดเรียกว่า PIXEL (ซึ่งย่อมาจากคำว่า Picture Element) เรียงสลับกันไปจนประกอบขึ้นมาเป็นภาพและเมื่อลองพิจารณาภาพจากหน้าจอโทรทัศน์ ที่แพร่ภาพออกอากาศในระบบ 525 เส้น ด้วยความละเอียด 720 Pixels/เส้น (มีการสแกนภาพจากด้านบนจอไปถึงด้านล่างจอด้วยที่ขนาด 525 เส้น แต่ละเส้นประกอบไปด้วยส่วนประกอบของภาพ 720 Pixels) แต่จริง ๆ แล้ว ในการคำนวณจะนำมาคิดแค่ 486 เส้น เพราะจำนวนเส้นที่เหลือ จะถูกซ่อนไว้ในขอบบนและขอบล่างของจอภาพ เพื่อให้เห็นว่ามีภาพอยู่เต็มจอ

ก่อนที่จะดูลึกไปถึงรายละเอียดของภาพแต่ละ PIXEL เราลองมาทำความเข้าใจ ถึงองค์ประกอบของภาพกันก่อน

ในยุคแรก ๆ โทรทัศน์ขาวดำจะใช้สัญญาณแทนค่าระดับความสว่าง (Luminance) ของแต่ละ PIXEL ถ้าสว่างมากก็จะเห็นเป็นสีขาวสว่างน้อยก็จะเป็นสีเทา ไม่สว่างเลยก็จะมืดเป็นสีดำ เมื่อนำ PIXEL เหล่านี้มาประติดประต่อกันอย่างเป็นระบบ ก็จะทำให้เห็นเป็นภาพขาวดำขึ้นในยุคต่อมา เมื่อมีการสร้างเครื่องรับโทรทัศน์สี ก็มีการพยายามจะแทนค่าเฉดสี หรือระดับสัญญาณของสี (Chrominance) ในแต่ละ PIXEL หลายแบบ แต่แบบที่นิยมใช้และรู้จักกันมากที่สุดได้แก่วิธีผสมสัญญาณสี จากปืนอีเล็กตรอนของจอภาพ โดยการผสมสัญญาณสีต่าง ๆ คือสีแดง (Red-R) เขียว

(Green-G) และ น้ำเงิน (Blue-B) ซึ่งต่อไปจะเรียกว่าสัญญาณ R-G-B ที่ระดับเฉดสีต่าง ๆ กัน (ถ้าเฉดสี อ่อน-แก่ แตกต่างกันมาก ๆ สีที่ผสมกันออกมาแล้วก็จะมีความแตกต่างหรือมีรายละเอียดมาก ส่วนใหญ่จะใช้กันที่ 256 เฉดสี) แล้วยิงแสงแต่ละอันไปบริเวณเดียวกัน ทำให้สีต่าง ๆ ผสมกันขึ้นมาเป็น Pixel หนึ่ง ที่อาจจะมีสีเหมือนหรือแตกต่างกันออกไป

เมื่อมาถึงจุดนี้ เราลองมาคำนวณกันเล่น ๆ ว่าภาพสีแต่ละเฟรม(ภาพ) ของโทรทัศน์เครื่องนี้ควรมีขนาดเท่าใด เมื่อภาพหนึ่งประกอบไปด้วยจุดสี (Pixel) รวมกันเท่ากับ 720x486 = 349,920 pixel และการแทนเฉดสี 256 เฉดสีในแต่ละ Pixel จะแทนค่าเป็นตัวเลขฐานสองซึ่งระบบคอมพิวเตอร์จะสามารถเข้าใจได้เอง ต้องแทนค่าเป็นข้อมูลขนาด 8 บิต (28 = 256) แต่การแทนค่าระดับเฉดสีต้องทำทั้งสามสี คือ Red-Green-Blue (R-G-B) หรือจะต้องมีการใช้ข้อมูลแทนค่าสีทั้งหมด 8 x 3 = 24 บิต หรือเท่ากับ 3 ไบต์นั่นเองดังนั้นในสัญญาณสีหนึ่งภาพ จะต้องใช้ข้อมูลถึง 349,920 x 3 = 1,025 KB จึงจะแทนสัญญาณภาพได้ละเอียดครบถ้วน มากพอที่จะให้ภาพที่มีสีไม่ผิดเพี้ยนไปได้ และถ้าสัญญาณวิดีโอหนึ่งวินาทีมีภาพอยู่ 30 ภาพ ดังนั้น 1 นาทีจะต้องใช้พื้นที่ในการจัดเก็บข้อมูลเท่ากับ 1025 x 30 x 60 = 1.76 GB/นาที หรือต้องใช้พื้นที่เป็น 180 เท่าของสัญญาณเสียง

จะเห็นได้ว่าข้อมูลภาพ จะมีขนาดมหาศาลเมื่อเทียบกับข้อมูลเสียงแต่เราก็สามารถที่จะลดขนาดของข้อมูลเหล่านี้ได้โดย

1.ลดขนาดภาพ ถ้าคุณต้องการรักษาคุณภาพของภาพให้ได้ความคมชัดเท่ากับระดับของการแพร่ภาพ การลดขนาดของภาพลงด้วย

2.ลดจำนวนเฟรมต่อวินาที การลดจำนวนเฟรมต่อวินาทีก็คือการลดจำนวนภาพต่อวินาทีลง ถึงแม้ว่าจะฟังดูแปลก ๆ แต่ก็เป็นวิธีการหนึ่งที่จะลดขนาดข้อมูลลงได้

3.ลดรับเฉดสี แทนที่จะคงระดับการแยกแยะเฉดสีของแต่ละสี (R-G-B) ไว้ที่ 256 ระดับ หรือ 8 บิต (256=28) ก็ให้ลดระดับของการแยกแยะเฉดสีแต่ละสีลงมาเหลือแค่ 4 บิต โดยอาจจะตั้งภาพขาวดำ ก็จะทำให้ข้อมูลของภาพแต่ละเฟรมลดลงไปได้มาก

4.ปรับสมมาตรเฉดสี วิธีอาจจะดูว่าซับซ้อนไปสักนิด แต่ก็เป็นลูกเล่นที่สามารถปรับเปลี่ยนเพื่อให้ภาพออกมาดี โดยใช้ข้อมูลสีเท่าเดิม ตัวอย่างเช่น ถ้าจะใช้ข้อมูลสีแค่ 12 บิต แทนที่จะตั้งค่าสีเป็น 4:4:4 ก็ให้เพิ่มระดับสีเขียวให้เป็น 6 และลดระดับสีน้ำเงินเป็น 2 คือตั้งข้อมูลสีไว้เป็น 4:6:2 ซึ่งอาจจะให้ภาพที่ออกมามีโทนสีคล้าย ๆ กัน แต่ให้ความรู้สึกประทับใจได้มากกว่า

5.บีบอัดสัญญาณให้เล็กลง สิ่งนี้เป็นสิ่งที่พวกเรารู้จักและคุ้นเคยกันในรูปแบบของไฟล์แบบต่าง ๆ เช่น ZIP ไฟล์, JPEG ไฟล์, M-JPEG, MPEG หรือเพลง MP-3 เป็นต้น สิ่งเหล่านี้มีบทบาทมากขึ้นในชีวิตประจำวันของเรา โดยที่เราอาจจะไม่ทราบมาก่อนว่าเราได้ใช้และคุ้นเคยกับเทคโนโลยีเหล่านี้อยู่แล้ว ซึ่งจะขอกล่าวถึงรายละเอียดของเรื่องนี้ในย่อหน้าถัด ๆ ไป

ก่อนที่จะเข้าสู่รายละเอียดของการบีบอัดสัญญาณ จะมีค่าถามว่า มีเหตุผลอะไรที่จะต้องบีบอัดสัญญาณ จะมีคำถามว่า มีเหตุผลอะไรที่จะต้องบีบอัดสัญญาณด้วย ? คำตอบก็คือ สัญญาณก็คือข้อมูลที่เราจะต้องนำไปใช้ และด้วยเหตุที่ว่าค่าใช้จ่ายในการเก็บรักษาและส่งผ่านข้อมูลต่าง ๆ จะ

คิดคำนวณเป็นราคาต่อหน่วยของข้อมูล หรือเป็นราคาหน่วยข้อมูลที่ใช้ในการจัดเก็บ ซึ่งจะเห็นได้ชัดจากกรณีของการบีบอัดข้อมูลของเพลงให้เล็กลง เช่น เพลง MP-3 ในแผ่น CD เพราะถ้าแผ่น MP-3 มีจำนวนเพลงไม่มากกว่าหรือเท่ากับ CD เพลงธรรมดา ก็คงจะไม่มีคนซื้อไปฟัง แต่สิ่งสำคัญที่สุดก็คือเวลาที่ใช้ในการรับส่งข้อมูล เพราะในยุคข้าวสารที่ต้องทำงานแข่งกับเวลา ในการจัดการและนำเสนอเรื่องราวต่าง ๆ ต้องเร็วและชัดเจนเสมอ

เทคนิคการบีบอัดข้อมูลต่าง ๆ ก็คือ การทำให้ไฟล์ข้อมูลขนาดใหญ่มีขนาดเล็กลง ซึ่งอาจจะเป็นการบีบอัดข้อมูลแบบต่าง ๆ เช่น แอนะล็อกวิดีโอ ดิจิทัลวิดีโอ ข้อมูล ซึ่งพวกเราอาจจะคุ้นเคยกันในรูปแบบของไฟล์แบบ ZIP ไฟล์ ซึ่งเป็นโปรแกรมที่นิยมใช้กันในปัจจุบัน แต่ก็ยังมีจุดที่น่าสนใจที่สุดก็คือในเครื่องคอมพิวเตอร์ที่ใช้กันอยู่ในปัจจุบันยังมีความสามารถในการส่งผ่านข้อมูลช้า เช่นฮาร์ดดิสก์อย่างไวที่สุดก็สามารถส่งผ่านข้อมูลได้เพียง 100MB ต่อวินาทีแค่นั้นเอง และเมื่อส่งข้อมูลออกมาจากฮาร์ดดิสก์แล้ว ก็มักจะไปติดรออยู่บนบัสต่าง ๆ เหมือนกับติดไฟแดงบนถนน เพราะในคอมพิวเตอร์เครื่องหนึ่ง ๆ จะมีอุปกรณ์ที่ต้องใช้บัสร่วมกันอยู่หลายตัว เปรียบเหมือนกับการที่รถมากมายต้องมาใช้ถนนร่วมกัน

การบีบอัดข้อมูลจะเข้ามามีส่วนร่วมในตอนนี้ เช่น Zip drive สามารถส่งผ่านข้อมูลได้ 8 Mb/second เปรียบเสมือนข้อมูลส่วนที่เคลื่อนไหวไม่ทันก็ต้องรอ ...

เทคนิคแบบนี้เรียกว่า เทคนิค "รู้กัน" เช่นเรื่องของพนักงานขายสองคนที่ชอบเล่าเรื่องตลกกัน โดยการใช้รหัส เช่นเขา ตั้งรหัส (อัด) เรื่องว่า 142 ซึ่งหมายถึง เรื่องเจ้านายอารมณ์เสีย เพราะรถเสียกลางทางด่วนอีกแล้ว เมื่อพวกลูกน้อยเจอหน้ากันและเอ่ยโจ๊กเรื่อง 142 ขึ้นมา เมื่อทุกคนได้ยินแล้วก็เฮ เพราะ ถอดรหัส (ขยาย) ออกมาแล้วก็เป็นที่เข้าใจกันว่า วันนี้เจ้านายจอมเหนียวรถเสียกลางทางด่วนอีกแล้ว หรืออีกตัวอย่างหนึ่งก็คือ การจดชวเลขซึ่งเป็นวิชาของพวกเลขานุการ ที่จดบันทึกเป็นคำย่อต่าง ๆ ซึ่งก็เป็นการเปรียบเทียบให้เห็นถึงการอัดขยายข้อมูล Compressor-DECompressor ซึ่งเรียกย่อลงมาว่าการทำ CODEC นั่นเอง.

ตัวอย่างของการบีบอัดข้อมูลแบบง่าย ๆ

CODEC นั้นอาจจะเป็นขบวนการที่ทำขึ้นโดยซอฟต์แวร์หรืออาจจะใช้ไอซีที่ออกแบบพิเศษมาช่วยจัดการก็ได้ แต่ก็สรุปออกมาเป็น 2 แบบ คือ

Losses Method ซึ่งจะทำการบีบอัดข้อมูลแล้วขยายกลับมาใช้งานใหม่ โดยไม่เสียราย-ละเอียดของข้อมูลเลย

Lossy Schemes จะเป็นการบีบอัดข้อมูล โดยกรองเอารายละเอียดบางส่วนทิ้งไปก่อน แล้วจึงบีบอัดข้อมูลลงไป ข้อมูลที่นำเอากลับมาขยายจะไม่ดีเท่ากับของเดิมเพราะมีการทิ้งเอารายละเอียดบางส่วนไปตั้งแต่ตอนแรกแล้ว

และอีกเหตุผลหนึ่งที่อุตสาหกรรมต่าง ๆ ต้องทำการวิจัยค้นหาวิธีการบีบอัดข้อมูลก็เพราะว่า อุตสาหกรรมบันเทิงในอเมริกามีความพยายามที่จะขยายธุรกิจบันเทิงเข้าไปสู่บ้านของผู้รับชม

รายการโดยตรง โดยที่ผู้ชมสามารถเข้าเลือกชมรายการที่วันเวลาต่าง ๆ โดยไม่ต้องรอเวลาออกอากาศของทางสถานี เช่น การดูภาพยนตร์ผ่านสายโทรศัพท์ (VIDEO over Phone line) และแม้แต่ระบบการแพร่ภาพในระบบความชัดสูง (HDTV) ซึ่งก็มีมาตรฐานแตกต่างกันออกไปตามแต่ความต้องการของอุตสาหกรรมนั้น ๆ แต่สำหรับอุตสาหกรรมการลำดับภาพยนตร์และวีดีทัศน์แล้ว จะสนใจอยู่แค่สองมาตรฐานคือ JPEG และ MPEG เท่านั้น

JPEG (อ่านว่า เจ-เพ็ก / JAY-peg) JPEG เป็นมาตรฐานการบีบอัดข้อมูลที่คิดค้นขึ้นราวยุคปลายทศวรรษ 1980 เนื่องจากมีความต้องการที่จะย่อภาพสีโดยให้คงรายละเอียดใกล้เคียงของเดิม โดยคอมพิวเตอร์จะทำการชักตัวอย่างของจุดภาพในส่วนต่าง ๆ ก่อนที่จะบีบอัด โดยจะตรวจดูว่า ในพื้นที่ของภาพขนาด 8x8 pixel นั้น เมื่อทำการคำนวณแล้ว น่าจะมีสีใดมากที่สุด จากนั้นก็ยุบพื้นที่ขนาด 8x8 Pixel ให้เหลือเป็นสีที่ต้องการเพียง 1 Pixel เท่านั้น JPEG มักจะนำมาใช้กับภาพเดี่ยวที่อัตราส่วนประมาณ 25:1, 40:1 จนถึง 100:1

Motion-JPEG หรือ M-JPEG บางทีก็ใช้แก้ไขตัดต่อภาพวิดีโอที่ละเฟรม JPEG

ความเป็นมา MPEG-4

MPEG ย่อมาจาก Moving Pictures Experts Group เป็นกลุ่มของคณะกรรมการที่ทำงานภายใต้องค์การมาตรฐาน ISO (International Standards Organization) เพื่อสร้างมาตรฐานสำหรับการบีบอัดข้อมูลวิดีโอและออดิโอแบบดิจิตอล (digital data compression) ซึ่งเป็นการกำหนดมาตรฐานขั้นพื้นฐานสำหรับการกำหนดรูปแบบของสายข้อมูลระดับบิตของวิดีโอและออดิโอ และรวมไปถึงวิธีการขยายกลับข้อมูลที่ถูกบีบอัด MPEG ตั้งขึ้นในปีค.ศ. 1988 ต่อมาในเดือนสิงหาคมปี ค.ศ.1993 MPEG ได้ออกมาตรฐาน MPEG-1 สำหรับการบีบอัดวีดีโอและออดิโอที่ 1.5 Mb/s ขณะเดียวกันในปี ค.ศ. 1990 ก็ได้มีการเริ่มคิดมาตรฐานใหม่ MPEG-2 โดย MPEG-1 มุ่งเน้นที่ CD-ROM แอพพลิเคชั่น ส่วนใน MPEG-2 ใช้กับภาพและเสียงคุณภาพสูงในช่วง 2 Mb/s ถึง 30 Mb/s เหมาะกับดิจิตอลTV และ HDTV แอพพลิเคชั่น

จากการคาดการในเร็วนี้อุตสาหกรรมการสื่อสาร คอมพิวเตอร์ และ TV กำลังจะมาร่วมกัน MPEG จึงได้เริ่มคิดมาตรฐานใหม่ MPEG-4 ในปี ค.ศ.1994 มีความ interactive, การบีบอัดสูง, มีความเป็นสากลในการเข้าใช้ในการไปใช้ในหลายแพลตฟอร์มสำหรับ video content บิตเรตเป้าหมายของวีดีโอมาตรฐาน 5-64 kb/s สำหรับแอพพลิเคชั่นบนมือถือและสูงไปถึง 2 Mb/s TV/ฟิล์ม แอพพลิเคชั่น ซึ่งใน MPEG-4 แบ่งเป็นหลาย profile ในแต่ล่ะ profile จะแบ่งหลาย level

MPEG-4 วีดีโอโคเดคที่เราใช้ในการทดลองของเราคือ XviD ที่เรา port มาทำงานบนระบบปฏิบัติการซิมเบียน (Symbian OS) ซึ่งเป็น Simple Profile Simple Level MPEG-4

2.2.2 ชนิดของเฟรมใน MPEG

สามชนิดของเฟรมใน MPEG-4

Intra frames (I-frames)

คำว่า "intra" หมายถึงภายในดังนั้น intraframe หรือ I-frame เป็นเฟรมที่เข้ารหัสโดยใช้ข้อมูลจากภายในเฟรมนั้นอย่างเดียว หรือจะกล่าวอีกอย่างหนึ่งว่าคือเฟรมที่ถูกบีบอัดโดยไม่ข้อมูลจากเฟรมอื่น-ไม่เกี่ยวกับการบีบอัดในเชิงเวลา การบีบอัดของ I-frame คล้ายกับการบีบอัดของรูปภาพใน JPEG ดีโค้ดเดอร์ต้องเริ่มจาก I-frame ไม่ใช้ P-frame I-frame จะใส่ในทุกๆ 12-15 frames

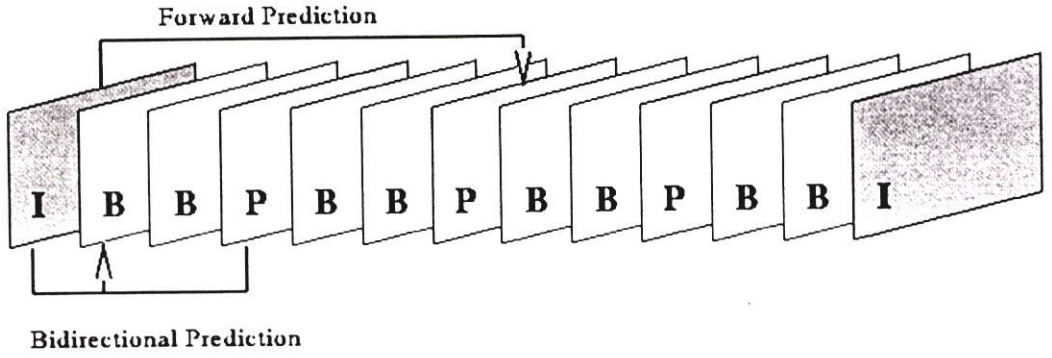
Intra เป็นหลักการที่สำคัญใน MPEG ซึ่งไม่ใช้แค่ตัวนำหน้า ตรงกันข้ามของ intraframe เฟรมที่ใช้ข้อมูลจากเฟรมอื่นโดยอธิบายใน inter หรือ non-intra

Predicted frames (P-frames)

เป็นการบีบอัดจากความแตกต่างๆของ I-frame หรือ P-frame ก่อนหน้า P-frame ใหม่จะเป็นการทำนายจาก I-frame หรือ P-frame ก่อนหน้า การทำนายจะทำกับทุกพิกเซล P-frame ใช้ motion prediction และ DCT encoding ผลของ P-frames จะมีอัตราการบีบอัดที่ดีกว่า I-frame แต่ขึ้นกับจำนวนการเคลื่อนไหน ความแตกต่างระหว่างจากที่ทำนายกับภาพปัจจุบันหรือ prediction error จะใช้ในการบีบอัด ค่า error นี้จะมีขนาดเล็กเพราะค่า pixel จะเปลี่ยนไม่มากในบริเวณเล็กๆ ดังนั้นการบีบอัดค่า error นี้จะดีกว่าบีบอัดค่า frame นั้น

Bidirectional frames (B-frames)

เป็นการบีบอัดจากความแตกต่างจาก I-frame หรือ P-frame ก่อนหรือหลัง B-frame จะทำนายเหมือน P-frame แต่ทุก block อย่างใดอย่างหนึ่งของ I-frame หรือ P-frame ก่อนหน้าหรือ I-frame หรือ P-frame ของเฟรมต่อไป B-frame ใช้ motion prediction และ DCT encoding เพราะ B-frame ต้องใช้ทั้งเฟรมก่อนและหลังสำหรับการถอดการบีบอัด ลำดับเฟรมที่ถูกอ่านไม่ใช้เหมือนกับลำดับแสดง ในการบีบอัดนี้จะให้คุณภาพที่ดีกว่าเมื่อเทียบกับ P-frame เพราะเป็นไปได้ที่ทุก macroblock ถูกเปรียบเทียบมาจากเฟรมก่อนหรือหลัง



รูปที่ 2.1 Typical MPEG Group Of Pictures Sequence, 12 Frames / GOP

2.2.3 เอ็นโค้ดเดอร์และดีโค้ดเดอร์ SP@L0

เอ็นโค้ดเดอร์ : การบีบอัดแบบMPEGมีขั้นตอนต่าง ๆ ดังนี้

อันดับแรก การวิเคราะห์ความเคลื่อนไหวจะนำมาใช้กับแต่ละแมกโครบล็อก นอกจากการวิเคราะห์การเคลื่อนไหวจากกรอบภาพที่อยู่ข้างหน้า MPEG ยังอนุญาตให้การทำนายทำกับกรอบภาพที่ผ่านมากแล้วหรือยังมาไม่ถึงหรือผสมกันได้อีกด้วย

เนื่องจากวัตถุในกรอบภาพอาจจะไม่ได้เคลื่อนไหวอย่างคงที่จากกรอบภาพหนึ่งสู่กรอบภาพหนึ่ง แต่ละแมกโครบล็อกจึงอนุญาตให้มีเวกเตอร์การเคลื่อนที่ได้สองเวกเตอร์ (อันหนึ่งสัมพันธ์กับกรอบภาพที่ผ่านมา อีกอันหนึ่งสัมพันธ์กับกรอบภาพในอนาคต) ในการทำนายจากกรอบภาพในอนาคต กรอบภาพที่เพิ่มเข้ามาจะต้องถูกเก็บไว้ชั่วคราวและภาพที่ถูกเข้ารหัสจะไม่เรียงตามลำดับ การวิเคราะห์ความเคลื่อนไหว อนุญาตให้มีพิสัยได้มาก ๆ (ถึง +/- 1023) และมีความละเอียดระดับครึ่งจุดภาพได้ การใช้ตัวกรองแบบวนกลับแบบ H.261 จะไม่รวมอยู่ใน MPEG เนื่องจากเวกเตอร์การเคลื่อนที่แบบครึ่งจุดภาพก็ตอบสนองความต้องการได้เช่นเดียวกัน

