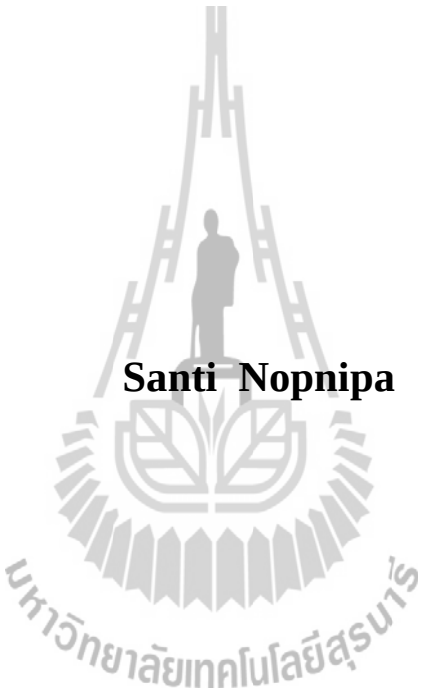
การศึกษาและออกแบบความหมายของการแนะนำเชิงลักษณะโดยใช้ตัวรวมไบต์โค้ด

นายสันติ นภนิภา

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์
มหาวิทยาลัยเทคโนโลยีสุรนารี
ปีการศึกษา 2557

A STUDY AND DESIGN OF THE SEMANTICS OF ASPECT-ORIENTED ADVICE USING BYTECODE COMBINATORS

Santi Nopnipa



A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Computer Engineering

Suranaree University of Technology

Academic Year 2014

การศึกษาและออกแบบความหมายของการแนะนำเชิงลักษณะโดยใช้ตัวรวมไบต์โค้ด

มหาวิทยาลัยเทคโนโลยีสุรนารี อนุมัติให้นับวิทยานิพนธ์ฉบับนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญามหาบัณฑิต

คณะกรรมการสอบวิทยานิพนธ์

_____

(รศ. ดร.กิตติศักดิ์ เกิดประสพ)
ประธานกรรมการ

_____

(ผศ. ดร.ชาญวิทย์ แก้วกสิ)
กรรมการ (อาจารย์ที่ปรึกษาวิทยานิพนธ์)

_____

(ผศ. ดร.พิชโยทัย มหัทธนาภิวัฒน์)
กรรมการ

_____

(ศ. ดร.ชูกิจ ลิมปีจำนงค์)
รองอธิการบดีฝ่ายวิชาการและนวัตกรรม

_____

(รศ. ร.อ. ดร.กนต์ธร ชำนิประศาสน์)
คณบดีสำนักวิชาวิศวกรรมศาสตร์

สันติ นภนิภา : การศึกษาและออกแบบความหมายของการแนะนำเชิงลักษณะโดยใช้ตัวรวมไบต์โค้ด (A STUDY AND DESIGN OF THE SEMANTICS OF ASPECT-ORIENTED ADVICE USING BYTECODE COMBINATORS) อาจารย์ที่ปรึกษา : ผู้ช่วยศาสตราจารย์ ดร.ชาญวิทย์ แก้วกสิ, 80 หน้า.

วัตถุประสงค์ของวิทยานิพนธ์นี้เพื่อศึกษาและพัฒนาระบบเชิงลักษณะแบบพลวัตโดยใช้กระบวนการทำงานของคำสั่ง invokedynamic ในการสร้างระบบเพื่อเพิ่มประสิทธิภาพการทำงานของระบบเชิงลักษณะแบบพลวัตให้ดีขึ้น และศึกษาขั้นตอนการสร้างตัวแจงส่วนการตัดจุด (Pointcut parser) เพื่อให้สามารถใช้งานระบบที่พัฒนาได้ง่ายขึ้นและตรงตามส่วนหนึ่งของไวยากรณ์ภาษา AspectJ รุ่น 1.7.1 โดยงานหลักของวิทยานิพนธ์นี้คือการสร้างตัวรวมไบต์โค้ดที่มีชื่อว่า Aspect-aware Bytecode Combinators หรือ AABC ซึ่งมีหน้าที่รวมตัวแนะนำแต่ละแบบ ในส่วนของเครื่องมือที่ใช้สำหรับการพัฒนานี้คือ Bytecode Outline รุ่น 2.4.1 สำหรับการจัดการแก้ไขปัญหาไบต์โค้ดซึ่งทำให้ง่ายต่อการพัฒนา และสำหรับการทดสอบประสิทธิภาพของระบบเชิงลักษณะแบบพลวัตที่สร้างขึ้นจะทำการทดสอบกับชุดทดสอบของ DaCapo และ SciMark 2.0 ซึ่งเป็นชุดทดสอบที่ได้มาตรฐานและเป็นที่ยอมรับในงานวิจัย

สาขาวิชา วิศวกรรมคอมพิวเตอร์ ลายมือชื่อนักศึกษา_____

ปีการศึกษา 2557 ลายมือชื่ออาจารย์ที่ปรึกษา_____

SANTI NOPNIPA : A STUDY AND DESIGN OF THE SEMANTICS OF ASPECT-ORIENTED ADVICE USING BYTECODE COMBINATORS. THESIS ADVISOR : ASST. PROF. CHANWIT KAEWKASI, Ph.D., 80 PP.

AOP/RUNTIME/ INVOKEDYNAMIC/POINTCUT/PARSER/DYNAMIC AOP

The purpose of the work described in this thesis is to study and develop a dynamic aspect-oriented system with the invokedynamic instruction to improve performance. In addition, this work studied techniques for creating a pointcut parser to make the system easier to use. The pointcut's syntax is a subset of AspectJ's syntax version 1.7.1. The main contribution of the thesis is the development of bytecode combinators, Aspect-aware Bytecode Combinators or AABC. AABC is responsible for combing each advice together. In part of tools, this work used Bytecode Outline version 2.4.1 for managing and debugging bytecodes. For the experiments, DaCapo benchmark and SciMark 2.0 benchmark were used because they are standard suites in this research area.

School of Computer Engineering Student's Signature_____

Academic Year 2014 Advisor's Signature_____

กิตติกรรมประกาศ

วิทยานิพนธ์นี้สำเร็จลุล่วงด้วยดี เนื่องจากได้รับความช่วยเหลืออย่างดียิ่ง ทั้งด้านวิชาการและด้านการดำเนินงานวิจัย จากบุคคลและกลุ่มบุคคลต่าง ๆ ได้แก่

ผู้ช่วยศาสตราจารย์ ดร.ชาญวิทย์ แก้วกสิ อาจารย์ที่ปรึกษาวิทยานิพนธ์ ที่ให้คำปรึกษาในการแก้ไขปัญหาต่าง ๆ ของงานวิจัย ผู้ช่วยศาสตราจารย์ ดร.พิชโยทัย มหัทธนาภิวัฒน์ รองศาสตราจารย์ ดร.กิตติศักดิ์ เกิดประสพ และรองศาสตราจารย์ ดร.นิตยา เกิดประสพ อาจารย์ประจำสาขาวิชาวิศวกรรมคอมพิวเตอร์ และ ดร.พิชญา แก้วกสิ ที่ให้คำปรึกษาด้านวิชาการ และการดำเนินการต่าง ๆ

คุณกัลญา พับโพธิ์ เลขานุการสาขาวิชาวิศวกรรมคอมพิวเตอร์ ที่ให้ความช่วยเหลือในการประสานงานด้านเอกสารระหว่างศึกษา

คุณนริศรา ธาระพุทธ คุณชลวิศว์ เสภาศีราภรณ์ คุณนันทวุฒิ คะอังกุ และนักศึกษาบัณฑิตสาขาวิชาวิศวกรรมคอมพิวเตอร์ ทุกท่านที่ให้คำปรึกษาและช่วยเหลือด้วยดีมาโดยตลอด

นอกจากนี้ขอขอบคุณ ครู อาจารย์ ทั้งในอดีตและปัจจุบันที่ให้ความรู้แก่ผู้วิจัยจนประสบความสำเร็จในชีวิต

ท้ายที่สุดที่จะลืมไม่ได้ ขอกราบขอบพระคุณ บิดา มารดา ที่ให้กำเนิด อบรม เลี้ยงดูด้วยความรัก และส่งเสริมการศึกษาเป็นอย่างดีโดยตลอด ทำให้ผู้วิจัยมีความรู้ ความสามารถ มีจิตใจที่เข็มแข็ง รวมทั้งเป็นกำลังใจที่ยิ่งใหญ่แก่ผู้วิจัย จนทำให้ผู้วิจัยประสบความสำเร็จในชีวิตเรื่อยมา

สันติ นภนิภา

สารบัญ

หน้า

สารบัญ (ต่อ)

หน้า

สารบัญ (ต่อ)

หน้า

สารบัญตาราง

ตารางที่ **หน้า**

สารบัญตาราง (ต่อ)

ตารางที่ **หน้า**

สารบัญรูป

รูปที่ **หน้า**

สารบัญรูป (ต่อ)

รูปที่ **หน้า**

สารบัญรูป (ต่อ)

รูปที่ **หน้า**

บทที่ 1

บทนำ

1.1 ความสำคัญและที่มาของปัญหาการวิจัย

ในปัจจุบันเทคโนโลยีได้เข้ามามีบทบาทในชีวิตประจำวันเป็นอย่างมาก ทำให้ความต้องการในการใช้ซอฟต์แวร์เพื่ออำนวยความสะดวกในการใช้งานหรือกิจกรรมต่าง ๆ เกิดขึ้นเป็นจำนวนมาก ในการตอบสนองต่อความต้องการในยุคปัจจุบันส่งผลให้เกิดการพัฒนาซอฟต์แวร์เป็นไปอย่างรวดเร็ว การพัฒนาซอฟต์แวร์ที่มีประสิทธิภาพและสามารถพัฒนาได้อย่างรวดเร็วนั้นเป็นสิ่งสำคัญ และเครื่องมือต่าง ๆ ที่ช่วยอำนวยความสะดวกในการพัฒนาและบำรุงรักษาระบบนั้นมีส่วนสำคัญอย่างยิ่ง ถ้าหากจำเป็นต้องมีการปรับปรุงแก้ไขระบบและต้องมีการปิดระบบและทำการเปิดการทำงานระบบใหม่ อาจทำให้เกิดความยุ่งยากในการเตรียมการเพื่อเริ่มทำงานระบบใหม่อีกครั้ง และยังแฝงมาด้วยค่าใช้จ่ายที่ต้องสูญเสียไป ดังนั้นจึงทำให้เกิดแนวคิดในการปรับปรุงแก้ไขระบบในช่วงเวลาโปรแกรมกำลังทำงาน ซึ่งการปรับปรุงแก้ไขระบบในลักษณะดังกล่าวนี้เรียกว่า การปรับปรุงแก้ไขแบบพลวัต

จากความจำเป็นที่ต้องมีการปรับปรุงแก้ไขระบบในช่วงเวลาโปรแกรมกำลังทำงานอยู่นั้น ได้รับความสนใจโดยได้มีการวิจัยเกิดขึ้นเป็นจำนวนมากซึ่งจะกล่าวต่อไปในบทที่ 2 และเมื่อเร็ว ๆ นี้ Java™ Platform Standard Edition 7 ได้เพิ่มขีดความสามารถในการปรับปรุงแก้ไขการทำงานของโปรแกรม โดยทำการเพิ่มคำสั่งใหม่ที่สามารถทำงานแบบพลวัตลงในจาวาเวอร์ชวลแมชีน (Java virtual machine หรือ JVM) เพื่อตอบสนองภาษาพลวัตที่ใช้จาวาเวอร์ชวลแมชีนในการทำงาน ซึ่งคำสั่งไบต์โค้ดที่เพิ่มเข้าไปในจาวาเวอร์ชวลแมชีนก็คือคำสั่ง invokedynamic (Rose, 2009) โดยช่วยให้ระบบสามารถปรับเปลี่ยนการทำงานช่วงเวลาโปรแกรมกำลังทำงานได้ง่ายขึ้น จึงทำให้คำสั่ง invokedynamic ได้รับการพัฒนาอย่างต่อเนื่องในช่วงที่ Java™ Platform Standard Edition 7 ยังเป็นช่วงทดลองใช้

การสร้างระบบขึ้นมาหนึ่งระบบและทำการเขียนโปรแกรมการทำงานไว้ที่เดียวกัน จะทำให้เกิดความยุ่งยากเป็นอย่างมากในการจัดการหรือแก้ไขการทำงานของระบบ และบางครั้งอาจมีการเขียนโปรแกรมที่มีหน้าที่เหมือนกันเกิดซ้ำหลายแห่งทำให้สิ้นเปลืองทรัพยากรเป็นอย่างมากในกรณีที่เป็นระบบขนาดใหญ่ ดังนั้นจึงเกิดแนวคิดในการแยกการทำงานตามลักษณะหน้าที่ออกจากกัน

โดยแต่ละชิ้นส่วนที่แยกออกมาเรียกว่าโมดูล ซึ่งช่วยให้นักพัฒนาโปรแกรมสามารถทำการพัฒนาโปรแกรมเป็นไปอย่างรวดเร็วและมีประสิทธิภาพ และลักษณะการแยกการทำงานออกเป็นโมดูลซึ่งพบในการเขียนโปรแกรมโดยใช้หลักการของการโปรแกรมเชิงวัตถุ (Object-oriented programming หรือ OOP) (Tim, 1982) ซึ่งจะแบ่งการทำงานต่าง ๆ ออกจากกันอย่างชัดเจนโดยจะมองหน้าที่การทำงานแต่ละอย่างเป็นวัตถุ ดังเช่นภาษาจาวาเป็นภาษาที่ใช้หลักการนี้ในการเขียนระบบซึ่งทำให้การสร้างและการปรับปรุงประสิทธิภาพของระบบทำได้ง่ายขึ้น ด้วยเหตุนี้ทำให้ภาษาจาวาได้รับความนิยมในการเขียนโปรแกรม

อย่างไรก็ตามการโปรแกรมเชิงวัตถุก็ยังมีข้อจำกัดบางประการ โดยโปรแกรมการทำงานบางอย่างที่มีลักษณะการทำงานที่เหมือนกันแต่ไม่สามารถแยกออกมาเป็นโมดูลได้ ทั้งในกระบวนการของการออกแบบโปรแกรม ซึ่งบางครั้งไม่สามารถออกแบบให้เป็นคลาสที่แยกการทำงานเฉพาะออกมาได้ และในบางกรณีอาจเกิดจากการพัฒนาระบบที่มีขนาดใหญ่ ซึ่งโอกาสที่จะเกิดการเขียนโปรแกรมที่มีการทำงานเหมือนกันเขียนซ้ำขึ้นได้ ด้วยเหตุนี้จึงทำให้เกิดแนวคิดในการเพิ่มประสิทธิภาพของการพัฒนาซอฟต์แวร์ระบบให้ดีขึ้นเพื่อขจัดข้อจำกัดที่อาจเกิดขึ้นนี้ และทำให้โปรแกรมสามารถแยกการทำงานออกจากกันตามลักษณะหน้าที่ของการทำงานได้ โดยวิธีสำหรับช่วยแก้ไขปัญหาดังกล่าวก็คือหลักการของการโปรแกรมเชิงลักษณะ (Aspect-oriented programming หรือ AOP) (Kiczales et al., 1997) ซึ่งหลักการนี้จะทำการมองจากลักษณะของการทำงานที่เหมือนกันและกระจายตัวอยู่ในโปรแกรมตามการตัดขวาง (Cross-cutting) โดยทำการเลือกจากการตัดจุดซึ่งเป็นเงื่อนไขที่กำหนดจากลักษณะของการทำงานที่เหมือนกันและกระจายตัวอยู่ วิธีการดังกล่าวนี้ทำให้เพิ่มสภาพโมดูลของระบบให้ดีขึ้น โดยโมดูลที่ได้มาจะสามารถทำการปรับเปลี่ยนแก้ไขการทำงานต่าง ๆ ของโปรแกรมได้ เครื่องมือที่ช่วยในการสร้างโมดูลตามหลักการของการโปรแกรมเชิงลักษณะได้มีการพัฒนาอย่างต่อเนื่อง โดยตัวที่นิยมและเป็นที่รู้จักก็คือภาษา AspectJ (Kiczales et al., 2001) ซึ่งเป็นภาษาและชุดของคอมไพเลอร์ (Compiler) ที่ออกแบบมาสำหรับใช้กับภาษาจาวา รายละเอียดของการโปรแกรมเชิงลักษณะจะอธิบายต่อไปในบทที่ 2

การทำงานตามหลักการของการโปรแกรมเชิงลักษณะใน AspectJ ที่กล่าวมาข้างต้นนี้ได้รับการพัฒนาขึ้นมาเพื่อทำงานร่วมกับภาษาจาวาที่มีอยู่ โดยกระทำการตอนช่วงเวลาโหลดโปรแกรมหรือช่วงเวลาแปลโปรแกรมเท่านั้น ยังไม่สามารถทำการปรับเปลี่ยนแก้ไขการทำงานต่าง ๆ ได้ ณ ช่วงโปรแกรมกำลังทำงาน กล่าวคือยังมีลักษณะการทำงานที่เป็นระบบเชิงลักษณะแบบสถิตย์ (Static aspect-oriented system) อยู่ จากความจำเป็นที่ต้องการให้ระบบซอฟต์แวร์สามารถทำการปรับเปลี่ยนแก้ไขได้ในช่วงโปรแกรมกำลังทำงานตามที่ได้กล่าวไว้ข้างต้น ทำให้มีนักพัฒนาซอฟต์แวร์หลายท่านให้ความสนใจในการพัฒนาและวิจัยเพื่อเพิ่มขีดความสามารถให้ระบบเชิงลักษณะ (Aspect-oriented system) ทำงานได้อย่างยืดหยุ่น ซึ่งจะเรียกว่าระบบเชิงลักษณะแบบพลวัต (Dynamic aspect-

oriented system) โดยงานวิจัยที่เกี่ยวข้องจะกล่าวต่อไปในบทที่ 2 แต่จากงานวิจัยพบว่าการทำงานของระบบเชิงลักษณะแบบพลวัตมีประสิทธิภาพช้ากว่าระบบเชิงลักษณะแบบสถิตย์เป็นอย่างมาก ด้วยเหตุนี้จึงทำให้เกิดมีการวิจัยเพื่อจัดการกับปัญหาดังกล่าวเกิดขึ้นเป็นจำนวนมาก

จากการศึกษากลไกการทำงานของคำสั่ง invokedynamic พบว่าคำสั่งนี้สามารถนำมาสร้างระบบเชิงลักษณะแบบพลวัตได้ งานวิจัยนี้จึงเห็นความสำคัญและแนวทางในการพัฒนาให้ระบบเชิงลักษณะแบบพลวัตสามารถทำงานอย่างมีประสิทธิภาพมากขึ้นกว่าก่อน

1.2 วัตถุประสงค์ของการวิจัย

1.2.1 เพื่อศึกษาวิธีการสร้างระบบเชิงลักษณะแบบพลวัตโดยใช้กลไกการทำงานของคำสั่ง invokedynamic ที่พบใน Java™ Platform Standard Edition 7

1.2.2 เพื่อเพิ่มประสิทธิภาพการใช้งานระบบเชิงลักษณะแบบพลวัตให้ดีขึ้น โดยทำการพัฒนาตัวรวมไบต์โค้ดใหม่และทำการเทียบประสิทธิภาพกับระบบเชิงลักษณะแบบพลวัตที่ประยุกต์ใช้จากตัวรวมไบต์โค้ดที่มีอยู่ในภาษาจาวา

1.2.3 เพื่อศึกษาวิธีการทดสอบประสิทธิภาพการทำงานของระบบเชิงลักษณะแบบพลวัตที่พัฒนาขึ้น เทียบกับระบบเชิงลักษณะแบบพลวัตที่ประยุกต์ใช้ตัวรวมไบต์โค้ดที่มีอยู่ในภาษาจาวา

1.3 สมมุติฐานการวิจัย

1.3.1 การประยุกต์ใช้คำสั่ง invokedynamic ในการสร้างระบบเชิงลักษณะแบบพลวัตและการพัฒนาตัวรวมไบต์โค้ดใหม่จะทำให้ประสิทธิภาพของระบบเชิงลักษณะแบบพลวัตดีขึ้นกว่าระบบเชิงลักษณะแบบพลวัตที่ใช้คำสั่ง invokedynamic ในการสร้างเหมือนกันแต่ใช้ตัวรวมไบต์โค้ดที่มีอยู่เดิมในภาษาจาวา

1.4 คำถามเกี่ยวกับการวิจัย

1.4.1 ประสิทธิภาพของระบบเชิงลักษณะแบบพลวัตที่สร้างขึ้นโดยการประยุกต์ใช้คำสั่ง invokedynamic จะดีขึ้นจริงหรือไม่อย่างไร

1.5 ข้อตกลงเบื้องต้น

1.5.1 ระบบเชิงลักษณะแบบพลวัตที่พัฒนาขึ้นนี้จะต้องทำการทดสอบกับระบบจำลองก่อนที่จะนำไปใช้กับระบบจริงเพื่อป้องกันความผิดพลาดที่อาจเกิดขึ้น

1.5.2 เครื่องมือที่พัฒนาขึ้นนี้ไม่สมบูรณ์ทั้งหมดเพราะพบว่ายังมีข้อจำกัดของภาษาจาวาที่ไม่อนุญาตให้ทำการแปลงคำสั่ง invokespecial ในส่วนของการเรียกเมธอด super() และเมธอด this() ได้เนื่องจากในส่วนของเมธอด findSpecial ที่ภาษาจาวามีอยู่นั้นไม่อนุญาตให้ค้นหาชื่อเมธอด <init> ซึ่งเป็นชื่อสำหรับใช้เรียกเมธอด super() หรือเมธอด this() ได้

1.6 ขอบเขตของการวิจัย

1.6.1 การศึกษาทดสอบประสิทธิภาพการใช้คำสั่ง invokedynamic ทำการอ้างอิงไบต์โค้ดในจาวาเวอร์ชวลแมชีน รุ่น 1.7 เท่านั้น

1.6.2 การทดสอบประสิทธิภาพการทำงานของระบบเชิงลักษณะแบบพลวัตที่พัฒนาขึ้นจะทำการทดสอบกับชุดทดสอบมาตรฐานบางรายการที่อยู่ในชุดทดสอบ DaCapo (Blackburn et al., 2006) และ SciMark 2.0 (Pozo and Miller, 2010) เท่านั้น

1.6.3 ในการทดสอบประสิทธิภาพของระบบเชิงลักษณะแบบพลวัตที่ได้ทำการพัฒนาขึ้นจะทำการทดสอบเทียบกับ AspectJ รุ่น 1.7.1 เท่านั้น

1.7 ประโยชน์ที่คาดว่าจะได้รับ

1.7.1 สามารถแยกโมดูลของระบบออกมาได้โดยใช้งานตัวรวมไบต์โค้ดที่พัฒนาขึ้นสำหรับใช้งานระบบเชิงลักษณะแบบพลวัต

1.7.2 สามารถช่วยให้นักพัฒนาปรับปรุงเปลี่ยนแปลงแก้ไขการทำงานของระบบได้ง่ายขึ้นโดยไม่ต้องทำการปิดการทำงานของระบบ

1.7.3 สามารถช่วยลดค่าใช้จ่ายที่ต้องสูญเสียไปจากการบำรุงรักษาระบบได้

1.7.4 ช่วยเพิ่มประสิทธิภาพการทำงานของระบบได้

1.8 คำอธิบายศัพท์

1.8.1 การโปรแกรมเชิงลักษณะ (Aspect-oriented programming) หรือ AOP

หมายถึง การเขียนโปรแกรมที่มองจากลักษณะการทำงานของโปรแกรมที่เหมือนกันและกระจายตัวอยู่ในโค้ด โดยจะนำส่วนของโปรแกรมที่มีลักษณะเหมือนกันนำมารวมกัน ซึ่งจะได้โมดูลที่สามารถทำการปรับเปลี่ยนการทำงานต่าง ๆ ของโปรแกรมง่ายขึ้น

1.8.2 เมธอดเริ่มต้น (Bootstrap method)

หมายถึง เมธอด (Method) ที่ถูกเรียกใช้งานเมื่อคำสั่ง invokedynamic ถูกเรียกใช้งานเป็นครั้งแรก โดยทำหน้าที่เชื่อมโยงคำสั่ง invokedynamic กับเมธอดเป้าหมาย

1.8.3 คอลไซต์ (Call site)

หมายถึง ตำแหน่งการเรียกใช้งานเมธอดในโปรแกรมโดยเป็นตัวแทนของตำแหน่ง ดังกล่าวคือคลาส CallSite ซึ่งอยู่ในระบบของ Java Development Kit หรือ JDK

1.8.4 วัตถุคอลไซต์ (Call site object)

หมายถึง วัตถุที่แทนตำแหน่งการใช้งานเมธอดซึ่งเป็นวัตถุของคลาส CallSite ที่ถูกส่งออกมาจากเมธอดเริ่มต้นสำหรับใช้งานกับคำสั่ง invokedynamic

1.8.5 โมดูล (Module)

หมายถึง การแบ่งโปรแกรมออกตามลักษณะหน้าที่ตามการทำงานที่แตกต่างกันเพื่อช่วยให้สามารถทำการปรับเปลี่ยนแก้ไขโปรแกรมได้ง่ายขึ้น