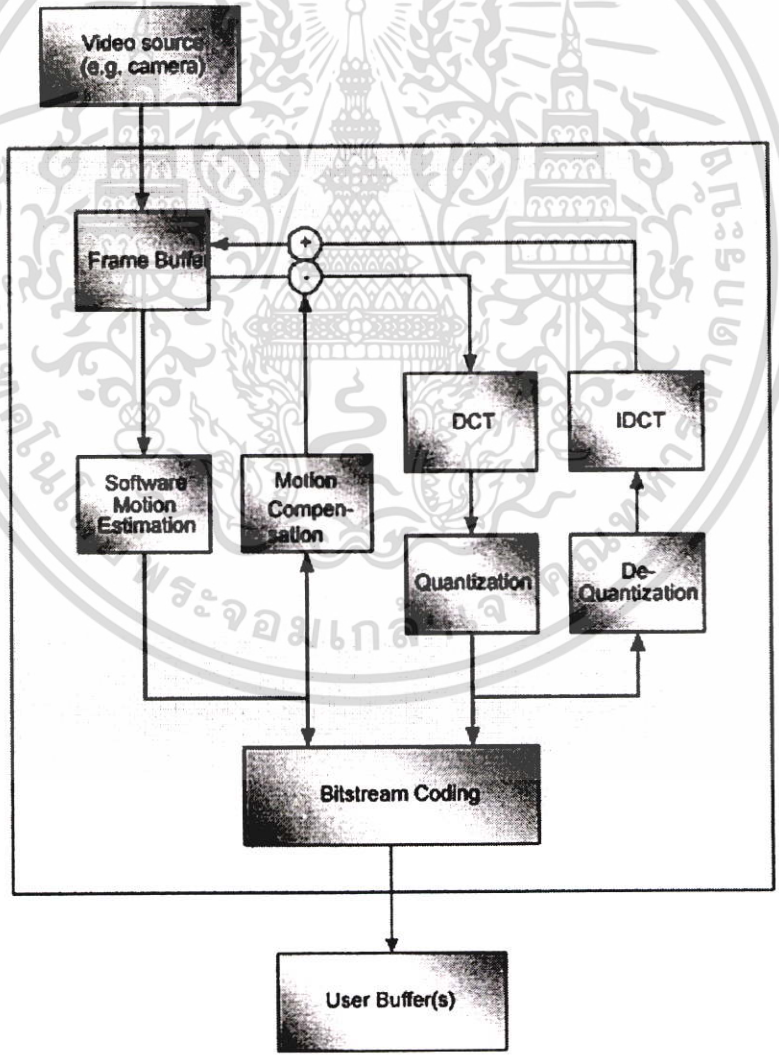
ลำดับต่อมา การตัดสินใจแบบ 4 ทางจะถูกทำขึ้น MPEG จะให้มีรูปแบบการทำนายจาก 1) ความแตกต่างทางคณิตศาสตร์ระหว่างแมกโครบล็อกปัจจุบันกับระยะแมกโครบล็อกจากกรอบภาพที่ผ่านมา 2) กรอบภาพอนาคต 3) ค่าเฉลี่ยระหว่างกรอบภาพอดีตและอนาคตที่ถูกเข้ารหัสไว้แล้ว หรือ 4) เข้ารหัสแมกโครบล็อกปัจจุบันแบบโดด ๆ อย่างเดียว การตัดสินใจที่แตกต่างสามารถทำกับแต่ละแมกโครบล็อกโดยขึ้นกับข้อจำกัดต่อไปนี้

คีย์เฟรม (Keyframe) เรียกว่า อินทราเฟรม (Intraframe) หรือ ไอเฟรม (I-frame) นี้จะไม่ให้มีแมกโครบล็อกที่เกิดจากการทำนายใด ๆ ถูกเข้ารหัสไว้เพื่อจะให้การเข้าถึงแบบสุ่มภายในกระแสวีดิทัศน์ได้กรอบภาพที่ถูกทำนายแบบไปข้างหน้า (P-frame) จะให้แมกโครบล็อกได้รับการทำนายจาก P-frame ที่ผ่านมา หรือ I-frame หรือแมกโครบล็อกที่เข้ารหัสมาโดด ๆ อย่างเดียว I-frame และ P-frame จะถูกใช้เป็นกรอบภาพอดีตและอนาคตสำหรับกรอบภาพที่ถูกทำนายแบบสองทิศทาง (เรียกว่า B-frame) B-frame จะอนุญาตให้มีแมกโครบล็อกได้ทั้ง 4 ชนิด

DCT ขนาด 8x8 จะถูกนำมาใช้กับแต่ละบล็อกที่เกิดจากความแตกต่างทางคณิตศาสตร์ หรือแมกโครบล็อกปัจจุบัน ในการทำควันไทซ์ MPEG จะใช้ตารางทั้งคู่ (เหมือนกับ JPEG) และส่วนประกอบการปรับขนาด (เหมือนกับ H.261) เนื่องจากช่อง DC เป็นช่องที่สำคัญที่สุด มันจึงถูกควันไทซ์เป็นขนาเที่คงที่ที่ 8 บิต

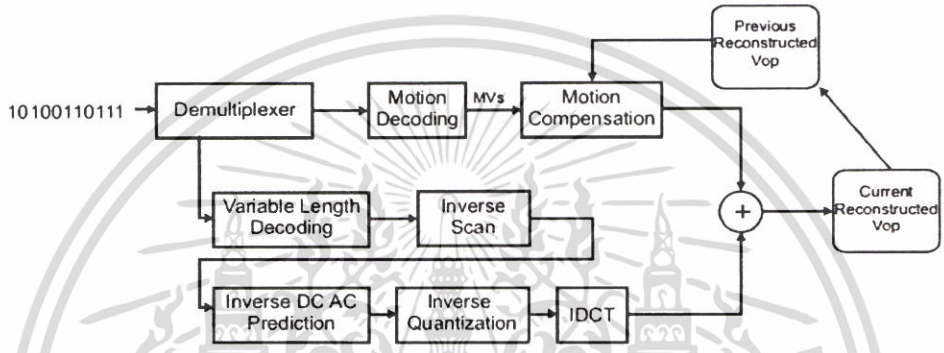
เนื่องจากผลที่มองเห็นได้จากการควันไทซ์ของช่องความถี่จะมีความแตกต่างกันสำหรับบล็อกที่ทำนายมาได้กับบล็อกปัจจุบัน MPEG จึงอนุญาตให้มีแมตทริกซ์สองตาราง (หนึ่งตารางต่อชนิด) โดยทั่วไปแมตทริกซ์จะถูกตั้งไว้ครั้งเดียวสำหรับลำดับของรูปภาพหนึ่ง และขนาดของการควันไทซ์จะถูกปรับเพื่อให้ควบคุมอัตราส่วนการบีบอัดได้

ขั้นตอนสุดท้ายของการบีบอัดก็คือการทำซิกแซกสแกน (Zigzag Scan) การเข้ารหัสรันเลงต์ (Run-length Encoding) แลการใส่รหัสเอนโทรปี (Entropy Coding) เช่นเดียวกับ H.261 MPEG จะระบุตารางเข้ารหัสอัฟฟ์แมนที่ตายตัวในการเข้ารหัสแอนโทรปีนี้



รูปที่ 2.2 Simple Profile Simple Level (SP@L0) Mpeg-4 วีดีโอเอ็นโค้ดเดอร์

ดีโค้ดเดอร์ : MPEG-4 SP@L0 ดีโค้ดเดอร์แสดงในรูปที่ 2.3 มันสามารถรองรับทั้งการบีบอัดเฟรม (frame) สี่เหลี่ยมแบบ intra (I) และแบบ predictive (P) เฟรม I ประกอบด้วยข้อมูลพื้นผิว (texture) ซึ่งจะถอดการบีบอัดข้อมูลนี้จะใช้ inverse quantization และ IDCT [3] ส่วนเฟรม P จะบีบอัดโดยการทำนาย motion compensation จากเฟรม P หรือ I ก่อนหน้า ดังนั้นในการสร้างภาพจากเฟรม P หมายถึงต้องเอา motion compensated และ motion estimation มารวมกันซึ่งเป็นกระบวนการที่ซับซ้อนและใช้หน่วยความจำเยอะมาก การถอดการบีบอัดจะได้เป็นกลุ่ม MacroBlock (MB) ใน MB จะประกอบด้วยบล็อกข้อมูลสี่ luminance (Y1, Y2, Y3, และ Y4)และสอง chrominance (U และ V) ทุกอันมีขนาด 8x8 พิกเซล [2]



รูปที่ 2.3 Simple Profile Simple Level (SP@L0) Mpeg-4 วีดีโอดีโค้ดเดอร์ [3]

2.3 สถาปัตยกรรม ARM920T โปรเซสเซอร์

ARM โปรเซสเซอร์เป็นโปรเซสเซอร์ที่ได้รับความนิยมอย่างมากในระบบสมองกลฝังตัวโดยเฉพาะอย่างยิ่งในมือถือและพีดีเอ ซึ่ง ARM โปรเซสเซอร์จะแบ่งออกเป็นหลายซีรี่เพื่อรองรับความต้องการที่หลากหลาย งานวิจัยเรามุ่งเน้นที่จะใช้ได้จริงบนมือถือและโค้ดของเราทำงานบนระบบปฏิบัติการซิมเบียน ซึ่งมือถือที่ใช้ระบบปฏิบัติการนี่จะใช้ ARM ซีรี่9 ซึ่งใน ARM ซีรี่ 9 มีสองตัวที่ใช้ในมือถือและพีดีคือ ARM920T และ ARM922T ทั้งสองตัวนี้มีสถาปัตยกรรมการทำงานที่เหมือนกันต่างกันตรงที่ขนาดของแคช (cache) ที่ ARM920T มีขนาด 16 กิโลไบต์ ส่วน ARM922T ขนาด 8 กิโลไบต์ ในงานวิจัยเราได้เลือกศึกษา ARM920T เพราะมีแคชขนาดที่ใหญ่กว่า ARM922T เพื่อที่จะได้เห็นผลในการทดลองที่ชัดเจน

2.3.1 เกี่ยวกับ ARM920T

ARM920T เป็นสมาชิกหนึ่งในตระกูล ARM9TDMT ที่เป็น microprocessor ที่ใช้ในงานทั่วไป ซึ่งประกอบด้วย

ARM9TDMI (core)
ARM940T (core บวก cache และ protection unit)

ARM920T และ ARM922T (core บวก cache และ MMU)

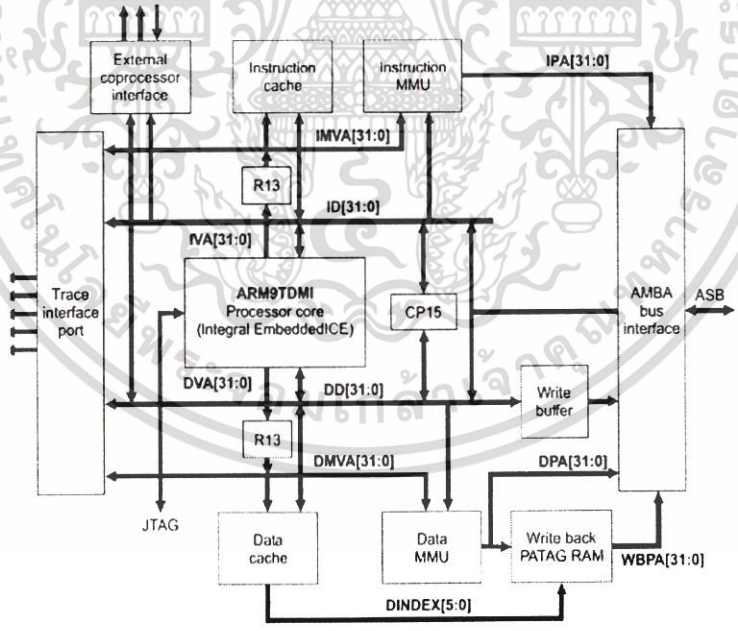
ARM9TDMI โพรเซสเซอร์ใช้สถาปัตยกรรม Harvard ที่มีห้า stage ของ pipeline ประกอบด้วย Fetch, Decode, Execute, Memory และ Write. ARM9TDMI สามารถใช้เป็น standalone core ได้ นั้นคือสามารถที่จะฝั่งไปในอุปกรณ์ที่ซับซ้อนได้ โดยการใช้เป็น standalone core จะมี simple bus interface ที่อนุญาตให้ออกแบบระบบ cache และ memory ได้

ตระกูล ARM9TDMI microprocessor รองรับทั้งชุดคำสั่ง 32 bits ARM และ 16 bit Thumb ทำให้เราสามารถเลือกได้ว่าจะเอาโค้ดที่มีประสิทธิภาพสูงหรือจะเอาโค้ดที่มีขนาดเล็ก

ARM920T โพรเซสเซอร์ใช้สถาปัตยกรรม Harvard cache จุดประสงค์ใช้กับ multiprogramer application ที่มี memory management, ประสิทธิภาพสูง และใช้พลังงานน้อย มีการแยก instruction cache และ data cache ขนาด 16 KB ที่มี line ล่ะ 8 word ARM920T มีการสร้าง enhanced ARM architecture v4 MMU ที่มีการจัดการการตรวจสอบสิทธิในการ access และ translation สำหรับตำแหน่ง instruction และ data

2.3.2 Block diagram ของฟังก์ชันการทำงานของโพรเซสเซอร์

รูป 2.4 แสดง Block diagram ของฟังก์ชันการทำงานของ ARM920T โพรเซสเซอร์



รูปที่ 2.4..ARM920T ไดอะแกรม

สถาปัตยกรรม ARM920T เป็น RISC โปรเซสเซอร์ขนาด 32 บิตที่ใน ARM920T เป็นโพรเซสเซอร์ภายในประกอบด้วย ARM9TDMI โพรเซสเซอร์กับ:

-แคชของชุดคำสั่งและแคชของข้อมูลขนาด 16 กิโลไบต์

-MMUหรือหน่วยจัดการหน่วยความจำคำสั่งและข้อมูล (Instruction and data Memory Management Units)

-ปัฟเฟอร์ก่อนเขียนลงหน่อยความจำ (Write buffer)

-AMBA™ (Advanced Microprocessor Bus Architecture)

บัสอินเตอร์เฟส

-บัสในการส่งข้อมูลและคำสั่งขนาด 32 บิต

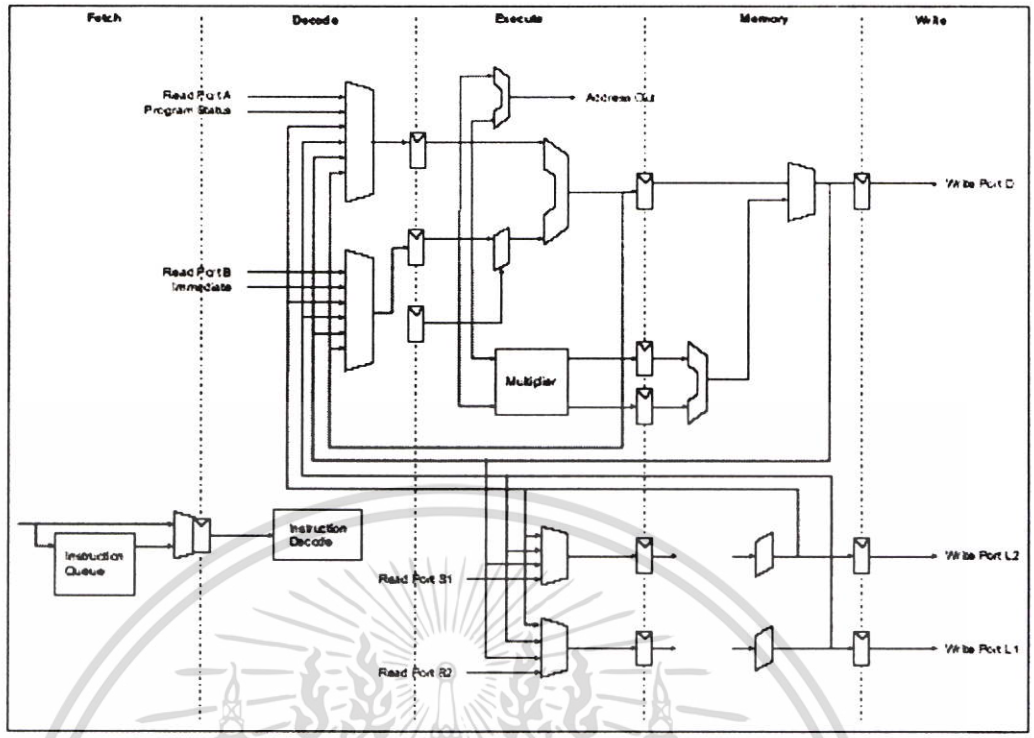
-Embedded Trace Macrocell (ETM) อินเตอร์เฟส

2.3.3 ARM920T Macrocell

ARM920T Macrocell อยู่บนพื้นฐานสถาปัตยกรรม ARM9TDMI Harvard โดยมีไปป์ไลนขนาดห้าขั้น: เฟ็ตช์ (fetch), ดีโค้ด (decode), ประมวลผล (execute), เข้าใช้หน่อยความจำ (memory access) และเขียนรีจีสเตอร์ (write register back) แสดงดังรูปที่ 2.5

เพื่อลดแบนด์วิดท์และความหนาแน่นข้อมูลที่หน่วยความจำหลัก ARM920T จึงมี

- แคชชุดคำสั่ง
- แคชข้อมูล
- MMU
- TLBs
- Write buffer

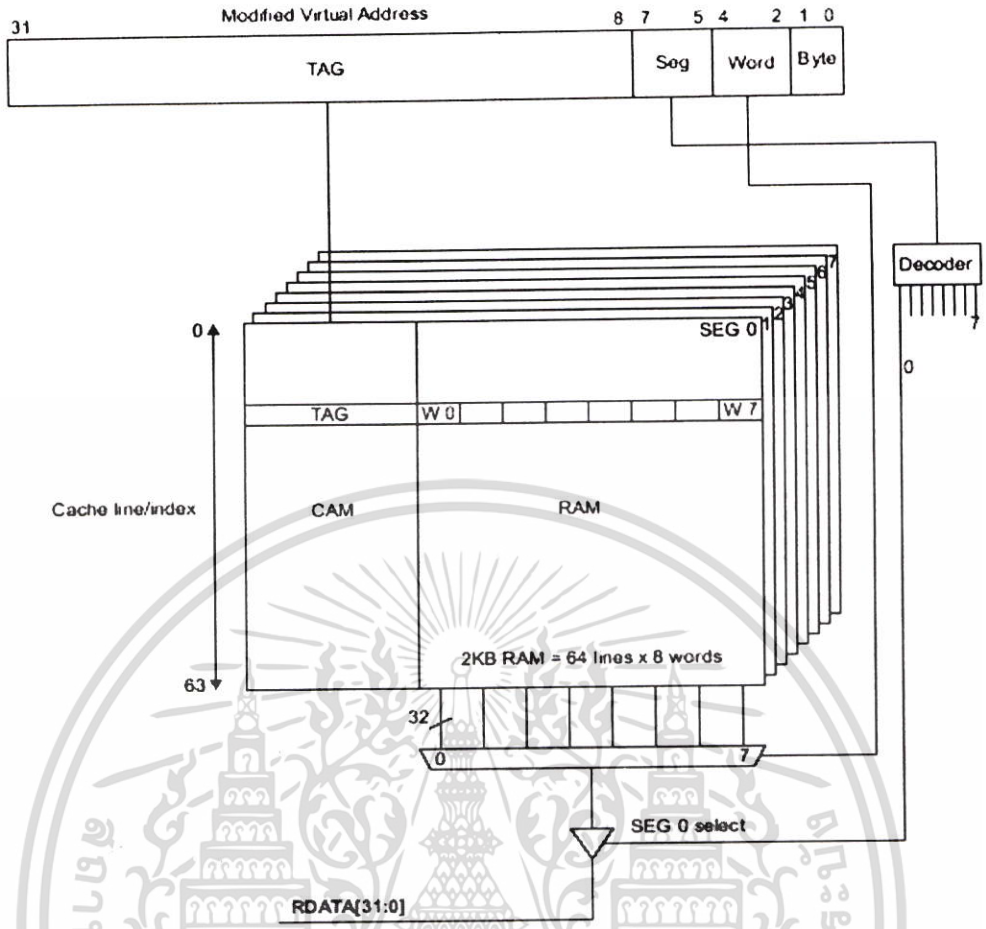


รูปที่ 2.5 ไปป์ไลนของ ARM9TDMI

2.3.4 แคชของ ARM920T

ทั้งแคชของคำสั่งและของข้อมูลทั้งสองมีขนาด 16 กิโลไบต์ มีขนาดรายละ 8 word และเป็น 64-way set associative [5] จึงมี 8 Segments และมีบัสข้อมูลขนาด 32 บิตต่อทุกแคชไปที่ ARM9TDMI

และเป็นจะ allocate ข้อมูลบนแคชเมื่ออ่านไม่เจอในแคช และยังใช้นโยบาย Write thought และ Non-allocate เมื่อการเขียนข้อมูลแล้วไม่เจอในแคช (Write miss) ส่วนการเขียนข้อมูลที่มีในแคช (Write hit) จะใช้ Write back



รูปที่ 2.6 โครงสร้างการจัดการของแคช

2.3.5 แอพพลิเคชั่นที่ใช้บน ARM920T

ระบบปฏิบัติการ

-EPOC (พวก Symbian OS)

-Linux

-WindowsCE

แอพพลิเคชั่นไร้สายประสิทธิภาพสูง

-สมาตร์โฟน

-พีดีเอ

เนตเวิร์กกิ้งแอพพลิเคชั่น

การเข้ารหัสและคล้ายรหัสของเสียงและภาพ

2.4 ระบบปฏิบัติการซิมเบียน (Symbian OS)

ระบบปฏิบัติการซิมเบียน เป็นระบบปฏิบัติการที่ออกแบบมาเพื่อรองรับเทคโนโลยีการสื่อสารแบบไร้สายบนโทรศัพท์เคลื่อนที่ ซึ่งสามารถที่จะทำงานได้โดยไม่ขึ้นต่อฮาร์ดแวร์ (Hardware) ของโทรศัพท์เคลื่อนที่ที่มีความแตกต่างกัน และสามารถจัดการกับการพัฒนาอย่างต่อเนื่องในอนาคตได้ ช่วยในด้านของการส่งข้อมูลของโทรศัพท์เคลื่อนที่ อีกทั้งยังเป็นระบบที่ใช้งานได้ง่าย มีความปลอดภัยสูง ช่วยประหยัดพลังงาน และใช้หน่วยความจำที่มีขนาดเล็ก เพื่อรองรับกับโทรศัพท์เคลื่อนที่ทั้งในปัจจุบันและอนาคต

ทั้งนี้ระบบปฏิบัติการซิมเบียนนั้นยังเป็นระบบปฏิบัติการเปิด ที่อนุญาตให้โปรแกรมเมอร์รวมถึงผู้ใช้สามารถพัฒนาซอฟท์แวร์ต่างๆ บนระบบปฏิบัติการซิมเบียนได้เอง เรียกได้ว่าในอนาคตจะมีแอพพลิเคชั่นมากมายที่ถูกสร้างขึ้นบนระบบปฏิบัติการซิมเบียน และยังส่งผลให้เป็นตลาดผลิตภัณฑ์ซอฟท์แวร์ที่ยิ่งใหญ่ในอนาคตอีกด้วย

2.4.1 ความเป็นมาและพัฒนาการของระบบปฏิบัติการซิมเบียน

ซิมเบียน คือ บริษัทผลิตซอฟท์แวร์ (Software) ที่เป็นผู้นำในด้านการผลิตซอฟท์แวร์ที่รองรับเทคโนโลยีการสื่อสารแบบไร้สาย (Wireless) ซึ่งผลิตภัณฑ์ที่ได้รับการยอมรับจากบริษัทคู่ค้า ก็คือ ระบบปฏิบัติการซิมเบียนนั่นเอง

ระบบปฏิบัติการซิมเบียนเกิดขึ้นในเดือนมิถุนายน ปี ค.ศ.1998 ซึ่งในตอนนั้นมีพันธมิตรร่วมกัน 4 ราย คือ Ericsson, Motorola, Nokia และ Psion มีสำนักงานใหญ่อยู่ที่ประเทศอังกฤษ

ถัดมาในปี ค.ศ. 1999 ซิมเบียนก็ได้พันธมิตรรายใหม่นั่นคือ Matsushita (Panasonic) ในปี ค.ศ. 2000 พันธมิตรของซิมเบียน ก็มากขึ้นอีกโดยมีการจับมือกับ Sony, Sanyo และ Kenwood รวมถึงการเปิดตัวโทรศัพท์ที่ใช้ระบบปฏิบัติการซิมเบียน เครื่องแรกในรูปแบบของสมาร์ทโฟนนั่นก็คือ Ericsson R380 ซึ่งมีคุณสมบัติการใช้งานที่มากมาย และยังมีทำงานด้วยหน้าจอระบบสัมผัสอีกด้วย

ในปี 2001 Nokia ที่เคยใช้ระบบปฏิบัติการ GEOS (Graphical Environment Operating System) ได้เปลี่ยนมาใช้ระบบปฏิบัติการซิมเบียนแทนระบบปฏิบัติการเดิม โดยทาง Nokia ได้พัฒนาคอมมิวนิเคเตอร์ (Communicator) รุ่นใหม่ออกมาคือ Nokia 9210 และโทรศัพท์ที่ใช้แพลทฟอร์ม (Platform) Series 60 รุ่นแรกคือ Nokia 7650 ซึ่งระบบปฏิบัติการที่ใช้นั้นแตกต่างจาก Ericsson R380 เนื่องจากระบบปฏิบัติการซิมเบียนของ Nokia 9210 และ Nokia 7650 นั้นเป็นระบบปฏิบัติการเปิด (Open Symbian OS) คือสามารถที่จะนำโปรแกรมอื่นที่สามารถทำงานบนระบบปฏิบัติการซิมเบียนมาลงเพิ่มในเครื่องได้

ในปี 2002 ทางซิมเบียนได้มีการพัฒนาระบบปฏิบัติการรุ่นใหม่ที่เรียกว่าระบบปฏิบัติการซิมเบียนเวอร์ชัน 7.0 และทาง Sony Ericsson ก็เข้ามาเป็นพันธมิตรและเป็นหุ้นส่วนรายใหญ่ รวมทั้งทาง Sony Ericsson ก็เปิดตัวสมาร์ทโฟนรุ่นล่าสุด คือ Sony Ericsson P800 และในปี 2003

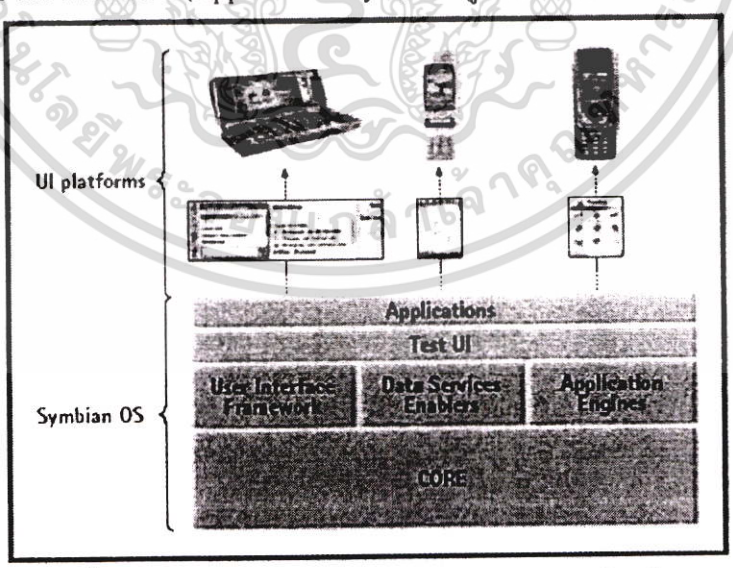
ก็ได้มีการพัฒนาระบบปฏิบัติการต่อจากเวอร์ชัน 7.0 เป็นเวอร์ชัน 7.0s โดยได้เพิ่มส่วนของการรองรับระบบโทรศัพท์แบบดับบลิวซีดีเอ็มเอ (WCDMA: Wideband Code Division Multiple Access) และได้มีการปรับปรุงการรองรับ Java เป็น MIDP 2.0

และในปี 2004 ทางซิมเบียนก็ได้ทำการเปิดตัวระบบปฏิบัติการซิมเบียนเวอร์ชันใหม่ล่าสุดนั่นก็คือ เวอร์ชัน8.0 ซึ่งระบบปฏิบัติการตัวนี้ได้ถูกสร้างขึ้นมาเพื่อพยายามลดค่าใช้จ่ายที่เกิดขึ้นจากการพัฒนาโทรศัพท์เคลื่อนที่ในอนาคต

2.4.2 โครงสร้างของระบบปฏิบัติการซิมเบียน

ในส่วนโครงสร้างของระบบปฏิบัติการซิมเบียนประกอบไปด้วยส่วนต่างๆ ที่สำคัญดังนี้

1. แกนหลัก (Core) ของระบบปฏิบัติการซิมเบียน เป็นแกนหลักในการทำงานของอุปกรณ์ เช่น เคอร์เนล (Kernel) ไฟล์เซิร์ฟเวอร์ (File server) การจัดการด้านหน่วยความจำ (Memory management) และไดรเวอร์ของอุปกรณ์ (Device driver) ซึ่งส่วนประกอบต่างๆ นั้นสามารถที่จะเพิ่มหรือลดได้ขึ้นอยู่กับอุปกรณ์แต่ละชนิด
2. ชั้นของระบบ (System Layer) เป็นส่วนที่จัดการเกี่ยวกับการคำนวณ การติดต่อสื่อสาร เช่น TCP/IP, IMAP4, SMS และ การจัดการฐานข้อมูล
3. แอพพลิเคชันเอนจิน (Application Engine) ซึ่งอยู่บนชั้นของระบบ ทำให้นักพัฒนาโปรแกรมสามารถสร้างส่วนติดต่อกับผู้ใช้ (User Interface) ติดต่อกับข้อมูลได้
4. ส่วนติดต่อกับผู้ใช้ โดยผู้ผลิตแต่ละรายสามารถที่จะสร้างขึ้นได้ เช่น แพลทฟอร์ม Series 60 ของบริษัท Nokia
5. ชั้นของแอพพลิเคชัน (Application Layer) ซึ่งอยู่ส่วนบนของส่วนติดต่อกับผู้ใช้



รูปที่ 2.7 โครงสร้างโดยรวมของระบบปฏิบัติการซิมเบียน

2.4.7 แอพพลิเคชันเฟรมเวอร์ค (Application Framework)

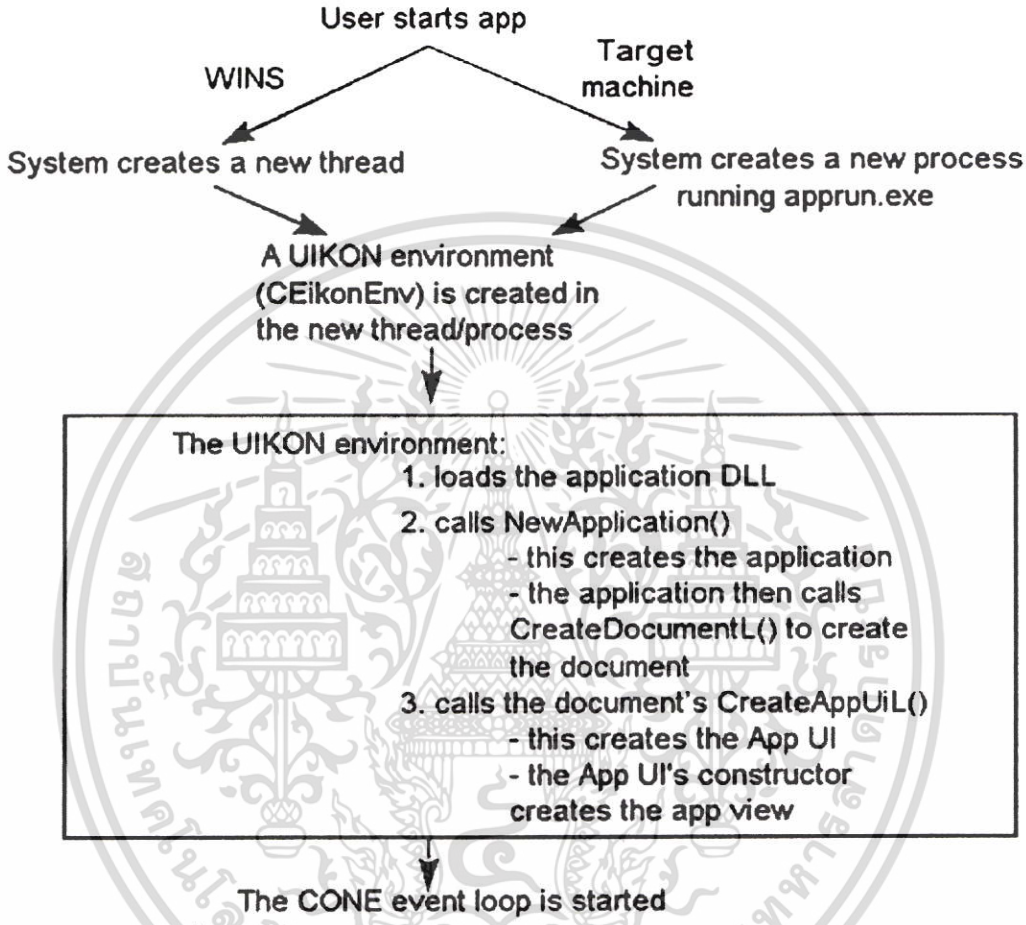
การทำงานของแอพลิเคชันที่ใช้ระบบปฏิบัติการซิมเบียน บนอีมูเลเตอร์ (Emulator) ที่อยู่บน ระบบปฏิบัติการวินโดวส์ (Windows) (ในทางเทคนิคเรียกว่า "WINS" ที่ทำงานแบบกระบวนการเดี่ยว) และอุปกรณ์จริง อย่างเช่น Nokia 3650 จะมีความแตกต่างกันอยู่เล็กน้อย ซึ่งความแตกต่างทำให้ผู้พัฒนาโปรแกรมนั้นมีปัญหากับความไม่สัมพันธ์กันระหว่างอีมูเลเตอร์กับอุปกรณ์จริง โดยเฉพาะผู้พัฒนาโปรแกรมที่เป็นโปรแกรมที่มีความซับซ้อนมาก จะเป็นห่วงเกี่ยวกับความแตกต่างระหว่าง WINS กับอุปกรณ์จริง

กุญแจหลักสำคัญของแอพพลิเคชันเฟรมเวอร์คนี้คือ UIKON ซึ่ง UIKON ก็คือ มาตรฐานของเฟรมเวอร์คที่ทุกๆ อุปกรณ์ซึ่งทำงานอยู่บนระบบระบบปฏิบัติการซิมเบียนต่างมีเหมือนกัน UIKON ไม่เพียงแต่จะมีเฟรมเวอร์คที่จัดเตรียมไว้สำหรับการทำงานของแอพพลิเคชันเท่านั้น แต่ยังมีส่วนที่สนับสนุน การทำงานของคอมโพเนนท์สำหรับการควบคุมทั่วๆ ไปด้วย เช่น กล่องข้อความ (dialog box) ตัวแก้ไขตัวเลข (number editor) และตัวแก้ไขวันที่ (date editor) ซึ่งกลุ่มของแอพพลิเคชันจะสามารถใช้ประโยชน์ได้ขณะทำงาน

แอพพลิเคชันบนระบบระบบปฏิบัติการซิมเบียน ประกอบด้วย 4 คอมโพเนนท์ ซึ่งแต่และส่วนมีความเกี่ยวข้องกับคลาส (Class) ใน UIKON เฟรมเวอร์ค ได้แก่

1. แอพพลิเคชันเชลล์ (Application shell) ซึ่งได้รับการถ่ายทอดคุณสมบัติมาจาก CEikApplication และคลาสนี้จะถูกสร้างขึ้นตอนแรกสุดโดยเฟรมเวอร์ค ทันทีที่ถูกสร้างขึ้นมันจะมีหน้าที่รับผิดชอบต่อการกำหนดค่าเริ่มต้นต่างๆ ให้กับโค้ด (Code) ส่วนที่เหลืออยู่
2. ดอคคิวเมนท์ (Document) ซึ่งได้รับการถ่ายทอดคุณสมบัติมาจาก CEikDocument ส่วนนี้จะทำให้เข้าใจผิดเล็กน้อยในเรื่องที่ว่า ไม่ใช่ทุกแอพพลิเคชันต้องมีดอคคิวเมนท์ที่จะต้องมาเกี่ยวข้องกับมัน เหมือนกับที่แอพพลิเคชันเหล่านี้จะต้องเกี่ยวข้องกับผู้ใช้ตัวอย่างเช่น โปรแกรม Notepad ที่เป็นโปรแกรมเกี่ยวข้องกับดอคคิวเมนท์โดยตรงอย่างชัดเจน ในขณะที่โปรแกรม เช่น นาฬิกา นั้นจะไม่ถูกกำหนดความสามารถในการจัดการดอคคิวเมนท์ทั้งการสร้าง การเปิดเอกสารและการแก้ไขดอคคิวเมนท์ใดๆ ในความเป็นจริงทุกๆ แอพพลิเคชันจะต้องมีคลาส CEikDocument ที่เฟรมเวอร์คจะต้องการสร้างขึ้นมา
3. ส่วนติดต่อผู้ใช้งานของแอพพลิเคชัน (App UI) ซึ่งได้รับการถ่ายทอดคุณสมบัติมาจาก CEikAppUi ซึ่งคลาสนี้จัดเตรียมหน้าที่การทำงานหลักสำหรับทุกๆ แอพพลิเคชัน เช่น การรับมือกับเหตุการณ์ (event handing) การควบคุมเหตุการณ์ (event control) การสร้างการควบคุม การเข้าถึงการเรียกของระบบ (system call) ที่เป็นประโยชน์หลายๆ ตัว เป็นต้น
4. วิว (View) ส่วนนี้คือส่วนที่ผู้ใช้สามารถมองเห็นได้จริงในหน้าจอของอุปกรณ์ มันสามารถใช้งานสำหรับการแสดงข้อมูลอย่างง่าย หรือรวบรวมข้อมูลจากผู้ใช้ในแอพพลิเค

ชันที่ซับซ้อนมากขึ้น ตัวอย่างเช่น ในแอพพลิเคชันประเภทการแก้ไขข้อความ ตัวหนังสือที่ถูกพิมพ์นั้น ก็คือการควบคุมมาตรฐานซึ่งถูกจัดไว้โดย UIKON โดยจัดเก็บอยู่ในส่วนของวิว ในขณะที่ทุกแอพพลิเคชันจะมีอยู่หนึ่งวิวโดยปกติ แต่แอพพลิเคชันที่มีความซับซ้อนมากสามารถมีได้หลายวิว



รูปที่ 2.8 แอพพลิเคชันเฟรมเวอร์คของระบบปฏิบัติการซิมเบียน

2.5 หลักการการออพติไมเซชั่นของARM Compiler

คอมไพเลอร์ ARM สามารถ optimize ได้ระดับสูง โดยเฉพาะขนาดของโค้ดจะมีขนาดเล็ก การ optimization หลักๆของคอมไพเลอร์นี้คือ [16], [17]

2.5.1. Common Subexpression Elimination (CSE)

เป็นการระบุ sub-expression ในโค้ด และใช้ผลลัพธ์ของ sub-expression ที่ได้กับทุก instance ดีกว่าการคำนวณมันใหม่ทุกครั้ง ตัวอย่างเช่น โค้ดอาจใช้ expression a+1 ในหลายที่ ต้องคำนวณค่าทุกครั้งที่ใช้ คอมไพเลอร์จะระบุ sub-expression นี้และคำนวณมันครั้งเดียว และใช้มันหลายครั้ง expression อาจจะซับซ้อนมาก ดังนั้นนี้เป็นการ optimization ที่มีประสิทธิ์ภาพมากของคอมไพเลอร์

2.5.2 Loop invariant Motion (Expression Lifting)

เป็นการยก expression ออกจากลูป คอมไพเลอร์จะระบุ expression ในลูปซึ่งไม่มีการเปลี่ยนแปลงขณะลูปกำลังทำงาน การที่เราคำนวณ expression ใหม่ทุกครั้งทำให้เกิด cost ดังนั้นคอมไพเลอร์จะยก expression จากลูปทำให้คำนวณมันเพียงครั้งเดียว

2.5.3 Live range splitting (for dynamic register allocation)

เป็นการระบุ live state ของ variable ภายในโปรแกรม ตัวอย่างเช่น variable ที่ใช้ในสถานการณ์หนึ่ง เช่นเป็นตัวนับของลูป จากนั้น variable ใช้ในการคำนวณ ถ้าทั้งสองไม่มีความเกี่ยวข้องกัน เราสามารถ allocate ใน register ที่ต่างกัน และเมื่อ variable ตาย (เมื่อค่าของมันไม่ถูกใช้แล้วภายหลัง) register ที่เก็บค่า variable นั้นก็จะถูกไปใช้อย่างอื่น

2.5.4 Constant Folding

แทนที่ constant expression ด้วยค่าที่คอมไพเลอร์คำนวณจาก expression เหล่านั้น

2.5.5 Tail Call Optimization and Tail Recursion

Tailcall เป็นการ call ที่เกิดขึ้นทันทีหลังจาก return ปกติแล้ว function จะถูก call และเมื่อมัน return ไปหา caller แล้วจากนั้น caller มันจะ return อีกที Tailcall optimization จะหลีกเลี่ยงเหตุการณ์นี้โดยการเก็บ saved registers ก่อน jump ไป tailcall ทำให้ function ที่ถูก call จะ return ตรงไป caller ของ caller เลย

คอมไพเลอร์สนับสนุน tail call recursion ซึ่งเป็นไปได้เมื่อ tailcall ทำให้ function เดียวกัน ในกรณีนี้มันเป็นไปได้ที่จะ skip ทั้งลำดับการเข้าและการออก แปลงการ call ใน loop

2.5.6 Cross Jump Elimination

เป็นการรวมของตั้งแต่สอง instance ของโค้ดที่เหมือนกัน ตัวอย่าง หลาย return จาก function จะเกิดจากโค้ดที่บ่อยครั้งจะเหมือนๆกัน และจะ optimize ให้ return เพียงหนึ่งเดียว การ optimization นี้ทำให้ประหยัดพื้นที่

2.5.7 Table Driven Peepholing

ระหว่างการทำการ compile โค้ดไปเป็น ARM หรือ Thumb code จะมีจุดที่พบว่า code จะแทนที่ด้วยโค้ดที่เป็น optimal version โดยทำจากการมองโค้ดผ่าน window ที่เรียกว่า peephole และแทนที่คำสั่งด้วย hand crafted version ตารางของ peephole มีการเพิ่มเติม optimal เข้าไปเพิ่มด้วย และเพิ่มโดยวิศวกรของ ARM

2.5.8 Structure Splitting

Structure splitting เป็นการแบ่งโครงสร้างเป็น component วิธีการนี้ component อาจถูก assign ให้ register ทำให้ access ได้เร็วขึ้น มันจะมีประโยชน์เมื่อ return ตัว structure จาก function structure สามารถถูก return ใน register ดีกว่าบน stack

2.5.9 Conditional Execution (or Branch Elimination)

ARM คอมไพเลอร์ ใช้ conditional execution เพื่อหลีกเลี่ยง branch Conditional execution save ทั้งพื้นที่และเวลาทำงาน เพราะลดหลาย conditional execution

บทที่ 3
วิธีการออพติไมเซชั่นเพื่อความเร็วและลดพลังงาน

3.1 วิธีการเพิ่มความเร็ว

ในส่วนนี้จะเสนอเทคนิคต่างๆที่ใช้ในการออพติไมเซชันเพื่อเพิ่มความเร็ว เช่นการลด function call การลด branch prediction การทำ loop unrolling ซึ่งเป็นการปรับปรุงประสิทธิภาพการช่วยทำให้ pipeline ทำงานได้ดี หรือเป็นการเปลี่ยน control flow ซึ่งทำให้ได้ความเร็วในการประมวลผลที่เร็วขึ้น ซึ่งเทคนิคต่างมีดังนี้

3.1.1 ย้ายการทำงานที่ซ้ำซ้อนในโค้ด

ในขั้นตอนนี้คือการทำการลดการทำงานซ้ำซ้อนที่ไม่จำเป็นต้องทำซึ่งเป็นการทำงานที่ศูนย์เปล่าในตัวอย่าง รูปที่ 3.1 (a) จะเห็นว่าสมการ j*stride+i มีการคำนวณซ้ำซ้อน ในหนึ่งรอบคำนวณซ้ำไปสามครั้ง และค่า j*stride ใน for ลูปของ i จะมีค่าเท่าเดิมเสมอ แต่ต้องคำนวณ j*stride ตลอด ดังนั้นสามารถย้ายความซับซ้อนออก โดยคำนวณ j*stride นอก for ลูปของ i และกำหนดค่าให้ induction variable ชื่อว่า temp และใน for ลูปของ i ก็ให้ คำนวณค่า j*strde + i แล้วกำหนดค่าให้ induction variable ชื่อว่า temp1 และยังมีสมการ 1-rounding ที่ยังทำงานซ้ำซ้อน และยังจะเห็นว่า tot จะทำการ allocate ใหม่ทุกครั้ง เมื่อทำการกำจัดความซับซ้อนออกทั้งหมดจะได้ดังในรูปที่ 3.1 (b)

```
uint32_t i, j;
for (j = 0; j < 8; j++)
{ for (i = 0; i < 8; i++)
  {int32_t tot =
    (int32_t)src[j*stride+i]+(int32_t)src[j*stride+i+1];
    tot = (tot + 1 - rounding) >> 1;
    dst[j * stride + i] = (uint8_t) tot;
  }
}
```

(a)

```
uint32_t i, j;
uint32_t temp,temp1;
int32_t tot,temp_round;
temp_round = 1-rounding;
for (j = 0; j < 8; j++) {
    temp = j * stride;
    for (i = 0; i < 8; i++) {
        temp1=temp+i;
        tot =
            (int32_t)(src[temp1]+src[temp1 + 1]+ temp_round)>>1;
        dst[temp1] = (uint8_t) tot;
    }
}
```

(b)

รูปที่ 3.1 ส่วนของโค้ดจากดีโค้ดเดอร์ของ MPEG-4(a) ส่วนโค้ดที่มีการทำงานซ้ำซ้อน (b) ที่ทำการกำจัดส่วนที่ซ้ำซ้อน

3.1.2 Loop Unrolling

ออพติไมเซชันที่มีประสิทธิภาพและง่ายที่สุดในการปรับปรุงเวลาในการ execution ของ loop คือการทำ unrolling ใน loop สั้นๆที่มีจำนวน iteration คงที่สามารถแทนด้วย unroll ได้และเป็นการกำจัด overhead ของ loop การออพติไมเซชันสามารถตัดลด effective cost ได้ครึ่งหนึ่งของ overhead ของ loop ในตัวอย่างของรูปที่ 2(b) จะเห็นว่าfor ลูปของ i เป็นลูปขนาดเล็ก และมีจำนวน iteration คงที่แน่นอน ซึ่งสามารถทำการ unrolling ได้ดังในรูปที่ 3.2

```
uint32_t i, j;
uint32_t temp;
int32_t tot,temp_round;

temp_round = 1-rounding;
for (j = 0; j < 8; j++) {
    temp = j * stride;
    tot = (int32_t)(src[temp]+src[temp+1]+ temp_round)>>1;
    dst[temp] = (uint8_t)tot;
    tot = (int32_t)(src[temp+1]+src[temp+2]+ temp_round)>>1;
    dst[temp+1]= (uint8_t)tot;
    tot = (int32_t)(src[temp+2]+src[temp+3]+ temp_round)>>1;
    dst[temp+2]= (uint8_t)tot;
    tot = (int32_t)(src[temp+3]+src[temp+4]+ temp_round)>>1;
    dst[temp+3]= (uint8_t)tot;
    tot = (int32_t)(src[temp+4]+src[temp+5]+ temp_round)>>1;
    dst[temp+4]= (uint8_t)tot;
    tot = (int32_t)(src[temp+5]+src[temp+6]+ temp_round)>>1;
    dst[temp+5]= (uint8_t)tot;
    tot = (int32_t)(src[temp+6]+src[temp+7]+ temp_round)>>1;
    dst[temp+6]= (uint8_t)tot;
    tot = (int32_t)(src[temp+7]+src[temp+8]+ temp_round)>>1;
    dst[temp+7]= (uint8_t)tot;
}
```

รูปที่ 3.2 ส่วนของโค้ดจากดีโค้ดเดอร์ของ MPEG-4 ที่ทำการunrolling

3.1.3 การทำกำจัดความซับซ้อน

Non-linear Operator Strength Reduction (NOSR)[2] คือการแทนที่สมการที่เป็น non-liner โดยการรวมกันของ conditional และ induction variable ในตัวอย่างรูปที่ 3.2 จะเห็นว่าสมการ j*stride ก็อยู่ในกรณีนี้ j เป็น loop iterator และ stride เป็น data dependent value แต่ค่า stride ถูกระบุค่าก่อนเข้า loop สมการนี้สามารถแทนโดย induction variable ที่ชื่อ temp ดังรูปที่ 3.3

```
uint32_t i, j;
uint32_t temp =0 ;
int32_t tot,temp_round;

temp_round = 1-rounding;
for (j = 0; j < 8; j++) {
    tot = (int32_t)(src[temp]+src[temp+1]+ temp_round)>>1;
    dst[temp] = (uint8_t)tot;
    tot = (int32_t)(src[temp+1]+src[temp+2]+ temp_round)>>1;
    dst[temp+1]= (uint8_t)tot;
    tot = (int32_t)(src[temp+2]+src[temp+3]+ temp_round)>>1;
    dst[temp+2]= (uint8_t)tot;
    tot = (int32_t)(src[temp+3]+src[temp+4]+ temp_round)>>1;
    dst[temp+3]= (uint8_t)tot;
    tot = (int32_t)(src[temp+4]+src[temp+5]+ temp_round)>>1;
    dst[temp+4]= (uint8_t)tot;
    tot = (int32_t)(src[temp+5]+src[temp+6]+ temp_round)>>1;
    dst[temp+5]= (uint8_t)tot;
    tot = (int32_t)(src[temp+6]+src[temp+7]+ temp_round)>>1;
    dst[temp+6]= (uint8_t)tot;
    tot = (int32_t)(src[temp+7]+src[temp+8]+ temp_round)>>1;
    dst[temp+7]= (uint8_t)tot;
    temp = temp+stride;
}
```