1.8.6 ช่วงเวลาโปรแกรมกำลังทำงาน (Run time)

หมายถึง ช่วงเวลาขณะโปรแกรมกำลังทำงานอยู่

1.8.7 ช่วงเวลาโหลดโปรแกรม (Load time)

หมายถึง ช่วงเวลาระหว่างโหลดโปรแกรม

1.8.8 ช่วงเวลาแปลโปรแกรม (Compile time)

หมายถึง ช่วงเวลาในการแปลโปรแกรม

1.8.9 ความเกี่ยวพันเชิงตัดขวาง (Cross-cutting concern)

หมายถึง การตัดขวางโปรแกรมที่มีลักษณะการทำงานที่เหมือนกันแต่กระจายตัวอยู่ให้สามารถเพิ่มโมดูลให้กับระบบได้

1.8.10 จุดร่วม (Join point)

หมายถึง จุดของโปรแกรมที่ทำการตัดขวางออกมาเพื่อรวมเป็นโมดูลโดยได้จากลักษณะที่เหมือนกันจากเงื่อนไขของการตัดจุด

1.8.11 การตัดจุด (Pointcut)

หมายถึง ส่วนที่ทำการกำหนดลักษณะในส่วนของโปรแกรมที่ต้องการตัดออกมา โดยอาจเป็นการเรียกใช้งานเมธอดหรือการเรียกใช้งานคอนสตรัคเตอร์ (Constructor) ก็ได้

1.8.12 ตัวกำหนดการตัดจุด (Pointcut designators) หรือ พีซีดี (PCD)

หมายถึง เงื่อนไขที่ใช้สำหรับการตัดจุดเพื่อทำการเลือกจุดร่วมที่ต้องการสำหรับสานตัวแนะนำเชิงลักษณะเข้าไป

1.8.13 ระบบเชิงลักษณะแบบสถิตย์ (Static aspect-oriented system)

หมายถึง ระบบเชิงลักษณะที่ไม่สามารถทำการปรับเปลี่ยนแก้ไขการทำงานของโปรแกรมได้ในช่วงเวลาโปรแกรมกำลังทำงานอยู่ได้

1.8.14 ระบบเชิงลักษณะแบบพลวัต (Dynamic aspect-oriented system)

หมายถึง ระบบเชิงลักษณะที่สามารถทำการปรับเปลี่ยนแก้ไขการทำงานของโปรแกรมได้ในช่วงเวลาโปรแกรมกำลังทำงาน

1.8.15 ไบต์โค้ด (Bytecode)

หมายถึง รหัสที่ใช้ทำงานกับเครื่องจักรเสมือนโดยในภาษาจาวาก็คือไฟล์ .class ที่ได้จากการคอมไพล์ไฟล์ .java

1.8.16 เครื่องจักรเสมือน (Virtual machine)

หมายถึง เครื่องจักรที่ทำหน้าที่เป็นตัวกลางของการทำงานระหว่างไบต์โค้ดของโปรแกรมกับเครื่องจักร ในภาษาจาวาเครื่องจักรเสมือนช่วยให้ภาษาจาวาสามารถทำงานกับสถาปัตยกรรมของเครื่องจักรที่มีสภาพแวดล้อมที่แตกต่างกันได้

1.8.17 โปรซีด (Proceed)

หมายถึง ส่วนที่ใช้สำหรับเรียกใช้งานคำสั่งเดิมของโปรแกรมที่ต้องการเรียกใช้งาน โดยโปรซีดจะถูกเรียกใช้งานในตัวแนะนำครอบ

1.8.18 ตัวแนะนำก่อน (Before advice)

หมายถึง ตัวแนะนำสำหรับใช้ทำงานก่อนโปรแกรม ณ จุดร่วมจะเริ่มทำงาน

1.8.19 ตัวแนะนำหลัง (After advice)

หมายถึง ตัวแนะนำสำหรับใช้ทำงานหลังจากการทำงานของโปรแกรมที่จุดร่วมต้องการเรียกใช้งานเสร็จสิ้นลง

1.8.20 ตัวแนะนำหลังการคืนค่า (After-return advice)

หมายถึง ตัวแนะนำสำหรับใช้ทำงานหลังเสร็จสิ้นการทำงานของโปรแกรมตรงจุดร่วมเสร็จสิ้นและตัวแนะนำนี้สามารถเปลี่ยนแปลงค่าของค่าคืนกลับ (Return value) ได้

1.8.21 ตัวแนะนำครอบ (Around advice)

หมายถึง ตัวแนะนำที่สามารถเปลี่ยนการทำงานตรงจุดร่วมได้ทั้งหมด และสามารถเรียกใช้งานของการทำงานเดิมผ่านการเรียกใช้โปรซีด

บทที่ 2

ปริทัศน์วรรณกรรมและงานวิจัยที่เกี่ยวข้อง

ในบทนี้จะเป็นส่วนที่เกี่ยวกับการทบทวนวรรณกรรมและงานวิจัยที่เกี่ยวข้องซึ่งประกอบไปด้วยรายละเอียดทฤษฎีกระบวนการทำงานของคำสั่ง invokedynamic ที่ใช้สำหรับสร้างระบบเชิงลักษณะแบบพลวัต ทฤษฎีการโปรแกรมเชิงลักษณะและงานวิจัยที่เกี่ยวข้อง

2.1 การโปรแกรมเชิงลักษณะ

ในทางด้านวิศวกรรมซอฟต์แวร์ การแยกหน่วยที่มีการทับซ้อนกันในระบบออกจากกันได้ถือเป็นสิ่งที่ช่วยลดความซับซ้อนของซอฟต์แวร์ซึ่งทำให้การจัดการพัฒนาดูแลซอฟต์แวร์เป็นไปง่ายขึ้น โดยโปรแกรมหรือกรอบงาน (Framework) ทั่วไปจะมีกลไกสำหรับช่วยแยกหน่วยเหล่านี้แต่สามารถแยกได้ในระดับหนึ่งเท่านั้น หรือในส่วนของการโปรแกรมเชิงวัตถุก็มีการแยกหน่วยโดยอยู่ในรูปคลาส ซึ่งถ้าใช้การโปรแกรมเชิงวัตถุเข้ามาช่วยในการพัฒนาซอฟต์แวร์จะทำให้ผู้ดูแลและพัฒนาซอฟต์แวร์สามารถทำงานได้ง่ายขึ้น อย่างไรก็ตามการโปรแกรมเชิงวัตถุก็ยังมีบางส่วนของโปรแกรมที่ยังขึ้นต่อกันอยู่ทำให้ไม่สามารถแยกหน่วยออกมาได้ จากปัญหาดังกล่าวทำให้ Kiczales และคณะได้คิดค้นและวิจัยเพื่อจัดการกับปัญหาดังกล่าว โดยได้นำเสนอหลักการการโปรแกรมเชิงลักษณะ (Kiczales et al., 1997) ซึ่งมีองค์ประกอบและรายละเอียดดังต่อไปนี้

2.1.1 ลักษณะ

ตามหลักการของการโปรแกรมเชิงวัตถุคลาสเปรียบได้กับหน่วยหนึ่งหน่วยที่แยกออกมา เช่นเดียวกับหน่วยในการโปรแกรมเชิงลักษณะก็คือ ลักษณะ (Aspect) โดยเป็นหน่วยที่เกิดจากกลไกจากการตัดขวางเลือกลักษณะที่เหมือนกันของโปรแกรมออกมา โดยการโปรแกรมเชิงวัตถุจะมีบางส่วนของโปรแกรมที่เหมือนกันและต้องเขียนซ้ำกันหลายตำแหน่งแต่ไม่สามารถแยกออกมาเป็นหน่วยได้ ดังนั้นโดยหลักการของการโปรแกรมเชิงลักษณะที่ได้กล่าวมาสามารถช่วยจัดการกับข้อจำกัดดังกล่าวได้

ในการศึกษารูปแบบการทำงานของลักษณะในการโปรแกรมเชิงลักษณะได้ทำการศึกษาจาก AspectJ (Kiczales et al., 2001) ซึ่งเป็นภาษาสำหรับใช้สร้างตามหลักการของการ

โปรแกรมเชิงลักษณะซึ่งใช้งานร่วมกับภาษาจาวา เพื่อให้สามารถทำความเข้าใจได้ง่าย ลักษณะประกอบไปด้วยเขตข้อมูลส่วนตัว (Private field) เมธอดและยังสามารถมีลักษณะย่อยได้โดยการใช้คีย์เวิร์ด “extends” เหมือนกับคลาสในการเขียนโปรแกรมตามหลักการโปรแกรมเชิงวัตถุ โดยภายในลักษณะประกอบไปด้วยการตัดจุดสำหรับการเลือกจุดตัดขวางที่มีลักษณะตรงตามเงื่อนไขที่กำหนด ซึ่งรายละเอียดของการตัดจุดจะกล่าวในส่วนถัดไป และประกอบด้วยตัวแนะนำเชิงลักษณะ (AOP advice) สำหรับใช้ในการดำเนินการต่าง ๆ โดยรายละเอียดจะกล่าวต่อไป สำหรับตัวอย่างการเขียนลักษณะใน AspectJ แสดงในรูปที่ 2.1

2.1.2 การตัดจุด

การตัดจุดจะถูกเขียนขึ้นในลักษณะซึ่งประกอบไปด้วยเงื่อนไขต่าง ๆ สำหรับใช้เลือกจุดที่ต้องการตัด โดยมีตัวกำหนดการตัดจุด (Pointcut designators หรือ พีซีดี) ซึ่งเป็นประโยคบูลีน (Boolean) สำหรับตำแหน่งที่ตรงกับเงื่อนไขการตัดจุดจะเรียกว่าจุดร่วม (รายละเอียดจะกล่าวในส่วนถัดไป) ซึ่งจะสามารถนำตัวแนะนำเชิงลักษณะสานเข้าไป ณ จุดร่วมแต่ละจุดได้

```
1 public aspect SimpleAspect {
2    pointcut sim1() : call(* *(..)) && within(Boo);
3    before() : sim1() {
4
5    }
6 }
```

รูปที่ 2.1 ตัวอย่างลักษณะที่เขียนในภาษา AspectJ

จากตัวอย่างของการกำหนดเงื่อนไขการตัดจุดได้แสดงในตัวอย่างรูปที่ 2.1 ซึ่งถูกเขียนขึ้นในบรรทัดที่ 2 ภายในลักษณะ (Aspect) ที่ชื่อว่า SimpleAspect โดยการตัดจุดในตัวอย่างชื่อว่า sim1() ซึ่งมีเงื่อนไขการตัดจุดเป็น call(* *(..)) && within(Boo); โดยประกอบไปด้วยพีซีดี 2 ตัว พีซีดีตัวแรกคือ call(* *(..)) หมายความว่าทำการเลือกจุด ณ ตำแหน่งที่มีการเรียกทั้งหมด และพีซีดีตัวที่สองคือ within(Boo) หมายถึงการเลือกจะต้องอยู่ในคลาส Boo แล้วทำการเชื่อมทั้งสองพีซีดีด้วยเงื่อนไข && (และ) ถ้าบริเวณของโปรแกรมมีลักษณะตรงตามเงื่อนไขนี้ก็จะถูกตัดออกมาแล้วทำการสานตัวแนะนำเชิงลักษณะเข้าไป นอกจากตัวอย่างการสร้างเงื่อนไขการตัดจุดนี้ ประโยคเงื่อนไขการ

ตัดจุดสามารถเขียนให้มีความซับซ้อนได้ตามสภาพแวดล้อมของจุดที่ต้องการตัด โดยสามารถนำพีซีดีมาประกอบเข้าด้วยกันได้

2.1.3 จุดร่วม

จุดร่วมเป็นองค์ประกอบที่จำเป็นในการออกแบบของการโปรแกรมเชิงลักษณะ โดยเป็นจุดที่ได้จากเงื่อนไขที่กำหนดในพีซีดีของการตัดจุด จุดร่วมสามารถเป็นตำแหน่งของการเรียกใช้งานเมธอดหรือคอนสตรัคเตอร์ได้ โดยสามารถทำการสานตัวแนะนำเชิงลักษณะเข้าไปตามที่ผู้พัฒนาซอฟต์แวร์กำหนดไว้ สำหรับรายละเอียดของตัวแนะนำเชิงลักษณะจะกล่าวในส่วนถัดไป

2.1.4 ตัวแนะนำ

ตัวแนะนำ (Advice) คือส่วนที่จะนำไปสาน ณ จุดร่วม โดยผู้พัฒนาซอฟต์แวร์สามารถสร้างขึ้นสำหรับใช้งานให้เหมาะสมตามการทำงานของโปรแกรมที่ต้องการ ซึ่งตัวแนะนำเชิงลักษณะแบ่งออกได้สามประเภทหลัก ๆ คือ ตัวแนะนำก่อน ตัวแนะนำหลัง และตัวแนะนำครอบ (Kiczales et al., 2001) โดยตัวแนะนำแต่ละประเภทมีหน้าที่ และความแตกต่างกันดังต่อไปนี้

1) ตัวแนะนำก่อนเป็นฟังก์ชันการทำงานที่ดำเนินการก่อนโปรแกรมจริง ณ จุดร่วมจะเริ่มทำงาน โดยความสามารถของตัวแนะนำก่อนคือสามารถทำการสังเกตค่าของพารามิเตอร์ที่ทำการส่งให้คอนสตรัคเตอร์หรือเมธอดของโปรแกรมจริงได้ และยังสามารถทำการเพิ่มการทำงาน ณ ตำแหน่งก่อนการทำงานของโปรแกรมจริงจะเริ่มทำงาน แต่ไม่สามารถทำการเปลี่ยนแปลงค่าของพารามิเตอร์ที่จะส่งไปให้โปรแกรมจริงได้ โดยตัวอย่างการสร้างตัวแนะนำก่อนที่ใช้งานจริงใน AspectJ แสดงในรูปที่ 2.2 ในส่วนที่ 1

2) ตัวแนะนำหลังเป็นฟังก์ชันที่ใช้สำหรับทำงานหลังโปรแกรม ณ จุดร่วมทำงานเสร็จแล้ว โดยตัวแนะนำหลังนี้สามารถทำการดูค่าคืนกลับและค่าพารามิเตอร์ต่าง ๆ ณ จุดร่วมนั้นได้ และสามารถทำการเพิ่มการทำงานหลังโปรแกรม ณ จุดร่วมทำงานเสร็จ แต่ไม่สามารถทำการเปลี่ยนแปลงค่าคืนกลับได้ ตัวอย่างการสร้างตัวแนะนำหลังที่เขียนขึ้นโดยใช้ AspectJ แสดงในรูปที่ 2.2 ในส่วนที่ 2 อย่างไรก็ตามมีตัวแนะนำอีกแบบหนึ่งที่สามารถทำการเปลี่ยนแปลงค่าคืนกลับได้ ซึ่งก็คือตัวแนะนำหลังการคืนค่าที่มีลักษณะการทำงานคล้ายกับตัวแนะนำหลัง แต่มีความสามารถพิเศษที่สามารถเปลี่ยนแปลงค่าคืนกลับได้ โดยรูปแบบการสร้างตัวแนะนำหลังการคืนค่าซึ่งสร้างใน AspectJ แสดงในรูปที่ 2.2 ในส่วนที่ 3

3) ตัวแนะนำครอบเป็นฟังก์ชันการทำงานที่สามารถปรับเปลี่ยนการทำงานทั้งหมดของโปรแกรม ณ จุดร่วมได้ โดยส่วนของโปรแกรมที่ถูกเรียกใช้งาน ณ จุดร่วมจะถูกเก็บไว้ในโปรซีด ซึ่งในการเขียนตัวแนะนำครอบสามารถทำการปรับเปลี่ยนการทำงานใหม่ได้ทั้งหมด และ

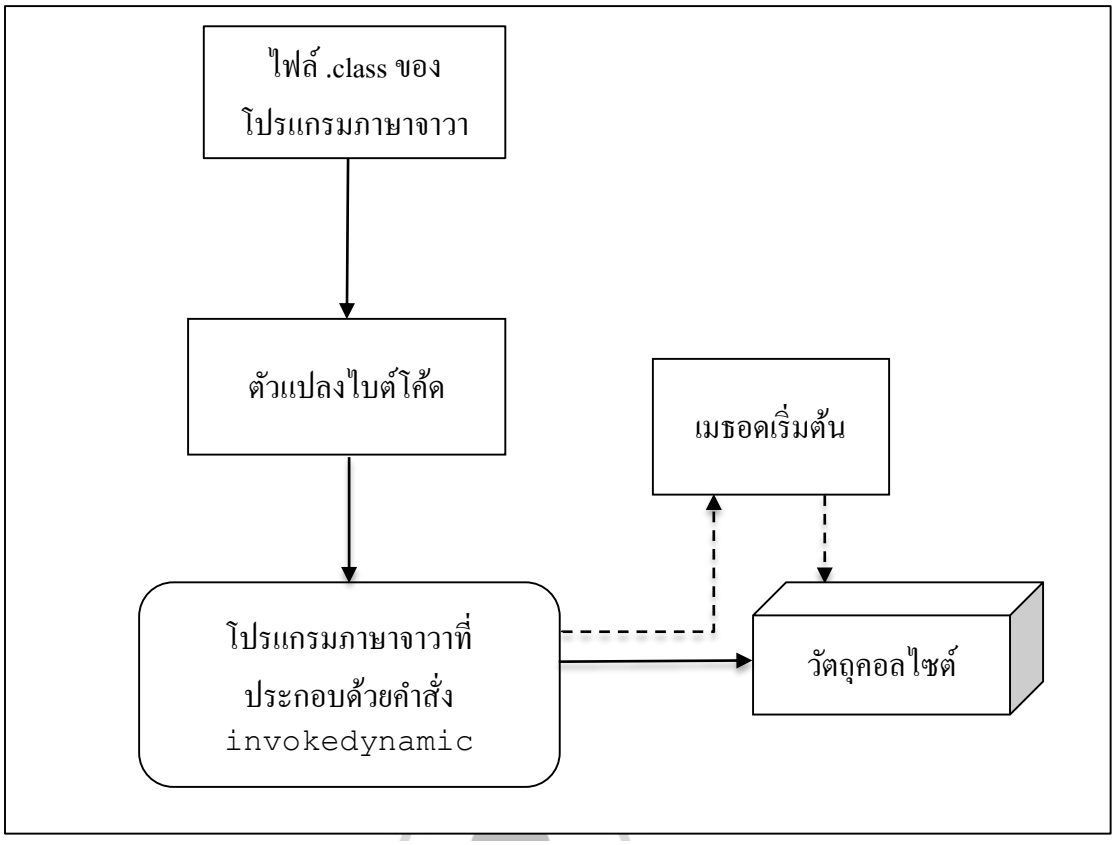
```
//1) Before advice
before() : fib() {
     // write some action here
}

//2) After advice
after() : fib() {
     // write some action here
}

//3) After-return advice
after() returning (Object o): fib() {
     // write some action here
}

//4) Around advice
Object around() : fib() {
     // write some action here
     return proceed();
}
```

รูปที่ 2.2 ตัวอย่างตัวแนะนำเชิงลักษณะแต่ละประเภทที่เขียนใน AspectJ

ยังสามารถเรียกใช้งานในส่วนของโปรแกรมเดิมได้ผ่านโปรซีดซึ่งช่วยให้ผู้พัฒนาซอฟต์แวร์ง่ายต่อการใช้งานในส่วนนี้ โดยการเขียนตัวแนะนำครอบได้แสดงตัวอย่างในรูปที่ 2.2 ในส่วนที่ 4

2.2 คำสั่ง invokedynamic

คำสั่ง invokedynamic (Rose, 2009) เป็นคำสั่งไบต์โค้ดที่ถูกกำหนดและเป็นมาตรฐานอยู่ใน Java Specification Request 292 (Rose, 2008) เพื่อสนับสนุนการทำงานของภาษาพลวัตที่ทำงานอยู่บนจาวาเวอร์ชวลแมชีน โดยแผนภาพกระบวนการทำงานทั้งหมดของคำสั่ง invokedynamic แสดงในรูปที่ 2.3 ซึ่งมีหลักการทำงานโดยเริ่มจากการนำไฟล์ .class เข้ามาประมวลผลเพื่อทำการค้นหา



รูปที่ 2.3 แผนภาพแสดงการทำงานโดยรวมของคำสั่ง invokedynamic

ไบต์โค้ดคำสั่ง invokestatic คำสั่ง invokevirtual คำสั่ง invokeinterface และคำสั่ง invokespecial เพื่อทำการแปลงแต่ละคำสั่งไปเป็นคำสั่ง invokedynamic ตามที่ผู้พัฒนาซอฟต์แวร์ต้องการ โดยองค์ประกอบสำหรับการทำงานตามกระบวนการของคำสั่ง invokedynamic มีดังต่อไปนี้

2.2.1 เมธอดเริ่มต้น

เมธอดเริ่มต้นเป็นเมธอดที่ถูกสร้างขึ้นในขั้นตอนของการแปลงคำสั่งไบต์โค้ดที่ใช้เรียกคอนสตรัคเตอร์หรือเมธอดแต่ละประเภทไปเป็นคำสั่ง invokedynamic โดยเมธอดเริ่มต้นจะถูกเรียกใช้งานก็ต่อเมื่อตำแหน่งที่ถูกแปลงไปเป็นคำสั่ง invokedynamic ถูกเรียกเป็นครั้งแรก ซึ่งเมธอดเริ่มต้นมีหน้าที่สำหรับเชื่อมตำแหน่งที่เรียกใช้งานเมธอดกับเมธอดจริงโดยมีวัตถุของคลาส MethodHandle และวัตถุคอลไซต์เป็นองค์ประกอบสำคัญสำหรับใช้สร้างเมธอดเริ่มต้นซึ่งจะกล่าวในส่วนถัดไป สำหรับตัวอย่างการเขียนเมธอดเริ่มต้นแสดงในรูปที่ 2.4

```
public static java.lang.invoke.CallSite bootstrap(
          MethodHandles.Lookup caller,
          String name,
          MethodType type) throws ...{
     String[] names = name.split(":");
     MethodHandle target =
          mh[Integer.valueOf(names[0])];
     return new ConstantCallSite(target);
}
```

รูปที่ 2.4 ตัวอย่างเมธอดเริ่มต้น

2.2.2 คลาส MethodHandle

คลาส MethodHandle ทำหน้าที่เป็นตัวอ้างอิงไปยังเมธอดหรือคอนสตรัคเตอร์โดยตรง ในบริบทของคำสั่ง invokedynamic หน้าที่ของคลาส MethodHandle เปรียบได้กับพอยเตอร์ที่ชี้ไปยังเมธอดหรือคอนสตรัคเตอร์ที่ต้องการเรียกใช้งานจริง โดยทำการค้นหาคอนสตรัคเตอร์หรือเมธอดที่ต้องการอ้างอิงถึงซึ่งการค้นหาสามารถแบ่งออกตามประเภทของการเรียกใช้งาน โดยหลักการแปลงของการเรียกแต่ละแบบไปเป็นคำสั่ง invokedynamic (Kaewkasi, 2010) ดังนี้

1) เมธอด findStatic ใช้สำหรับค้นหาเมธอดที่เป็นแบบสถิตย์ (Static method)

2) เมธอด findSpecial ใช้สำหรับค้นหาเมธอดที่มีการสืบทอด (Inherited method)

3) เมธอด findConstructor ใช้สำหรับค้นหาคอนสตรัคเตอร์

4) เมธอด findVirtual ใช้สำหรับค้นหาเมธอดทั่วไปหรือเมธอดที่เป็นแบบอินเตอร์เฟส

สำหรับตัวอย่างการใช้งานแสดงในรูปที่ 2.5 ซึ่งเป็นตัวอย่างของเมธอด mhCreator สำหรับมอบค่าวัตถุของคลาส MethodHandle ให้กับตัวแปร mh ซึ่งเป็นตัวแปรชุด (Array) ของคลาส MethodHandle และสำหรับแต่ละคำสั่งทำการค้นหาตัวอย่างภาษาจาวาในรูปที่ 2.6

```
public static void mhCreator() throws Throwable {
     mh = new MethodHandle[METHOD_HANDLE_MAX];
     mh[0] = MethodHandles.lookup().findStatic(
             SUT.class,
             "student",
             MethodType.methodType(
                    void.class,
                    int.class)
             );
     mh[1] =MethodHandles.lookup().findVirtual(
             SUT.class,
             "println",
             MethodType.methodType(
                    void.class,
                    String.class)
             );
     mh[2] = MethodHandles.lookup().findConstructor(
             SUT.class,
             MethodType.methodType(void.class));
}
```

รูปที่ 2.5 ตัวอย่างเมธอด mhCreator สำหรับการมอบค่าวัตถุของคลาส MethodHandle ให้กับตัวแปร mh ซึ่งเป็นตัวแปรชุดของคลาส MethodHandle

2.2.3 วัตถุคอลไซต์

วัตถุคอลไซต์เป็นวัตถุที่แทนตำแหน่งการใช้งานเมธอดซึ่งเป็นวัตถุของคลาส CallSite ที่ถูกส่งออกมาจากเมธอดเริ่มต้นสำหรับใช้งานกับคำสั่ง invokedynamic ซึ่งวัตถุคอลไซต์จะถูกส่งออกมาจากเมธอดเริ่มต้นในครั้งแรกที่คำสั่ง invokedynamic ถูกเรียกใช้งาน โดยถ้าคำสั่ง invokedynamic ถูกเรียกขึ้นอีกครั้งก็จะมาใช้วัตถุคอลไซต์โดยตรงซึ่งไม่ต้องเข้ามาทำงานในเมธอดเริ่มต้นอีก