รูปที่ 3.3 ส่วนของโค้ดจากดีโค้ดเดอร์ของ MPEG-4 ที่ทำ NOSR

3.1.4 การทำการลดจำนวนของ branch prediction

การลดจำนวนของ branch prediction โดยการทำ control flow transformation ซึ่งเป็นสิ่งที่สำคัญมากในการทำงานของระบบ pipeline นั้นคือถ้าเราทำการลดจำนวนของ branch prediction ก็จะทำให้ความเร็วในการทำงานเพิ่มขึ้น

ในดีโค้ดเดอร์ของ MPEG-4 บนโทรศัพท์มือถือก็มีเงื่อนไขที่มี branch มาก ซึ่งเป็นลูปที่ภายในมีสมการ branch condition ที่เป็นอิสระจาก loop ซึ่งนั่นคือสามารถใส่ลูปเข้ามาภายใต้ branch condition ได้โดยไม่ทำให้เกิดการละเมิด data dependency ดังรูปที่ 3.4

```
for (i = 0; i < 6; i++) {
    if (cbp & (1 << (5 - i)))   // coded
    {
        Mem::Fill(&block[i * 64], 64 * sizeof(int16_t), 0);
        get_inter_block(bs, &block[i * 64]);

        if (dec->quant_type == 0) {
            dequant_inter(&data[i * 64], &block[i * 64], iQuant);
        } else {
            dequant4_inter(&data[i * 64], &block[i * 64], iQuant);
        }
        idct(&data[i * 64]);
    }
}
```

(a)

```
if (dec->quant_type == 0) {
    for (i = 0; i < 6; i++) {
        if (cbp & (1 << (5 - i)))   // coded
        {   Mem::Fill(&block[i << 6], sizeof(int16_t) << 6, 0);
            get_inter_block(bs, &block[i << 6]);
            dequant_inter(&data[i << 6], &block[i << 6], iQuant);
            idct(&data[i << 6]);
        }
    }
}else{
    for (i = 0; i < 6; i++) {
        if (cbp & (1 << (5 - i)))   // coded
        {   Mem::Fill(&block[i << 6], sizeof(int16_t) << 6, 0);
            get_inter_block(bs, &block[i << 6]);
            dequant4_inter(&data[i * 64], &block[i << 6], iQuant);
            idct(&data[i << 6]);
        }
    }
}
```

(b)

รูปที่ 3.4 ส่วนของโค้ดจากดีโค้ดเดอร์ของ MPEG-4 (a) ยังไม่ได้ทำการลด branch condition (b) ทำการลด branch condition

3.1.5 การทำการลดจำนวน function call

ใน function call จะมี overhead ที่เกิดขึ้น ซึ่งการลด function call ซึ่งมีผลต่อประสิทธิภาพของ pipeline แต่การที่เราลด function call จะต้องเพิ่มโค้ด ดังนั้นการที่เราจะลด function call ควรที่จะใช้กับ function ที่มีการเรียกใช้งานบ่อยครั้ง ดังในตัวอย่าง ฟังก์ชัน BitstreamGetBit ภายในมีเรียกใช้ BitstreamShowBits และ BitstreamSkip ซึ่ง ฟังก์ชัน BitstreamGetBit มีการเรียกใช้หลายหมื่นครั้งซึ่งทำให้เกิด overhead จาก function call สูง แก้โดยการนำโค้ดของ BitstreamShowBits และ BitstreamSkip มาไว้ในฟังก์ชัน BitstreamGetBit ดังรูปที่ 3.5

```
static uint32_t __inline
BitstreamGetBit (Bitstream * const bs,
                 const uint32_t n)
{
    uint32_t ret = BitstreamShowBits(bs, n);

    BitstreamSkip(bs, n);
    return ret;
}
```

(a)

```
uint32_t ret;
int nbit = (n + bs->pos) - 32;

if (nbit > 0) {
    ret = ((0xffffffff >> (32-n))
        & (bs->bufa << nbit) | (bs->bufb >> (32-nbit)));
} else {
    ret = (bs->bufa & (0xffffffff >> bs->pos)) >> (-nbit);
}
    bs->pos += n;
    if (bs->pos >= 32) {
            uint32_t tmp;
            bs->bufa = bs->bufb;
            tmp = *((uint32_t *) bs->tail + 2);
            bs->bufb = tmp;
            bs->tail++;
            bs->pos -= 32;
    }
return ret;
```

(b)

รูปที่ 3.5 (a) ส่วนโค้ดจากดีโค้ดเดอร์ของ MPEG-4 ซึ่งเรียกใช้ระดับหมื่นครั้งซึ่งมีการใช้ function-call (b) ทำการกำจัด function call

3.2 วิธีการลดการใช้พลังงาน

จากพื้นฐานโมเดลพลังงานใน [4] ทำให้วิธีออพติไมซ์เพื่อลดพลังงานที่เราใช้จะมุ่งไปการลดจำนวนการอ้างหน่วยความจำโดยทำออพติไมเซชั่นในภาษาระดับสูง

3.2.1 โมเดลกำลังและพลังงาน(Power/Energy Model)

ในระบบมัลติมีเดียสมองกลฝังตัว การเข้าใช้หน่วยความจำจะกินพลังงานถึง 50 – 80 % ของทั้งหมด [6] การประเมินพลังงานที่ใช้ไปง่ายๆแสดงในสมการ (1) และ (2) [4] :

$$P_{Tr} = E_{Tr} * \frac{\#Transfers}{Second} \qquad (1)$$

$$E_{Tr} = f(\#words, \#bits) \qquad (2)$$

โดย P_{Tr} เป็นพลังงานที่ให้ต่อวินาที และ E_{Tr} เป็นพลังงานที่ใช้ต่อหนึ่งการส่งข้อมูลซึ่งขึ้นกับขนาดบิต สมมุติฐานของเราคือ E_{Tr} เป็นค่าคงในทุกการเข้าใช้หน่วยความจำ และผลของพลังงานที่ใช้จะแปรผันตรงจำนวนการส่งข้อมูล

$$E_{Total} \propto \#Transfers \qquad (3)$$

3.2.2 อัลกอริธึมในออพติไมเซชั่นเพื่อลดการใช้พลังงาน

กรณีการเขียนข้อมูลแล้วไม่เจอในแคช (Write miss) ในโพรเซสเซอร์ ARM920T ใช้นโยบาย Write thought และ Non-allocate ส่วนการเขียนข้อมูลที่มีในแคช (Write hit) ในโพรเซสเซอร์นี้จะใช้ Write back และเขียนใน Write back บัฟเฟอร์ อัลกอริธึมในออพติไมเซชั่นของเรา minimizing buffer และ read before write สามารถลดจำนวนการเข้าใช้หน่วยความจำโดยปรับปรุงเปลี่ยนแปลงโปรแกรมเพื่อประโยชน์ต่อแคชมากขึ้น โดยผลที่ออกมาจะมีจำนวน Cache miss น้อยลงซึ่งก็คือลดเหตุการณ์การใช้หน่วยความจำใน ARM ไมโครโพรเซสเซอร์

3.2.2.1 วิธี Minimizing Buffer Allocationในการพัฒนา C++ บนระบบปฏิบัติการซิมเบียน ใน subroutine จะมีบ่อยๆที่ใช้บัฟเฟอร์ในการเก็บผลก่อนในระหว่างการคำนวณ จากที่สังเกตดูบัฟเฟอร์พวกนี้จะ allocate แบบ local แล้ว assign(written) ต่อมาจะเอาค่าไปใช้ (read) ท้ายสุดทำลายทิ้ง (deallocate) เมื่อ subroutine ถูกเรียกทำงาน ตำแหน่งหน่วยความจำจะถูก allocate ในตำแหน่งที่แตกต่างกันในแต่ละครั้ง ซึ่งคือปัญหาที่ยากในการจัดการกับแคชข้อมูลที่มีขนาดจำกัด

วิธีของเราในการแก้ปัญหานี้โดยการประกาศบัฟเฟอร์ข้อมูลเป็น Attribute ของ class ทำให้สามารถเข้าสโคปที่ใหญ่ขึ้น โดยผลที่ได้จากที่บัฟเฟอร์ข้อมูล allocate ครั้งเดียวแล้วใช้มันอีกเรื่อยๆเป็นผลให้ cache miss ลดลงดังในรูปที่ 3.6

```
void Cdecoder::ConstructL()
{
    int32_t roundtab1[16]=
    { 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2 };
        •
        •
}
void Cdecoder::decoder_mbinter(DECODER * dec, ...)
{
    int16_t *block,*data;
    block = new int16_t[6*64];
    data = new int16_t[6*64];
    uint32_t stride = dec->edged_width;
    uint32_t stride2 = stride >>1;
        •
        •
}
void Cdecoder::decoder_mbintra(DECODER * dec, ...)
{
    int16_t *block,*data;
    block = new int16_t[6*64];
     data  = new int16_t[6*64];
    uint32_t stride = dec->edged_width;
    uint32_t stride2 = stride / 2;
        •
        •
}
```

(a)

```
void Cdecoder::ConstructL()
{
    block = new int16_t[6*64];
    data = new int16_t[6*64];
    int32_t roundtab1[16]=
    { 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2 };
        •
        •
}

void Cdecoder::decoder_mbinter(DECODER * dec, ...)
{
    uint32_t stride = dec->edged_width;
    uint32_t stride2 = stride >>1;
        •
        •
}

void Cdecoder::decoder_mbintra(DECODER * dec, .. )
{
    uint32_t stride = dec->edged_width;
    uint32_t stride2 = stride / 2;
        •
        •
}
```

(b)

รูปที่ 3.6 โค้ด C++ ของ XviD MPEG-4 วีดีโอดีโค้ดเดอร์บนซิมเบียน (a) ก่อนและ (b) หลังใช้วิธีMinimizing Buffer Allocation

ในวิธีนี้จะทำให้บัฟเฟอร์เข้าไปอยู่ในแคช แม้ว่าในสถาปัตยกรรมของ ARM จะมีความสามารถทำ lock down ข้อมูลในแคช แต่มันยากสำหรับการเขียนโปรแกรมบน C/C++ให้มีประสิทธิภาพ

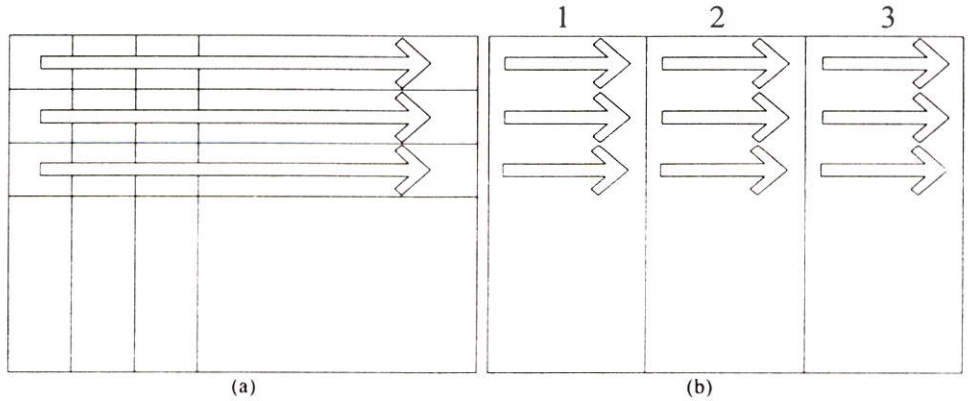
3.2.2.2 วิธี Read before Write เราพบว่าบัฟเฟอร์ที่ใช้เป็น temp จะถูกเขียน (ข้อมูลในระหว่างการคำนวณ) ครั้งแรกและจะถูกอ่านเอาข้อมูลไปใช้ต่อ ในการเขียนข้อมูลลงบัฟเฟอร์เป็นเหตุของการ write miss จึงเขียนลงหน่วยความจำเลย (write thought) ต่อมาจะเรียกใช้ข้อมูลในบัฟเฟอร์อีกก็จะไม่พบในแคช (read miss) โค้ดลักษณะแบบนี้เป็นโค้ดที่ไม่มีประสิทธิ์ภาพในการบริโภคพลังงานคือตอนแรกเขียนก็ไม่เจอในแคชต่อมาจะใช้ก็ไม่พบบนแคชทำให้เกิดการ data cache miss สองครั้งและเกิดการเข้าใช้หน่วยความจำที่ไม่จำเป็น
เราจึงเสนอทางแก้ง่ายคือทำการเริ่มตอนก็อ่านบัฟเฟอร์เพื่อให้บัฟเฟอร์เข้าไปอยู่ในแคชข้อมูลก่อนจะใช้งาน ดังนั้นจำนวนของการเข้าใช้หน่วยความจำจะลดลงเหมือนที่กินพลังงานน้อยลง แสดงในรูปที่ 3.7

```
//Load block to Dcache
int count;
for(count=0;count<6*64;count++)
{       tempdata+ = block[count];
          tempdata+ = data[count];}
if (dec->quant_type == 0)
{       for (i = 0; i < 6; i++)
        {       if (cbp & (1 << (5 - i)))     // coded
                {Mem::Fill(&block[i * 64], 64 * sizeof(int16_t), 0);
                  ..
                }
        }
}
else
{       for (i = 0; i < 6; i++)
        {       if (cbp & (1 << (5 - i)))     // coded
                {Mem::Fill(&block[i * 64], 64 * sizeof(int16_t), 0);
                 ..
                }
        }
}
```

รูปที่ 3.7 ส่วนโค้ดจาก MPEG-4 วีดีโอดีโค้ดเดอร์ที่ออฟติไมต์โดยวิธี read before write

3.2.2.3 วิธี Cache-Conscious Motion Estimation Pattern: Algorithms ในการ Encoding จะใช้พลังงานมากในการทำ motion estimation (ME) แม้ในหลายวิธีการ fast ME algorithm จะทำให้ลดการคำนวณและการเข้าใช้หน่วยความจำลงไป (ลดการใช้พลังงาน) แล้ว อย่างไรก็ตามขนาด cache และ cache replacement policy จะมีผลกระทบต่อการลดการคำนวณ และการเข้าใช้หน่วยความจำลงไป วีดีโอ Encoder โดยส่วนมากจะเข้าใช้ ME ตามแนวแถวจาก บนลงล่าง ดังรูป 3.8(a) ในการทำ motion estimation MB ที่อยู่ขวาไกลอาจถึงแทนที่ pixel ก่อน หน้าเนื่องด้วยความกว้างของ frame ที่กว้างและ cache replacement policy ของ ARM920T ที่เป็น FIFO

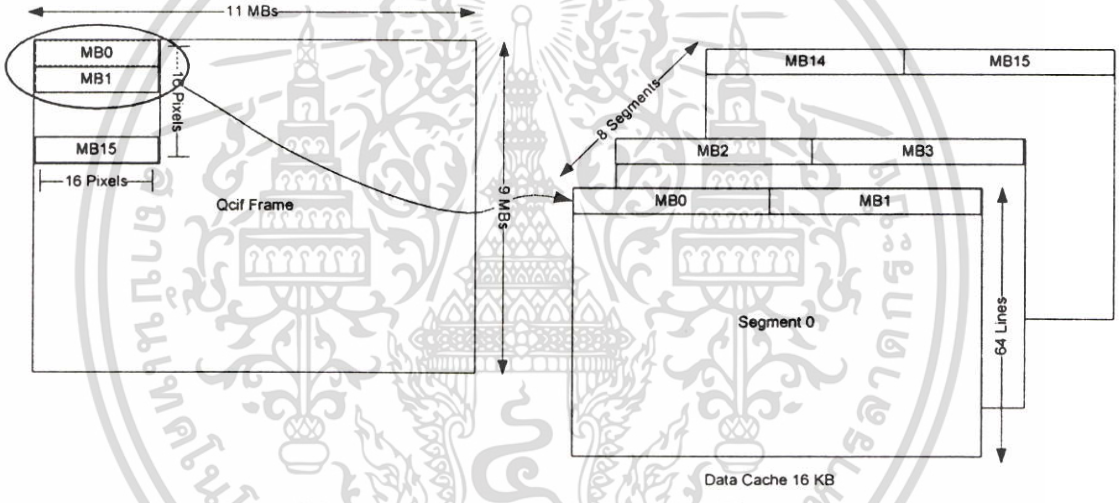
และ pattern ใหม่ที่แสดงในรูป 3.8(b) คือ ME จะเริ่มจาก MB ซ้ายไปขวาและบนลงล่าง แต่ทำตาม column ก่อน ซึ่งทำให้ pixel ของ reference frame ใน data cache ถูกใช้มากขึ้นก่อนถูก ไล่ออกไป โดยเราไม่ต้องออพติไมซ์ cache replacement policy ดังนั้นเราจึงเรียกวิธีนี้ว่า **Cache-Conscious Motion Estimation Pattern**



รูปที่ 3.8 ลำดับการเข้าใช้ MBs ในการทำ motion estimation (a) รูปแบบเก่า (b) รูปแบบใหม่

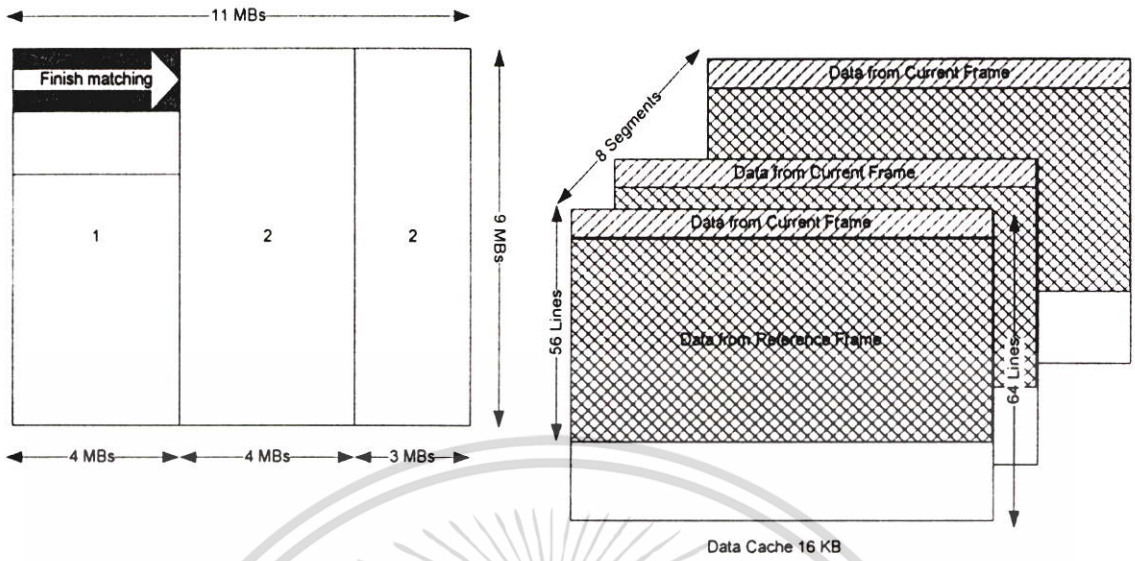
หลักการแบ่ง Pattern ของ Cache-Conscious Motion Estimation Pattern

วีดีโอที่เราใช้ในการทดลองคือ QCIF มีขนาด 176 * 144 Pixels คิดเป็น 11 * 9 MB โดย MB มีขนาด 16 * 16 Pixels เราใช้ pattern เป็น 4-4-3 ทำ Motion Estimation เริ่มคอลัมน์แรกจาก 4 MB คอลัมน์สอง 4 MB และคอลัมน์ท้าย 3 MB เหตุผลที่ใช้ pattern นี่ เราเฉลี่ยการทำ motion matching MB คือการไปหา MB ใดใน reference frame ใกล้เคียงกับ MB ใน current frame ทำไม่เกิน 4 ครั้ง หรือ 4 MB ซึ่งมีการเคลื่อนไหวอยู่ในขอบเขตวงกลมรัศมี 4 MB หรือรัศมี 64 Pixels สำหรับบนโทรศัพท์มือถือเวลาใช้งานจะเป็นภาพวีดีโอจากการถ่ายจากกล้องมือถือ ซึ่งถือว่าขอบเขตขนาดนี้ครอบคลุมการเคลื่อนไหวของภาพจากโทรศัพท์มือถือ โครงสร้างสถาปัตยกรรมของ Data Cache ใน ARM920T ที่มี 8 segments ในหนึ่ง segment มี 64 line โดย line ละ 8 word หรือ 32Bytes ให้หนึ่ง MB เวลา load ลง Data Cache จะตั้งแต่ segment 0 ถึง segment 7 ใช้ segment ละ 1 line ดังในรูปที่ 3.9



รูปที่ 3.9 แสดงข้อมูล MB ที่ถูกเก็บอยู่ใน Data Cache ของ ARM920T

ดังนั้นใน Motion matching ที่ใช้ใน XviD MPEG-4 video encoder คือ PMVfast ในตอนแรกจะทำการทำนายการเคลื่อนไหวจาก MB รอบตัวแล้วจากนั้นทำการ search แบบ diamond ทำการ search 4 ครั้งต้องอ้าง MB ใน reference frame 13 MB และ current frame 1 MB รวมต่อ 1 Motion matching อ้าง 14 MB การที่ทำ motion estimation แบบรูป 3.8(a) ที่ทำ motion matching ไป 11 MB ทางแนวนอนต้องอ้าง 11 * 14 MB เท่ากับ 154 MB แล้วกลับมาทำในแถวต่อมาข้อมูล reference frame ที่จำเป็นในการทำ motion matching เคยอยู่ใน Data Cache จากการอ้างในแถวก่อนหน้านี้ จะโดยข้อมูลอื่น load เข้าไปแทนที่อย่างแน่นอน เราใช้ pattern 4-4-3 ทำ motion matching ไป 4 MB มีการอ้าง 4 * 14 เท่ากับ 56 MB คือใช้ไป 56 line ในแต่ละ segment เมื่อกลับมาทำ motion matching แถวต่อไปข้อมูล MB ของ reference frame ก็จะยังอยู่ใน Data Cache ดังรูปที่ 3.10



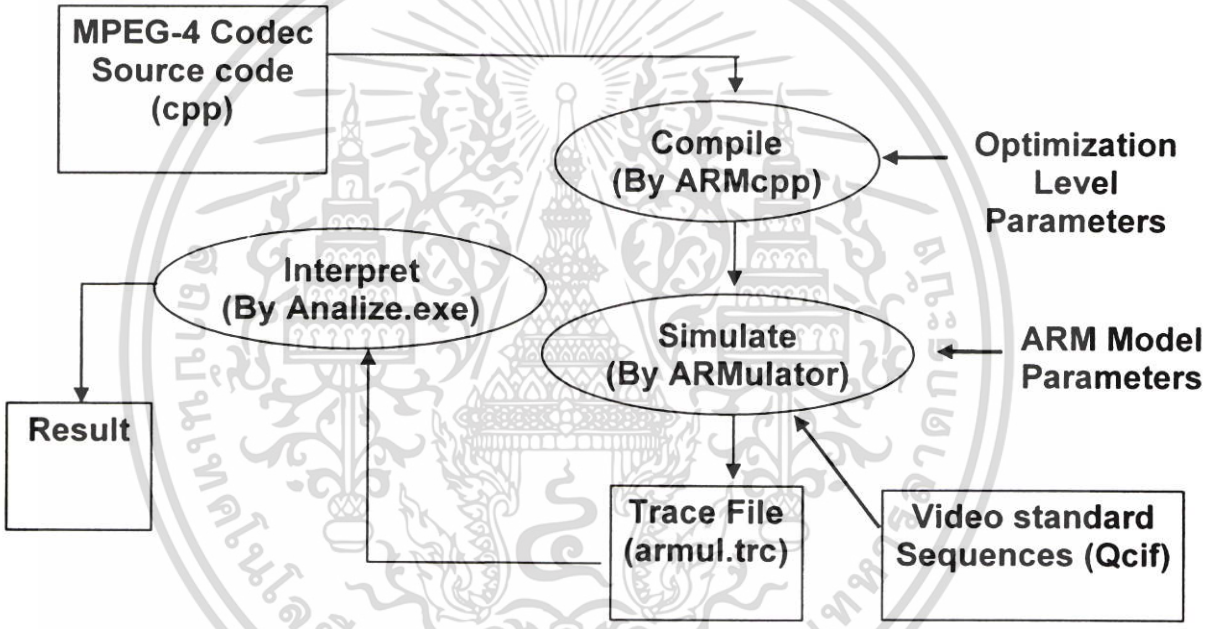
รูปที่ 3.10 แสดงข้อมูลที่อยู่ใน Data Cache เมื่อทำ Motion matching MB แล้ว 4 MB