2.2.4 ตัวอย่างโดยรวมการทำงานของคำสั่ง invokedynamic

จากรูปที่ 2.6 จะเห็นโปรแกรมที่สามารถอ่านแล้วทำความเข้าใจได้ทันที แต่จากรูปนี้โปรแกรมภาษาจาวาจะถูกคอมไพล์ด้วยจาวาคอมไพเลอร์เพื่อแปลงไปเป็นไบต์โค้ดที่สามารถ

```
public class SUT {
     public SUT(String idNumber) {
          System.out.println(idNumber);
     }
     public static void student(int number){
          System.out.println(number);
     }
     public static void main(String args[]){
          SUT n = new SUT("B55xxxxx");
          student(200);
          System.out.println("SURANAREE");
     }
}
```

รูปที่ 2.6 แสดงตัวอย่างโปรแกรมภาษาจาวามีชื่อไฟล์ว่า SUT.java

ทำงานบนจาวาเวอร์ชวลแมชีนได้ โดยสามารถเรียกใช้คำสั่งคอมไพล์ด้วยคำสั่ง “javac ชื่อไฟล์.java” ในตัวอย่างนี้จะทำการคอมไพล์ด้วยคำสั่ง “javac SUT.java” หลังจากการคอมไพล์ตัวโปรแกรมแล้วก็จะได้ไฟล์ที่มีชื่อว่า SUT.class ดังแสดงในรูปที่ 2.7 ต่อไปนี้

สำหรับรูปที่ 2.7 ได้แสดงไบต์โค้ดในส่วนของเมธอด main เท่านั้น สำหรับคำสั่งที่ใช้แสดงไบต์โค้ดคือคำสั่ง javap เหตุผลที่ตัวอย่างใช้ –verbose เพราะต้องการแสดงข้อมูลรายละเอียดของไฟล์ .class ทั้งหมด จากคำสั่งไบต์โค้ดที่แสดงในตัวอย่างนี้พบว่ามีคำสั่ง invokespecial คำสั่ง invokevirtual คำสั่ง invokestatic และยังมีคำสั่ง invokeinterface ซึ่งไม่ได้ถูกแสดงในตัวอย่างนี้ คำสั่งเหล่านี้จะถูกแปลงไปเป็นคำสั่ง invokedynamic ตามความต้องการของผู้พัฒนาซอฟต์แวร์ว่าต้องการแปลงคำสั่งส่วนใดของโปรแกรม สำหรับในส่วนนี้จะทำการพัฒนาตัวแจงส่วนการตัดจุดซึ่งจะกล่าวโดยละเอียดในบทที่ 3 ต่อไป

นอกจากนี้ยังมีเครื่องมือที่ช่วยให้การทำวิจัยได้สะดวกขึ้นโดยใช้ Bytecode Outline (Loskutov, 2005) เป็นเครื่องมือที่พัฒนาขึ้นเพื่อใช้จัดการไบต์โค้ดโดยสามารถใช้งานกับ Eclipse ดังตัวอย่างที่แสดงในรูปที่ 2.8 โดยส่วนที่ 1 ทางด้านซ้ายจะเป็นโปรแกรมภาษาจาวา และส่วนที่ 2 ทางด้านขวาคือส่วนของ Bytecode Outline จะเห็นว่าสามารถใช้วิเคราะห์และจัดการไบต์โค้ดได้สะดวก อย่างไรก็ตามผู้วิจัยได้ลองใช้งาน Bytecode Outline เพื่อค้นหาจุดที่ผิดพลาดของโปรแกรมในส่วนของไบต์โค้ด พบว่า Bytecode Outline ไม่สามารถบอกให้ทราบถึงจุดโปรแกรมที่ผิดพลาดได้อย่างแม่นยำครบถ้วน แต่สามารถบอกได้บางส่วนเท่านั้น แต่ทว่า Bytecode Outline ก็เป็นเครื่องมือที่สำคัญสำหรับช่วยพัฒนาระบบเชิงลักษณะแบบพลวัตนี้

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=2, args_size=1
       0: new           #1    // class sut/jdynmic/SUT
       3: dup
       4: ldc           #36   // String B55xxxxx
       6: invokespecial #38   // Method
          "<init>":(Ljava/lang/String;)V
       9: astore_1
      10: sipush        200
      13: invokestatic  #40   // Method student:(I)V
      16: getstatic     #11   // Field
          java/lang/System.out:Ljava/io/PrintStream;
      19: ldc           #42   // String SURANAREE
      21: invokevirtual #17 // Method
          java/io/PrintStream.println:
          (Ljava/lang/String;)V
      24: return
    LineNumberTable:
      line 11: 0
      line 12: 10
      line 13: 16
      line 14: 24
    LocalVariableTable:
    Start  Length  Slot  Name   Signature
      0      25     0    args   [Ljava/lang/String;
     10      15     1     n     Lsut/jdynmic/SUT;
```

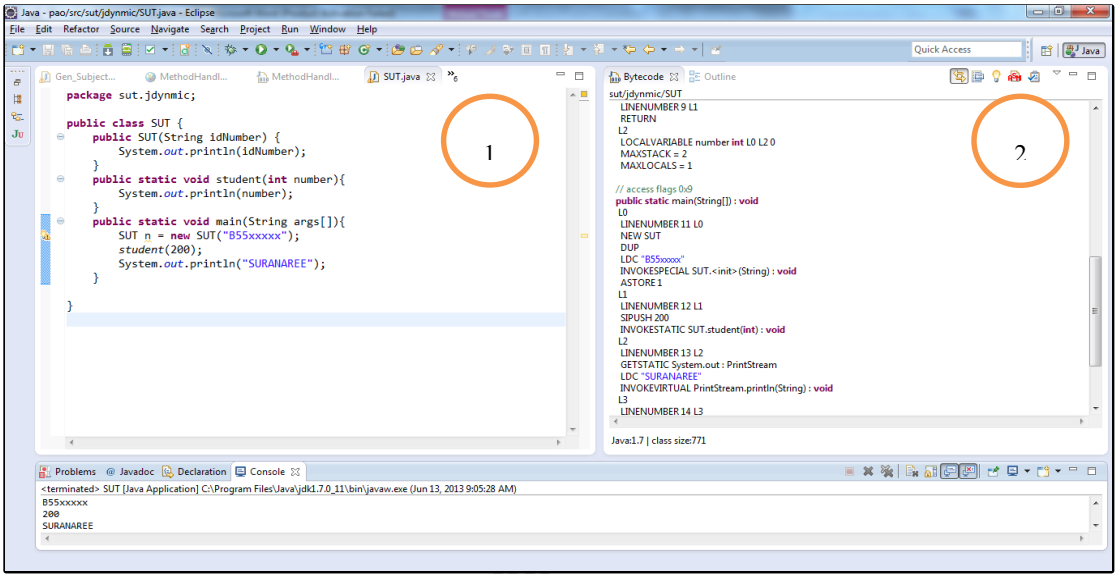
รูปที่ 2.7 แสดงตัวอย่างไฟล์ SUT.class ที่อ่านด้วยคำสั่ง javap –verbose SUT.class

2.3 ระบบเชิงลักษณะ

ระบบเชิงลักษณะ คือ ระบบที่นำเอาหลักการของการโปรแกรมเชิงลักษณะมาใช้ในการพัฒนาระบบโดยก่อให้เกิดประสิทธิภาพในการจัดการระบบที่ดีขึ้นซึ่งมีวิวัฒนาการของเครื่องมือที่ใช้ในการพัฒนาระบบเชิงลักษณะดังต่อไปนี้

2.3.1 ระบบเชิงลักษณะแบบสถิตย์

AspectJ (Kiczales et al., 2001) เป็นการพัฒนาให้การโปรแกรมเชิงลักษณะสามารถใช้งานได้จริง และเป็นประโยชน์อย่างมากเพราะ AspectJ พัฒนาเพื่อใช้งานกับภาษาจาวา ซึ่งสามารถใช้งานกับระบบที่มีอยู่แล้วเพื่อเพิ่มโมดูลให้กับระบบ ทำให้การจัดการดูแลระบบทำได้ง่ายขึ้น



รูปที่ 2.8 ตัวอย่างแสดงโปรแกรม Bytecode Outline ที่สามารถ
ใช้งานวิเคราะห์ไบต์โค้ดได้จากโปรแกรม Eclipse

อย่างไรก็ตามการทำงานของ AspectJ ยังมีข้อจำกัดอยู่ตรงที่ AspectJ ไม่สามารถทำการปรับเปลี่ยนแก้ไขระบบในขณะที่ระบบกำลังทำงานอยู่ได้ โดยระบบที่มีการทำงานในลักษณะนี้จะเรียกว่า ระบบเชิงลักษณะแบบสถิตย์ ซึ่งจะเกิดความยุ่งยากกับผู้ดูแลระบบเมื่อต้องทำการปรับปรุงแก้ไขระบบ เพราะต้องปิดและเปิดการทำงานของระบบใหม่ทั้งหมด บางครั้งในระยะเวลาการปรับปรุงระบบอาจแฝงไปด้วยค่าใช้จ่ายที่อาจสูญเสียไปถ้าระบบของธุรกิจนั้นมีความจำเป็นต้องทำงานตลอดเวลา

2.3.2 ระบบเชิงลักษณะแบบพลวัต

จากปัญหาดังกล่าวของการทำงานระบบเชิงลักษณะแบบสถิตย์ทำให้เกิดการวิจัยและพัฒนาต่อยอดขึ้นโดยเพิ่มขีดความสามารถแก่ระบบเชิงลักษณะให้สามารถทำการปรับปรุงแก้ไขระบบได้ ณ เวลาที่ระบบกำลังทำงานอยู่ โดยระบบที่มีการทำงานในลักษณะดังกล่าวนี้เรียกว่า ระบบเชิงลักษณะแบบพลวัต ซึ่งมีการวิจัยและพัฒนากันมาอย่างต่อเนื่องโดยงานวิจัยที่เกี่ยวข้องมีดังต่อไปนี้

ในปี 2001 ได้มีงานวิจัยซึ่งนำเสนอแพลตฟอร์มที่มีชื่อว่า AspectS (Hirschfeld, 2001) ซึ่งเป็นส่วนขยายของ Smalltalk โดยใน Smalltalk มี Meta-object protocols หรือ MOP สำหรับควบคุมโครงสร้างของวัตถุ AspectS ได้พัฒนาสภาพแวดล้อมที่มีอยู่ใน Smalltalk ให้สามารถใช้งานระบบเชิงลักษณะแบบพลวัตได้ โดยอาศัยตัวห่อบล็อกของเมธอด (Block method wrapper) ซึ่งช่วยสำหรับสานตัวแนะนำเชิงลักษณะแต่ละประเภทที่เหมาะสมเข้าไป สำหรับประสิทธิภาพการทำงาน

ของ AspectS ไม่ได้ถูกกล่าวไว้ ในบทความเป็นเพียงการเสนอความเป็นไปได้ในการพัฒนาระบบเชิงลักษณะแบบพลวัตกับ Smalltalk

ในปีเดียวกัน Pawlak และคณะ ได้เสนอกรอบงานชื่อ JAC (Pawlak et al., 2001) ซึ่งเป็นกรอบงานสำหรับใช้งานการโปรแกรมเชิงลักษณะกับภาษาจาวาที่ได้พัฒนาตามไวยากรณ์ของภาษาจาวา โดย JAC ต่างจาก AspectJ ตรงที่ AspectJ เป็น class-based แต่ JAC เป็น object-based สำหรับลักษณะใน JAC เป็นเซตของวัตถุลักษณะที่สามารถนำไปใช้งานบนวัตถุของโปรแกรมประยุกต์ (Application) ที่กำลังทำงานอยู่ได้ โดยการสร้างลักษณะแบ่งออกเป็น 3 วิธีดังนี้ วิธีที่ 1 คือ wrapping methods วิธีที่ 2 คือ role methods และวิธีที่ 3 คือ exception handlers ซึ่ง JAC ใช้แนวคิดของตัวควบคุมการห่อ (wrapping controller) เพื่อใช้งานลักษณะจะสังเกตเห็นว่า AspectJ จะสนใจไปที่การสร้างการตัดจุดด้วยภาษาใหม่ ส่วนจุดสนใจของ JAC เน้นการพัฒนาให้เข้ากับนิพจน์แบบเดิมของภาษาจาวา

PROSE (PROgrammable extenSions of sErvices) เป็นแพลตฟอร์มที่ใช้สำหรับสร้างระบบเชิงลักษณะแบบพลวัต (Popovici et al., 2002) PROSE สนับสนุนการปรับเปลี่ยนการทำงานของระบบให้มีความยืดหยุ่นโดยสามารถสานและไม่สานตัวแนะนำเชิงลักษณะในช่วงเวลาโปรแกรมกำลังทำงานอยู่ได้ ซึ่งลักษณะที่ใช้ใน PROSE เขียนได้ด้วยภาษาจาวา โดยประยุกต์จาก Debugger interface ซึ่งเป็นองค์ประกอบของจาวาเวอร์ชวลแมชีน แต่ประสิทธิภาพการทำงานที่ได้พบว่า PROSE มีการทำงานที่ช้า

ถึงแม้ว่าประสิทธิภาพของ PROSE มีการทำงานที่ช้าแต่วิธีการของ PROSE ก็มีประโยชน์ในการพัฒนาต่อ โดย Sato และคณะได้เห็นประโยชน์ดังกล่าวจึงได้พัฒนาต่อยอดโดยใช้หลักการดังกล่าวในชื่อแพลตฟอร์มที่พัฒนาขึ้นนี้ว่า Wool (Sato et al., 2003) ซึ่งใช้ Java Platform Debugger Architecture หรือ JPDA สำหรับแทรกตัวเกี่ยว (Hook) ไปยังจุดพัก (Breakpoint) ซึ่งเป็นวิธีที่แทรกโค้ดแบบคงที่ไปยังทุกจุดร่วม Wool มีความสามารถที่เหนือกว่า PROSE คือความสามารถในการรวมสองวิธีเข้าด้วยกันโดยทำการเลือกวิธีที่ดีและเหมาะสมของแต่ละจุดร่วมที่สุดซึ่งมีกลไกการทำงานดังนี้ เริ่มแรกคือทำการแทรกตัวเกี่ยวไปยังจุดพักผ่านทาง JPDA อีกวิธีจะทำการสร้างโปรแกรมในที่ซึ่งตัวเกี่ยวถูกฝังไว้กับการเรียกใช้เมธอด แล้วทำการเริ่มต้นโปรแกรมไปยังจาวาเวอร์ชวลแมชีนใหม่อีกครั้ง ทั้งสองวิธีนี้ไม่ต้องการจาวาเวอร์ชวลแมชีนที่มีการปรับปรุงให้เหมาะสมสำหรับการใช้งานแต่อย่างใด โดยสามารถทำงานกับจาวาเวอร์ชวลแมชีนมาตรฐานได้โดยตรง วิธีดังกล่าวนี้จะหลีกเลี่ยงการแทรกตัวเกี่ยวที่ไม่มีความจำเป็น ส่งผลให้ประสิทธิภาพของ Wool ทำงานได้เร็วกว่า PROSE

ต่อมา JAsCo (Suvée et al., 2003) ได้ทำการปรับปรุงภาษาเชิงลักษณะซึ่งเป็นองค์ประกอบพื้นฐานของการพัฒนาซอฟต์แวร์ และองค์ประกอบของ Java Beans โดยเฉพาะขึ้นใหม่

โดยลักษณะที่ดำเนินการด้วยภาษา JAsCo พัฒนาจากโมเดลซึ่งเป็นองค์ประกอบชนิดใหม่ที่รวมตัวดักสัญญาณซึ่งเป็นกับดักที่สามารถกระทำการต่าง ๆ ได้โดยการนำตัวดักสัญญาณที่สำคัญมารวมไว้ด้วยกันเพื่อให้สามารถใช้งานลักษณะแบบพลวัตได้ อย่างไรก็ตามความสามารถที่เป็นแบบพลวัตและความซับซ้อนที่ได้มาจากการใช้งาน JAsCo นี้กลับต้องแลกด้วยประสิทธิภาพการทำงานที่ช้าลง

ในปี 2004 ได้มีการดัดแปลง Jikes Research Virtual Machine (RVM) ซึ่งเป็นเครื่องจักรเสมือนสำหรับทดลองในทางวิจัยเพื่อพัฒนาเครื่องมือที่ชื่อว่า Steamloom (Bockisch et al., 2004) โดยได้นำเสนอการทดสอบการตัดจุดแบบพลวัต (Dynamic pointcut) ซึ่งกำหนดเงื่อนไขการตัดจุดของตัวตัดจุดหรือพีซีดี จะกำหนดเงื่อนไขที่ไม่สามารถพบเมธอดหรือคอนสตรัคเตอร์ในส่วนของโปรแกรมจริงได้ ณ ขณะช่วงเวลาโหลดโปรแกรมหรือช่วงเวลาแปลโปรแกรม แต่จะสามารถค้นหาและพบเมธอดหรือคอนสตรัคเตอร์ที่ต้องการได้ในช่วงโปรแกรมกำลังทำงานอยู่ โดยผลการทดสอบประสิทธิภาพแสดงว่า Steamloom ที่พัฒนาโดยทำการดัดแปลง RVM ให้สามารถใช้งานระบบเชิงลักษณะที่วัดประสิทธิภาพของการตัดจุดแบบพลวัตแล้วนั้นมีความเร็วกว่าการตัดจุดแบบพลวัตที่ใช้ AspectJ ทำงานบนทั้ง RVM ซึ่งไม่มีการดัดแปลง และบนเครื่องจักรเสมือนของ Sun Microsystems ในงานนี้ผู้วิจัยกล่าวไว้ว่ายังมีข้อผิดพลาดที่ยังเกิดขึ้นอีกมาก แต่ก็แสดงให้เห็นความเป็นไปได้ในการพัฒนาระบบเชิงลักษณะแบบพลวัต

ต่อมาในปี 2012 ได้มีการนำเสนอ JooFlux (Ponge and Mouel, 2012) ซึ่งเครื่องมือสำหรับสร้างระบบเชิงลักษณะแบบพลวัตโดยใช้คำสั่ง invokedynamic โดยใช้เมธอด filterArguments และเมธอด filterReturnValue ในแพ็คเกจ java.lang.invoke ในการรวมไบต์โค้ดสำหรับการสร้างตัวแนะนำก่อนและตัวแนะนำหลังแต่ในความเป็นจริงแล้วตัวแนะนำของการโปรแกรมเชิงลักษณะของ JooFlux ที่สร้างขึ้นมายังมีความหมายของตัวแนะนำหลังผิดอยู่เพราะจากความสามารถของตัวแนะนำหลังที่ได้กล่าวมาแล้วไม่สามารถทำการเปลี่ยนแปลงค่าคืนกลับได้ แต่จากการศึกษาพบว่าวิธีที่ JooFlux ใช้สำหรับสร้างตัวแนะนำหลังยังสามารถเปลี่ยนแปลงค่าคืนกลับได้ นอกจากนี้ในส่วนของตัวแนะนำก่อนที่ JooFlux สร้างขึ้น ผู้วิจัยพบว่าสามารถปรับการทำงานในส่วนนี้ให้มีความเร็วขึ้นได้ซึ่งจะกล่าวต่อไปในบทที่ 3

สำหรับงานวิจัยที่เกี่ยวข้องทั้งหมดนี้สามารถสรุปได้ดังตารางที่ 2.1 โดยทำการสรุปจาก 3 หัวข้อหลักดังนี้ หัวข้อที่ 1 ความสามารถของการใช้งานซึ่งทำการแบ่งตามความสามารถได้ดังนี้ สามารถใช้งานกับภาษาจาวา สามารถใช้งานตัวแนะนำก่อน สามารถใช้งานตัวแนะนำหลัง สามารถใช้งานตัวแนะนำหลังการคืนค่า สามารถใช้งานตัวแนะนำครอบ หัวข้อที่ 2 การพัฒนาซึ่งมีรายละเอียดหัวข้อย่อยดังนี้ การพัฒนาโดยใช้ภาษาจาวา การพัฒนาโดยเพิ่มประสิทธิภาพของระบบเชิงลักษณะแบบพลวัต การพัฒนาโดยให้มีไวยากรณ์การใช้งานตาม AspectJ การพัฒนาโดยทำการ

ดัดแปลงไบต์โค้ด และหัวข้อที่ 3 ขอบเขตการวิจัยโดยมีหัวข้อย่อยดังนี้ วิจัยเพื่อทดสอบประสิทธิภาพ วิจัยเพื่อเสนอแนวคิด และหัวข้อย่อยสุดท้ายคือ มีการประยุกต์ใช้กับชุดทดสอบมาตรฐาน

ตารางที่ 2.1 สรุปเปรียบเทียบงานวิจัยที่เกี่ยวข้องกับการเปรียบเทียบความสามารถในการใช้งานระบบเชิงลักษณะแบบพลวัต

กระบวนการทำงาน	งานวิจัยที่เกี่ยวข้อง*								
	1	2	3	4	5	6	7	8	9
ความสามารถของงาน									
สามารถใช้งานกับภาษาจาวา	✓		✓		✓	✓		✓	✓
สามารถใช้งานตัวแนะนำก่อน	✓	✓	✓		✓	✓	✓	✓	✓
สามารถใช้งานตัวแนะนำหลัง	✓	✓	✓		✓	✓	✓		✓
สามารถใช้งานตัวแนะนำหลังการคืนค่า	✓								✓
สามารถใช้งานตัวแนะนำครอบ	✓	✓	✓			✓	✓		✓
การพัฒนา									
พัฒนาโดยใช้ภาษาจาวา			✓	✓	✓	✓		✓	✓
พัฒนาโดยเพิ่มประสิทธิภาพการทำงานของระบบโปรแกรมเชิงลักษณะแบบพลวัต		✓	✓	✓	✓	✓	✓	✓	✓
พัฒนาโดยให้มีไวยากรณ์การใช้งานตาม AspectJ	✓	✓					✓		✓
พัฒนาโดยทำการดัดแปลงไบต์โค้ด	✓		✓	✓	✓		✓	✓	✓
ขอบเขตของการวิจัย									
วิจัยเพื่อทดสอบประสิทธิภาพ	✓			✓	✓	✓	✓	✓	✓
วิจัยเพื่อเสนอแนวคิด	✓	✓	✓	✓		✓	✓	✓	✓
มีการประยุกต์ใช้กับชุดทดสอบมาตรฐาน	✓			✓	✓		✓		✓

*งานวิจัยที่เกี่ยวข้องประกอบด้วย

1 = Kiczales, G. และคณะ (2001)
2 = Hirschfeld, R. (2001)
3 = Pawlak, R. และคณะ (2001)
4 = Popovici, A. และคณะ (2002)
5 = Sato, Y. และคณะ (2003)
6 = Suvée, D. และคณะ (2003)
7 = Bockisch, C. และคณะ (2004)
8 = Ponge, J. และ Mouël, Le F. (2012)
9 = การศึกษาและออกแบบความหมายของการแนะนำเชิงลักษณะโดยใช้ตัวรวมไบต์โค้ด

บทที่ 3

วิธีดำเนินการวิจัย

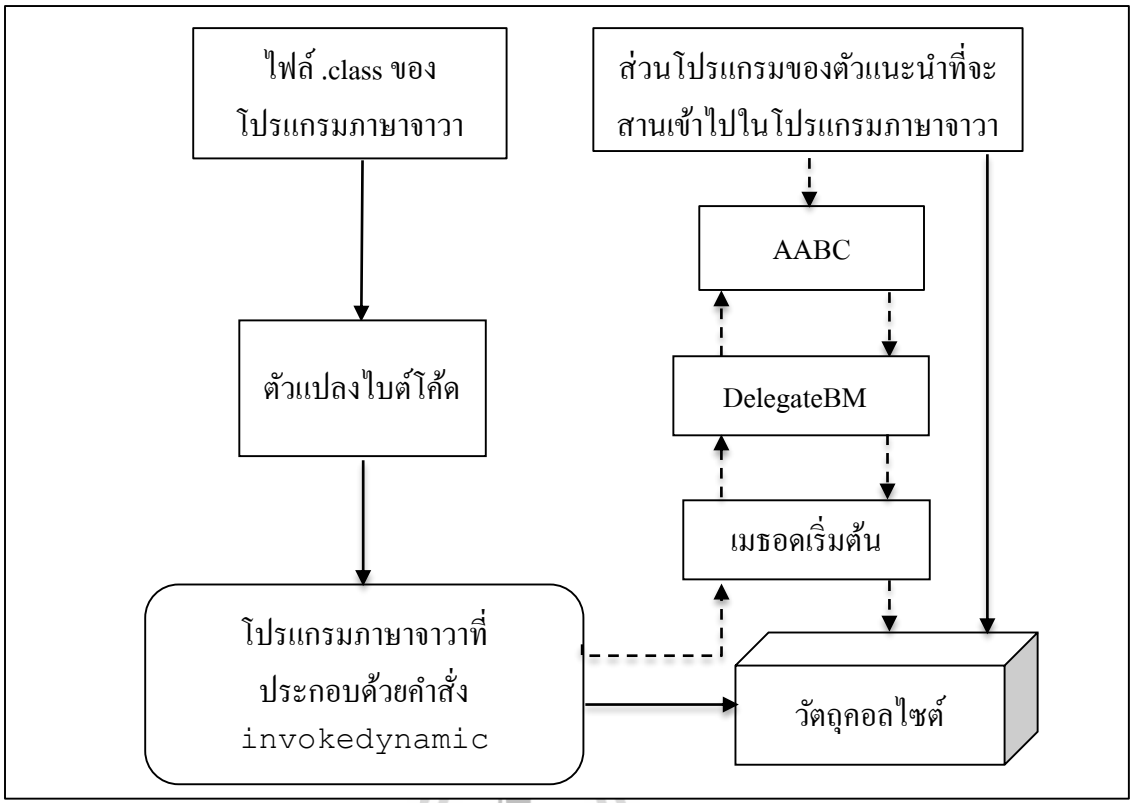
ในบทนี้จะกล่าวถึงวิธีการวิจัย และองค์ประกอบของการวิจัยสำหรับการทำวิจัยทั้งหมดซึ่งประกอบด้วยขั้นตอนการพัฒนาเครื่องมือสำหรับใช้สร้างระบบเชิงลักษณะแบบพลวัตโดยใช้คำสั่ง invokedynamic ของ Java™ Platform Standard Edition 7 โดยมีลำดับขั้นตอนการศึกษาดังนี้

1. ศึกษาการทำงานของการโปรแกรมเชิงลักษณะ (รายละเอียดอยู่ในส่วนที่ 2.1)
2. ศึกษากระบวนการทำงานของคำสั่ง invokedynamic (รายละเอียดอยู่ในส่วนที่ 2.2)
3. ทำการออกแบบระบบเชิงลักษณะแบบพลวัตโดยใช้คำสั่ง invokedynamic (รายละเอียดอยู่ในส่วนที่ 3.1)
4. ทำการออกแบบไวยากรณ์สำหรับสร้างตัวแจงส่วนการตัดจุดเพื่อการใช้งานเครื่องมือโดยเทียบไวยากรณ์กับการประกาศการตัดจุดใน AspectJ (รายละเอียดอยู่ในส่วนที่ 3.2)
5. ทำการสร้างระบบเชิงลักษณะแบบพลวัตโดยใช้คำสั่ง invokedynamic ตามที่ได้ออกแบบในขั้นตอนที่ 3 (รายละเอียดอยู่ในส่วนที่ 3.3)
6. จัดเตรียมเครื่องมือสำหรับการสร้างและทดสอบระบบเชิงลักษณะแบบพลวัตโดยใช้คำสั่ง invokedynamic ที่พัฒนาขึ้น (รายละเอียดอยู่ในส่วนที่ 4.1)
7. ทำการออกแบบการวัดประสิทธิภาพของระบบเชิงลักษณะแบบพลวัตโดยใช้กลไกการทำงานของคำสั่ง invokedynamic ที่พัฒนาขึ้น (รายละเอียดอยู่ในส่วนที่ 4.2)
8. ทำการทดสอบประสิทธิภาพของระบบเชิงลักษณะแบบพลวัตที่พัฒนาขึ้นโดยใช้กลไกการทำงานของคำสั่ง invokedynamic (รายละเอียดอยู่ในส่วนที่ 4.3)

สำหรับในส่วนขั้นตอนที่ 3 – 5 ซึ่งอธิบายในหัวข้อที่ 3.1 – 3.3 ของบทนี้โดยรายละเอียดแต่ละขั้นตอนมีดังต่อไปนี้

3.1 การออกแบบระบบเชิงลักษณะแบบพลวัตโดยใช้คำสั่ง invokedynamic

การออกแบบในส่วนนี้เป็นการนำเอากลไกการทำงานของคำสั่ง invokedynamic มาต่อยอดให้สามารถใช้งานระบบเชิงลักษณะแบบพลวัตได้ ซึ่งมีรายละเอียดการออกแบบโดยรวมของการทำงานระบบที่สร้างขึ้นแสดงในรูปที่ 3.1 ดังนี้



รูปที่ 3.1 แผนภาพการทำงานของเครื่องมือการสร้างระบบเชิงลักษณะแบบพลวัต
โดยใช้คำสั่ง invokedynamic

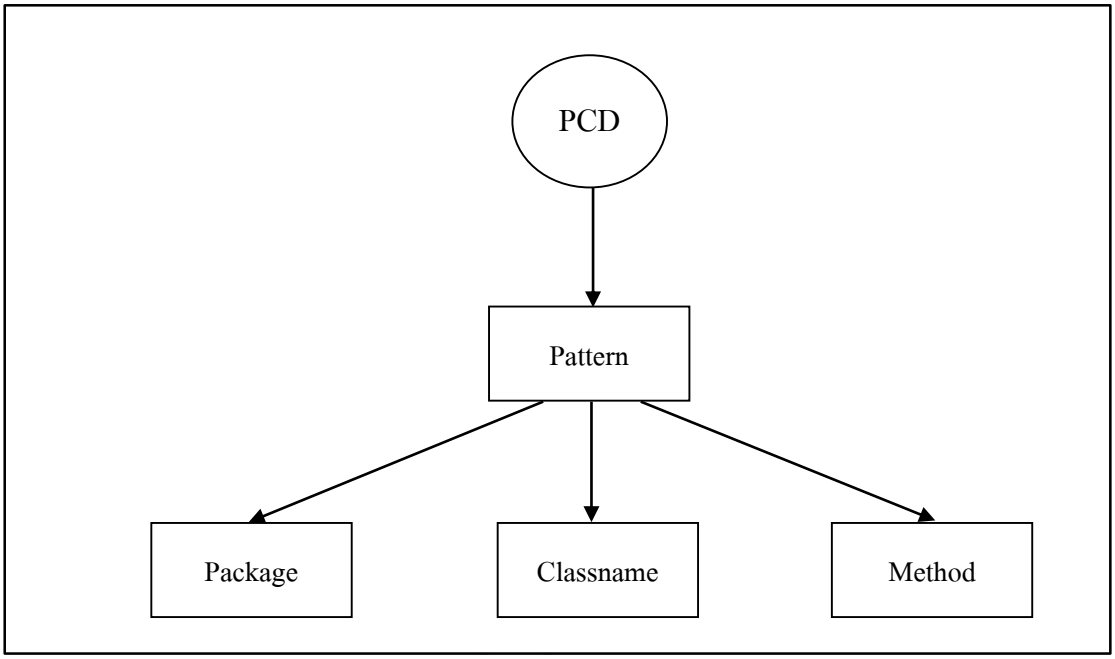
เริ่มต้นด้วยการนำไฟล์ .class แต่ละไฟล์มาสู่ตัวแปลงไบต์โค้ดเพื่อเตรียมการสำหรับใช้งานระบบเชิงลักษณะแบบพลวัตโดยทำการค้นหาคำสั่งไบต์โค้ดที่ใช้สำหรับเรียกคอนสตรัคเตอร์หรือเมธอดโดยไบต์โค้ดเหล่านี้อยู่ในรูปของคำสั่ง invokestatic คำสั่ง invokevirtual คำสั่ง invokeinterface และคำสั่ง invokespecial ซึ่งจะทำการแปลงคำสั่งเหล่านี้ไปเป็นคำสั่ง invokedynamic นอกจากจะทำการแปลงไบต์โค้ดแล้ว ในส่วนนี้ยังรับผิดชอบในการผลิตไบต์โค้ดที่เป็นองค์ประกอบสำคัญสำหรับการทำงานร่วมกับคำสั่ง invokedynamic โดยจะทำการผลิตไบต์โค้ดในส่วนของเมธอดเริ่มต้นขึ้นเพราะเมธอดเริ่มต้นจะถูกเรียกใช้งานเมื่อคำสั่ง invokedynamic ถูกเรียกเป็นครั้งแรก ซึ่งภายในเมธอดเริ่มต้นจะทำการสร้างคำสั่งไบต์โค้ดที่เป็นองค์ประกอบสำคัญคือคำสั่งเรียกใช้งาน Delegate Bootstrap Method หรือเมธอด DelegateBM สำหรับการเรียกใช้งานตัวแจงส่วนการตัดจุดโดยจะตรวจสอบคุณสมบัติของตำแหน่งที่คำสั่ง invokedynamic ถูกเรียกใช้งานว่าตรงตามเงื่อนไขของการตัดจุดหรือไม่ ถ้าตำแหน่งดังกล่าวตรงตามเงื่อนไขของการตัดจุดที่ผู้พัฒนาซอฟต์แวร์กำหนดขึ้นก็จะทำการสานไบต์โค้ดโดยใช้ตัวรวมไบต์โค้ด (Bytecode combinator) ซึ่งในงานที่พัฒนาขึ้นนี้ใช้ชื่อว่า

Aspect-aware Bytecode Combinators หรือ AABC ในการสานระหว่างตัวแนะนำแต่ละประเภทที่ถูกเขียนไว้ในคลาส aspect กับเมธอดที่คำสั่ง invokedynamic ต้องการเรียกใช้งาน โดยจะกระทำการในรูปวัตถุของคลาส MethodHandle ซึ่งเป็นเหมือนพอยเตอร์ที่ชี้ไปยังตำแหน่งจริง ๆ ของตัวแนะนำและตำแหน่งที่คำสั่ง invokedynamic ต้องการเรียกใช้งาน เสร็จแล้วก็จะส่งวัตถุของคลาส MethodHandle ที่ได้ผูกแล้วให้กับคลาส ConstantCallSite เพื่อได้วัตถุคอลไซต์ออกมาสำหรับใช้งานกับคำสั่ง invokedynamic และเมื่อโปรแกรมทำงานมาพบคำสั่ง invokedynamic ในตำแหน่งเดิมอีกครั้งก็สามารถเรียกใช้งานวัตถุคอลไซต์ได้โดยตรงโดยไม่ต้องเข้าไปทำงานในเมธอดเริ่มต้นอีก แต่ถ้ามีการปรับปรุงตัวแนะนำเชิงลักษณะใหม่ก็จะเข้ามาทำงานที่เมธอดเริ่มต้นอีกครั้งเพื่อปรับปรุงในส่วนที่ได้แก้ไปแล้ว สำหรับตัวรวมไบต์โค้ดแต่ละประเภทจะอธิบายในส่วนถัดไป

3.2 การออกแบบไวยากรณ์สำหรับสร้างตัวแจงส่วนการตัดจุด

รูปแบบประโยคที่ใช้งานจริงของ AspectJ สำหรับใช้งานในการสร้างการตัดจุดได้อธิบายหน้าที่และความหมายของแต่ละวลีแล้วในส่วนที่ 2.1.2 สำหรับในส่วนนี้จะทำการออกแบบไวยากรณ์สำหรับสร้างตัวแจงส่วนการตัดจุด ในการออกแบบจะอาศัยรูปแบบการเขียนเพื่อใช้งานตัวแนะนำแต่ละประเภทใน AspectJ โดยการใช้งานจริงของตัวแจงส่วนการตัดจุดจะนำมาใช้ในเมธอดเริ่มต้นเพื่ออำนวยความสะดวกแก่ผู้พัฒนาซอฟต์แวร์ให้สามารถทำการเลือกจุดตัดที่ต้องการ โดยมีรายละเอียดดังต่อไปนี้

ไวยากรณ์ของตัวแจงส่วนการตัดจุดได้ทำการพัฒนาโดยเทียบรูปแบบการเขียนและความหมายตามหลักการของการโปรแกรมเชิงลักษณะกับ AspectJ (Kiczales et al., 2001) ซึ่งเป็นภาษาที่พัฒนาขึ้นโดยเฉพาะสำหรับใช้งานตามหลักการของการโปรแกรมเชิงลักษณะกับภาษาจาวา โดยเครื่องมือสำหรับใช้สร้างตัวแจงส่วนการตัดจุดของงานวิจัยนี้คือ ANTLR (Parr, 2010) ซึ่งแบบอย่าง (Pattern) สำหรับตัวแจงส่วนการตัดจุดแสดงในรูปที่ 3.2 และไวยากรณ์ที่ทำการสร้างขึ้นดังแสดงในรูปที่ 3.3 เป็นการนิยามกฎของพีซีดีซึ่งเป็นเงื่อนไขสำหรับการเลือกจุดที่ต้องการ การตั้งเงื่อนไขของการตัดจุดสามารถทำได้โดยนำพีซีดีมาต่อกันโดยใช้ และ (&&) หรือ (||) และเครื่องหมายนิเสธ (!) ในการเพิ่มเงื่อนไขของการตัดจุดซึ่งจะทำให้การเขียนโปรแกรมสามารถเลือกจุดที่ต้องการสานตัวแนะนำแต่ละประเภทได้ง่ายขึ้น โดยตัวแจงส่วนการตัดจุดที่สร้างขึ้นจะทำงานร่วมกับตัวรวมไบต์โค้ด AABC ซึ่งกระบวนการทำงานของตัวแจงส่วนการตัดจุดนี้จะถูกเรียกใช้งานในส่วนของเมธอด DelegateBM ตามรูปที่ 3.1 โดยตัวแจงส่วนการตัดจุดนี้จะสนับสนุนเพียงบางส่วนของพีซีดีทั้งหมดใน AspectJ เท่านั้น ได้แก่ พีซีดี call พีซีดี execution และพีซีดี withincode



รูปที่ 3.2 แผนภาพแสดงแบบอย่างสำหรับตัวแจงส่วนการตัดจุด

```
// part 1/3

unit
     :    expression EOF
     ;

expression
     :    '!' '(' expression ')'
     |    '(' expression ')'
     |    expression '&&' expression
     |    expression '||' expression
     |    pcd
     |    '!' pcd
     ;

pcd
     :    call
     |    execution
     |    withincode
     ;
```

```
// part 2/3

withincode
     :    'withincode' '('
               access* returntype' '
               method'('arguments*')'
          ')'
     ;

call
     :    'call' '('
               Returntype' 'pkc '(' heckargument ')'
          ')'
     ;

access
     :    'public '
     |    'static '
     ;
pkc
     :    pk* classname method
     |    '*'
     ;

returntype
     :    '*'
     |    Identifier2
     |    Identifier
     ;

pk
     :    Identifier'.'
     |    '*.'
     ;

classname
     :    Identifier2 '.'
     |    '*.'
     ;
```

```
// part 3/3

method
     :    Identifier
     |    '*'
     ;

checkargument
     :    arguments
     |    '..'
     |    ''
     ;

arguments
     :    argument',' arguments
     |    argument
     ;
argument
     :    Identifier2
     |    Identifier
     ;

Identifier
     :    Smallletter Letter*
     ;

Identifier2
     :    Bigletter Letter*
     ;
```


รูปที่ 3.3 ไวยากรณ์ตัวแจงส่วนการตัดจุดสำหรับใช้งานกับตัวรวมไบต์โค้ด AABC

สำหรับรูปที่ 3.4 เป็นการใช้งานจริงร่วมกับตัวรวมไบต์โค้ด AABC ที่ได้ทำการพัฒนาขึ้น โดยเทียบลักษณะการใช้งานใน AspectJ ตามตัวอย่างรูปที่ 3.5 สำหรับการใช้งานจริงร่วมกับตัวรวมไบต์โค้ด AABC นี้อาศัยการประมวลผลแอนโนเทชัน (Annotation) โดยการเรียกใช้งานตัวแจงส่วนการตัดจุดจะทำการรับค่าจากแอนโนเทชันแล้วเข้าสู่กระบวนการแจงส่วนการตัดจุด จากตัวอย่างที่ 1 ในรูปที่ 3.4 เป็นการกำหนดเงื่อนไขสำหรับการสานตัวแนะนำก่อนซึ่งเทียบรูปแบบการใช้งานในตัวอย่างที่ 1 รูปที่ 3.5 โดยอาศัยการใช้แอนโนเทชันชื่อว่า @Before จากตัวอย่างการใช้งานสำหรับ

```
public class Aspect{

//************(1). for Before advice ********************//
      @Before("call(* *(..))&&
               !withincode(public static void main())")
      public static void before(Context ct)
      {
          // write some action here
      }

//************(2). for After advice ********************//
      @After("call(* *(..))&&
              !withincode(public static void main())")
      public static void after(Context ct, Object
               returnValue){
          // write some action here
      }

//************ (3). for After-return advice **************//
      @AfterReturn("call(* *(..))&&
                    !withincode(public static void main())")
      public static Object afterReturn(Object returnValue)
      {
          // write some action here
          return returnValue;
      }

//************ (4). for Around advice ********************//
      @Around("call(* *(..)) &&
               !withincode(public static void main())")
      public static Object around(Context ct)
               throws Throwable{
          // write some action here
          return ct.proceed.invokeWithArguments(ct.argument);
      }
}
```

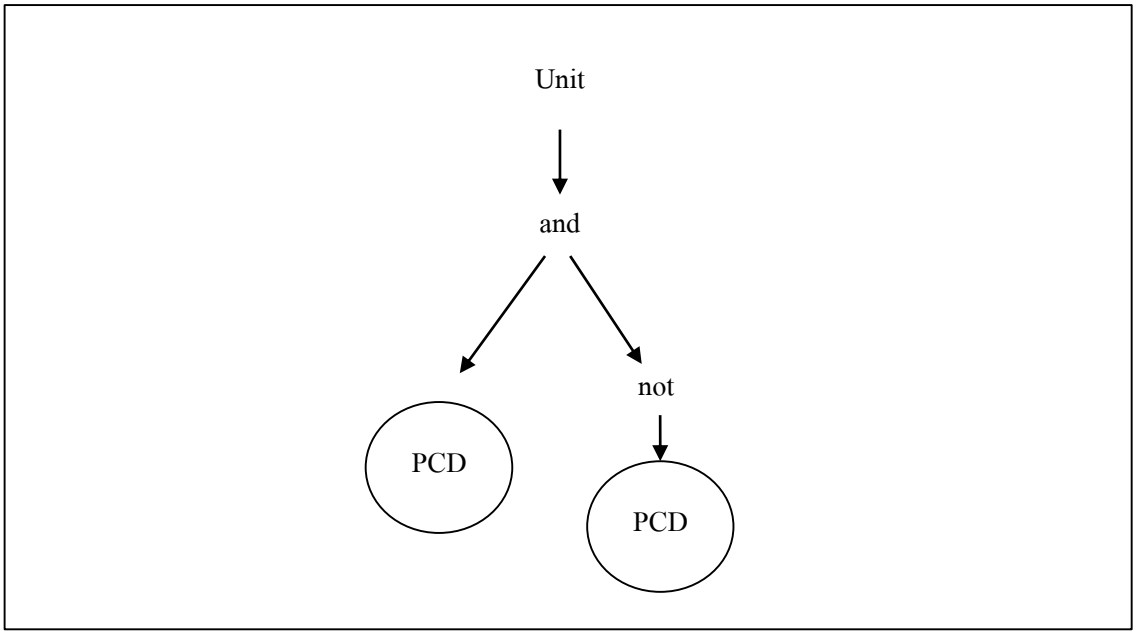
รูปที่ 3.4 ตัวอย่างการเขียนตัวแนะนำเชิงลักษณะแต่ละประเภทสำหรับใช้งานกับระบบเชิงลักษณะแบบพลวัตที่พัฒนาขึ้น

```
public aspect Aspect{

//************(1). for Before advice ********************//
      before() : call(* *(..)) &&
            !withincode(public static void main()){
            //write some action here
      }

//************(2). for After advice *********************//
      after() : call(* *(..)) &&
            !withincode(public static void main()){
            write some action here
      }

//************ (3). for After-return advice **************//
      after() returning (Object o): call(* *(..)) &&
            !withincode(public static void main()){
            write some action here
      }