บทที่ 4

การทดลองและผลการทดลอง

4.1 การทดลอง

เพื่อแสดงถึงประสิทธิภาพที่เราเสนอ รูปที่ 4.1 แสดงวิธีการทดลอง เราจะใช้ simple-profile simple-level MPEG-4 วีดีโอ Codec ของ XviD [7] ที่ทำการแปลงโค้ดไปทำงานบนระบบปฏิบัติการซิมเบียน ใช้ ARM คอมไพเลอร์ (ARMcpp) ซึ่งมีความสามารถออพติไมซ์ได้สูง คือมีโค้ดขนาดเล็ก และมีสามลำดับชั้นในการออพติไมซ์ความเร็ว Opt0, Opt1 และ Opt2



รูปที่ 4.1 แสดงขั้นตอนการทดลองของเรา

หลังจากคอมไพล์เสร็จได้ Binary ไฟล์แล้วนำไปจำลองการทำงานโดย ARMulator [11] ซึ่งสามารถจำลองการทำงานหลายเวอร์ชันของสถาปัตยกรรม ARM instruction set สามารถนับจำนวนสัญญาณนาฬิกา, การเข้าใช้หน่วยความจำ และ เหตุการณ์พิเศษ (เช่น แคชข้อมูลหรือชุดคำสั่ง, write back stall) เราใช้ ARMulator จำลองการทำงานของ ARM920T โดยไม่มีหน่วย floating-point

ในการจำลองการทำงาน MPEG-4 video codec เราใช้สอง QCIF วีดีโอในการทดลองมี Foreman และ News โดยบีบอัดที่ 15 เฟรมต่อวินาทีและ 24 กิโลบิตต่อวินาที เพียง 10 วีดีโอเฟรม (หนึ่ง I และเก้า P เฟรม) ที่ดีโค้ดเดอร์

หลังจากการจำลองจะได้ trace ไฟล์คือ armul.trc โดยภายในประกอบด้วยเหตุการณ์ของโพรเซสเซอร์ต่างเช่นการเข้าใช้หน่วยความจำ, การไม่เจอข้อมูลในแคชและข้อมูลที่เกี่ยวข้อง เราจะแปลข้อมูล trace ไฟล์โดยใช้โปรแกรมวิเคราะห์ที่เราพัฒนาขึ้นคือ Analize.exe เพื่อเอาข้อมูลที่เกี่ยวกับการเข้าใช้หน่วยความจำและเหตุการณ์ที่เกี่ยวกับแคช ผลลัพธ์ที่ได้จากการวิเคราะห์โดย Analize.exe จะได้จำนวนการอ้างหน่วยความจำทั้งแบบการอ่านและการเขียน จำนวนการเกิด Data Cache Misses และ จำนวนการเกิด Instruction Cache Misses และนำเอาข้อมูลเหล่านี้ไปแสดงเป็นกราฟ และตาราง เราจะวัดผลการทดลองของ MPEG-4 video codec ก่อนและหลังการทดลองแสดงในส่วนของผลการทดลอง ในรูปที่ 4.2 แสดงบางส่วนของ armul.trc

```
Date: Sat May 07 18:47:11 2005
Source: Armul
Options: Trace Events

E 00008000 00000000 10005
BSR4O___ A0000000 00000C1E
E 00008000 00000000 10002
BSR8O___ 00008018 E240B001 E242C001
IT 00008000 e28f8090 ADD r8,pc,#0x90 ; #0x8098
IT 00008004 e898000f LDMIA r8,{r0-r3}
BNR4O___ A0000000 00000C1E
BNR8O___ 00008098 00007804 00007828
BSR8O___ 00008080 10844009 E3C44003
BSR8O___ 00008088 E2555004 24847004
```

รูปที่ 4.2 แสดงตัวอย่างข้อมูลใน "armul.trc"

ต่อไปเราจะแสดงวิธีการแปลข้อมูลการเข้าใช้หน่วยความจำและเหตุการณ์ต่างๆใน trace ไฟล์

1) Memory Bus Trace (Bline): บรรทัดที่เริ่มต้นด้วย B แสดงถึงการเข้าใช้บัสของหน่วยความจำ โดยจะมีรูปแบบดังนี้ [12]

B<type><rw><size>[O][L][S] <address> <data>

โดย

<type> แสดงถึงชนิดของสัญญาณนาฬิกา

S sequential

N nonsequential

<rw> แสดงถึงการอ่านหรือการเขียน

R การอ่าน (Read)

W การเขียน (Write)

<size> แสดงถึงขนาดในการเข้าใช้บัสหน่วยความจำ

4 word (32 บิต)

2 halfword (16 บิต)

1 ไบต์ (8บิต)

O คือ Opcode fetch (instruction fetch)

L คือ locked access (SWR instruction)

S คือ speculative instruction fetch.

<address> จะอยู่ในรูปแบบเลขฐาน 16 ตัวอย่างเช่น 00008008

<data> จะแสดงตามนี้:

ค่าจากการอ่านหรือเขียน เช่น EB00000C

2) Events Trace (E lines): รูปแบบของบรรทัดเหตุการณ์ (E) เป็นดังนี้ [12]:

E <word1> <word2> <event_number>

ตัวอย่างเช่น E 00000048 00000000 10001

โดย

<word1> แสดงเป็น word บอกเช่นค่า PC

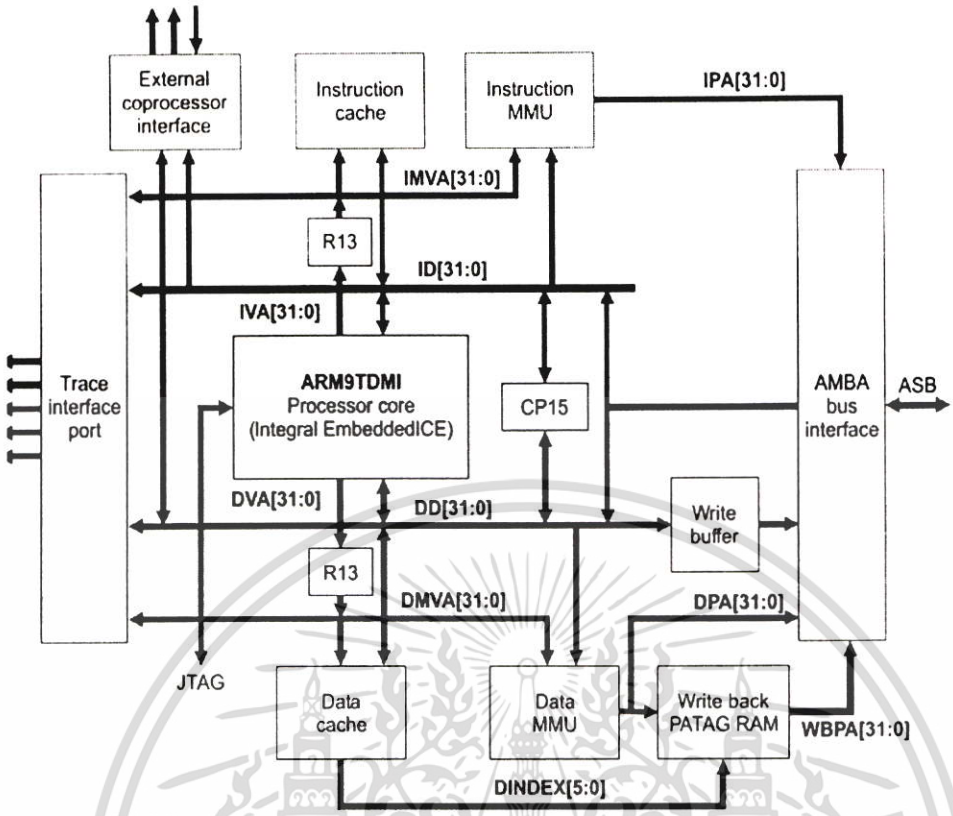
<word2> แสดงเป็น word บอกเช่นตำแหน่งที่เกิด event

<event_number> คือตัวเลขของเหตุการณ์ ตัวอย่างเช่น 0x10001 คือ MMUEvent_DLineFetch โดยเหตุการณ์ทั้งหมดจะอธิบายในตารางที่ 4.1

ตารางที่ 4.1 เหตุการณ์พิเศษต่างๆที่ตรวจได้จาก ARMulator

Event name (MMUEvent_)	**Word 1**	**Word 2**	**Event number**
DLineFetch	Miss address	Victim address	0x10001
ILineFetch	Miss address	Victim address	0x10002

รูปที่ 4.3 แสดงฟังก์ชันการทำงานของ ARM920T [5] โดย ARMulator ตรวจจับข้อมูลจาก trace interface port มันถูกกำหนดให้ตรวจจับการเข้าใช้บัสของหน่วยความจำในการอ่านและเขียนจาก AMBA 32-bit bus interface port



รูปที่ 4.3 ARM920T ไดอะแกรมจาก ARM Limited [5]

4.1.1 ขั้นตอนการทดลองการเพิ่มประสิทธิภาพด้านเพิ่มความเร็ว

การทดลองเพิ่มประสิทธิภาพแอพพลิเคชั่น MPEG-4 วีดีโอโคเด็คด้านเพิ่มความเร็วที่ทำงานบน ARM โปรเซสเซอร์ทำตามวิธีการข้างบน แต่ทำเราจะการทดลองเปรียบเทียบผลความเร็ว (จำนวน clock cycle) ก่อนและหลังการออพติไมเซชั่นเพื่อเพิ่มความเร็วในการทำงานของ MPEG-4 วีดีโอโคเดคของ XviD เวอร์ชั่นที่แปลงลงบนมือถือระบบปฏิบัติการซิมเบียน

ทำออพติไมซ์สี่แบบที่แตกต่างเปรียบเทียบกัน Opt0, Opt2, Opt0+OptSP และ OptSP โดย OptSP คือการออพติไมเซชั่นความเร็วที่เราเสนอไปในบทที่ 3.1 โดยใช้สี่วีดีโอคือ foreman, news, hall และ container ที่ encode ที่ 10 และ 15 fps กับ decoder และใช้สองวีดีโอ foreman และ news กับ encoder ให้ encode ที่ 15 fps และ 24 bps

เราจะหาประสิทธิภาพของวิธีการออพติไมเซชั่นความเร็ว (OptSP) ของเราจากตัวแปลพวกนี้: จำนวน clock cycles และ Speed up

4.1.2 วิธีการทดลองการเพิ่มประสิทธิภาพด้านการลดการใช้พลังงาน

ออพติไมซ์สี่แบบที่แตกต่างกันคือ Opt0, Opt2, Opt0+OptPW และ Opt2+OptPW โดย OptPW คือวิธีการออพติไมซ์พลังงานสองวิธีที่เราเสนอไปในบทที่ 3.2 เราใช้สองวีดีโอ Foreman และ News จำนวน 10 frame (1 I-frame และ 9 P-frames) โดยใช้ทำการ Encode ที่ 15 fps ที่ 24 bps

เ ราจะหาประสิทธิภาพของวิธีการออพติไมเซชั่นพลังงาน(OptPW) ของเราจากตัวแปลพวกนี้: การไม่พบข้อมูลในแคช (cache miss), การเข้าใช้บัสของหน่วยความจำ (memory/bus access), เวลาในการทำงาน (execution time) และจำนวนพลังงานที่ลดลง

4.1.3 วิธีการทดลองการเพิ่มประสิทธิภาพด้านการลดพลังงานและเพิ่มความเร็ว

วิธีการทดลองในการวัดการเพิ่มประสิทธิภาพด้านการลดพลังงานและเพิ่มความเร็ว ทำเหมือนการทดลองการเพิ่มประสิทธิภาพด้านลดพลังงานบวกกับการทดลองการเพิ่มประสิทธิภาพด้านเพิ่มความเร็ว คือจะวัดประสิทธิภาพก่อนและหลังการออพติไมซ์ทั้งด้านความเร็ว พลังงาน และความเร็วกับพลังงานร่วมกัน โดยดูทั้งการเข้าให้หน่วยความจำ, cache miss และจำนวน clock cycles โดยให้ ARMulator ในการทำ profile ใช้ ARMCpp ในการ compile โค้ด โดย complier มีหลาย level ในการออพติไมเซชั่นตั้งแต่ opt0, opt1 และ opt2

เราใช้สอง QCIF วีดีโอในการทดสอบมี Foreman และ News โดยบีบอัดที่ 15 เฟรมต่อวินาทีและ 24 กิโลบิตต่อวินาที เพียง 10 วีดีโอเฟรม (หนึ่ง I และเก้า P เฟรม) ของดีโค้ดเดอร์และเอ็นโค้ดเดอร์ โดยทำการออพติไมซ์สี่แบบที่แตกต่างกันคือ Opt0, Opt2, Opt0+Ours และ Opt2+Ours โดย Ours คือวิธีการออพติไมซ์วิธีที่เราเสนอคือ OptSP+OptPW

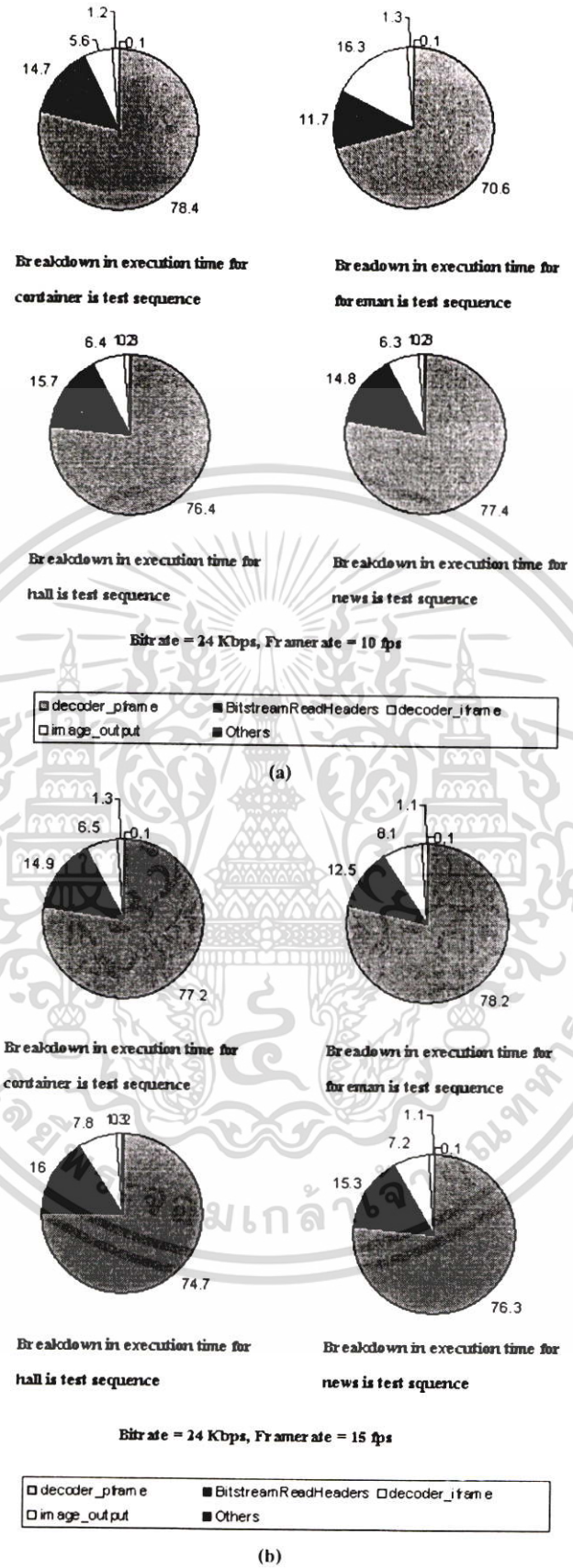
4.2 วิเคราะห์ประสิทธิภาพและผลการทดลอง

4.2.1 วิเคราะห์ประสิทธิภาพและผลการทดลองด้านความเร็ว

โดยสิ่งที่สนใจคือเวลาที่ใช้ของฟังก์ชันและจำนวนการเรียกใช้ที่ได้จากการทำ profile ตัว MPEG-4 วีดีโอ Codec โดย ARMulator นำมาวิเคราะห์ประสิทธิภาพของ Codec และวิเคราะห์หาคอขวดของ Codec

-ดีโค้ดเดอร์

เราใช้ QCIF วีดีโอในการทดสอบมี Foreman, Hall, Container และ News โดยบีบอัดที่ 10 และ 15 เฟรมต่อวินาทีและ 24 กิโลบิตต่อวินาทีที่ใช้กับการทดลองการออพติไมเซชั่นดีโค้ดเดอร์ โดยทำการออพติไมซ์สี่แบบที่แตกต่างกันคือ Opt0, Opt2, Opt0+OptSP และ Opt2+OptSP โดย OptSP คือวิธีการออพติไมซ์เพิ่มความเร็วด้วยวิธีการในบทที่ 3.1



รูปที่ 4.4 เวลาที่ใช้ของฟังก์ชันต่างๆของดีโค้ดเดอร์ที่ยังไม่ได้ทำการออพติไมเซชัน (a) ซีเควนซ์ที่ 10 fps (b)ซีเควนซ์ที่ Encode ที่ 15 fps เข้าไป

จากรูปที่ 4.4 ชุดวีดีโอซีเควนซ์ (Container, Foreman, Hall และ News) ที่ 10 fps และ 15 fps โดย bitrate เท่ากับ 24 Kbit ต่อวินาที ฟังก์ชันที่กินเวลามากสุดคือ P-frame ที่ถูกดีโค้ด โดยฟังก์ชัน decoder_pframe ทำการวิเคราะห์ฟังก์ชัน decoder_pframe กับฟังก์ชันลูกหลาน (child function) สามฟังก์ชันลูกของ decoder_pframe ที่ใช้เวลามากที่สุด คือฟังก์ชันdecode_mbinter, ฟังก์ชัน get_motion_vector, และฟังก์ชัน image_setedge ตามลำดับ

-เอ็นโค้ดเดอร์

ส่วนเอ็นโค้ดเดอร์ เราใช้ QCIF วีดีโอในการทดสอบมี Foreman และ News โดยบีบอัดที่ 15 เฟรมต่อวินาทีและ 24 กิโลบิตต่อวินาที โดยทำการออพติไมซ์สี่แบบที่แตกต่างกันคือ Opt0, Opt2, Opt0+OptSP และ Opt2+OptSP โดย OptSP คือวิธีการออพติไมซ์เพิ่มความเร็วในบทที่ 3

ในการ encode จะให้เวลาในการคำนวณส่วนมากที่ motion estimation (ME) และยิ่งในการทำ sad ดังนั้นเราจะมึงเน้นที่ 2 ส่วนนี้ในการออพติไมเซชั่น

ความเร็วที่เพิ่มขึ้นโคเดคของ MPEG4 วีดีโอจากการออพติไมเซชั่น

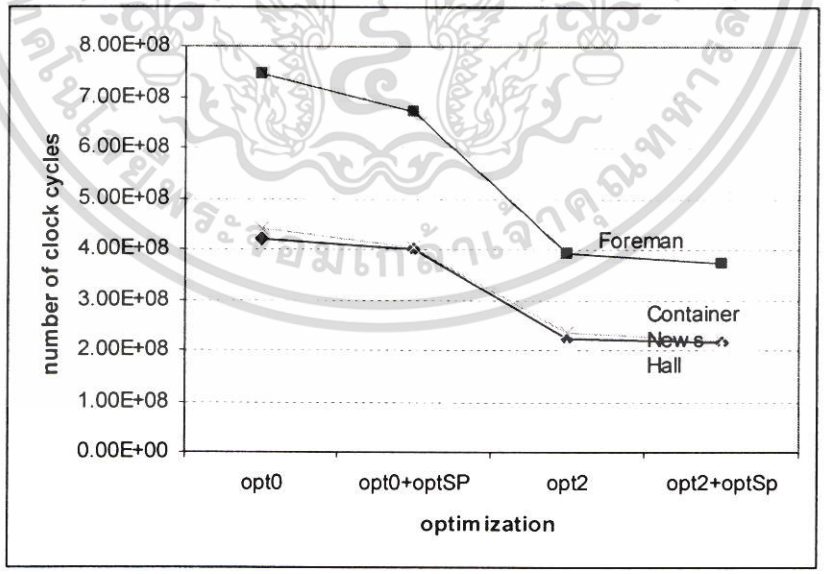
-ดีโค้ดเดอร์

ในการออพติไมเซชั่นพลังงานหลายวิธีที่นำมาให้ซึ่งทำให้ประสิทธิภาพของดีโค้ดเดอร์ดีขึ้นมาก (ในชุด hall ที่เป็นวีดีโอซีเควนซ์ที่ 15 fps จากที่ใช้ cycles ทั้งหมด 5.74E+08 เมื่อออพติไมเซชั่นเหลือเพียง 5.58E+08) แต่ประสิทธิภาพจะเพิ่มขึ้นกับวีดีโอซีเควนซ์ที่มีความซับซ้อนสูงเนื่องจากต้องคำนวณมากขึ้น (ทำงานโค้ดมาก) ซึ่งเมื่อเทียบที่ยังไม่ออพติไมซ์จึงเร็วขึ้นกว่าเดิมมาก (ในชุด Foreman ที่เป็นวีดีโอซีเควนซ์ที่ 15 fps จากที่ใช้เวลาทั้งหมด 1.03E+09 เมื่อออพติไมเซชั่นเหลือเพียง 9.15E+08) ซึ่งก็ยังอยู่ในผลที่น่าพอใจ ผลลัพธ์หลังการออพติไมเซชั่น ดังในตารางที่ 4.2

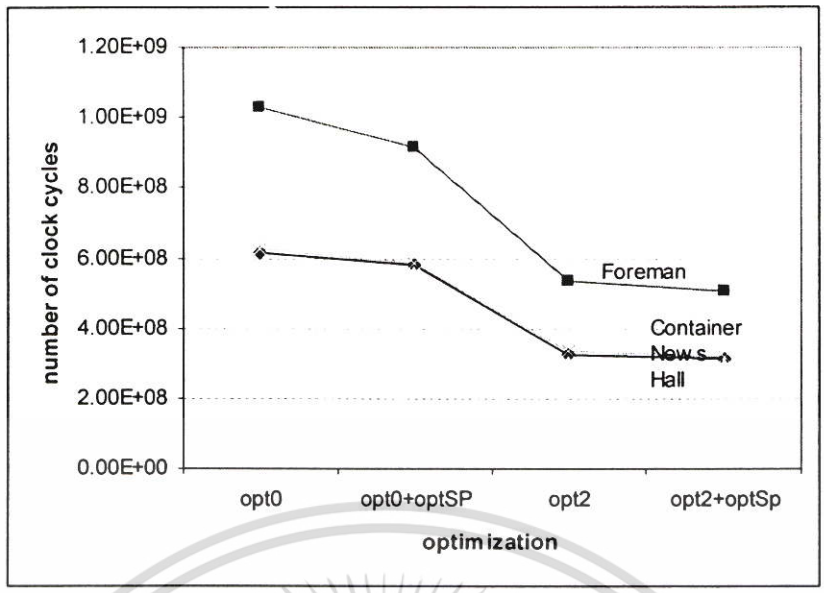
ตารางที่ 4.2 ผลก่อนและผลหลังออพติไมเซชั่นดีโค้ดเดอร์ของ MPEG-4 โดยใช้วีดีโอซีเควนซ์ที่ 24 Kbps

Test case	Encode fps	Clock Cycles		Speed Up (%)
		Opt0	OptSP+Opt0	
container	10	4.41E+08	4.03E+08	9.34
	15	6.28E+08	5.85E+08	7.27
foreman	10	7.45E+08	6.70E+08	11.20
	15	1.03E+09	9.15E+08	12.41
hall	10	3.93E+08	3.81E+08	3.17
	15	5.74E+08	5.58E+08	2.88
news	10	4.21E+08	4.03E+08	5.49
	15	6.14E+08	5.82E+08	5.44

แต่เมื่อวิธีการของเราเทียบกับการออพติไมซ์ (opt2) ของคอมไพเลอร์ ARMCpp วิธีการของเราสามารถลดจำนวนของ clock cycles ได้น้อยกว่า อย่างไรก็ตามเมื่อเรานำวิธีการของเราใช้ร่วมกับ opt2 ยิ่งทำให้ลดจำนวน clock cycles ได้มากขึ้นดังรูปที่ 4.5 และ รูปที่ 4.6