//************ (4). for Around advice *******************//
      Object around() : call(* *(..)) &&
            !withincode(public static void main()){
            write some action here
            return proceed();
       }
}
```

รูปที่ 3.5 ตัวอย่างการเขียนลักษณะด้วยของภาษา AspectJ

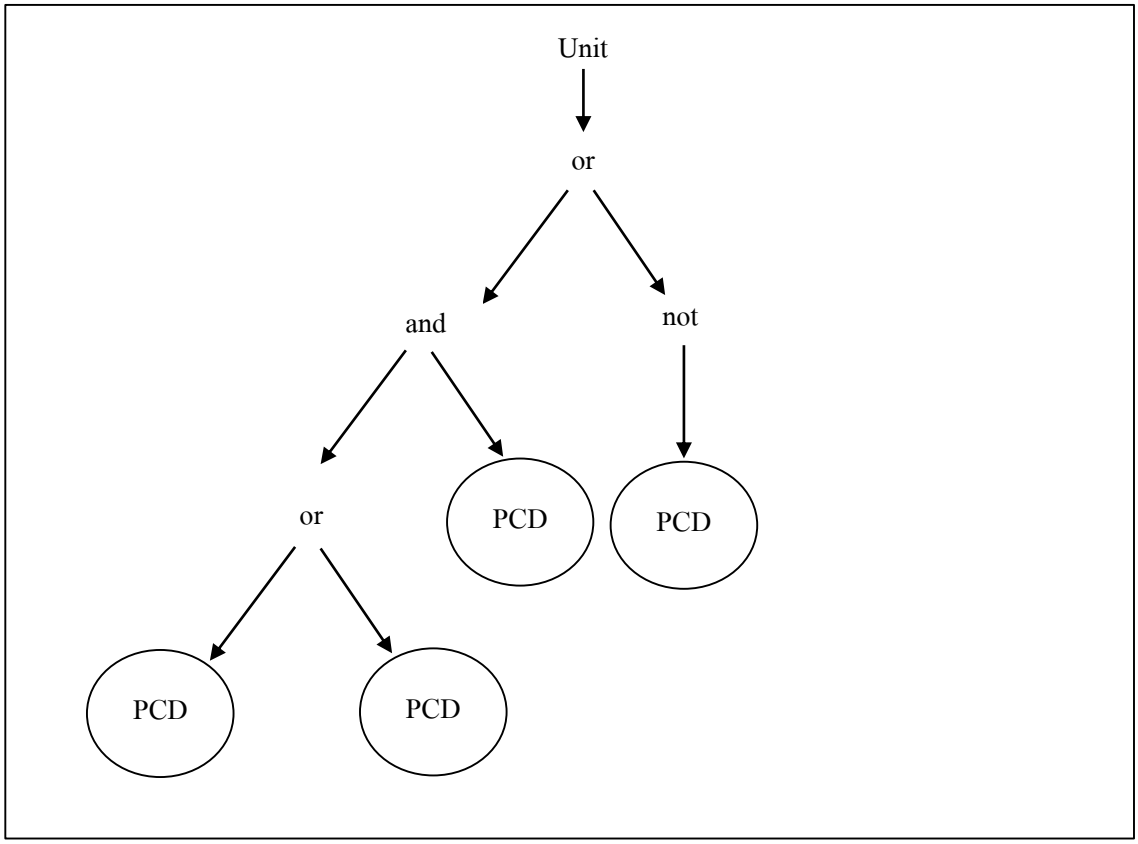
การเขียนในส่วนของตัวแนะนำก่อนดังนี้ @Before("call(* *(..)) && !withincode(public static void main())") ซึ่งประกอบไปด้วย 2 พีซีดี สำหรับพีซีดีตัวแรกคือ call(* *(..)) หมายถึงการเลือกส่วนของการเรียกใช้งานทั้งหมดโดยที่ * ตัวแรกหมายถึงทำการเลือกชนิดของค่าคืนกลับ (Return type) ทุกชนิด สำหรับ * ตัวที่สองหมายถึงทำการเลือกการเรียกใช้งานทุกคลาส ทุกแพ็คเกจ และ (..) หมายถึงเลือกพารามิเตอร์ทุกจำนวนทุกรูปแบบ

สำหรับพีซีดีตัวที่สองคือ !withincode(public static void main()) หมายถึงทำการยกเว้นการเรียกภายในเมธอด main ทั้งหมดจากนั้นพีซีดีตัวที่หนึ่งและพีซีดีตัวที่สองจะถูกเชื่อมกันด้วยเงื่อนไข



รูปที่ 3.6 แผนภาพต้นไม้แสดงตัวอย่างของประโยค PCD && !(PCD)

และ (&&) โดยตัวอย่างนี้ตรงตามเงื่อนไข PCD && !PCD ดังแสดงในรูปที่ 3.6 ซึ่ง Unit คือประโยคเงื่อนไขทั้งประโยคและเชื่อมเงื่อนไขแต่ละประโยคด้วย และ (and) กับพีซีดีทั้งสองตัวโดยพีซีดีตัวที่สองคั่นด้วยนิเสธ (!) เพราะต้องการให้เป็นประโยคปฏิเสธ สำหรับตัวอย่างที่ 2 ที่ 3 และที่ 4 ในรูปที่ 3.4 เป็นตัวอย่างการใช้งานการกำหนดเงื่อนไขของตัวแนะนำหลัง ตัวแนะนำหลังการคืนค่า และตัวแนะนำครอบ โดยเทียบรูปแบบการเขียนด้วย AspectJ ในตัวอย่างที่ 2 ที่ 3 และที่ 4 ของรูปที่ 3.5 ตามลำดับ โดยตัวแนะนำหลังจะใช้แอนโนเทชันชื่อว่า @After และจะสานตัวแนะนำนี้ตามการทำงานของตัวแนะนำหลัง ส่วนตัวแนะนำหลังการคืนค่าจะใช้แอนโนเทชันชื่อว่า @AfterReturn ซึ่งจะสานเมธอดที่อยู่ด้านล่างของแอนโนเทชันนี้ตามหลักการทำงานของตัวแนะนำหลังการคืนค่า สำหรับแอนโนเทชันตัวสุดท้ายชื่อว่า @Around จะถูกใช้งานสำหรับสานเมธอดตามการทำงานของตัวแนะนำครอบ ซึ่งถ้าหากเงื่อนไขที่กำหนดในแอนโนเทชันแต่ละตัวตรงตามตำแหน่งที่คำสั่ง invokedynamic ถูกเรียกใช้งานก็จะถูกสานเข้าไปตอนช่วงโปรแกรมกำลังทำงาน และสำหรับการใช้งานการกำหนดเงื่อนไขที่ซับซ้อนขึ้นได้แสดงตัวอย่างแผนภาพตามรูปที่ 3.7 โดยมีเงื่อนไขดังนี้คือ ((PCD || PCD) && PCD) || !(PCD) ซึ่งลำดับความสำคัญในการแจงส่วนจะทำการพิจารณาจากส่วนของวงเล็บในสุดก่อนคือ (PCD || PCD) โดยผลลัพธ์ที่ได้จะมีค่าเป็นจริงหรือเท็จ เพื่อให้การอธิบายในส่วนนี้ให้เข้าใจง่ายค่าที่ได้ออกมาจะเก็บไว้ในตัวแปร A หลังจากนั้นขั้นตอนที่สองจะทำการพิจารณาจากวงเล็บถัดมาคือ (A && PCD) ซึ่งค่าความจริงที่ได้จากการพิจารณาในขั้นตอนนี้จะเก็บ



รูปที่ 3.7 แผนภาพต้นไม้แสดงตัวอย่างของประโยค ((PCD || PCD) && PCD) || !(PCD)

ไว้ในตัวแปร B และขั้นตอนสุดท้ายจะเทียบค่าความจริงกับส่วนของวงเล็บถัดไปซึ่งตัวอย่างนี้ส่วนนอกสุดคือ (B || !(PCD) จากนั้นค่าที่ออกมามีค่าเป็นจริงก็จะทำการสานตัวแนะนำลงไป ณ ตำแหน่งดังกล่าว แต่ถ้ามีค่าเป็นเท็จก็จะไม่มีการสานตัวแนะนำใด ๆ ลงไปในระบบจากตัวอย่างที่ได้กล่าวมาสามารถรองรับประโยคเงื่อนไขได้ครอบคลุมขึ้นตามที่ผู้พัฒนาซอฟต์แวร์ต้องการสำหรับใช้แทรกตัวแนะนำแต่ละประเภทเข้าสู่ระบบที่ต้องการ

3.3 การสร้างระบบเชิงลักษณะโดยใช้คำสั่ง invokedynamic

สำหรับการสร้างระบบเชิงลักษณะโดยใช้คำสั่ง invokedynamic นั้นสามารถแบ่งขั้นตอนตามกระบวนการสร้างได้ดังต่อไปนี้

3.3.1 ทำการสร้างตัวแปลงไบต์โค้ดสำหรับการเปลี่ยนคำสั่งไบต์โค้ด

ในขั้นตอนนี้จะทำการสร้างตัวแปลงไบต์โค้ดคำสั่ง invokestatic คำสั่ง invokevirtual คำสั่ง invokeinterface และคำสั่ง invokespecial โดยแปลงไปเป็นคำสั่ง invokedynamic สำหรับสูตรที่ใช้แปลงได้นำเสนออยู่ในบทความวิจัยของ Kaewkasi (2010)

3.3.2 ทำการสร้างองค์ประกอบต่าง ๆ สำหรับใช้งานคำสั่ง invokedynamic

องค์ประกอบสำหรับใช้งานคำสั่ง invokedynamic ได้แก่ ส่วนที่ใช้สร้างการค้นหาคอนสตรัคเตอร์ หรือเมธอดจริงที่ได้แสดงแล้วในส่วนที่ 2.2.2 และส่วนที่สร้างเมธอดเริ่มต้นและได้ทำการพัฒนาต่อให้มีการเรียกเมธอด DelegateBM สำหรับใช้ตรวจสอบและใช้งานร่วมกับตัวรวมไบต์โค้ด AABC ซึ่งจะกล่าวถึงในขั้นตอนต่อไป

3.3.3 ทำการสร้างตัวรวมไบต์โค้ด AABC

การสร้างตัวรวมไบต์โค้ดซึ่งในงานวิจัยนี้ใช้ชื่อว่า AABC จะทำการสร้างตัวรวมไบต์โค้ดสำหรับใช้งานตัวแนะนำเชิงลักษณะแต่ละแบบซึ่งมีรายละเอียดดังนี้

1) ตัวรวมไบต์โค้ด AABC สำหรับตัวแนะนำก่อน

ตัวรวมไบต์โค้ดสำหรับตัวแนะนำก่อนจะถูกเรียกใช้งานสำหรับรวมไบต์โค้ดก็ต่อเมื่อตำแหน่งที่คำสั่ง invokedynamic ถูกเรียกอยู่ในตำแหน่งที่ตรงตามเงื่อนไขการตัดจุด จากตัวอย่างได้แสดงการกำหนดเงื่อนไขการตัดจุดในรูปที่ 3.5 ในส่วนที่ 1 โดยตัวรวมไบต์โค้ดในส่วนนี้จะทำการพัฒนาต่อยอดให้เข้ากับตัวแนะนำก่อนโดยใช้ประโยชน์จากเมธอด filterArguments ซึ่งอยู่ในคลาส MethodHandles ของแพ็คเกจ java.lang.invoke มาทำการดัดแปลงสำหรับใช้งานตามหลักการของตัวแนะนำก่อน เหตุผลที่ทำการดัดแปลงเมธอด filterArguments เพราะมีการทำงานบางส่วนที่ไม่เกี่ยวข้องกับการทำงานตามหลักการของตัวแนะนำก่อนซึ่งอาจทำให้ประสิทธิภาพการทำงานของระบบช้าลงได้ โดยตัวอย่างของตัวรวมไบต์โค้ด AABC สำหรับตัวแนะนำก่อนแสดงในรูปที่ 3.8 ในเมธอด DelegateBM โดยตัวรวมไบต์โค้ด AABC ในส่วนนี้มีชื่อว่า CallBeforeAdvice ซึ่งทำการผูกวัตถุ target และวัตถุ beforeAdvice ที่เป็นวัตถุของคลาส MethodHandle ไว้ด้วยกัน ในการผูกจะทำการผูกวัตถุ beforeAdvice ไว้ในส่วนก่อนการทำงานของคอนสตรัคเตอร์หรือเมธอดจะเริ่มทำงานและได้ทำการพัฒนา CallBeforeAdvice ให้สามารถผูกตัวแนะนำตามความสามารถของตัวแนะนำก่อนโดยสามารถทำการสำรวจค่าของพารามิเตอร์ที่จะส่งให้วัตถุ target แต่ไม่สามารถปรับเปลี่ยนค่าของพารามิเตอร์ได้ และยังสามารถทำการสร้างการทำงานต่าง ๆ แทรกเข้าไปตรงจุดนี้ได้ โดยเมื่อวัตถุ beforeAdvice ทำงานเสร็จก็จะมาทำงานในส่วนของวัตถุ target ต่อไป

```
if(declare.annotationType().toString().
      endsWith("Before"))
      {
      if(
            check(
                  new ANTLRInputStream(
                        mt.getAnnotation(
                              Before.class
                        ).value()
                  ),
                  info, tp, callfromMethod,
                  access,tpcallfromMethod
            )!=null
      ){
            mhCombine =
                  CallBeforeAdvice.beforeAd(
                     aopAdvice,
                     mhCombine,
                     type.parameterCount()
                  ).asType(type);

      }
}
```

รูปที่ 3.8 ตัวอย่างการเรียกใช้งานตัวรวมไบต์โค้ด AABC สำหรับตัวแนะนำก่อน

2) ตัวรวมไบต์โค้ด AABC สำหรับตัวแนะนำหลัง

ตัวรวมไบต์โค้ดนี้จะถูกเรียกใช้งานก็ต่อเมื่อตำแหน่งของคำสั่ง invokedynamic ตรงตามเงื่อนไขที่กำหนดในการตัดจุดในรูปที่ 3.5 ส่วนที่ 2 โดยตัวรวมไบต์โค้ดส่วนนี้ได้นำเมธอด filterReturnValue ซึ่งอยู่ในคลาส MethodHandles ของแพ็คเกจ java.lang.invoke มาทำการดัดแปลงพัฒนาให้สามารถใช้งานกับตัวแนะนำหลังได้ โดยตัวรวมไบต์โค้ด AABC สำหรับตัวแนะนำหลังมีชื่อว่า CallAfterAdvice ตัวอย่างการใช้งานถูกเขียนขึ้นในเมธอดเริ่มต้นซึ่งได้แสดงในรูปที่ 3.9 สำหรับความสามารถของตัวแนะนำหลังซึ่งได้กล่าวไว้แล้วในส่วนที่ 2.1.4 โดยได้มีการพัฒนาให้ CallAfterAdvice ทำการผูกไบต์โค้ดตามนิยามของตัวแนะนำหลังคือทำการผูกวัตถุในรูปของคลาส MethodHandle ของวัตถุ target กับวัตถุ afterAdvice เข้าด้วยกันโดยจะทำการผูกวัตถุ afterAdvice ไว้ตรงส่วนท้ายของของวัตถุ target ซึ่งจะกำหนดให้วัตถุ afterAdvice สามารถทำการสำรวจค่าที่ส่งออก

```
else if(declare.annotationType().toString().
     endsWith("After"))
     {
     if(
          check(
               new ANTLRInputStream(
                    mt.getAnnotation(
                         After.class
                    ).value()
               ),
               info, tp, callfromMethod,
               access, tpcallfromMethod
          )!=null
     ){
          mhCombine =
               CallAfterAdvice.afterAd(
                    aopAdvice,
                    mhCombine.asType(
                        mhCombine.type().generic()
                    ),
                    names[1], names[2],
                    type.parameterCount()
               ).asType(type);
     }
}
```

รูปที่ 3.9 ตัวอย่างการเรียกใช้งานตัวรวมไบต์โค้ด AABC สำหรับตัวแนะนำหลัง

มาจากวัตถุ target ได้ แต่ไม่มีสิทธิในการเปลี่ยนแปลงค่าดังกล่าว และยังสามารถแทรกการทำงานตามการใช้งานของระบบที่ต้องการให้เกิดขึ้นหลังจากที่วัตถุ target ทำงานเสร็จสิ้นแล้ว

3) ตัวรวมไบต์โค้ด AABC สำหรับตัวแนะนำหลังการคืนค่า

สำหรับตัวรวมไบต์โค้ด AABC ในส่วนนี้จะถูกเรียกใช้งานเมื่อคำสั่ง invokedynamic ตรงตามเงื่อนไขการตัดจุดในตัวอย่างรูปที่ 3.5 ส่วนที่ 3 โดยเช่นเดียวกับตัวแนะนำก่อนซึ่งได้นำเมธอด filterReturnValue ซึ่งอยู่ในคลาส MethodHandles ของแพ็คเกจ java.lang.invoke มาทำการดัดแปลงให้สามารถทำงานตามความสามารถของตัวแนะนำหลังการคืนค่าได้ ในความเป็นจริงสามารถนำเมธอด filterArguments มาดัดแปลงได้ แต่เหตุผลที่เลือกเมธอด filterReturnValue เพราะว่าเมธอด filterReturnValue มีการทำงานที่รวดเร็วกว่าเมธอด filterArguments มาก สำหรับรายละเอียดของตัวแนะนำหลังการคืนค่าได้อธิบายแล้วในบทที่ 2 โดยตัวรวมไบต์โค้ดในส่วนนี้มีชื่อ

```
else if(declare.annotationType().toString().
     endsWith("AfterReturn"))
     {
     if(
          check(
               new ANTLRInputStream(
                    mt.getAnnotation(
                         AfterReturn.class
                    ).value()
               ),
               info, tp, callfromMethod,
               access,tpcallfromMethod
          ) !=null
     ){
          MethodHandle asObject =
               target.asType(
                    type.changeReturnType(Object.class))
               ;
          mhCombine =
               (
                    CallAfterReturnAdvice.afterAd(
                         aopAdvice, asObject
                    )
               ).asType(type);
     }
}
```

รูปที่ 3.10 ตัวอย่างการเรียกใช้งานตัวรวมไบต์โค้ด AABC สำหรับตัวแนะนำหลังการคืนค่า

ว่า CallAfterReturnAdvice ซึ่งทำการผูก MethodHandle ของวัตถุ target และวัตถุ afterReturnAdvice เข้าด้วยกัน โดยในส่วนนี้จะทำการผูกวัตถุ afterReturnAdvice ไว้ตรงส่วนท้ายหลังจบการทำงานของวัตถุ target สำหรับ CallAfterReturnAdvice นี้จะอนุญาตให้ตัวแนะนำหลังการคืนค่าที่สร้างขึ้นสามารถทำการแก้ไขค่าคืนกลับที่วัตถุ target ส่งออกมาได้ สำหรับตัวอย่างของตัวรวมไบต์โค้ดนี้ได้แสดงในรูปที่ 3.10

4) ตัวรวมไบต์โค้ด AABC สำหรับตัวแนะนำครอบ

ตัวรวมไบต์โค้ด AABC ในส่วนตัวแนะนำครอบนี้จะถูกเรียกเพื่อใช้สำหรับสานไบต์โค้ดก็ต่อเมื่อตำแหน่งที่คำสั่ง invokedynamic ที่ถูกเรียกใช้งานตรงตามเงื่อนไขการตัดจุดในรูปตัวอย่างที่ 3.5 ส่วนที่ 4 โดยตัวรวมไบต์โค้ดส่วนนี้ได้นำเมธอด filterArguments ซึ่งอยู่ในคลาส MethodHandles ของแพ็คเกจ java.lang.invoke มาปรับแต่งใหม่ ซึ่งตัวรวมไบต์โค้ดที่พัฒนาขึ้นนี้ชื่อ

```
else if(declare.annotationType().toString().
     endsWith("Around"))
     {
     if(
          check(
               new ANTLRInputStream(
                    mt.getAnnotation(
                         Around.class
                    ).value()
               ),
               info, tp, callfromMethod,
               access, tpcallfromMethod
          ) !=null
     ){
          mhCombine =
               AroundAdvice.aroundAd(
                    mhCombine.asType(
                         target.type().generic()
                    ),
                    aopAdvice,
                    type.parameterCount()
               ).asType(type);

     }
}
```

รูปที่ 3.11 ตัวอย่างการเรียกใช้งานตัวรวมไบต์โค้ด AABC สำหรับตัวแนะนำครอบ

ว่า CallAroundAdvice ได้ทำการพัฒนาให้มีคุณสมบัติพิเศษที่สามารถจัดการกับนิพจน์โปรซีด (Proceed) ได้ สำหรับตัวอย่างการใช้งานในเมธอดเริ่มต้นแสดงในรูปที่ 3.11 โดยรูปแบบการทำงานของตัวแนะนำครอบได้อธิบายในบทที่ 2 ซึ่งความสามารถของตัวแนะนำครอบสามารถทำการเปลี่ยนรูปแบบการทำงานของโปรแกรมที่ถูกเรียกได้ทั้งหมด ซึ่งสามารถเรียกโปรแกรมการทำงานเดิมผ่านโปรซีดโดยในตัวอย่างที่ได้แสดงวัตถุ target ที่เป็น MethodHandle ถูกส่งไปให้วัตถุ aroundAdvice โดยวัตถุ MethodHandle ที่มารับค่าจะมีชื่อว่าโปรซีดโดยผู้พัฒนาซอฟต์แวร์สามารถเรียกใช้การทำงานเดิมของโปรแกรมได้ผ่านการเรียกใช้งานโปรซีด

ในบทที่ 3 นี้ได้อธิบายส่วนของเนื้อหาและองค์ประกอบของการสร้างตัวรวมไบต์โค้ด AABC สำหรับการรวมไบต์โค้ดตามตัวแนะนำแต่ละประเภทและการพัฒนาสร้างตัวแจงส่วนการตัดจุด ในส่วนของเครื่องมือและการวัดประสิทธิภาพการทำงานของตัวรวมไบต์โค้ด AABC ที่ได้พัฒนาขึ้นนี้จะกล่าวถึงในบทถัดไป

บทที่ 4

การทดสอบและอภิปรายผล

ในบทนี้จะกล่าวถึงเครื่องมือสำหรับทำการทดสอบประสิทธิภาพของตัวรวมไบต์โค้ด AABC และตัวแจงส่วนการตัดจุดสำหรับใช้งานระบบเชิงลักษณะแบบพลวัตโดยใช้คำสั่ง invokedynamic ที่พัฒนาขึ้นและส่วนสุดท้ายจะเป็นการอภิปรายผลการทดลอง โดยรายละเอียดมีดังต่อไปนี้

4.1 เครื่องมือสำหรับสร้างและทดสอบการโปรแกรมเชิงลักษณะแบบพลวัตโดยใช้คำสั่ง invokedynamic

ส่วนนี้จะกล่าวถึงเครื่องมือและองค์ประกอบสำหรับใช้ในการทดสอบซึ่งมีดังต่อไปนี้

- เครื่องคอมพิวเตอร์แบบพกพา Acer ASPIRE 4830G หน่วยประมวลผล (CPU) Intel Core i5-2410M 2.3 GHz หน่วยความจำหลัก (RAM) 8 GB ระบบปฏิบัติการ Windows 7 Ultimate ระบบประมวลผลแบบ 64-bit
- Java Development Kit (JDK) รุ่น 1.7.0_09 สำหรับประมวลผลแบบ 64 bit
- ASM 4.0 สำหรับจัดการไบต์โค้ด
- AspectJ รุ่น 1.7.1 สำหรับสร้างและใช้เป็นเกณฑ์วัดผลประสิทธิภาพ
- Eclipse Java EE IDE สำหรับนักพัฒนาซอฟต์แวร์รุ่น Juno Service Release 1 สำหรับการพัฒนางานวิจัยชิ้นนี้
- Bytecode Outline รุ่น 2.4.1 สำหรับวิเคราะห์และแก้ไขปัญหาไบต์โค้ด
- ชุดทดสอบมาตรฐาน DaCapo และชุดทดสอบมาตรฐาน SciMark 2.0 สำหรับใช้ทดสอบประสิทธิภาพการใช้งานตัวรวมไบต์โค้ด AABC และตัวแจงส่วนการตัดจุด
- bb.util.Benchmark สำหรับจับเวลาในการทดสอบกับชุดทดสอบ SciMark 2.0

4.2 การออกแบบการทดสอบประสิทธิภาพ

การทดสอบประสิทธิภาพของงานวิจัยนี้สามารถแบ่งออกได้ 2 การทดสอบคือ การทดสอบประสิทธิภาพของตัวรวมไบต์โค้ด AABC และการทำสอบการใช้งานตัวแจงส่วนการตัดจุดที่ได้ทำการพัฒนาขึ้นโดยมีรายละเอียดดังต่อไปนี้

4.2.1 การทดสอบประสิทธิภาพของตัวรวมไบต์โค้ด AABC

ในงานวิจัยนี้จะดำเนินการทดสอบประสิทธิภาพของระบบเชิงลักษณะแบบพลวัตที่ใช้คำสั่ง invokedynamic ซึ่งจะแบ่งการทดลองออกเป็น 4 การทดลองโดยแบ่งตามประเภทของตัวแนะนำแต่ละประเภทซึ่งมีรายละเอียดสำหรับการทดสอบประสิทธิภาพดังต่อไปนี้

1) ตัวแนะนำก่อน

การทดสอบประสิทธิภาพการทำงานของตัวรวมไบต์โค้ดในส่วนของการทำงานตามหลักการของตัวแนะนำก่อนจะทำการทดสอบประสิทธิภาพโดยเทียบการทำงานระหว่าง 3 ตัวรวมไบต์โค้ดคือ ตัวแรกเป็นตัวรวมไบต์โค้ด AABC ที่ได้พัฒนาขึ้นสำหรับรวมไบต์โค้ดตามหลักการของตัวแนะนำก่อนเทียบกับการทำงานร่วมกับตัวรวมไบต์โค้ดมาตรฐานที่ภาษาจาวามีมาให้ (Standard bytecode combinators) ซึ่งเป็นตัวรวมไบต์โค้ดที่ประยุกต์การทำงานจากตัวรวมไบต์โค้ดที่มีในภาษาจาวาตามหลักการทำงานของตัวแนะนำก่อนโดยทั้งสองตัวรวมไบต์โค้ดนี้อาศัยกลไกการทำงานของคำสั่ง invokedynamic และสำหรับระบบเชิงลักษณะมาตรฐานคือ AspectJ จะเป็นตัวมาตรฐานสำหรับการวัดประสิทธิภาพของการทำงานของระบบเชิงลักษณะ โดยจะใช้เป็นฐานสำหรับการวัดประสิทธิภาพของตัวรวมไบต์โค้ดสองตัวแรก

2) ตัวแนะนำหลัง

สำหรับการทดสอบประสิทธิภาพของตัวรวมไบต์โค้ด AABC ที่พัฒนาสำหรับการใช้งานตามหลักการของตัวแนะนำหลังซึ่งทำงานโดยอาศัยกลไกการทำงานของคำสั่ง invokedynamic จะทำการทดสอบเทียบประสิทธิภาพกับการทำงานของ AspectJ เท่านั้นเพราะตัวรวมไบต์โค้ดมาตรฐานที่ภาษาจาวามีมาให้นั้นไม่สนับสนุนการรวมไบต์โค้ดตามหลักการทำงานของตัวแนะนำหลัง

3) ตัวแนะนำหลังการคืนค่า

ในการทำสอบประสิทธิภาพของตัวรวมไบต์โค้ดตามหลักการของตัวแนะนำหลังการคืนค่าจะทำการทดสอบเทียบประสิทธิภาพทั้งหมด 3 ตัวรวมไบต์โค้ด โดยตัวแรกทำการทดสอบประสิทธิภาพกับตัวรวมไบต์โค้ด AABC สำหรับตัวรวมไบต์โค้ดในส่วนนี้ได้ทำการพัฒนาให้สอดคล้องกับการทำงานตามหลักการของตัวแนะนำหลังการคืนค่า สำหรับตัวที่สองคือตัวรวม

ไบต์โค้ดมาตรฐานที่ภาษาจาวามีมาให้ซึ่งในส่วนนี้ภาษาจาวาสามารถประยุกต์ใช้งานตามหลักการของตัวแนะนำหลังการคืนค่าได้ และตัวสุดท้ายจะทำการเทียบประสิทธิภาพกับระบบเชิงลักษณะของ AspectJ ซึ่งจะเป็นฐานในการวัดประสิทธิภาพของทั้ง 2 ตัวรวมไบต์โค้ดแรก

4) ตัวแนะนำครอบ

การทดสอบประสิทธิภาพของตัวแนะนำตัวสุดท้ายคือตัวแนะนำครอบจะทำการทดสอบประสิทธิภาพระหว่างตัวรวมไบต์โค้ด AABC ที่ได้พัฒนาขึ้นตามหลักการของตัวแนะนำครอบโดยในส่วนนี้จะทำงานผ่านกลไกของคำสั่ง invokedynamic ซึ่งจะทำการเทียบประสิทธิภาพกับ AspectJ ที่ทำงานผ่านกลไกการทำงานปกติของระบบเชิงลักษณะแบบสถิตย์ โดยในการทดสอบประสิทธิภาพในส่วนนี้จะทำการทดสอบกับตัวรวมไบต์โค้ดเพียงสองตัวที่ได้กล่าวมาข้างต้นเท่านั้น เพราะตัวรวมไบต์โค้ดมาตรฐานของภาษาจาวาไม่สามารถนำมาประยุกต์ใช้สร้างตัวรวมไบต์โค้ดตามหลักการทำงานของตัวแนะนำครอบได้

```
"call(* *(..)) &&
!withincode(public static void main())"
```

รูปที่ 4.1 เงื่อนไขการตัดจุดสำหรับทดสอบประสิทธิภาพ

ในการทดลองจะทำการควบคุมสภาพแวดล้อมการทำงานของเครื่องคอมพิวเตอร์โดยทำการปิดโปรแกรมที่ไม่เกี่ยวข้องทั้งหมดเพื่อลดค่าความไม่ถูกต้องของเวลาที่อาจเกิดขึ้นได้ หลังจากนั้นจะกำหนดเงื่อนไขการตัดจุดตามตัวอย่างในรูปที่ 4.1 ซึ่งหมายความว่าทำการเลือกจุดที่มีการ call ทั้งหมดยกเว้นในเมธอด main และจะทำการทดสอบแต่ละการทดลองอย่างละ 10 ครั้งและจะตัดค่าสองตัวบนที่มากสุด และสองตัวล่างที่น้อยสุดออก จากนั้นจะนำค่าที่ได้มาหาค่าเฉลี่ยเพื่อให้ผลการทดลองมีความถูกต้องมากที่สุด หลังจากทำการทดลองแล้วจะนำผลการทดลองของตัวแนะนำแต่ละประเภทมาเปรียบเทียบกัน โดยจะทำการทดสอบกับชุดทดสอบมาตรฐานของ SciMark 2.0 ทั้งหมด 3 ตัว ได้แก่ตัวทดสอบ Fibonacci (Fib) ตัวทดสอบ Fast Fouier Transformations (FFT) และตัวทดสอบ LU matrix factorization (LU) และชุดทดสอบมาตรฐาน DaCapo ทั้งหมด 6 ตัว ได้แก่ตัวทดสอบ antlr ตัวทดสอบ avrora ตัวทดสอบ h2 ตัวทดสอบ luindex ตัวทดสอบ lusearch และตัวทดสอบ sunflow จากนั้นจะอภิปรายผลการทดลองต่อไปในส่วนที่ 4.3.1

4.2.2 การออกแบบการทดสอบประสิทธิภาพของตัวแจงส่วนการตัดจุด

การทดสอบประสิทธิภาพการทำงานของตัวแจงส่วนการตัดจุดจะทำการควบคุมสภาพแวดล้อมการทดสอบโดยกำหนดการแปลงคำสั่ง invokestatic คำสั่ง invokevirtual คำสั่ง invokeinterface และคำสั่ง invokespecial ไปเป็นคำสั่ง invokedynamic ซึ่งจะไม่ทำการแปลงการเรียกภายในเมธอด main ทั้งหมด โดยตรงกับเงื่อนไขการตัดจุดที่ได้ยกตัวอย่างไว้ข้างต้นในส่วนที่ 4.2.1 และทำการเทียบประสิทธิภาพกับการทดสอบโดยไม่ใช้ตัวแจงส่วนการตัดจุดซึ่งทั้งสองแบบจะเลือกสานเฉพาะตัวแนะนำก่อนเพียงประเภทเดียวเท่านั้นเพราะในส่วนนี้ต้องการวัดผลประสิทธิภาพของการใช้งานตัวแจงส่วนการตัดจุด โดยรูปแบบการวัดประสิทธิภาพจะคล้ายกับส่วนที่ 4.2.1 คือทำการวัดประสิทธิภาพทั้งหมด 10 ครั้ง หลังจากนั้นนำค่าเวลาที่ได้มาเรียงลำดับแล้วตัดสองตัวบนที่มากที่สุด และสองตัวล่างที่น้อยที่สุดออกไป แล้วนำค่าที่เหลือมาเฉลี่ยก็จะได้เวลาเฉลี่ยที่มีค่าถูกต้องมากที่สุด เมื่อได้ค่าเฉลี่ยของเวลาที่ได้แล้วก็นำมาเทียบผลระหว่างการทดสอบทั้งสองแบบที่ได้กล่าวมาข้างต้น โดยในส่วนนี้จะอภิปรายผลในส่วนที่ 4.3.2

4.3 อภิปรายผลการทดลอง

สำหรับการอภิปรายผลการทดลองในส่วนนี้สามารถแบ่งการอภิปรายผลการทดลองได้ 2 แบบคือ การอภิปรายผลการทดลองในส่วนประสิทธิภาพการทำงานของตัวรวมไบต์โค้ด AABC และการอภิปรายผลการทดลองในส่วนการทดสอบประสิทธิภาพการใช้งานตัวแจงส่วนการตัดจุดดังต่อไปนี้

4.3.1 อภิปรายผลการทดสอบประสิทธิภาพของตัวรวมไบต์โค้ด AABC

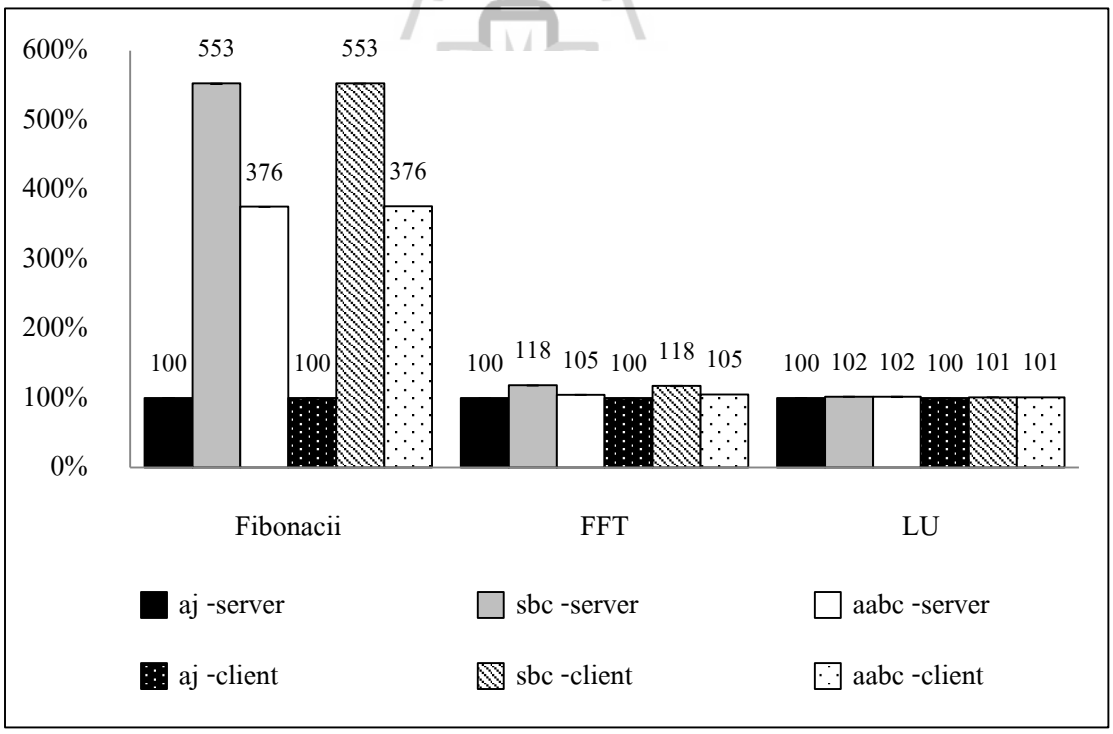
จากการทดสอบประสิทธิภาพของตัวรวมไบต์โค้ด AABC ซึ่งขั้นตอนการทดสอบในส่วนนี้ได้อธิบายแล้วในส่วนที่ 4.2.1 โดยตัวรวมไบต์โค้ด AABC ใช้สัญลักษณ์ในตารางและกราฟการทดลองเป็น aabc ส่วนตัวรวมไบต์โค้ดมาตรฐานที่ภาษาจาวามีมาให้ใช้ sbc แทนในกราฟและตารางการทดลอง และสำหรับ AspectJ ใช้สัญลักษณ์ aj โดยการทดสอบประสิทธิภาพของการทำงานสามารถวิเคราะห์ผลการทดลองตามตัวแนะนำแต่ละประเภทได้ดังนี้

1) ผลการทดสอบกับตัวแนะนำก่อน

ผลการทดสอบในส่วนของตัวแนะนำก่อนกับชุดทดสอบมาตรฐาน SciMark 2.0 ดังแสดงในตารางที่ 4.1 และรูปที่ 4.2 จะเห็นได้ชัดว่าประสิทธิภาพการทำงานของตัวทดสอบ Fibonacci ในส่วนของเซิร์ฟเวอร์พบว่าการทำงานของ aabc ช้ากว่า aj อยู่ 453% แต่เร็วกว่า

ตารางที่ 4.1 ตารางแสดงผลประสิทธิภาพการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนของตัวแนะนำก่อนกับชุดทดสอบ SciMark 2.0

แพลตฟอร์มการทำงาน	เวลาเฉลี่ย (s)			โอเวอร์เฮด (%)		
	Fib	FFT	LU	Fib	FFT	LU
aj -server	0.53	1.34E-04	1.04E-02	100	100	100
sbc -server	2.91	1.59E-04	1.06E-02	553	118	102
aabc -server	1.98	1.41E-04	1.06E-02	376	105	102
aj -client	0.53	1.34E-04	1.04E-02	100	100	100
sbc -client	2.91	1.58E-04	1.05E-02	553	118	101
aabc -client	1.98	1.41E-04	1.05E-02	376	105	101



รูปที่ 4.2 แผนภูมิแสดงการเปรียบเทียบโอเวอร์เฮดของเวลาการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนตัวแนะนำก่อนกับชุดทดสอบ SciMark 2.0
(โอเวอร์เฮดน้อยประสิทธิภาพสูง)

sbc อยู่ 177% และในส่วนของไคลเอนต์ aabc ช้ากว่า aj อยู่ 453% แต่มีการทำงานที่เร็วกว่า sbc อยู่ 177 % สำหรับการทำงานของตัวรวมไบต์โค้ดกับตัวทดสอบ FFT และตัวทดสอบ LU ในส่วนของ aabc และ sbc ไม่ต่างกันมากเพราะตัวทดสอบ FFT และตัวทดสอบ LU มีการเรียกใช้คำสั่ง invokedynamic ไม่มากแต่การทำงานของตัวทดสอบ Fibonacci มีการเรียกตัวเองซ้ำหลายครั้งทำให้เกิดการเรียกใช้คำสั่ง invokedynamic เกิดขึ้นหลายครั้งส่งผลให้ประสิทธิภาพการทำงานของ aabc แตกต่างกับ sbc อย่างชัดเจน

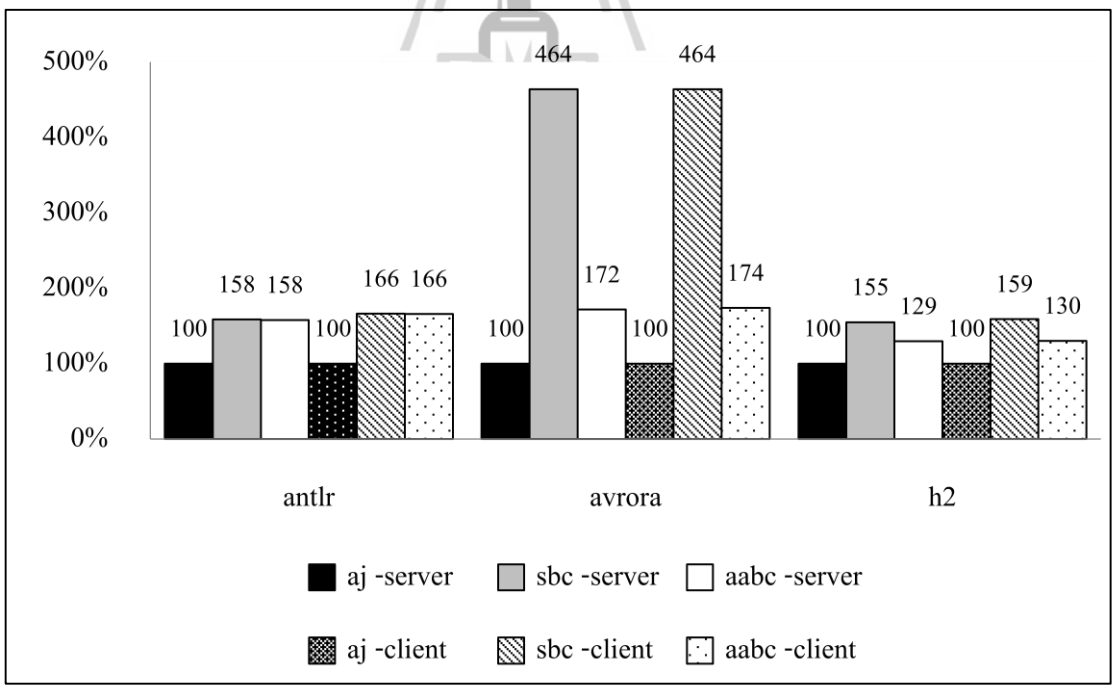
สำหรับประสิทธิภาพการทำงานกับชุดทดสอบมาตรฐาน DaCapo กับการทดสอบการทำงานของตัวทดสอบ antlr ตัวทดสอบ avrora และตัวทดสอบ h2 ดังแสดงในตารางที่ 4.2 และรูปที่ 4.3 โดยในส่วนการทดสอบกับตัวทดสอบ antlr พบว่าการทำงานในส่วนของเซิร์ฟเวอร์ของ aabc ช้ากว่า aj อยู่ 58% แต่มีโอเวอร์เฮดเท่ากับ sbc เช่นเดียวกับไคลเอนต์ที่ aabc มีการทำงานที่ช้ากว่า aj อยู่ 66% และมีโอเวอร์เฮดไม่ต่างกับ sbc แต่ถ้าพิจารณาจากค่าของเวลาแล้วจะพบว่า aabc มีการทำงานที่เร็วกว่า sbc สำหรับการทดสอบกับตัวทดสอบ avrora พบว่าการทำงานของ aabc ในส่วนของเซิร์ฟเวอร์มีการทำงานที่ช้ากว่า aj อยู่ 72% และเร็วกว่า sbc ถึง 292% และในส่วนของไคลเอนต์ aabc มีการทำงานที่ช้ากว่า aj เพียง 74% แต่เร็วกว่า sbc อยู่ 290% สำหรับการทดสอบประสิทธิภาพกับตัวทดสอบ h2 พบว่าการทำงานของตัวรวมไบต์โค้ดของ aabc ในส่วนของเซิร์ฟเวอร์มีการทำงานที่ช้ากว่า aj อยู่ 29% และเร็วกว่า sbc อยู่ 26% และในส่วนการทำงานบนไคลเอนต์ก็เช่นเดียวกัน aabc มีการทำงานที่ช้ากว่า aj เพียง 30% แต่มีการทำงานที่เร็วกว่า sbc ถึง 29%

สำหรับในส่วนของการทดสอบการทำงานกับตัวทดสอบ luindex ตัวทดสอบ lusearch และตัวทดสอบ sunflow แสดงในตารางที่ 4.3 และ รูปที่ 4.4 ในส่วนการทดสอบกับตัวทดสอบ luindex พบว่าการทำงานของตัวรวมไบต์โค้ดของ aabc ในส่วนการทำงานบนเซิร์ฟเวอร์มีการทำงานที่ช้ากว่า aj เพียง 42% แต่เร็วกว่า sbc ถึง 38% และในส่วนการทำงานบนไคลเอนต์ aabc มีการทำงานที่ช้ากว่า aj อยู่ 40% และมีการทำงานที่เร็วกว่า sbc อยู่ 3% สำหรับการทดสอบกับตัวทดสอบ lusearch พบว่า aabc ในส่วนของเซิร์ฟเวอร์มีการทำงานที่ช้ากว่า aj อยู่ 64% แต่มีการทำงานที่เร็วกว่า sbc ถึง 36% และในส่วนการทำงานบนไคลเอนต์ aabc ช้ากว่า aj เพียง 55% แต่มีการทำงานที่เร็วกว่า sbc ถึง 49% และตัวทดสอบตัวสุดท้ายคือตัวทดสอบ sunflow การทำงานของ aabc บนเซิร์ฟเวอร์กับชุดทดสอบนี้มีการทำงานที่ช้ากว่า aj อยู่ 163% เท่านั้น และมีการทำงานที่เร็วกว่า sbc ถึง 274% สำหรับการทำงานบนไคลเอนต์ aj มีการทำงานที่เร็วกว่า aabc อยู่ 165% และ sbc มีการทำงานที่ช้ากว่า aabc ถึง 263%

จากการทดสอบแต่ละตัวพบว่าการทำงานของ aabc และ sbc มีการทำงานที่ช้ากว่า aj เพราะว่า aj ได้มีการเตรียมการสานไบต์โค้ดแล้วก่อนถึงช่วงเวลาโปรแกรมกำลังทำงาน แต่สำหรับ aabc และ sbc อาศัยกลไกการทำงานของคำสั่ง invokedynamic ซึ่งจะทำการสานไบต์โค้ด

ตารางที่ 4.2 ตารางแสดงผลประสิทธิภาพการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนของตัวแนะนำก่อนกับชุดทดสอบ DaCapo ชุดที่ 1

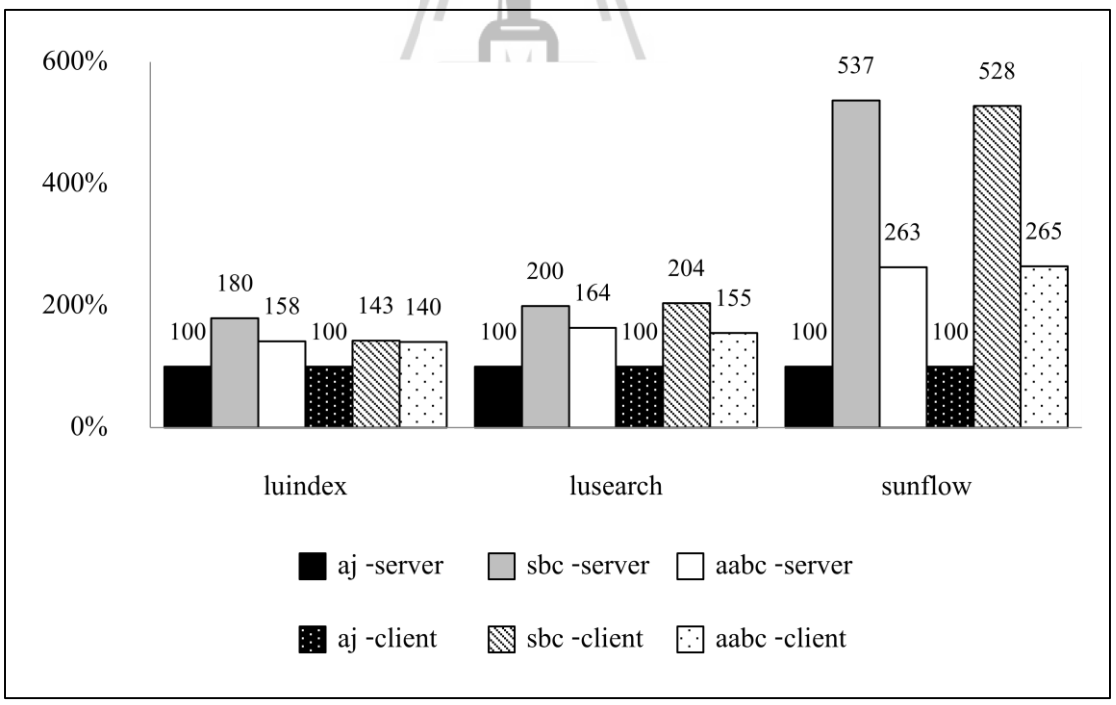
แพลตฟอร์มการทำงาน	เวลาเฉลี่ย (ms)			โอเวอร์เฮด (%)		
	antlr	avrora	h2	antlr	avrora	h2
aj -server	3,760	4,115	7,434	100	100	100
sbc -server	5,959	19,094	11,508	158	464	155
aabc -server	5,928	7,059	9,626	158	172	129
aj -client	3,698	4,111	7,384	100	100	100
sbc -client	6,146	19,074	11,731	166	464	159
aabc -client	6,131	7,140	9,612	166	174	130



รูปที่ 4.3 แผนภูมิแสดงการเปรียบเทียบโอเวอร์เฮดของเวลาการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนตัวแนะนำก่อนกับชุดทดสอบ DaCapo ชุดที่ 1
(โอเวอร์เฮดน้อยประสิทธิภาพสูง)

ตารางที่ 4.3 ตารางแสดงผลประสิทธิภาพการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนของตัวแนะนำก่อนกับชุดทดสอบ DaCapo ชุดที่ 2

แพลตฟอร์มการทำงาน	เวลาเฉลี่ย (ms)			โอเวอร์เฮด (%)		
	luindex	lusearch	sunflow	luindex	lusearch	sunflow
`aj -server`	2,304	3,026	5,229	100	100	100
`sbc -server`	4,137	6,037	28,088	180	200	537
`aabc -server`	3,734	4,961	13,775	142	164	263
`aj -client`	2,294	2,995	5,216	100	100	100
`sbc -client`	3,279	6,115	27,534	143	204	528
`aabc -client`	3,713	4,649	13,817	140	155	265



รูปที่ 4.4 แผนภูมิแสดงการเปรียบเทียบโอเวอร์เฮดของเวลาการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนตัวแนะนำก่อนกับชุดทดสอบ DaCapo ชุดที่ 2
(โอเวอร์เฮดน้อยประสิทธิภาพสูง)

ตอนช่วงเวลาโปรแกรมกำลังทำงานจึงส่งผลให้การวัดประสิทธิภาพการทำงานของ aabc และ sbc มีการทำงานที่ช้ากว่า aj อย่างไรก็ตามจากการอภิปรายข้างต้นในส่วนการทดลองกับตัวแนะนำก่อน พบว่าการทำงานของ aabc ซึ่งเป็นตัวรวมไบต์โค้ดที่ได้ทำการพัฒนาขึ้นนี้มีการทำงานที่เร็วกว่าการทำงานของ sbc เพราะ sbc มีการทำงานบางอย่างที่ไม่เกี่ยวข้องกับการทำงานตามหลักการของตัวแนะนำก่อนทำให้ต้องเสียเวลาในการทำงานในส่วนนี้และ aabc ถูกพัฒนาขึ้นเพื่อใช้สำหรับการใช้งานตัวแนะนำก่อนโดยเฉพาะจึงเป็นเหตุผลที่ทำให้การทำงานของตัวรวมไบต์โค้ด aabc ในส่วนของตัวแนะนำก่อนมีการทำงานที่เร็วกว่า sbc

1) ผลการทดสอบกับตัวแนะนำหลัง

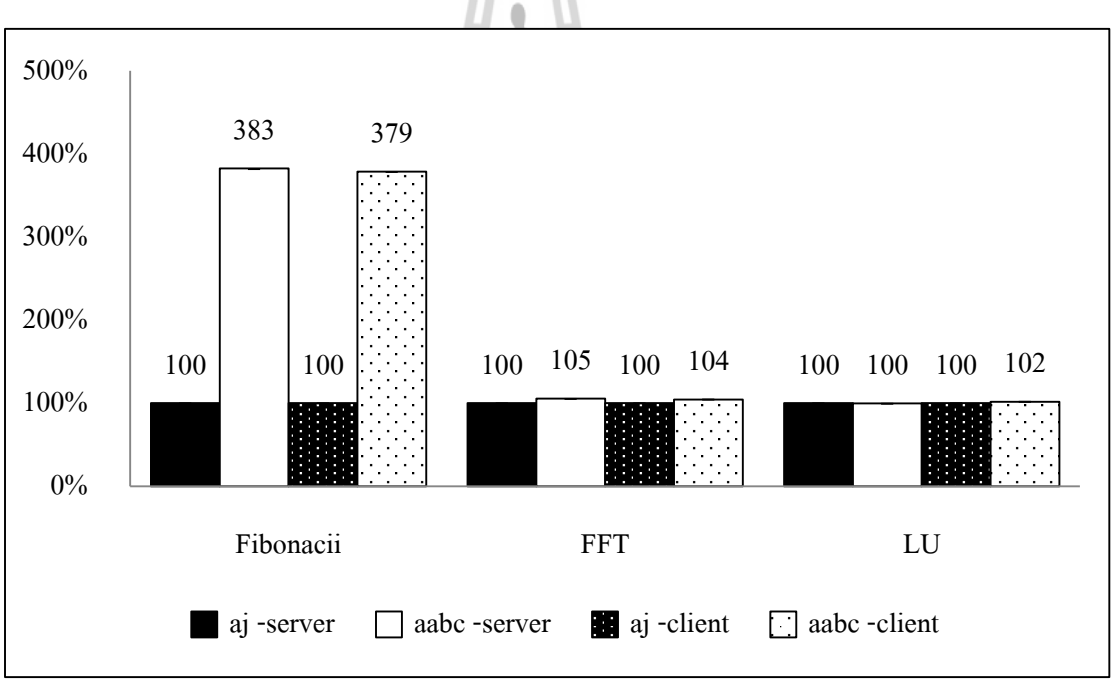
สำหรับผลการทดสอบในตารางที่ 4.4 และ รูปที่ 4.5 จะเป็นส่วนของชุดทดสอบมาตรฐาน SciMark 2.0 โดยการทดสอบในส่วนของตัวทดสอบ Fib พบว่า aabc มีการทำงานที่ช้ากว่า aj ในส่วนของเซิร์ฟเวอร์อยู่ 283% และในส่วนของไคลเอนต์อยู่ 279% สำหรับการทดสอบกับตัวทดสอบ FFT พบว่า aabc มีการทำงานที่ช้ากว่า aj ในส่วนของเซิร์ฟเวอร์อยู่ 5% และไคลเอนต์อยู่ 4% และตัวทดสอบตัวสุดท้ายของชุดทดสอบมาตรฐาน SciMark 2.0 คือตัวทดสอบ LU ซึ่ง aabc มีการทำงานที่ไม่ต่างจาก aj ทั้งในส่วนของการทำงานบนเซิร์ฟเวอร์และไคลเอนต์เนื่องจากตัวทดสอบ LU มีการเรียกใช้งานคำสั่ง invokedynamic ไม่มากทำให้ค่าโอเวอร์เฮดแทบไม่เกิดขึ้น

ผลการทดสอบจากตารางที่ 4.5 และรูปที่ 4.6 เป็นการทดสอบประสิทธิภาพในส่วนของตัวแนะนำหลังกับชุดทดสอบ DaCapo ของ 3 ตัวแรกคือตัวทดสอบ antlr ตัวทดสอบ avrora และตัวทดสอบ h2 พบว่าการทดสอบกับตัวทดสอบ antlr ในส่วนของ aabc มีการทำงานที่ช้ากว่า aj ในส่วนของของเซิร์ฟเวอร์เพียง 54% และไคลเอนต์เพียง 59% สำหรับการทดสอบกับตัวทดสอบ avrora พบว่า aabc มีการทำงานที่ช้ากว่า aj โดยในส่วนของเซิร์ฟเวอร์อยู่ 74% และไคลเอนต์อยู่ 72% การทดสอบกับตัวสุดท้ายใน 3 ตัวแรกนี้คือตัวทดสอบ h2 ซึ่งการทำงานของตัวรวมไบต์โค้ดของ aabc มีการทำงานที่ช้ากว่า aj เพียง 85% ในส่วนของเซิร์ฟเวอร์ และ 69% ในส่วนของไคลเอนต์

สำหรับผลการทดสอบประสิทธิภาพในตารางที่ 4.6 และรูปที่ 4.7 เป็นการทดสอบกับตัวทดสอบ luindex ตัวทดสอบ lusearch และตัวทดสอบ sunflow ของชุดทดสอบมาตรฐาน DaCapo โดยการทดสอบกับตัวทดสอบ luindex พบว่าประสิทธิภาพการทำงานของตัวรวมไบต์โค้ด aabc มีการทำงานที่ช้ากว่า aj เพียง 64% ในส่วนของเซิร์ฟเวอร์ และ 69% ในส่วนของไคลเอนต์ สำหรับการทดสอบกับตัวทดสอบ lusearch พบว่า aabc มีการทำงานที่เร็วกว่า aj ถึง 13% ในส่วนของเซิร์ฟเวอร์ และ 4% ในส่วนของไคลเอนต์ และสำหรับการทดสอบกับตัวทดสอบ sunflow พบว่า aabc ทำงานช้ากว่า aj ในส่วนการทำงานบนเซิร์ฟเวอร์มีโอเวอร์เฮดเกิดขึ้น 169% และบนไคลเอนต์เกิดขึ้น 172%