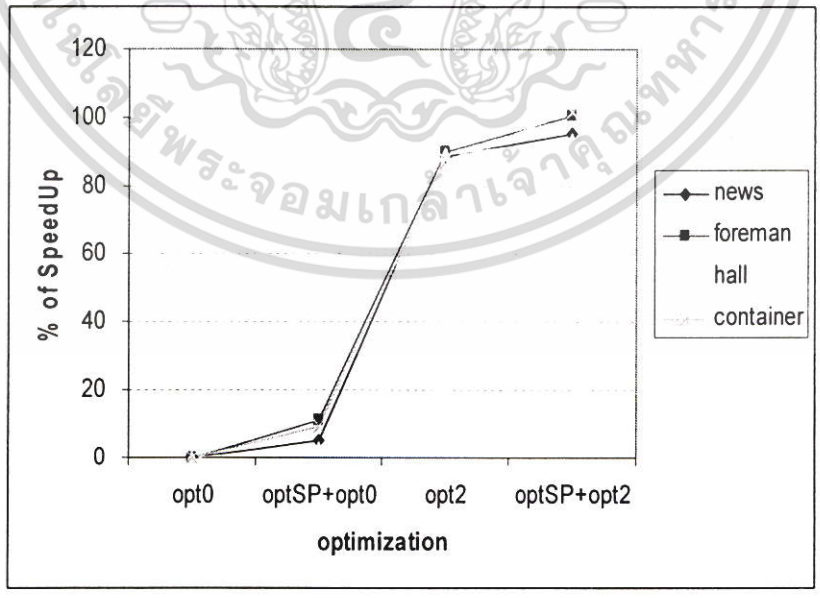


รูปที่ 4.5 จำนวนของ clock cycles ในการ optimization ที่ต่างกันที่ใช้ในการดีโค้ดวีดีโอซีเควนซ์ทั้ง 4 แบบที่ 10 เฟรมต่อวินาที

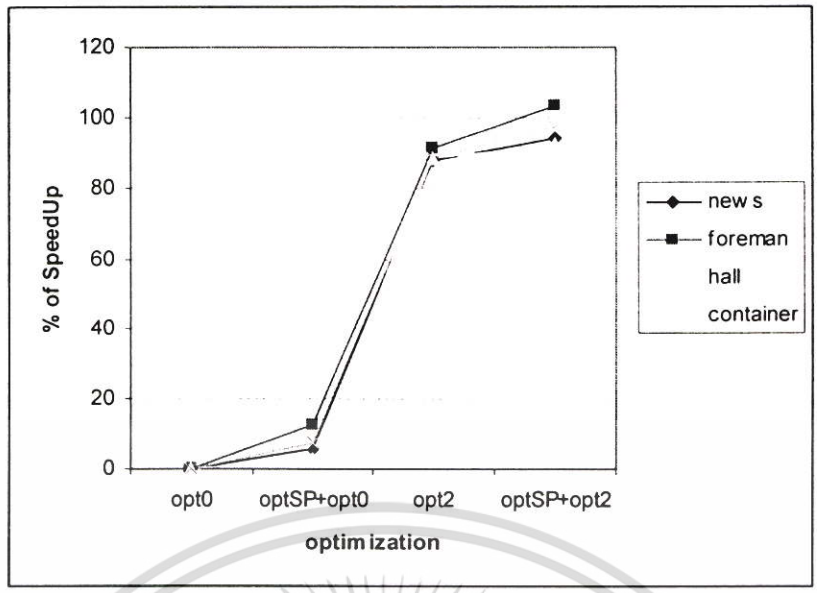


รูปที่ 4.6 จำนวนของ clock cycles ในการ optimization ที่ต่างกันที่ใช้ในการดีโค้ดวีดีโอซีเควนซ์ทั้ง 4 แบบที่ 15 เฟรมต่อวินาที

เมื่อคำนวณ SpeedUp ที่เพิ่มขึ้นเมื่อเทียบกับ opt0 จะเห็นว่า optSP ของเราสามารถเพิ่ม SpeedUp ได้ถึง 12.41% ส่วน opt2 ทำให้ SpeedUp เพิ่มขึ้น 90 % แต่เมื่อเราใช้ optSP กับ opt2 SpeedUp เพิ่มขึ้นถึง 103.5% ดังรูปที่ 4.7 และ 4.8 ถึงแม้ว่าวิธีการออพติไมเซชันของเรา optSP ของเราเพิ่มความเร็วได้น้อยการออพติไมเซชันของคอมไพเลอร์ แต่เมื่อเราใช้ optSP ของเราร่วมกับ opt2 ยังสามารถเพิ่มความเร็วได้มากขึ้นอีก



รูปที่ 4.7 เปอร์เซ็นต์ Speed Up ในแต่ optimization ที่ต่างกันที่เทียบกับ opt 0 ของ ARMcpp ของการดีโค้ดวีดีโอซีเควนซ์ทั้ง 4 แบบที่ 10 เฟรมต่อวินาที



รูปที่ 4.8 เปอร์เซ็นต์ Speed Up ในแต่ optimization ที่ต่างกันที่เทียบกับ opt 0 ของ ARMcpp ของการดีโค้ดวีดีโอซีเควนซ์ทั้ง 4 แบบที่ 15 เฟรมต่อวินาที

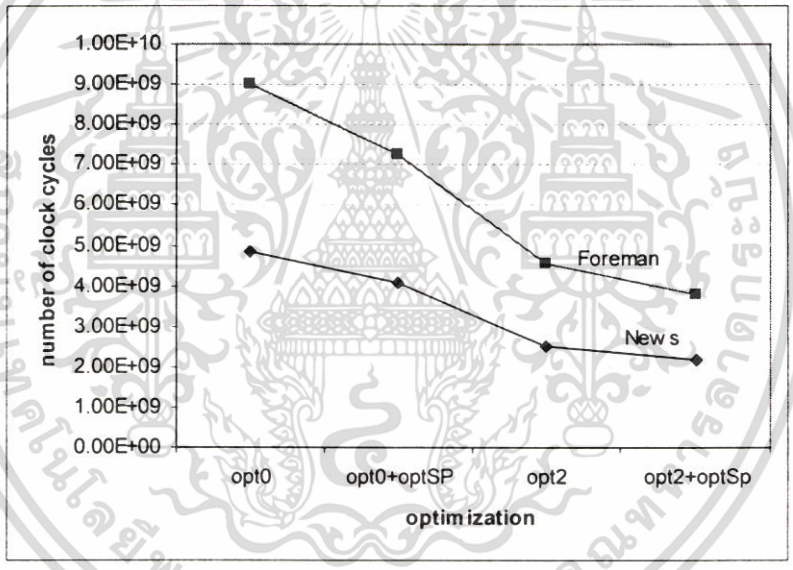
-เอ็นโค้ดเดอร์

ส่วนของเอ็นโค้ดเดอร์ผลลัพท์ที่ได้จะคล้ายคลึงกับดีโค้ดเดอร์ แต่จะปรับปรุงประสิทธิ์ภาพได้มากกว่าดีโค้ดเดอร์ จากที่ใช้การออพติไมเซชั่นวิธีที่เราเสนอนำมาให้ ได้ผลลัพธ์ที่ทำให้ประสิทธิภาพของเอ็นโค้ดเดอร์ดีขึ้นมาก (ในชุด foreman ที่เป็นวีดีโอซีเควนซ์ที่ 15 fps จากที่ใช้ cycles ทั้งหมด 8.98E+09 เมื่อทำการออพติไมเซชั่นความเร็วเหลือเพียง 7.23E+09) แต่ประสิทธิภาพจะเพิ่มขึ้นกับวีดีโอซีเควนซ์ที่มีความซับซ้อนสูงเนื่องจากต้องคำนวณมากขึ้น (ทำงานโค้ดมาก) ซึ่งเมื่อเทียบที่ยังไม่ออพติไมซ์จึงเร็วขึ้นกว่าเดิมดังในตารางที่ 4.3

ตารางที่ 4.3 ผลก่อนและผลหลังออพติไมเซชั่นเอ็นโค้ดเดอร์ของ MPEG-4 โดยใช้วีดีโอซีเควนซ์ที่ 24 Kbps

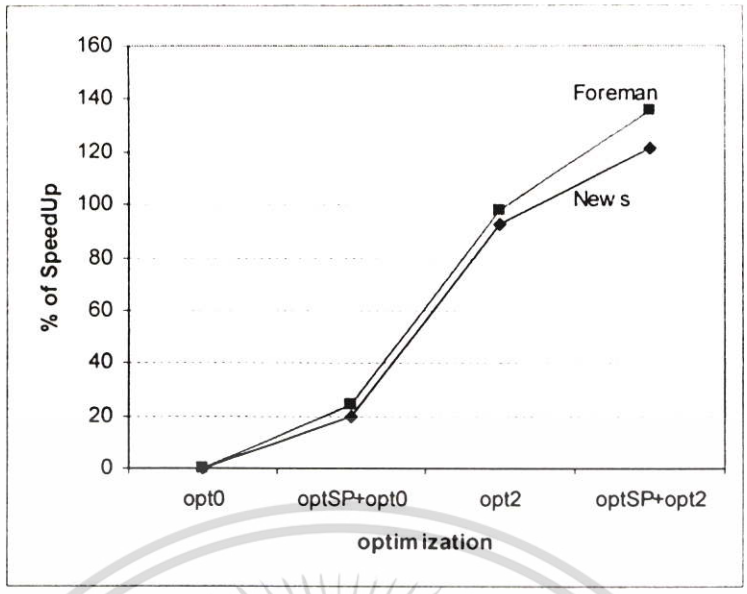
Test case	Encode fps	Clock Cycles		Speed Up (%)
		Opt0	Opt0+OptSP	
foreman	15	8.98E+09	7.23E+09	24.22
news	15	4.85E+09	4.06E+09	19.50

แต่เมื่อวิธีการของเราเทียบกับการออพติไมซ์ (opt2) ของคอมไพเลอร์ ARMCpp วิธีการของเราสามารถลดจำนวนของ clock cycles ได้น้อยกว่า แต่อย่างไรก็ตามเมื่อเรานำวิธีการของเราใช้ร่วมกับ opt2 ยิ่งทำให้ลดจำนวน clock cycles ได้มากขึ้นดังรูปที่ 4.9



รูปที่ 4.9 จำนวนของ clock cycles ในการ optimization ที่ต่างกัน ที่ใช้ในการเอ็นโค้ดวีดีโอซีเควนซ์ทั้ง 2 แบบที่ 15 เฟรมต่อวินาที

เมื่อเป็นคำนวณ SpeedUp ที่เพิ่มขึ้นเมื่อเทียบกับ opt0 จะเห็นว่า optSP ของเราสามารถเพิ่ม SpeedUp ได้ถึง 24.22% ส่วน opt2 ทำให้ SpeedUp เพิ่มขึ้น 103.31 % แต่เมื่อเราใช้ optSP กับ opt2 SpeedUp เพิ่มขึ้นถึง 139.38% ดังรูปที่ 4.10 ถึงแม้ว่าวิธีการออพติไมเซชั่นพลังงานของเรา optSP ของเราเพิ่มความเร็วได้น้อยการออพติไมเซชั่นของคอมไพเลอร์ แต่เมื่อเราใช้ optSP ของเราร่วมกับ opt2 ยังสามารถเพิ่มความเร็วได้มากขึ้นอีก



รูปที่ 4.10 เปอร์เซ็นต์ Speed Up ในแต่ optimization ที่ต่างกันที่เทียบกับ opt 0 ของ ARMcpp ของการเอ็นโค้ดสองวีดีโอซีเควนซ์ ที่ 15 เฟรมต่อวินาที

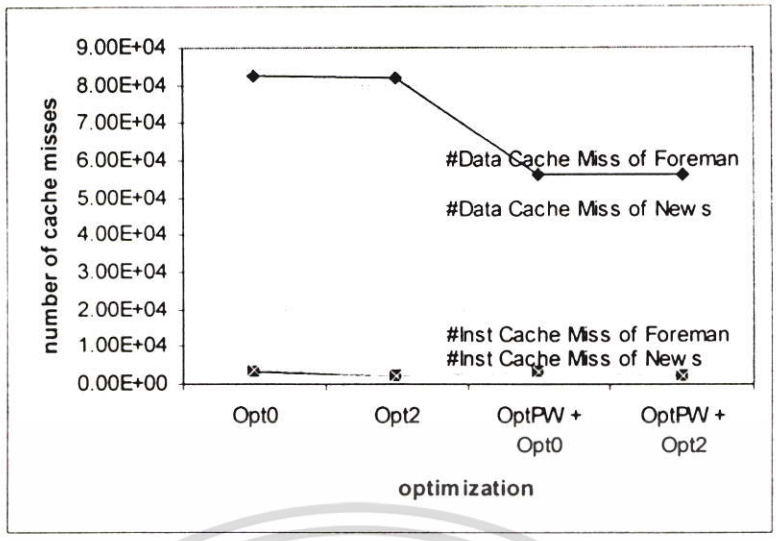
วิธีการออพติไมเซชั่นความเร็ว (OptSP) ที่เรานำมาใช้ได้ สามารถปรับปรุงทำให้เอ็นโค้ดเดอร์และดีโค้ดเดอร์ทำงานได้เร็วขึ้นกว่าเดิม

4.2.2 การทดลองและผลการทดลองในการเพิ่มประสิทธิภาพด้านการลดพลังงาน

เราจะหาประสิทธิภาพของวิธีการออพติไมเซชั่นพลังงาน(OptPW) ของเราจากตัวแปลพวกนี้: การไม่พบข้อมูลในแคช (cache miss), การเข้าใช้บัสของหน่วยความจำ (memory/bus access), เวลาในการทำงาน (execution time) และจำนวนพลังงานที่ลดลง

-ส่วนดีโค้ดเดอร์ (Decoder)

1) การไม่พบข้อมูลในแคช (cache miss): ในดีโค้ดเดอร์ส่วนมากจะเป็นการทำเข้าให้หน่วยความจำ, การเข้าใช้หน่วยความจำส่วนข้อมูล (data) จะบริโภคพลังงานมากกว่าส่วนชุดคำสั่ง (instruction)



รูปที่ 4.11 จำนวนของการไม่พบข้อมูลในแคชสำหรับทุกการออพติไมซ์ของดีโค้ดเดอร์ (24 กิโลบิตต่อวินาที, 15 เฟรมต่อวินาที, 10 วีดีโอเฟรมต่อวินาที)

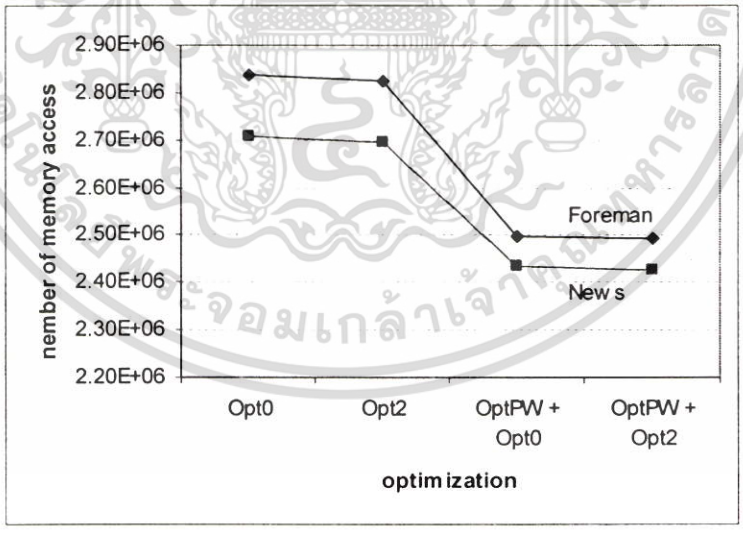
รูปที่ 4.11 อธิบายถึงวิธีการออพติไมซ์พลังงานของเรามีประสิทธิภาพในการลดจำนวนการไม่พบข้อมูลในแคชข้อมูลขณะที่การไม่พบข้อมูลคำสั่งในแคชชุดคำสั่งยังคงไม่เปลี่ยนแปลง เพียง Opt2 อย่างเดียวสามารถลดจำนวนการไม่พบข้อมูลในแคชได้น้อย รูปนี่แสดงให้เห็นว่าวิธีการของเรามีประสิทธิภาพทั้งที่แคชคำสั่งและแคชข้อมูลมี 16 กิโลไบต์ใน ARM 920T

2) การเข้าใช้หน่วยความจำ: ตารางที่ 4.4 แสดงจำนวนของการเข้าใช้หน่วยความจำ จำนวนการเข้าใช้หน่วยความจำทั้งหมดรวมกันจากจำนวนการอ่านข้อมูลในแคชคำสั่ง, จำนวนการอ่านข้อมูลในแคชข้อมูล และจำนวนการเขียนข้อมูลลงในแคชข้อมูลสำหรับสองวีดีโอทดสอบ จะเห็นว่าวิธีการออพติไมซ์ของเราสามารถลดจำนวนลดการเข้าใช้หน่วยความจำได้ทุกแบบย่อย ผลที่ได้จำนวนการเข้าใช้หน่วยความจำทั้งหมดลดลงอย่างที่เห็น

ตารางที่ 4.4 จำนวนและชนิดของการเข้าใช้หน่วยความจำของทุกการออพติไมเซชันของดีโค้ดเดอร์(15 เฟรมต่อวินาที, 10วีดีโอเฟรม)

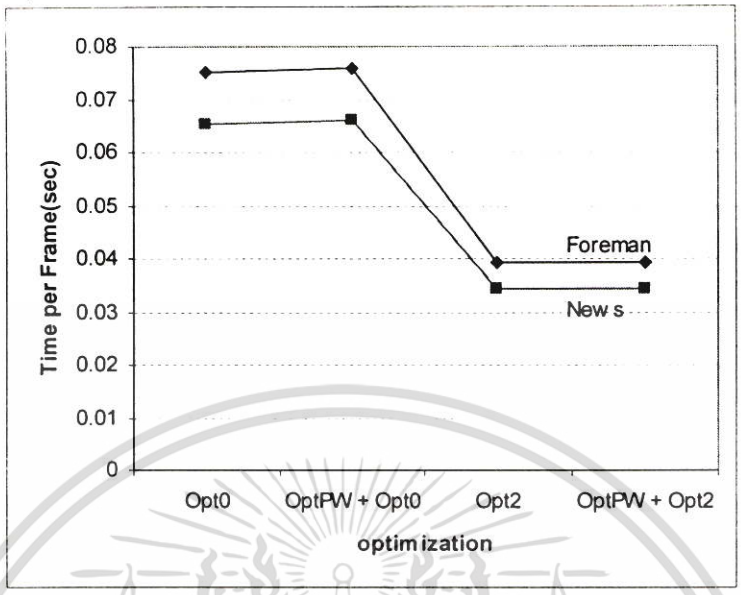
Video	Type	Opt0	OptPW+ Opt0	Opt2	OptPW+ Opt2
Foreman	Inst mem Read	27,321	24,297	17,649	15,689
	Data mem Read	659,038	449,004	654,622	451,428
	Data mem Write	2,150,179	2,024,930	2,150,723	2,025,343
	Total mem Access	2,836,538	2,498,231	2,822,994	2,492,460
News	Inst mem Read	28,473	25,465	17,513	15,585
	Data mem Read	558,333	399,516	554,549	400,652
	Data mem Write	2,119,976	2,007,870	2,121,856	2,010,156
	Total mem Access	2,706,782	2,432,851	2,693,918	2,426,393

รูปที่ 4.12 แสดงจำนวนการเข้าใช้หน่วยความจำทั้งหมดของทุกการออพติไมซ์สำหรับสองวีดีโอทดสอบ จากรูปนี้เราจะเห็นว่าวิธีการออพติไมซ์ของเรามีประสิทธิภาพในการลดจำนวนการเข้าใช้หน่วยความจำมากกว่าการออพติไมซ์ Opt2 ของ ARMcpp



รูปที่ 4.12 จำนวนของการเข้าใช้หน่วยความจำของทุกการออพติไมซ์ของดีโค้ดเดอร์ (15 เฟรมต่อวินาที, 10 วีดีโอเฟรม)

3) เวลาการทำงาน (Execution Time): รูปที่ 4.13 แสดงเวลาเฉลี่ยต่อหนึ่งเฟรมในการดีโค้ดเดอร์ของแต่ละวิธีการออพติไมซ์ จากการสังเกตจะเห็นว่าวิธีของเรายังทำงานได้ดีกับคอมไพเลอร์

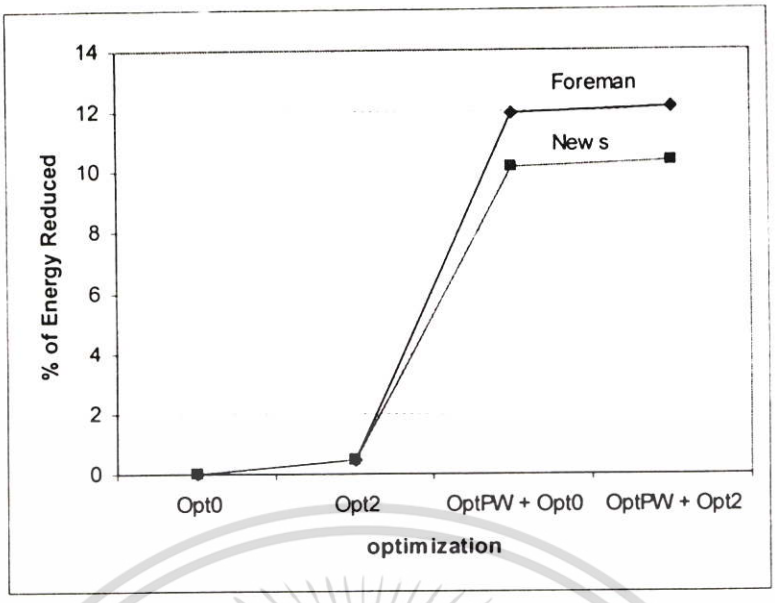


รูปที่ 4.13 เวลาในการทำงานต่อเฟรมสำหรับทุกการออพติไมซ์ของดีโค้ดเดอร์ (24กิโลบิตต่อวินาที, 15เฟรมต่อวินาที, 10วีดีโอเฟรม)

4) เปอร์เซ็นต์การลดพลังงาน (Reduced Energy Percentage): จากสมการที่ 3 เราคำนวณเปอร์เซ็นต์การลดพลังงานโดยวิธีของเรากับวิธี Opt0 จึงได้สมการตามนี้

$$\%\,Energy\ Reduced = \left(\frac{\#Tranfer_{opt0} - \#Tranfer_{new}}{\#Tranfer_{opt0}} \right) * 100 \qquad (4)$$

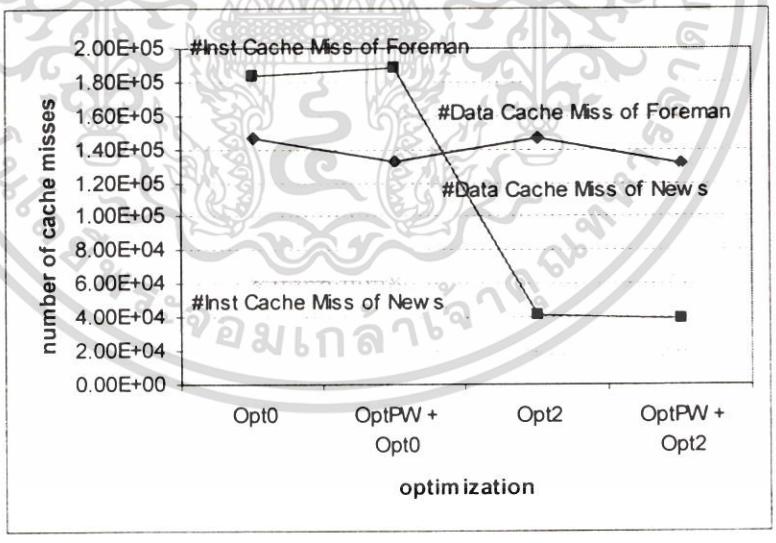
ผลที่ได้อยู่ในรูปที่ 4.14 จะเห็นว่าวิธีการออพติไมซ์ของเราสามารถลดได้ 10.6-12.5 เปอร์เซ็นต์ของการใช้พลังงาน



รูปที่ 4.14 เปอร์เซ็นต์การลดการบริโภคพลังงานของทุกการออพติไมซ์ของดีโค้ดเดอร์ (15 เฟรมต่อวินาที, 10 วีดีโอเฟรม)

-ส่วนเอ็นโค้ดเดอร์ (Encoder)

1) การไม่พบข้อมูลในแคช (cache miss): ในเอ็นโค้ดเดอร์ส่วนมากจะเป็นการบีบอัดภาพ และการคำนวณการชัดเชยการเคลื่อนไหนซึ่งเป็นการคำนวณส่วนมาก ดังนั้นการเข้าใช้หน่วยความจำส่วนชุดคำสั่ง (instruction) จะบริโภคพลังงานมากกว่าส่วนชุดข้อมูล (data)