ตารางที่ 4.4 ตารางแสดงผลประสิทธิภาพการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนของตัวแนะนำหลังกับชุดทดสอบ SciMark 2.0

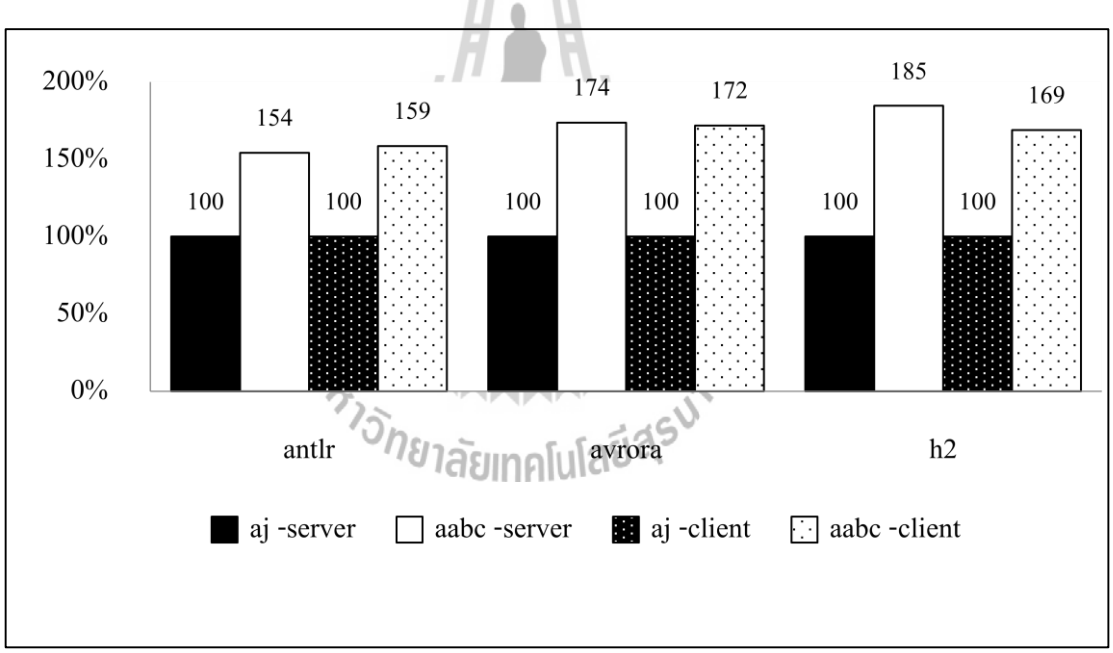
แพลตฟอร์มการทำงาน	เวลาเฉลี่ย (s)			โอเวอร์เฮด (%)		
	Fib	FFT	LU	Fib	FFT	LU
`aj -server`	0.52	1.34E-04	1.05E-02	100	100	100
`aabc -server`	1.98	1.41E-04	1.05E-02	383	105	100
`aj -client`	0.52	1.35E-04	1.07E-02	100	100	100
`aabc -client`	1.98	1.41E-04	1.09E-02	379	104	102



รูปที่ 4.5 แผนภูมิแสดงการเปรียบเทียบโอเวอร์เฮดของเวลาการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนตัวแนะนำหลังกับชุดทดสอบ SciMark 2.0
(โอเวอร์เฮดน้อยประสิทธิภาพสูง)

ตารางที่ 4.5 ตารางแสดงผลประสิทธิภาพการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนของตัวแนะนำหลังกับชุดทดสอบ DaCapo ชุดที่ 1

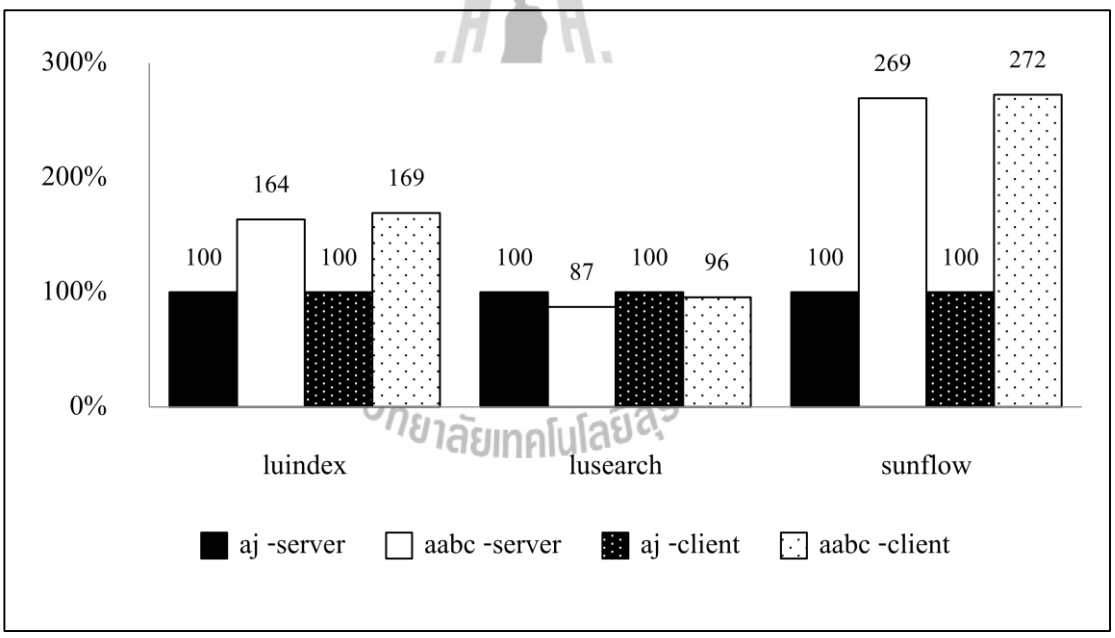
แพลตฟอร์มการทำงาน	เวลาเฉลี่ย (ms)			โอเวอร์เฮด (%)		
	antlr	avrora	h2	antlr	avrora	h2
aj -server	4,118	4,207	9,846	100	100	100
aabc -server	6,349	7,311	18,192	154	174	185
aj -client	4,103	4,230	9,878	100	100	100
aabc -client	6,505	7,272	16,676	159	172	169



รูปที่ 4.6 แผนภูมิแสดงการเปรียบเทียบโอเวอร์เฮดของเวลาการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนตัวแนะนำหลังกับชุดทดสอบ DaCapo ชุดที่ 1
(โอเวอร์เฮดน้อยประสิทธิภาพสูง)

ตารางที่ 4.6 ตารางแสดงผลประสิทธิภาพการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนของตัวแนะนำหลังกับชุดทดสอบ DaCapo ชุดที่ 2

แพลตฟอร์มการทำงาน	เวลาเฉลี่ย (ms)			โอเวอร์เฮด (%)		
	luindex	lusearch	sunflow	luindex	lusearch	sunflow
`aj -server`	2,327	4,290	5,455	100	100	100
`aabc -server`	3,806	3,744	14,688	164	87	269
`aj -client`	2,296	4,306	5,419	100	100	100
`aabc -client`	3,887	4,119	14,760	169	96	272



รูปที่ 4.7 แผนภูมิแสดงการเปรียบเทียบโอเวอร์เฮดของเวลาการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนตัวแนะนำหลังกับชุดทดสอบ DaCapo ชุดที่ 2
(โอเวอร์เฮดน้อยประสิทธิภาพสูง)

จากการทดสอบประสิทธิภาพในส่วนของตัวแนะนำหลังสามารถอภิปรายแบบสรุปได้ว่าในส่วนนี้ไม่ได้ทำการทดสอบร่วมกับ sbc เพราะว่า sbc ไม่สนับสนุนการทำงานในส่วนของตัวแนะนำหลัง และผลการทดสอบประสิทธิภาพพบว่าการทำงานของ aabc ทั้งในส่วนที่ทำการทดสอบกับ SciMark 2.0 และ DaCapo ในแต่ละตัวมีการทำงานที่ช้ากว่า aj ยกเว้นตัวทดสอบ lusearch โดยเกิดขึ้นได้จากกระบวนการเพิ่มประสิทธิภาพการทำงานของจาวาเวอร์ชวลแมชีนในตัวเอง

2) ผลการทดสอบกับตัวแนะนำหลังการคืนค่า

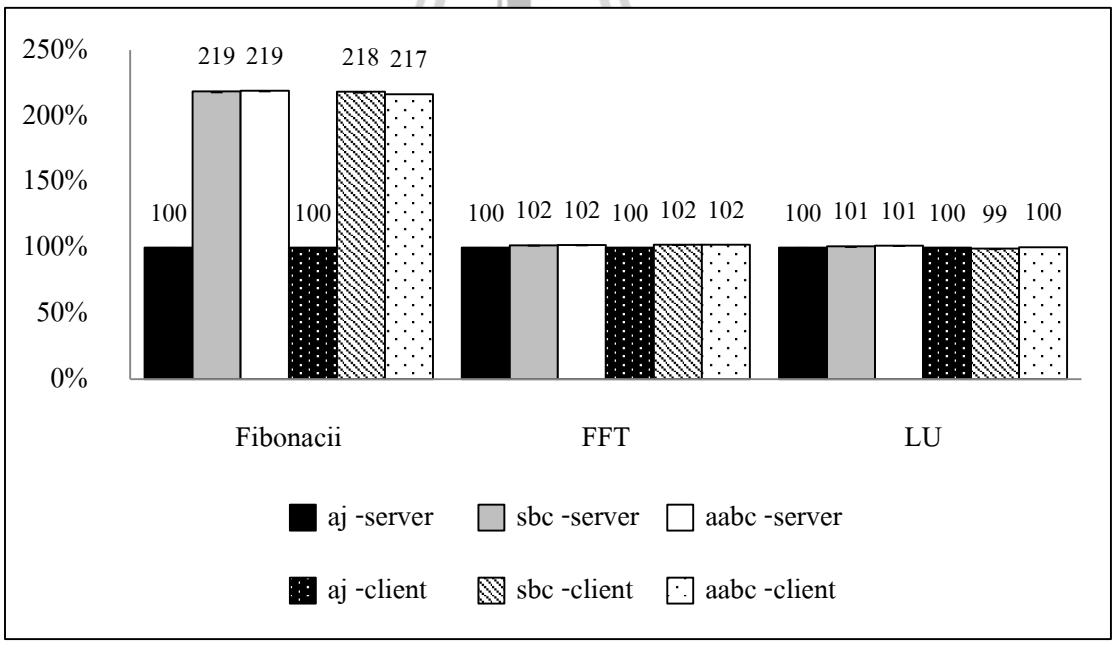
สำหรับผลการทดสอบประสิทธิภาพกับตัวแนะนำหลังการคืนค่าได้แสดงในตารางที่ 4.7 และรูปที่ 4.8 ซึ่งเป็นการทดสอบกับชุดทดสอบใน SciMark 2.0 โดยในการทดสอบกับตัวทดสอบ Fib พบว่าประสิทธิภาพการทำงานของของ aabc และ sbc ทั้งในส่วนของเซิร์ฟเวอร์และไคลเอนต์ไม่ต่างกัน และ aabc ช้ากว่า aj อยู่ 119% เมื่อทดสอบบนเซิร์ฟเวอร์ และ 117% เมื่อทดสอบบนไคลเอนต์ สำหรับการทดสอบกับตัวทดสอบ FFT และตัวทดสอบ LU ก็เช่นกันพบว่าประสิทธิภาพการทำงานของ aabc และ sbc เมื่อเทียบการทำงานกับ aj พบว่าโอเวอร์เฮดที่เกิดไม่ต่างกันมาก

ในตารางที่ 4.8 และรูปที่ 4.9 สามารถอภิปรายผลการทดลองได้ว่าการทดสอบกับตัวทดสอบ antlr ของตัวรวมไบต์โค้ด aabc มีการทำงานที่เร็วกว่า sbc ประมาณ 3% ในส่วนของเซิร์ฟเวอร์แต่ในส่วนของไคลเอนต์ไม่ต่างกัน และ aabc มีการทำงานที่ช้ากว่า aj เพียง 29% ในส่วนของเซิร์ฟเวอร์ และ 25% ในส่วนของไคลเอนต์ สำหรับการทดสอบกับตัวทดสอบ avrora พบว่าประสิทธิภาพการทำงานของ aabc และ sbc ไม่ต่างกันมากทั้งในส่วนการทำงานบนเซิร์ฟเวอร์และไคลเอนต์ และตัวสุดท้ายของชุดนี้คือตัวทดสอบ h2 โดย aabc มีการทำงานที่เร็วกว่า sbc ประมาณ 6% ในส่วนของเซิร์ฟเวอร์และไคลเอนต์ และ aabc มีการทำงานที่ช้ากว่า aj เพียง 4% ทั้งในส่วนการทำงานบนเซิร์ฟเวอร์และไคลเอนต์

สำหรับในตารางที่ 4.9 และรูปที่ 4.10 แสดงผลการทดลองกับตัวทดสอบ luindex ตัวทดสอบ lusearch และตัวทดสอบ sunflow โดยในส่วนของตัวทดสอบ luindex พบว่าประสิทธิภาพการทำงานของ aabc มีการทำงานที่ช้ากว่า aj เพียง 13% และ sbc อยู่ 1% ทั้งในส่วนการทำงานบนเซิร์ฟเวอร์และไคลเอนต์แต่ว่าถ้าพิจารณาจากเวลาพบว่าไม่ต่างกันมาก สำหรับการทดสอบกับตัวทดสอบ lusearch พบว่าทั้ง aabc และ sbc มีการทำงานที่เร็วกว่า aj ทั้งในส่วนการทำงานบนเซิร์ฟเวอร์และไคลเอนต์ ตัวทดสอบตัวสุดท้ายคือตัวทดสอบ sunflow พบว่าประสิทธิภาพการทำงานของ aabc และ sbc เมื่อพิจารณาจากเวลาแล้วแทบไม่ต่างกันมากทั้งในส่วนการทำงานบนเซิร์ฟเวอร์และไคลเอนต์

ตารางที่ 4.7 ตารางแสดงผลประสิทธิภาพการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนของตัวแนะนำหลังการคืนค่ากับชุดทดสอบ SciMark 2.0

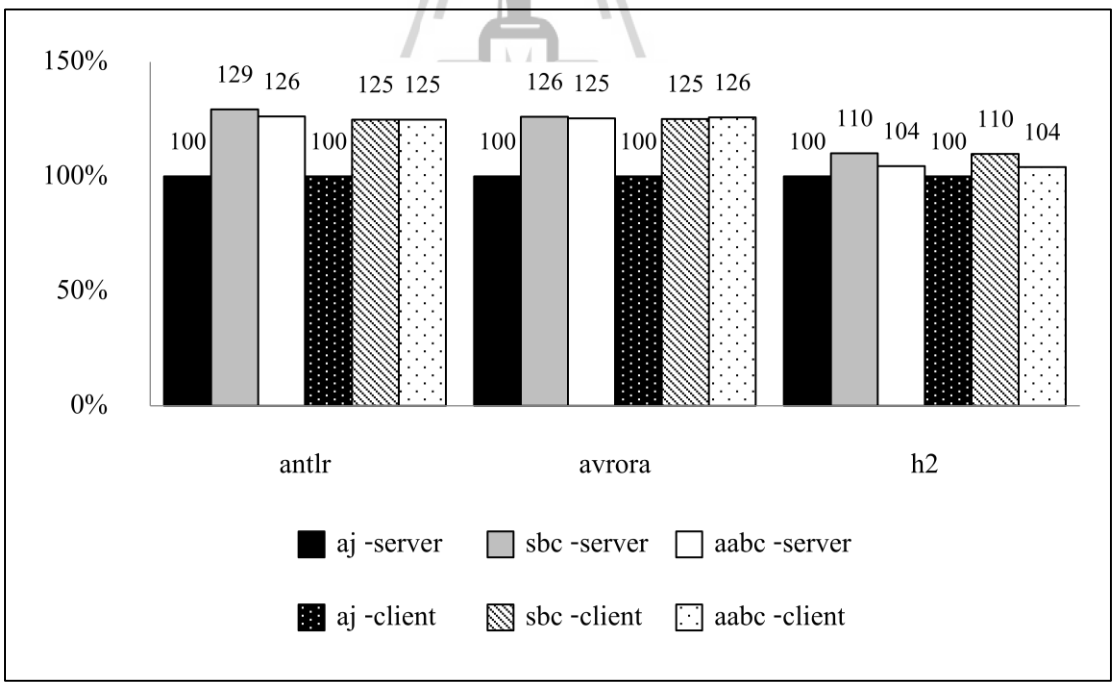
แพลตฟอร์มการทำงาน	เวลาเฉลี่ย (s)			โอเวอร์เฮด (%)		
	Fib	FFT	LU	Fib	FFT	LU
`aj -server`	0.52	1.34E-04	1.05E-02	100	100	100
`sbc -server`	1.14	1.36E-04	1.06E-02	219	102	101
`aabc -server`	1.15	1.36E-04	1.06E-02	219	102	101
`aj -client`	0.53	1.33E-04	1.06E-02	100	100	100
`sbc -client`	1.15	1.36E-04	1.05E-02	218	102	99
`aabc -client`	1.14	1.36E-04	1.06E-02	217	102	100



รูปที่ 4.8 แผนภูมิแสดงการเปรียบเทียบโอเวอร์เฮดของเวลาการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนตัวแนะนำหลังการคืนค่ากับชุดทดสอบ SciMark 2.0
(โอเวอร์เฮดน้อยประสิทธิภาพสูง)

ตารางที่ 4.8 ตารางแสดงผลประสิทธิภาพการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนของตัวแนะนำหลังการคืนค่ากับชุดทดสอบ DaCapo ชุดที่ 1

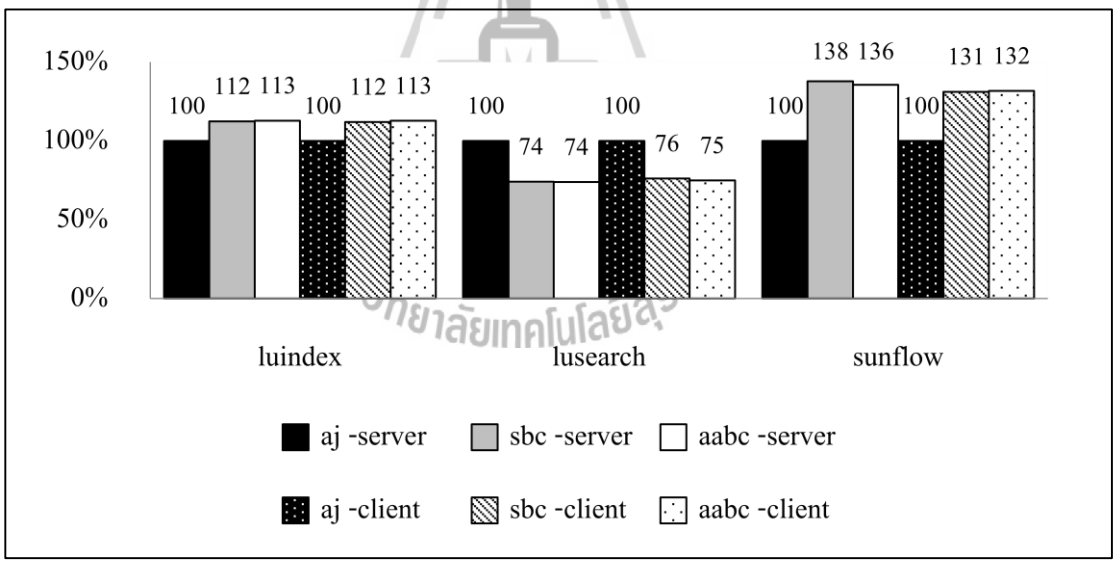
แพลตฟอร์มการทำงาน	เวลาเฉลี่ย (ms)			โอเวอร์เฮด (%)		
	antlr	avrora	h2	antlr	avrora	h2
`aj -server`	4,056	4,345	8,328	100	100	100
`sbc -server`	5,242	5,476	9,168	129	126	110
`aabc -server`	5,117	5,445	8,702	126	125	104
`aj -client`	4,150	4,371	8,299	100	100	100
`sbc -client`	5,179	5,468	9,111	125	125	110
`aabc -client`	5,179	5,496	8,638	125	126	104



รูปที่ 4.9 แผนภูมิแสดงการเปรียบเทียบโอเวอร์เฮดของเวลาการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนตัวแนะนำหลังการคืนค่ากับชุดทดสอบ DaCapo ชุดที่ 1 (โอเวอร์เฮดน้อยประสิทธิภาพสูง)

ตารางที่ 4.9 ตารางแสดงผลประสิทธิภาพการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนของตัวแนะนำหลังการคืนค่ากับชุดทดสอบ DaCapo ชุดที่ 2

แพลตฟอร์มการทำงาน	เวลาเฉลี่ย (ms)			โอเวอร์เฮด (%)		
	luindex	lusearch	sunflow	luindex	lusearch	sunflow
`aj -server`	2,681	5,070	5,546	100	100	100
`sbc -server`	3,013	3,760	7,644	112	74	138
`aabc -server`	3,026	3,744	7,519	113	74	136
`aj -client`	2,676	5,039	5,621	100	100	100
`sbc -client`	2,998	3,837	7,374	112	76	131
`aabc -client`	3,016	3,775	7,400	113	75	132



รูปที่ 4.10 แผนภูมิแสดงการเปรียบเทียบโอเวอร์เฮดของเวลาการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนตัวแนะนำหลังการคืนค่ากับชุดทดสอบ DaCapo ชุดที่ 2 (โอเวอร์เฮดน้อยประสิทธิภาพสูง)

สำหรับผลการทดสอบประสิทธิภาพในตัวแนะนำหลังการคืนค่าที่ได้อภิปรายผลการทดลองข้างต้นนี้พบว่าประสิทธิภาพการทำงานของ aabc ของตัวแนะนำหลังการคืนค่ามีประสิทธิภาพการทำงานไม่ต่างจาก sbc มากนักเป็นผลมาจากในส่วนนี้ได้พัฒนาตัวรวมไบต์โค้ดโดยตัดส่วนที่ไม่เกี่ยวข้องกับการทำงานตามหลักการของตัวแนะนำหลังการคืนค่าออกไปเพียงเล็กน้อยเท่านั้นส่งผลให้ประสิทธิภาพการทำงานของ aabc ที่พัฒนาขึ้นและ sbc ไม่ต่างจากการใช้งานกับ sbc มากนัก

3) ผลการทดสอบกับตัวแนะนำครอบ

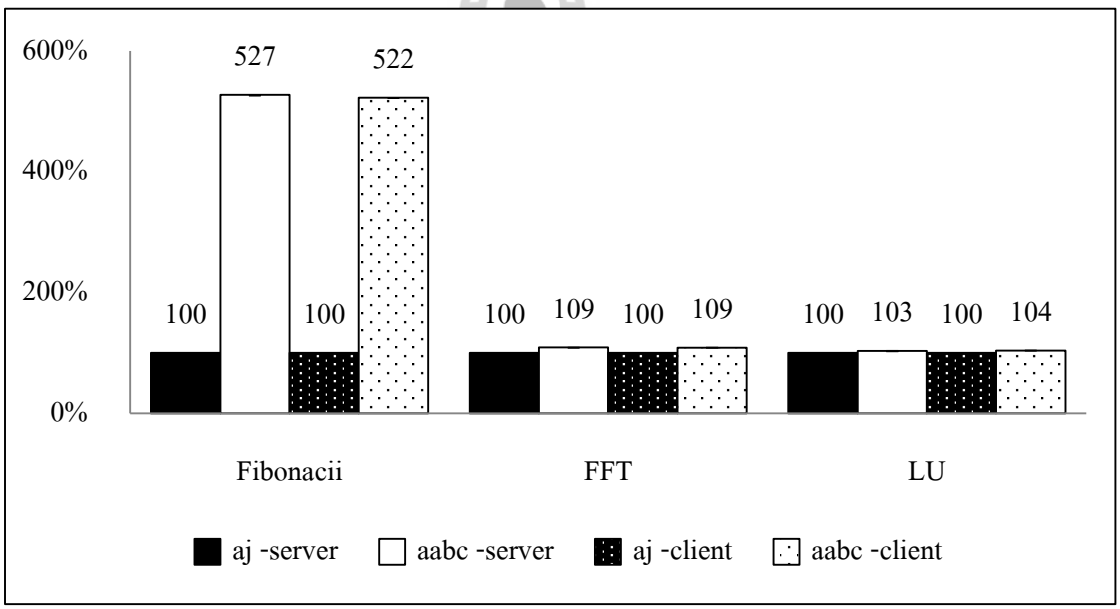
สำหรับผลการทดลองของตัวแนะนำครอบตามตารางที่ 4.10 และรูปที่ 4.11ในส่วนของตัวทดสอบ Fib พบว่าการทำงานของ aabc มีการทำงานที่ช้ากว่า aj อยู่ 427% ในส่วนของเซิร์ฟเวอร์ และ 422% ในส่วนของไคลเอนต์ สำหรับการทดสอบกับตัวทดสอบ FFT การทำงานของ aabc ช้ากว่า aj เพียง 9% ทั้งบนเซิร์ฟเวอร์และไคลเอนต์ ส่วนการทดสอบกับตัวทดสอบ LU พบว่า aabc ทำงานช้ากว่า aj อยู่ 3% ในส่วนของเซิร์ฟเวอร์และช้ากว่า aj อยู่ 4% ในส่วนของไคลเอนต์

ในการทดสอบกับตัวทดสอบ antlr ตัวทดสอบ avrora และตัวทดสอบ h2 ในตารางที่ 4.11 และรูปที่ 4.12 พบว่าการทดสอบกับตัวทดสอบ antlr การทำงานของ aabc มีการทำงานที่ช้ากว่า aj อยู่ 226% ในส่วนของเซิร์ฟเวอร์ และในส่วนของไคลเอนต์เป็น 229% สำหรับตัวทดสอบ avrora พบว่า aabc ช้ากว่า aj อยู่ 562% ในส่วนของเซิร์ฟเวอร์และ 567% ในส่วนของไคลเอนต์ และตัวทดสอบ h2 พบว่าการทำงานของ aabc มีการทำงานที่ช้ากว่า aj อยู่ 1,396% ในส่วนของเซิร์ฟเวอร์และไคลเอนต์

สำหรับในตารางที่ 4.12 และรูปที่ 4.13 ในส่วนการทดสอบกับตัวทดสอบ luindex พบว่าประสิทธิภาพการทำงานของ aabc มีการทำงานที่ช้ากว่า aj อยู่ 116% ในส่วนของเซิร์ฟเวอร์และ 119% ในส่วนของไคลเอนต์ สำหรับการทดสอบกับตัวทดสอบ lusearch พบว่าประสิทธิภาพการทำงานของ aabc ช้ากว่า aj ในส่วนของเซิร์ฟเวอร์ 184% และในส่วนของไคลเอนต์เป็น 165% ตัวทดสอบตัวสุดท้ายคือตัวทดสอบ sunflow พบว่าประสิทธิภาพการทำงานของ aabc ช้ากว่า aj อยู่ 1,251% ในส่วนของเซิร์ฟเวอร์และ 1,253% ในส่วนของไคลเอนต์

ตารางที่ 4.10 ตารางแสดงผลประสิทธิภาพการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนของตัวแนะนำครอบกับชุดทดสอบ SciMark 2.0

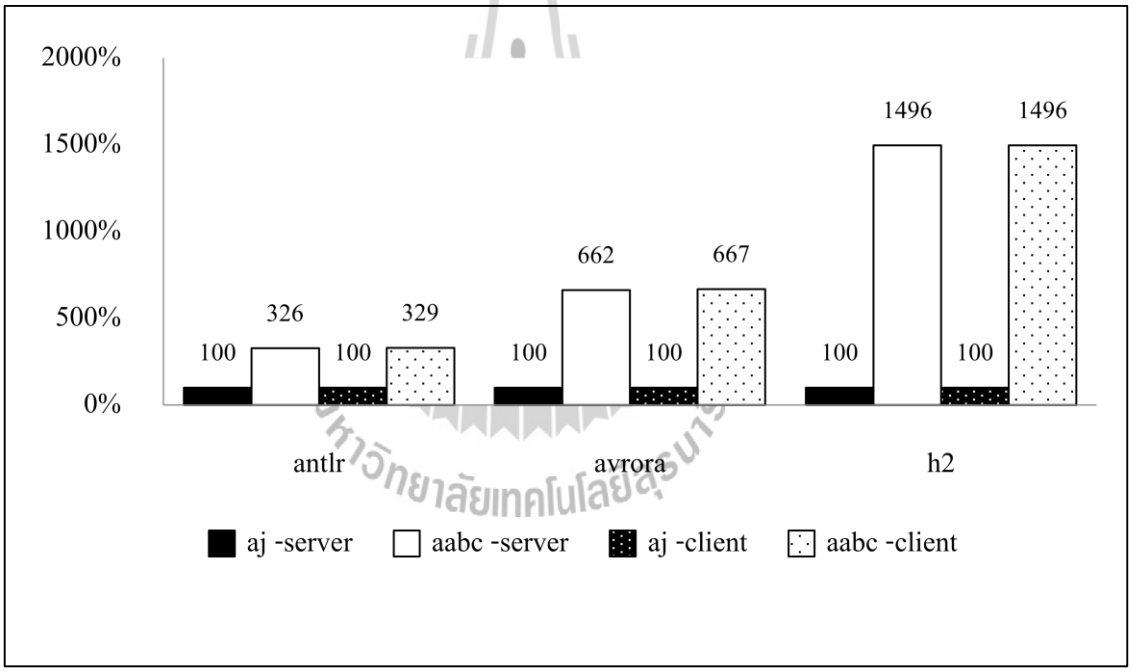
แพลตฟอร์มการทำงาน	เวลาเฉลี่ย (s)			โอเวอร์เฮด (%)		
	Fib	FFT	LU	Fib	FFT	LU
aj -server	0.82	1.44E-04	1.04E-02	100	100	100
aabc -server	4.34	1.57E-04	1.06E-02	527	109	103
aj -client	0.82	1.44E-04	1.06E-02	100	100	100
aabc -client	4.31	1.57E-04	1.04E-02	522	109	104



รูปที่ 4.11 แผนภูมิแสดงการเปรียบเทียบโอเวอร์เฮดของเวลาการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนตัวแนะนำครอบกับชุดทดสอบ SciMark 2.0
(โอเวอร์เฮดน้อยประสิทธิภาพสูง)

ตารางที่ 4.11 ตารางแสดงผลประสิทธิภาพการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนของตัวแนะนำครอบกับชุดทดสอบ DaCapo ชุดที่ 1

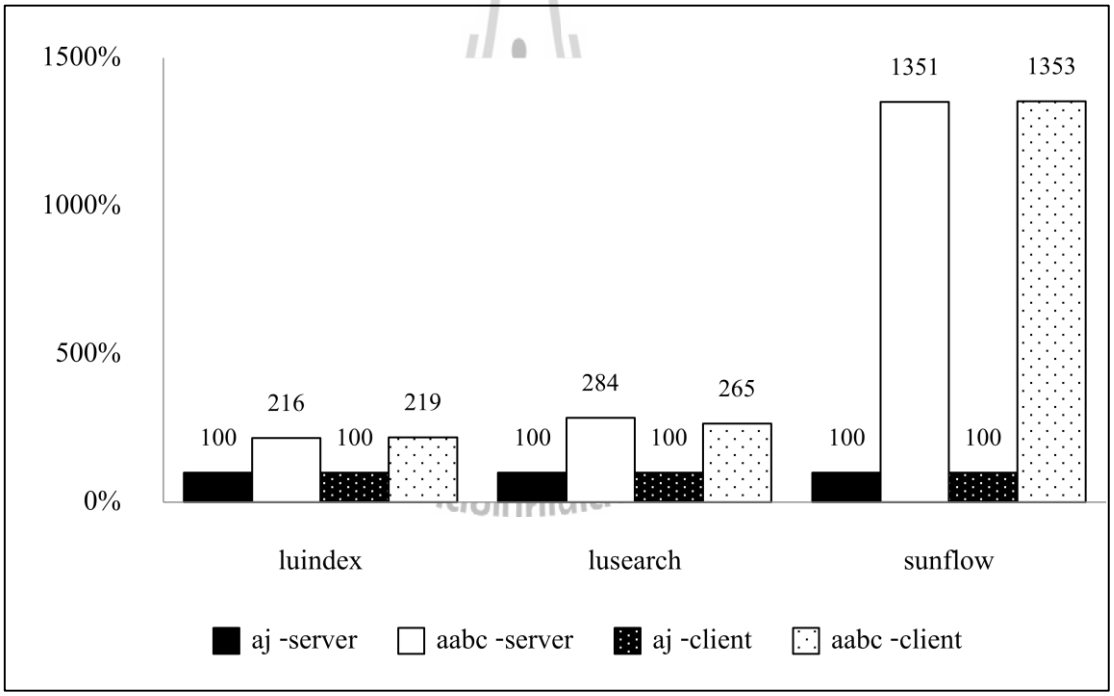
แพลตฟอร์มการทำงาน	เวลาเฉลี่ย (ms)			โอเวอร์เฮด (%)		
	antlr	avrora	h2	antlr	avrora	h2
`aj -server`	5,460	5,871	9,555	100	100	100
`aabc -server`	17,815	38,849	142,940	326	662	1,496
`aj -client`	5,414	5,863	9,602	100	100	100
`aabc -client`	17,815	39,130	143,656	329	667	1,496



รูปที่ 4.12 แผนภูมิแสดงการเปรียบเทียบโอเวอร์เฮดของเวลาการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนตัวแนะนำครอบกับชุดทดสอบ DaCapo ชุดที่ 1
(โอเวอร์เฮดน้อยประสิทธิภาพสูง)

ตารางที่ 4.12 ตารางแสดงผลประสิทธิภาพการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนของตัวแนะนำครอบกับชุดทดสอบ DaCapo ชุดที่ 2

แพลตฟอร์มการทำงาน	เวลาเฉลี่ย (ms)			โอเวอร์เฮด (%)		
	luindex	lusearch	sunflow	luindex	lusearch	sunflow
`aj -server`	3,003	6,340	6,856	100	100	100
`aabc -server`	6,499	18,034	92,633	216	284	1,351
`aj -client`	2,956	6,505	6,802	100	100	100
`aabc -client`	6,477	17,254	92,063	219	265	1,353



รูปที่ 4.13 แผนภูมิแสดงการเปรียบเทียบโอเวอร์เฮดของเวลาการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนตัวแนะนำครอบกับชุดทดสอบ DaCapo ชุดที่ 2
(โอเวอร์เฮดน้อยประสิทธิภาพสูง)