รูปที่ 4.15 จำนวนของการไม่พบข้อมูลในแคชสำหรับทุกการออพติไมซ์ของเอ็นโค้ดเดอร์ (24 กิโลบิตต่อวินาที, 15 เฟรมต่อวินาที, 10 วีดีโอเฟรมต่อวินาที)

ดังในรูปที่ 4.15 แสดงถึงในเอ็นโค้ดเดอร์ จำนวนการไม่พบคำสั่งในแคชคำสั่งเยอะมากใน Opt0 แต่การออพติไมเซชั่นให้แคชชุดคำสั่งทำงานดีขึ้นมีการพัฒนามานานมากมายหลายวิธี และถูกใช้อย่างมากมายใน compiler ต่างๆ แต่วิธีการออพติไมเซชั่นพลังงานของเราที่ได้เสนอไปใน เป็นวิธี

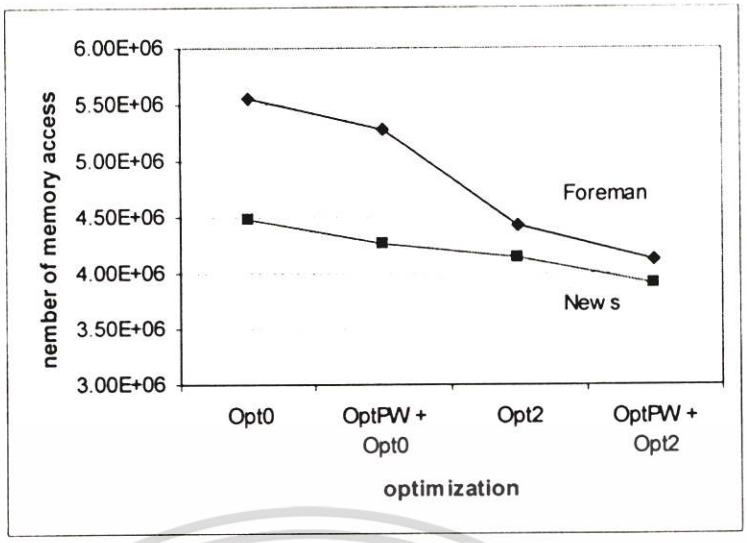
ที่จะปรับปรุงการทำงานของแคชของข้อมูลให้ดีขึ้น วิธีการออพติไมซ์พลังงานของเรามีประสิทธิภาพในการลดจำนวนการไม่พบข้อมูลในแคชข้อมูล เพียง Opt2 อย่างเดียวลดได้แต่เพียงการไม่พบข้อมูลคำสั่งในแคชชุดคำสั่งและสามารถลดจำนวนการไม่พบข้อมูลในแคชข้อมูลได้น้อย รูปนี่แสดงให้เห็นว่าวิธีการของเรามีประสิทธิภาพทั้งที่แคชคำสั่งและแคชข้อมูลมี 16 กิโลไบต์ใน ARM 920T

2) การเข้าใช้หน่วยความจำ: ตารางที่ 4.5 แสดงจำนวนของการเข้าใช้หน่วยความจำ จำนวนการเข้าใช้หน่วยความจำทั้งหมดรวมกันจากจำนวนการอ่านข้อมูลในแคชคำสั่ง, จำนวนการอ่านข้อมูลในแคชข้อมูล และจำนวนการเขียนข้อมูลลงในแคชข้อมูลสำหรับสองวีดีโอทดสอบ จะเห็นว่าวิธีการออพติไมซ์ของเราสามารถลดจำนวนลดการเข้าใช้หน่วยความจำได้ทั้งการอ่านและเขียนข้อมูล แต่จะเพิ่มการอ่านชุดคำสั่งมากขึ้นนิดหน่อย ซึ่งคุ้มค่าเมื่อเทียบกับจำนวนการเข้าถึงหน่วยความจำทั้งหมดลดลงอย่างที่เห็น

ตารางที่ 4.5 จำนวนและชนิดของการเข้าใช้หน่วยความจำของทุกการออพติไมเซชั่นของเอ็นโค้ด-เดอร์ (15 เฟรมต่อวินาที, 10วีดีโอเฟรม)

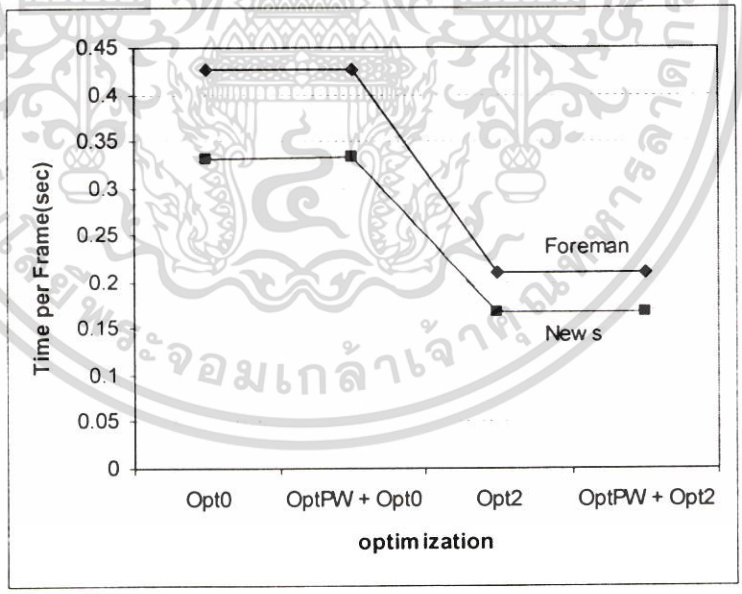
Video	Type	Opt0	OptPW+ Opt0	Opt2	OptPW+ Opt2
Foreman	Inst mem Read	1,465,461	1,537,685	326,756	335,052
	Data mem Read	1,176,691	984,378	1,176,723	1,009,866
	Data mem Write	2,917,379	2,770,846	2,923,459	2,784,276
	Total mem Access	5,559,531	5,292,909	4,426,938	4,129,194
News	Inst mem Read	490,949	516,623	142,852	154,463
	Data mem Read	1,090,907	968,382	1,090,476	971,573
	Data mem Write	2,905,996	2,787,725	2,912,728	2,790,617
	Total mem Access	4,487,852	4,272,730	4,146,056	3,916,653

รูปที่ 4.16 แสดงจำนวนการเข้าใช้หน่วยความจำทั้งหมดของทุกการออพติไมซ์สำหรับสองวีดีโอทดสอบ จากรูปนี้เราจะเห็นว่าวิธีการออพติไมซ์ของเรามีประสิทธิภาพในการลดจำนวนการเข้าใช้หน่วยความจำ และเมื่อใช้กับ opt2 ยังทำให้ลดการเข้าถึงหน่วยความจำลงไปอีก



รูปที่ 4.16 จำนวนของการเข้าใช้หน่วยความจำของทุกการออพติไมซ์ของเอ็นโค้ดเดอร์ (15 เฟรมต่อวินาที, 10 วีดีโอเฟรม)

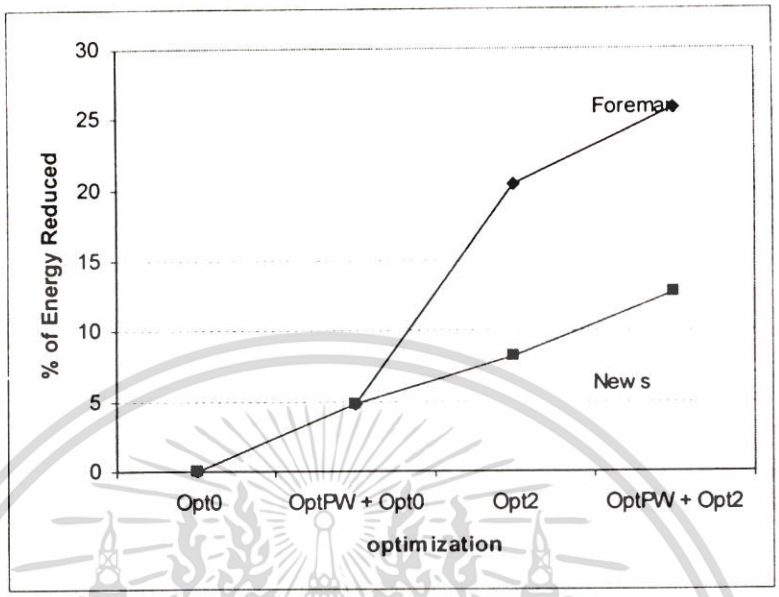
3) เวลาการทำงาน (Execution Time): รูปที่ 4.17 แสดงเวลาเฉลี่ยต่อหนึ่งเฟรมในการเอ็นโค้ดของแต่ล่ะวิธีการออพติไมซ์ จากการสังเกตจะเห็นว่าวิธีของเรายังทำงานได้ดีกับคอมไพเลอร์ ไม่ได้ทำให้ใช้เวลามากขึ้นแต่อย่างไร



รูปที่ 4.17 เวลาในการทำงานต่อเฟรมสำหรับทุกการออพติไมซ์ของเอ็นโค้ดเดอร์ (24กิโลบิตต่อวินาที, 15เฟรมต่อวินาที, 10วีดีโอเฟรม)

4) เปอร์เซ็นต์การลดพลังงาน (Reduced Energy Percentage): จากสมการที่ 3 เราคำนวณเปอร์เซ็นต์การลดพลังงานโดยวิธีของเรากับวิธี Opt0 ผลที่ได้อยู่ในรูปที่ จะเห็นว่าวิธีการออพติ

ไมซ์ของเราสามารถลดได้ 4.75 เปอร์เซ็นต์ของการใช้พลังงาน และเมื่อใช้กับ opt2 เพิ่มลดได้เพิ่มถึง 12.70 ถึง 25.72 ดังรูปที่ 4.18ห



รูปที่ 4.18 เปอร์เซ็นต์การลดการบริโภคพลังงานของทุกการออพติไมซ์ของเอ็นโค้ดเดอร์ (15 เฟรมต่อวินาที, 10 วีดีโอเฟรม)

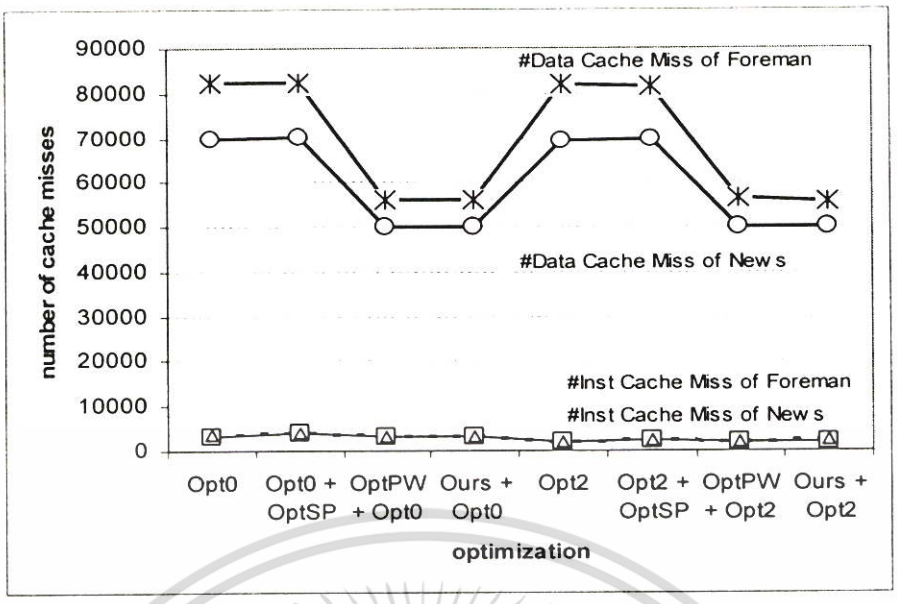
4.2.3 ผลการทดลองในการเพิ่มประสิทธิภาพทั้งความเร็วและพลังงาน

ผลที่เราสนใจในการวัดประสิทธิภาพทั้งความเร็วและพลังงานคือเราจะดูที่ cache miss, memory access และจำนวน clock cycles

- ส่วนดีโค้ดเดอร์ (Decoder)

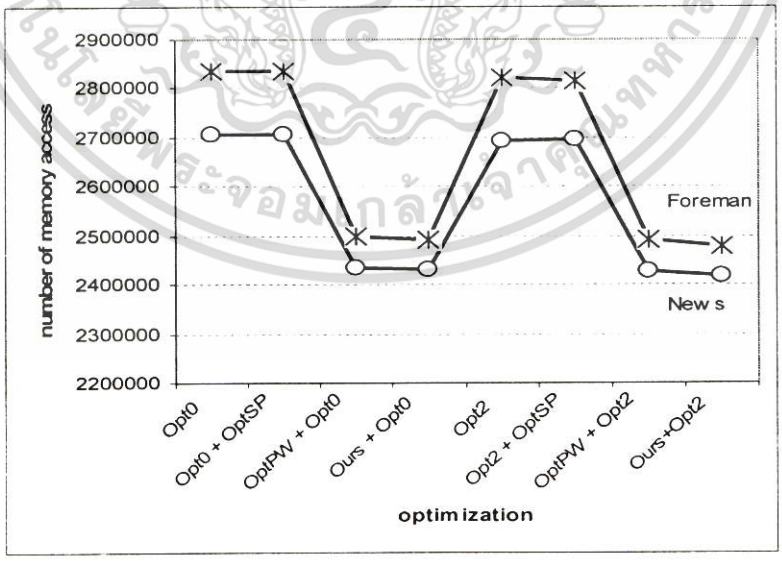
1. ผลการทดลองและวิเคราะห์ผลในแง่เมื่อเรานำการออพติไมเซชั่นความเร็ว (optSP) มาใช้กับการออพติไมเซชั่นพลังงานมาใช้กับดีโค้ดเดอร์ โดยพิจารณาในด้านพลังงานถึงผลกระทบต่างๆ

รูปที่ 4.19 เราจะเห็นว่าเมื่อใช้ออพติไมเซชั่นทั้งความเร็วและพลังงานร่วมกัน (Ours) แล้วไม่ได้ทำให้เกิด data cache miss มากขึ้นกว่าการใช้ออพติไมเซชั่นพลังงานอย่างเดียว (OptPW) ซึ่งจริงๆแล้วจำนวน data cache miss ลดลงนิดหน่อยด้วยเมื่อเทียบกับออพติไมเซชั่นพลังงานอย่างเดียว (OptPW) คือในออพติไมเซชั่นความเร็วช่วยลดจำนวน data cache miss ได้นิดหน่อย



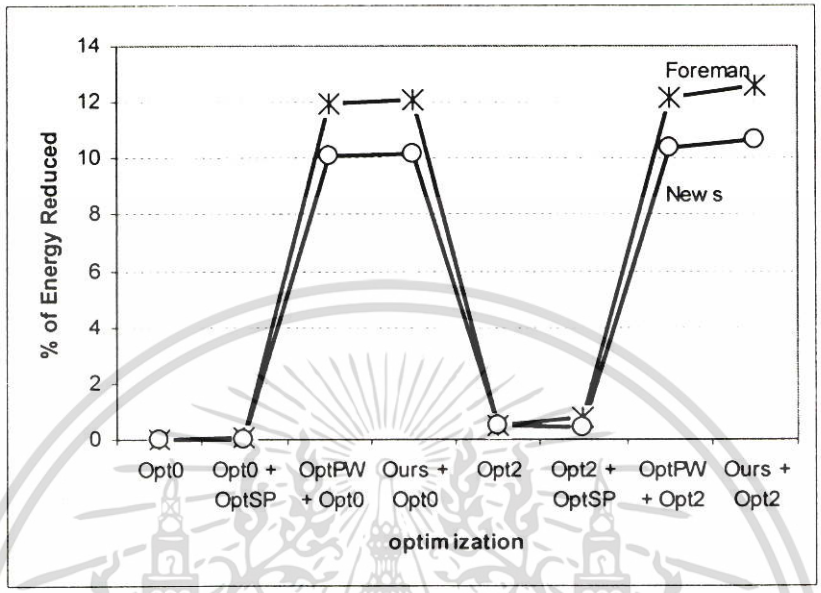
รูปที่ 4.19 จำนวนของการไม่พบข้อมูลในแคชสำหรับทุกการออพติไมซ์ (24 กิโลบิตต่อวินาที, 15 เฟรมต่อวินาที, 10 วีดีโอเฟรมต่อวินาที)

ดังที่แสดงในรูปที่ 4.20 เราจะเห็นว่าเมื่อเราออพติไมเซชั่นทั้งความเร็วและพลังงานพร้อมกัน(Outs)ไม่ได้ทำให้จำนวนการเข้าใช้หน่วยความจำเพิ่มขึ้นเมื่อเทียบกับการออพติไมเซชั่นพลังงานอย่างเดียวนั้นคือการออพติไมเซชั่นความเร็วไม่ได้ขัดแย้งกับการออพติไมเซชั่นพลังงาน แต่จริงแล้วยังสนับสนุนกับการออพติไมเซชั่นพลังงานอีกด้วย ถ้าสังเกตดูจะเห็นว่าการออพติไมเซชั่นความเร็วยังช่วยลดการเข้าใช้หน่วยความจำลดลงเล็กน้อย



รูปที่ 4.20 จำนวนของการเข้าใช้หน่วยความจำของวงทุกการออพติไมซ์ (15 เฟรมต่อวินาที, 10 วีดีโอเฟรม)

ส่วนในรูปที่ 4.21 แสดงถึงเปอร์เซ็นต์การบริโภคพลังงานที่ลดลงจากการใช้ออพติไมเซชั่น Ours จะเห็นว่าวิธีออพติไมเซชั่น Ours ลดการบริโภคพลังงานได้มากว่าการออพติไมเซชั่น OptPW อย่างเดียว

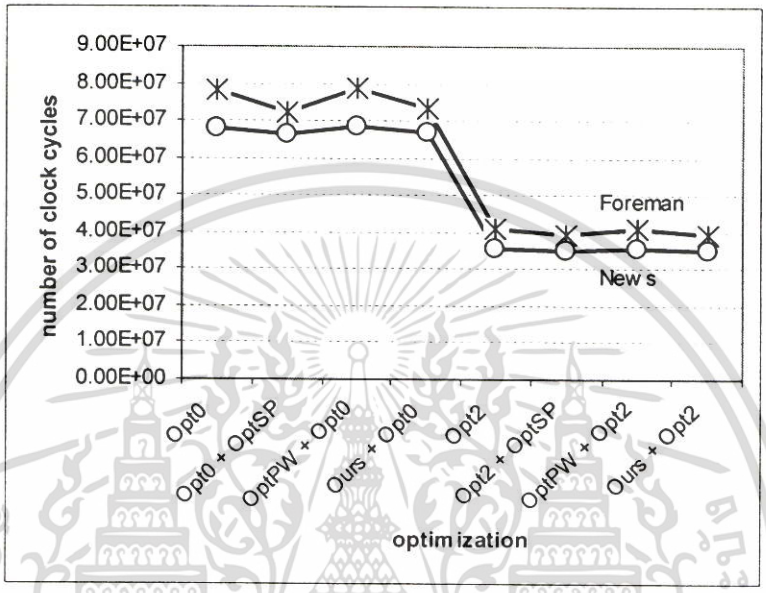


รูปที่ 4.21 เปอร์เซ็นต์การลดการบริโภคพลังงานของทุกการออพติไมซ์ (15 เฟรมต่อวินาที, 10 วีดีโอเฟรม)

2 ผลการทดลองและวิเคราะห์ผลในแง่เมื่อเรานำการออพติไมเซชั่นพลังงาน (optPW) มากับการออพติไมเซชั่นความเร็ว(optSP) โดยพิจารณาในด้านความเร็วถึงผลกระทบต่างๆ ในส่วนก่อนหน้าเราได้วิเคราะห์ว่าการออพติไมเซชั่นความเร็ว(optSP) จะส่งผลกระทบอย่างไรบ้างเมื่อนำไปใช้กับการออพติเซชั่นพลังงานซึ่งผลที่ได้จะเห็นว่าการออพติไมเซชั่นความเร็ว(optSP) ของดีโค้ดเดอร์ ไม่ได้ส่งผลเสียในด้านการใช้พลังงานแต่อย่างใด แต่ยังช่วยลดการเข้าใช้หน่วยความจำลงได้บางส่วนอีกด้วย ดังนั้นในส่วนนี้เราจะมาวิเคราะห์ผลว่าการออพติไมเซชั่นพลังงาน (optPW) จะมีผลกระทบอย่างไรบ้าง ในแง่ความเร็ว จากรูปที่ 4.22 แสดงให้เห็นว่าวิธีการออพติไมเซชั่น Ours จำนวน clock cycles ใกล้เคียงกับออพติไมเซชั่น optSP ทั้งในออพติไมเซชั่นโดยคอมไพเลอร์ทั้ง opt0 และ opt2 และจากตารางที่ 4.6 แสดงถึงเปอร์เซนต์ของ SpeedUp เมื่อเทียบกับการออพติไมเซชั่นของคอมไพเลอร์ ARMcpp ที่ opt0 โดยไม่มีการออพติไมเซชั่นทั้งความเร็วและพลังงาน จากตารางนี้เราการออพติไมเซชั่นพลังงานทำให้ใช้เวลาในการทำงานเพิ่มขึ้น แต่เพิ่มน้อยมากดูจาก opt0+optPW ในวีดีโอ foreman ซึ่งได้ค่า -0.72 และ opt0+optSP ในวีดีโอ foreman จาก 7.86 เมื่อใช้ออพติไมเซชั่น opt0+Ours ทำให้ Speed Up ลดลงไปเป็น 7.02 ซึ่งน้อยมากเมื่อกับพลังงานที่เราประหยัดมากขึ้นซึ่งถือว่าคุมให้การเพิ่มเวลาเล็กน้อยแต่ประหยัดพลังงานได้มากกว่าเดิมมาก แต่ที่แปลกคือใช้ Ours+Opt2 ได้เปอร์เซ็นต์ของ Speed up เพิ่มขึ้นจาก

OptSP+Opt2 นั้นแสดงว่าออพติไมเซชั่น OptPW ทำให้ออพติไมเซชั่นของคอมไพเลอร์ ARMcpp ออพติไมซ์ความเร็วได้ดีขึ้น

โดยร่วมแล้วออพติไมเซชั่นพลังงาน (optPW) เมื่อใช้ร่วมกับออพติไมเซชั่นความเร็ว (optSP) ทำให้ใช้ clock cycles เพิ่มขึ้นแต่เพิ่มขึ้นน้อยมาก ดังนั้นจึงเป็นการคุ้มค่าที่เราจะทำการออพติไมเซชั่นพลังงานแลกกับเวลาที่ใช้มากขึ้นน้อยมาก



รูปที่ 4.22 จำนวนการใช้ clock cycles ในแต่ละการออพติไมเซชั่นของสองชุดวีดีโอทดสอบ

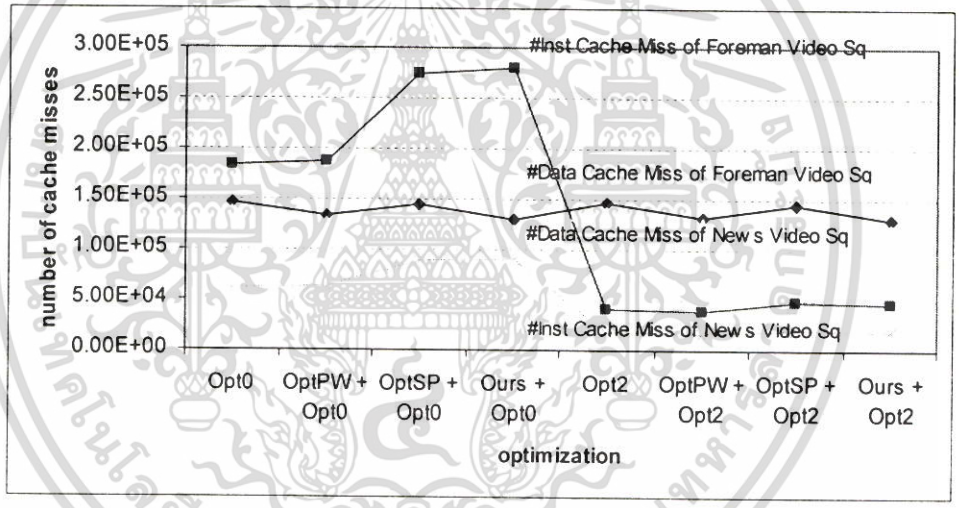
ตารางที่ 4.6 เปอร์เซ็นต์ SpeedUp เมื่อเทียบกับ Opt0 ของ ARMCpp ของแต่ละออพติไมเซชัน ในสองชุดวีดีโอทดสอบ

Optimization	Speed Up (%)	
	Foreman	News
Opt0	0	0
Opt0+OptSP	7.86	3.07
Opt0+OptPW	-0.72	-0.76
Opt0+Ours	7.02	2.26
Opt2	91.02	91.31
Opt2+OptSP	98.71	94.72
Opt2+OptPW	92.00	91.43
Opt2+Ours	99.80	94.91

-ส่วนเอ็นโค้ดเดอร์ (Encoder)

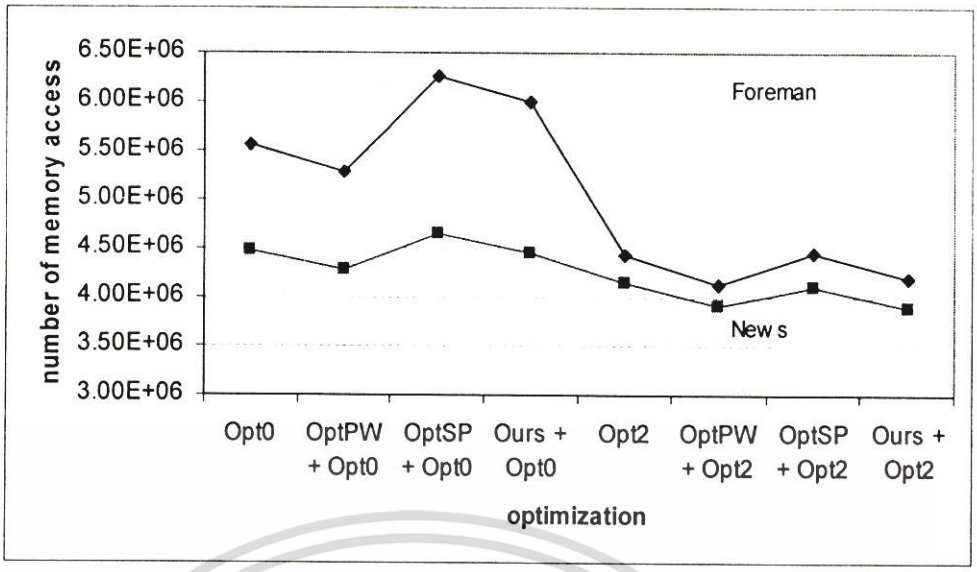
1. ผลการทดลองและวิเคราะห์ผลในแง่เมื่อเรานำการออพติไมเซชั่นความเร็ว(optSP) มากับการออพติไมเซชั่นพลังงานโดยพิจารณาในด้านพลังงานถึงผลกระทบต่างๆ

รูปที่ 4.23 เราจะเห็นว่าเมื่อใช้ออพติไมเซชั่นความเร็วโดยไม่ใช้ opt2 ทำให้ Instruction cache miss มากขึ้นกว่าเดิมมาก เมื่อนำมาใช้กับออพติไมซ์พลังงานทำให้ data cache miss ลดลงก็จริงแต่ไม่คุ้มค่าเนื่องจาก instruction cache miss เพิ่มมากว่า แต่เมื่อใช้การออพติไมเซชั่นทั้งความเร็วและพลังงานคู่กับ Opt2 ของ ARMcpp ผลที่ได้ instruction cache miss เพิ่มขึ้นนิดหน่อย ส่วน data cache miss ลดลงกว่าเดิมมาก ซึ่งถือว่าลดจำนวน cache miss โดยรวมได้มากขึ้น ที่เป็นเช่นนี้เนื่องจากการออพติไมเซชั่นความเร็วทำให้ใช้คำสั่งมากขึ้น ทำให้ instruction cache miss มากขึ้นมาก แต่เมื่อเรามาใช้บน opt2 ซึ่งมีความสามารถออพติไมเซชั่นความเร็ว(ลดจำนวน instruction)ที่ดีมาก จึงทำให้การออพติไมเซชั่นพลังงานและความเร็ว(Ours)ใช้ได้ผลใน opt2 และไม่มีจำนวน instruction cache miss เพิ่มมากขึ้นมากนัก



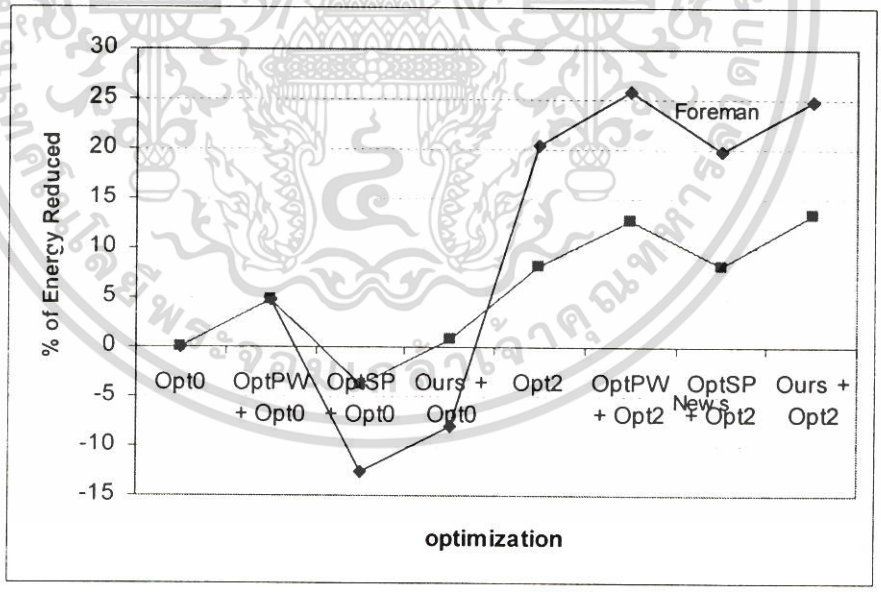
รูปที่ 4.23 จำนวนของการไม่พบข้อมูลในแคชสำหรับทุกการออพติไมซ์ (24 กิโลบิตต่อวินาที, 15 เฟรมต่อวินาที, 10 วีดีโอเฟรมต่อวินาที)

ดังที่แสดงในรูปที่ 4.24 จะเห็นว่าเมื่อเราออพติไมเซชั่นทั้งความเร็วและพลังงานพร้อมกัน(Outs) กับ Opt0 ทำให้จำนวนการเข้าใช้หน่วยความจำเพิ่มขึ้นเมื่อเทียบกับการออพติไมเซชั่นพลังงานกับ Opt0 นั้นคือการออพติไมเซชั่นความเร็วขัดแย้งกับการออพติไมเซชั่นพลังงานเนื่องจากการออพติไมเซชั่นความเร็วจำทำให้ต้องใช้คำสั่งมากขึ้นซึ่งแก้ได้โดยการใช้การออพติไมเซชั่นของ compiler เมื่อเราใช้ opt2 มาช่วยลดการเข้าใช้หน่วยความจำชุดข้อมูล เราจะเห็นว่าเมื่อเราออพติไมเซชั่นทั้งความเร็วและพลังงานพร้อมกัน(Outs) กับ Opt2 ไม่ได้ทำให้จำนวนการเข้าใช้หน่วยความจำเพิ่มขึ้นเมื่อเทียบกับการออพติไมเซชั่นพลังงานกับ Opt2 อย่างเดียว



รูปที่ 4.24 จำนวนของการเข้าใช้หน่วยความจำของทุกการออพติไมซ์ (15 เฟรมต่อวินาที, 10 วีดีโอเฟรม)

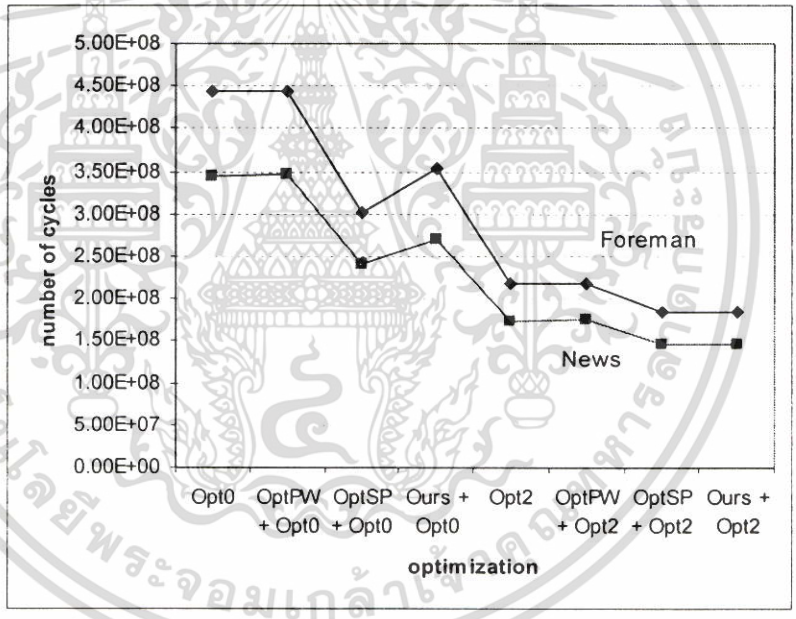
ส่วนในรูปที่ 4.25 แสดงถึงเปอร์เซ็นต์การบริโภคพลังงานที่ลดลงจากการใช้ออพติไมเซชั่น Ours กับ opt2 จะเห็นว่าวิธีออพติไมเซชั่น Ours ลดการบริโภคพลังงานได้มากขึ้นเมื่อเทียบกับการออพติไมเซชั่น opt2 อย่างเดียว



รูปที่ 4.25 เปอร์เซ็นต์การลดการบริโภคพลังงานของทุกการออพติไมซ์ (15 เฟรมต่อวินาที, 10 วีดีโอเฟรม)

2 ผลการทดลองและวิเคราะห์ผลในแง่เมื่อเรานำการออพติไมเซชั่นพลังงาน (optPW) มากับการออพติไมเซชั่นความเร็ว(optSP) โดยพิจารณาในด้านความเร็วถึงผลกระทบต่างๆ ส่วนนี้เราจะมาวิเคราะห์ผลว่าการออพติไมเซชั่นพลังงาน (optPW) จะมีผลกระทบอย่างไรบ้าง ในแง่ความเร็ว จากรูปที่ 4.26 แสดงให้เห็นว่าวิธีการออพติไมเซชั่น Ours จำนวน clock cycles ใกล้เคียงกับออพติไมเซชั่น optSP ทั้งในออพติไมเซชั่นโดยคอมไพเลอร์ทั้ง opt0 และ opt2 และจากตารางที่ 4.7 แสดงถึงเปอร์เซนต์ของ SpeedUp เมื่อเทียบกับการออพติไมเซชั่นของคอมไพเลอร์ ARMcpp ที่ opt0 โดยไม่มีการออพติไมเซชั่นทั้งความเร็วและพลังงาน

แม้ออพติไมเซชั่นพลังงาน (optPW) เมื่อใช้ร่วมกับออพติไมเซชั่นความเร็ว (optSP) ทำให้ใช้ clock cycles เพิ่มขึ้นเมื่อใช้กับ opt0 แต่ในความเป็นจริงแล้วในการใช้งานจะใช้งานโค้ดที่ทำการออพติไมเซชั่นโดย compiler แล้วซึ่งการออพติไมเซชั่น Ours ของเราใช้กับใน opt2 และได้ผลที่ดีคือทำให้ใช้ clock cycles เพิ่มขึ้นจาก optSP น้อยมาก ดังนั้นจึงเป็นการคุ้มค่าที่เราจะทำการออพติไมเซชั่นพลังงานแลกกับเวลาที่ใช้มากขึ้นน้อยมาก



รูปที่ 4.26 จำนวนการใช้ clock cycles ในแต่ละการออพติไมเซชั่นของสองชุดวีดีโอทดสอบ

ตารางที่ 4.7 เปอร์เซ็นต์ SpeedUp เมื่อเทียบกับ Opt0 ของ ARMCpp ของแต่ละออพติไมเซชัน ในสองชุดวีดีโอทดสอบ

Optimization	**Speed Up (%)**	
	Foreman	News
Opt0	0	0
Opt0+OptSP	47.36	43.64
Opt0+OptPW	0.28	0.51
Opt0+Ours	25.44	27.90
Opt2	103.31	98.67
Opt2+OptSP	139.38	137.50
Opt2+OptPW	102.06	97.55
Opt2+Ours	138.83	137.35

บทที่ 5
สรุป

5.1 สรุปงานวิจัยที่นำเสนอ

วิทยานิพนธ์นี้ได้ทำการเพิ่มประสิทธิภาพของ MPEG-4 Video Codec บนระบบปฏิบัติการซิมเบียน (Symbian OS) ซึ่งทำการแปลง source code มาจาก XviD MPEG-4 Video Codec มาทำงานบนระบบปฏิบัติการซิมเบียน โดยแต่เดิม XviD MPEG-4 Video Codec ทำงานบนพีซีซึ่งเป็น Open Source ของ XviD ปัญหาของ MPEG-4 Video Codec ที่ทำงานบนโทรศัพท์เคลื่อนที่คือหนึ่งมีโพรเซสเซอร์ที่มีความสามารถในการประมวลผลต่ำและมีแบตเตอรี่ที่จำกัด ดังนั้นเราจะทำการเพิ่มประสิทธิภาพความเร็วในการทำงานและลดการใช้พลังงานในการทำงาน โดยได้เสนอวิธีเพิ่มประสิทธิภาพซึ่งนำมาแก้ไข MPEG-4 Video Codec บนระบบปฏิบัติการซิมเบียนดังต่อไปนี้

การเพิ่มประสิทธิภาพด้านความเร็วในการทำงาน

- ย้ายการทำงานที่ซ้ำซ้อนในโค้ดคือการทำการลดการทำงานซ้ำซ้อนที่ไม่เป็นต้องทำซึ่งเป็นการทำงานที่ศูนย์เปล่า

- Loop Unrolling ออพติไมเซชันที่มีประสิทธิภาพและง่ายที่สุดในการปรับปรุงเวลาในการ execution ของ loop คือการทำ unrolling ใน loop สั้นๆที่มีจำนวน iteration คงที่สามารถแทนด้วย unroll ได้และเป็นการกำจัด overhead ของ loop การออพติไมเซชันสามารถตัดลด effective cost ได้ครึ่งหนึ่งของ overhead ของ loop

- การทำกำจัดความซับซ้อนหรือNon-linear Operator Strength Reduction (NOSR) คือการแทนที่สมการที่เป็น non-liner โดยการรวมกันของ conditional และ induction variable

- การทำการลดจำนวนของ branch prediction การลดจำนวนของ branch prediction โดยการทำ control flow transformation ซึ่งเป็นสิ่งที่สำคัญมากในการทำงานของระบบ pipeline นั้นคือถ้าเราทำการลดจำนวนของ branch prediction ก็จะทำให้ความเร็วในการทำงานเพิ่มขึ้น

- การทำการลดจำนวน function call ใน function call จะมี overhead ที่เกิดขึ้นซึ่งการลด function call ซึ่งมีผลต่อประสิทธิภาพของ pipeline แต่การที่เราลด function call จะต้องเพิ่มโค้ด ดังนั้นการที่เราจะลด function call ควรที่จะใช้กับ function ที่มีการเรียกใช้งานบ่อยครั้ง

การเพิ่มประสิทธิภาพด้านลดการใช้พลังงาน

- Cache-Conscious Motion Estimation Pattern: cache replacement policy จะมีผลกระทบต่อการลดการคำนวณและการเข้าใช้หน่วยความจำลงไป วีดีโอ Encoder โดยส่วนมากจะเข้าใช้ ME ตามแนวแถวจากบนลงล่าง ดังรูป *(a) ในการทำ motion estimation MB ที่อยู่ขวาไกลอาจถึงแทนที่ pixel ก่อนหน้าเนื่องด้วยความกว้างของ frame ที่กว้างและ cache replacement policy ของ ARM920T ที่เป็น FIFO และ pattern ใหม่ที่แสดงในรูป *(b) คือ ME จะเริ่มจาก MB ซ้ายไปขวาและบนลงล่างแต่ทำตาม column ก่อน ซึ่งทำให้ pixel ของ reference frame ใน data cache ถูกใช้มากขึ้นก่อนถูกไล่ออกไป โดยเราไม่ต้องออพติไมซ์ cache replacement policy

- Minimizing Buffer Allocation ในการพัฒนา C++ บนระบบปฏิบัติการซิมเบียน ใน subroutine จะมีบ่อยๆที่ใช้บัฟเฟอร์ในการเก็บผลก่อนในระหว่างการคำนวณ จากที่สังเกตดูบัฟเฟอร์พวกนี้จะ allocate แบบ local แล้ว assign(written) ต่อมาจะเอาค่าไปใช้ (read) ท้ายสุดทำลายทิ้ง (deallocate) เมื่อ subroutine ถูกเรียกทำงาน ตำแหน่งหน่วยความจำจะถูก allocate ในตำแหน่งที่แตกต่างกันในแต่ละครั้ง ซึ่งคือปัญหาที่ยากในการจัดการกับแคชข้อมูลที่มีขนาดจำกัด
วิธีของเราในการแก้ปัญหานี้โดยการประกาศบัฟเฟอร์ข้อมูลเป็น Attribute ของ class ทำให้สามารถเข้าสโคปที่ใหญ่ขึ้น โดยผลที่ได้จากที่บัฟเฟอร์ข้อมูล allocate ครั้งเดียวแล้วใช้มันอีกเรื่อยๆเป็นผลให้ cache miss ลดลง

- Read before Write เราพบว่าบัฟเฟอร์ที่ใช้เป็น temp จะถูกเขียน (ข้อมูลในระหว่างการคำนวณ) ครั้งแรกและจะถูกอ่านเอาข้อมูลไปใช้ต่อ ในการเขียนข้อมูลลงบัฟเฟอร์เป็นเหตุของการ write miss จึงเขียนลงหน่วยความจำเลย (write thought) ต่อมาจะเรียกใช้ข้อมูลในบัฟเฟอร์อีกก็จะไม่พบในแคช (read miss) โค้ดลักษณะแบบนี้เป็นโค้ดที่ไม่มีประสิทธิ์ภาพในการบริโภคพลังงานคือตอนแรกเขียนก็ไม่เจอในแคช ต่อมาจะใช้ก็ไม่พบบนแคชทำให้เกิดการ data cache miss สองครั้งและเกิดการเข้าใช้หน่วยความจำที่ไม่จำเป็น

เราจึงเสนอทางแก้ง่ายคือทำการเริ่มตอนก็อ่านบัฟเฟอร์เพื่อให้บัฟเฟอร์เข้าไปอยู่ในแคชข้อมูลก่อนจะใช้งาน ดังนั้นจำนวนของการเข้าใช้หน่วยความจำจะลดลงเหมือนที่กินพลังงานน้อยลง

ซึ่งจากผลการทดลองนี้แสดงให้เห็นว่า จากการเพิ่มประสิทธิภาพเพิ่มความเร็วและลดการบริโภคพลังงาน ได้ผลที่ดีทั้งใน Decoder และ Encoder ในด้านความเร็ว Speed up เพิ่มขึ้นจากเดิม 3.17 – 12.41% ของ Decoder และ 19.50-24.22% ของ Encoder ในด้านการลดพลังงานสามารถลด

ได้ถึง 10.6-12.5% ของ Decoder และ 4.5-5.7% ของ Encoder และถึงแม้เราจะใช้วิธีการเพิ่มประสิทธิภาพกับการออตไมเซชั่นของ compiler เช่น ARMCpp ที่ opt2 ก็ยังมีประสิทธิภาพอยู่

5.2 ปัญหาและอุปสรรค

ในปัจจุบัน MPEG-4 video codec บนโทรศัพท์มือถือไม่สามารถดาวน์โหลด source code มาทำวัดเปรียบเทียบความเร็วและพลังงานที่ใช้ในการทำงานได้โดยใช้ ARM developer suite เราจึงเปรียบเทียบให้ดู MPEG-4 video codec ก่อนและหลังการเพิ่มประสิทธิภาพ

5.3 แนวทางการพัฒนาต่อ

ในอนาคตเราจะพัฒนาเสียงเพิ่มเข้าไปใน MPEG-4 video codec ที่ทำงานบนระบบปฏิบัติการซิมเบียน โดยจะทำการ synchronization ระหว่างภาพและเสียง

บรรณานุกรม

[1] L. Nachtergaele, D. Moolenaar, B. Vanhoof, F. Catthoor, and H. De Man, **System-level power optimization of video codecs on embedded cores : a systematic approach**, Journal of VLSI Signal Processing Systems, vol. 18, no. 2, 1998, pp. 89-109

[2] T. Sikora, **The MPEG-4 video standard verification model**, IEEE Trans. on Circuits and Systems for Video Technology, Vol. 7, No. 1, pp. 19-31, Feb 1997.

[3] M. Palkovic, M. Miranda, K. Denolf, P. Vos, F. Catthoor, **Systematic Address and Control Code Transformations for Performance Optimisation of a MPEG-4 Video Decoder**, ASP-DAC/VLSI Design 2002, Jan 2002, pp. 547.

[4] K. Denolf, P. Vos, J. Bormans, and I. Bolsens. **Cost-efficient C-Level Design of anMPEG-4 Video Decoder**, PATMOS 2000, Germany, Sep 2000, pp. 233-24

[5] ARM Limited, 20 April 2001, **ARM920T (Rev1) Technical Reference Manual**

[6] R. Gonzales and M. Horowitz, **Energy dissipation in general-purpose microprocessors.** IEEE J. of Solid-state Circ., SC-31(9):1277-1283, 1996

[7] XviD, http://www.xvid.org

[8] ARM Limited, 21 Nov 2001, **ARM Developer Suite - Version 1.2 - AXD and armsd Debuggers Guide**

[9] J. Bormans, K. Denolf, S. Wuytack, L. Nachtergaele, I. Bolsens, **Integrating system-level low power methodologies into a real-life design flow**, PATMOS'99 Ninth International Workshop Power and Timing Modeling, Optimization and Simulation, pp. 19-28, Oct 6-8, 1999

[10] K. Ramkishor and V. Gunashree, **Real time implementation of MPEG-4 video decoder on ARM7TDMI**, in IEEE Proceedings of International Symposium on Intelligent Multimedia, Video and Speech Processing, pp. 3653, 2001.

[11] K. Tatas, K. Siozios, D. Soudris, K. Masselos, K. Potamianos, S. Blionas, and A. Thanailakis, **Power Optimization Methodology for Multimedia Applications Implementation on Reconfigurable Platforms**, PATMOS 2003, LNCS 2799, pp. 430–439, 2003.

[12] ARM Limited, 21 Nov 2001, **ARM Developer Suite - Version 1.2 -Debug Target Guide**

[13] ARM Limited, **ARM Programming Technique**

[14] Michael J. Flynn, **Computer Architecture Pipelined and Parallel Processor Design**

[15] L. Edwards, R Barker, and Staff of EMCC Software Ltd, **Developing Series 60 Applications A Guide for Symbian OS C++ Developers**

[16] ARM Limited, January 1998, **Writing Efficient C for ARM**

[17] http://www.arm.com/support/faqdev/1492.html

ผลงานวิจัยในระหว่างการศึกษาที่ได้รับการตีพิมพ์เผยแพร่

1. P Pekdeepaiboonpol and S Kittitornkun, "Performance Optimization of MPEG-4 Video Decoder on Symbian OS", Electrical Engineering Conference 27, Thailand 2004

2. P Pekdeepaiboonpol and S Kittitornkun, "ENERGY OPTIMIZATION FOR MOBILE MPEG-4 VIDEO DECODER", IEE MOBILITY CONFERENCE 2005, 15-17 November 2005, Guangzhou, China

3. P Pekdeepaiboonpol and S Kittitornkun, "Low Power Optimization Algorithms for ARM-based MPEG-4 Video Decoder" The 9th National Computer Science and Engineering Conference, October 27 - 28, 2005, Bangkok, THAILAND

4. P Pekdeepaiboonpol and S Kittitornkun, "Energy Optimization for MPEG-4 Video Decoder", Electrical Engineering Conference 28, Bangkok, Thailand 2005

5. P Pekdeepaiboonpol and S Kittitornkun, "POWER OPTIMIZATION FOR MOBILE MPEG-4 VIDEO DECODER", the Fifth International Conference on Information, Communications and Signal Processing 2005, Bangkok, Thailand 2005

6. P Pekdeepaiboonpol and S Kittitornkun, "Low Energy Optimization for MPEG-4 Video Encoder on ARM-based Mobile Phones", International Symposium on Wireless Pervasive Computing 2006, Thailand 2006

ประวัติผู้เขียน

ชื่อ-นามสกุล	นาย ไพโรจน์ ภักดีไพบูลย์ผล
ประวัติการศึกษา	สำเร็จการศึกษาระดับปริญญาตรี หลักสูตรวิศวกรรมศาสตรบัณฑิต สาขาวิศวกรรมคอมพิวเตอร์ จากคณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง ปีการศึกษา 2545
งานวิจัยที่สนใจ	Mobile Computing, Computer Architecture และ Multimedia