```
1) ct.proceed.invokeWithArguments(ct.arg);
2) ct.proceed.invokeWithArguments(ct.arg[0]);
```

รูปที่ 4.14 ตัวอย่างการใช้งานโปรซีด 2 รูปแบบ

สำหรับการทดสอบประสิทธิภาพในส่วนของตัวแนะนำครอบนี้ได้ทำการทดลองเทียบประสิทธิภาพกับ AspectJ เท่านั้นเพราะ sbc ไม่สนับสนุนการทำงานของตัวแนะนำครอบ สำหรับในส่วนของประสิทธิภาพการทำงานในส่วนตัวแนะนำครอบพบว่า aabc มีการทำงานที่ช้ากว่าการทำงานกับ aj ซึ่งสาเหตุได้อธิบายแล้วในส่วนการทดลองกับตัวแนะนำหลัง และอีกสาเหตุหนึ่งที่พบคือในการเรียกใช้โปรซีดดังตัวอย่างที่ 1 รูปที่ 4.14 ซึ่งจะเป็นการเรียกใช้โปรซีดที่รองรับทุกอาร์กิวเมนต์ซึ่งการเรียกใช้งานโปรซีดในลักษณะนี้จะทำให้การทำงานแต่ละครั้งในการเรียกใช้โปรซีดมีการทำงานที่ช้าเพราะตัวแปร ct.arg จะเป็นตัวแปรชนิดตัวแปรชุดทำให้ต้องเสียเวลาในการค้นหาอาร์กิวเมนต์ที่เข้ากับอาร์กิวเมนต์ที่ต้องการเรียกใช้งาน ส่งผลให้การทำงานของชุดทดสอบแต่ละตัวโดยรวมที่ทำการทดสอบมีการทำงานที่ช้าตามไปด้วย แต่ถ้าผู้ใช้งานจริงสามารถเขียนให้มีการทำงานที่เร็วขึ้นโดยการระบุอาร์กิวเมนต์ตามตัวอย่างที่ 2 ในรูปที่ 4.14 ซึ่งเป็นตัวอย่างสำหรับการส่งอาร์กิวเมนต์หนึ่งตัวโดยจะไม่เสียเวลาในการค้นหาคำสั่ง invokeWithArguments ที่มีจำนวนอาร์กิวเมนต์ที่ตรงกับจำนวนอาร์กิวเมนต์ที่รับเข้ามา ส่งผลให้การทำงานในการเรียกใช้งานโปรซีดลักษณะนี้มีการทำงานที่เร็วขึ้นและทำให้การทำงานของระบบโดยรวมเร็วขึ้น

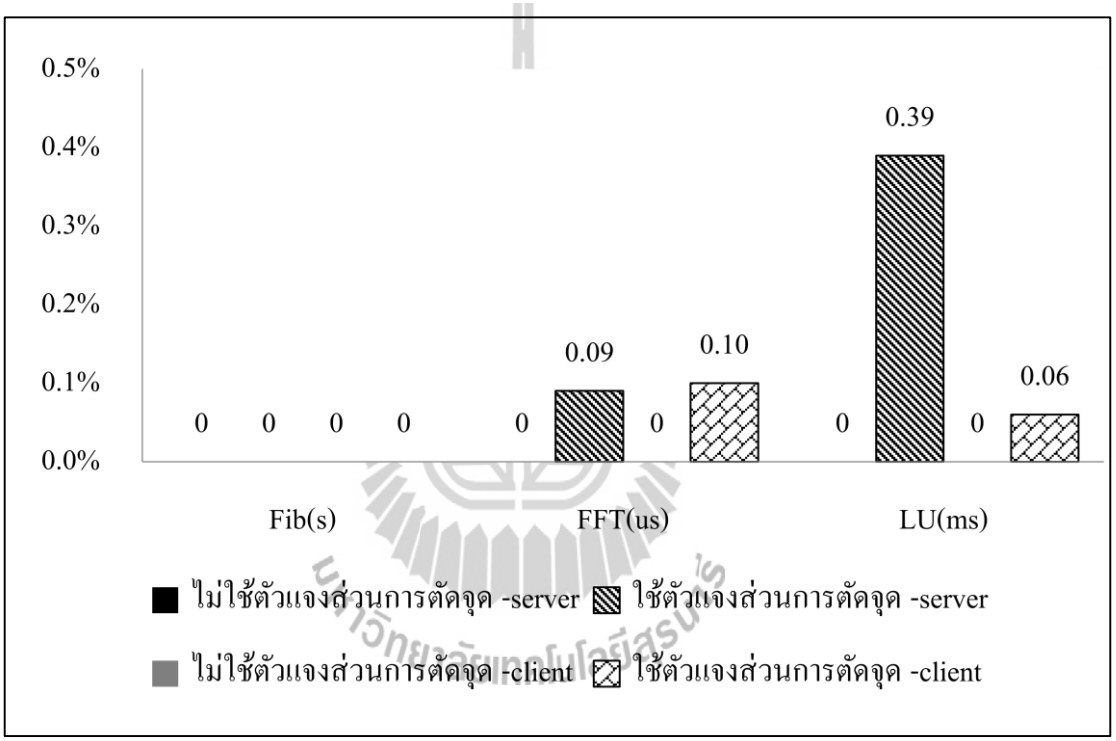
4.3.2 อภิปรายผลการทดสอบประสิทธิภาพการใช้งานตัวแจงส่วนการตัดจุด

จากการทดสอบประสิทธิภาพการใช้งานตัวแจงส่วนการตัดจุด ซึ่งขั้นตอนการทดสอบในส่วนนี้ได้อธิบายอยู่ในส่วนที่ 4.2.2 โดยในส่วนนี้สามารถวิเคราะห์ผลการทดลองได้ดังต่อไปนี้

จากตารางที่ 4.13 และรูปที่ 4.15 แสดงผลของการวัดประสิทธิภาพของการใช้งานตัวแจงส่วนการตัดจุดซึ่งจะเห็นว่าในส่วนของการวัดประสิทธิภาพกับตัวทดสอบ Fibonacci ซึ่งในตารางและรูปใช้สัญลักษณ์ Fib สำหรับการทำงานของระบบโดยรวมของการใช้ตัวแจงส่วนการตัดจุดเทียบกับระบบที่ไม่ใช้งานตัวแจงส่วนการตัดจุดพบว่าประสิทธิภาพของเวลาที่ได้ไม่แตกต่างกัน เนื่องจากการใช้งานตัวแจงส่วนการตัดจุดจะถูกเรียกใช้งานเพียงครั้งแรกเมื่อคำสั่ง invokedynamic ถูกเรียกใช้งาน หลังจากที่ได้วัตถุคอลไซต์สำหรับเรียกใช้งานกับคำสั่ง invokedynamic แล้วเมื่อคำสั่ง invokedynamic ถูกเรียกใช้งานอีกครั้งก็สามารถเรียกใช้งานวัตถุคอลไซต์ได้โดยตรงซึ่งไม่ต้องเสียเวลาในกระบวนการใช้งานตัวแจงส่วนการตัดจุดอีก ทั้งหมดนี้ส่งผลให้ประสิทธิภาพการทำงาน

ตารางที่ 4.13 ผลการทดสอบประสิทธิภาพตัวแจงส่วนการตัดจุด

แพลตฟอร์มการทำงาน	เวลาเฉลี่ย			โอเวอร์เฮด (%)		
	Fib(s)	FFT(us)	LU(ms)	Fib	FFT	LU
ไม่ใช้ตัวแจงส่วนการตัดจุด (บนเซิร์ฟเวอร์)	1.951	139.741	10.364	0	0	0
ใช้ตัวแจงส่วนการตัดจุด (บนเซิร์ฟเวอร์)	1.951	139.869	10.404	0	0.09	0.39
ไม่ใช้ตัวแจงส่วนการตัดจุด (บนไคลเอนต์)	1.951	139.961	10.404	0	0	0
ใช้ตัวแจงส่วนการตัดจุด (บนไคลเอนต์)	1.951	140.104	10.410	0	0.10	0.06



รูปที่ 4.15 แผนภูมิแสดงการเปรียบเทียบโอเวอร์เฮดของเวลาการใช้งานตัวแจงส่วนการตัดจุดเทียบกับไม่ใช้งานตัวแจงส่วนการตัดจุด (โอเวอร์เฮดน้อยประสิทธิภาพสูง)

ของระบบโดยรวมไม่ต่างจากตอนไม่เรียกใช้งานตัวแจงส่วนการตัดจุดโดยในกรณีที่มีการเรียกจุดเดิมซ้ำหลาย ๆ ครั้งดังตัวอย่างการทำงานกับตัวทดสอบ Fibonacci แต่สำหรับตัวทดสอบ FFT และตัวทดสอบ LU มีผลทางด้านเวลาที่เสียไปกับการใช้งานตัวแจงส่วนการตัดจุดเพียงเล็กน้อยเท่านั้นโดยจากตัวเลขในตางรางที่ 4.13 ตัวทดสอบ FFT ที่ใช้งานบนเซิร์ฟเวอร์ที่ทำงานร่วมกับตัวแจงส่วนการตัดจุดเกิดโอเวอร์เฮดขึ้นเพียง 0.09% เท่านั้น และกับไคลเอนต์เกิดขึ้นเพียง 0.10% สำหรับในส่วนของ LU ในส่วนของการทดสอบกับเซิร์ฟเวอร์มีโอเวอร์เฮดเกิดขึ้นเพียง 0.39% และไคลเอนต์เกิดขึ้น

เพียง 0.06% เท่านั้น โดยสามารถวิเคราะห์ได้ว่าจำนวนเปอร์เซ็นต์ที่เกิดขึ้นกับการทดลองแต่ละตัวนี้เป็นตัวเลขที่ยิ่งน้อยยิ่งดีเพราะเวลาที่เกิดขึ้นจากการใช้งานตัวแจงส่วนการตัดจุดเกิดขึ้นเพียงเล็กน้อยเท่านั้น ซึ่งหมายความว่าการใช้งานตัวแจงส่วนการตัดจุดสามารถนำมาใช้งานได้ดีและไม่มีผลต่อการทำงานของระบบมาก และที่สำคัญผู้ใช้งานในการเลือกจุดตัดก็สามารถทำงานได้ง่ายขึ้น

บทที่ 5

สรุปผลการวิจัยและข้อเสนอแนะ

งานวิจัยนี้เป็นการศึกษาการสร้างระบบเชิงลักษณะแบบพลวัตด้วยคำสั่ง invokedynamic และส่วนหลักของงานวิจัยนี้คือการพัฒนาตัวรวมไบต์โค้ดชื่อว่า Aspect-aware Bytecode Combinators หรือย่อว่า AABC ซึ่งได้ทำการพัฒนาให้รองรับกับตัวแนะนำตามหลักการของการโปรแกรมเชิงลักษณะทั้ง 4 ประเภทคือ ตัวแนะนำก่อน ตัวแนะนำหลัง ตัวแนะนำหลังการคืนค่า และตัวแนะนำครอบ และได้ทำการพัฒนาต่อในส่วนของตัวแจงส่วนการตัดจุดเพื่ออำนวยความสะดวกให้กับผู้ใช้งานสามารถเลือกจุดที่ต้องการตัดในระบบง่ายขึ้น โดยจากการพัฒนาดังกล่าวสามารถสรุปงานวิจัยได้ในส่วนที่ 5.1 และสำหรับส่วนที่ 5.2 เป็นข้อเสนอแนะ

5.1 สรุปผลการวิจัย

5.1.1 ประสิทธิภาพการทำงานของตัวรวมไบต์โค้ด AABC

จากการทดสอบประสิทธิภาพการใช้งานจริงของตัวรวมไบต์โค้ด AABC ของตัวแนะนำแต่ละประเภทกับชุดทดสอบมาตรฐานของ SciMark 2.0 และ DaCapo กับการทดสอบทั้งบนเซิร์ฟเวอร์และไคลเอนต์ซึ่งได้อภิปรายผลการทดลองไปแล้วในส่วนที่ 4.3.1 สามารถสรุปค่าเฉลี่ยด้านเวลาของโอเวอร์เฮดที่เกิดขึ้นทั้งหมดดังแสดงในตารางที่ 5.1 โดยสรุปได้ว่าประสิทธิภาพการใช้งานจริงในส่วนของตัวแนะก่อนมีการทำงานที่เร็วขึ้นกว่าการใช้งานตัวรวมไบต์โค้ดที่ภาษาจาวามีมาให้หรือ sbc โดยในส่วนการทดสอบบนเซิร์ฟเวอร์เร็วกว่าเฉลี่ย 95.11% และบนไคลเอนต์เร็วกว่าเฉลี่ย 91.56% เนื่องจากตัวรวมไบต์โค้ด AABC มีการทำงานที่เฉพาะเจาะจงกับการทำงานในความหมายของตัวแนะนำก่อนแต่ตัวรวมไบต์โค้ดที่ภาษาจาวามีให้ที่สามารถนำมาประยุกต์ใช้ยังมีบางส่วนของการทำงานที่ไปเกี่ยวข้องกับส่วนอื่นนอกเหนือจากการทำงานของตัวแนะนำก่อนจึงส่งผลให้ประสิทธิภาพการทำงานของตัวรวมไบต์โค้ด AABC เร็วกว่าตัวรวมไบต์โค้ดที่มีอยู่ในภาษาจาวา และประสิทธิภาพการทำงานของตัวรวมไบต์โค้ด AABC มีการทำงานที่ช้ากว่า AspectJ หรือ aj ในส่วนเซิร์ฟเวอร์เฉลี่ยเพียง 79.00% และบนไคลเอนต์เฉลี่ยเพียง 79.11% เท่านั้น

ตารางที่ 5.1 เวลาเฉลี่ยผลประสิทธิภาพการใช้งานตัวรวมไบต์โค้ด AABC
เทียบกับตัวรวมไบต์โค้ดของภาษาจาวาและ AspectJ

ผลสรุปเวลาเทียบกับ	sbc (%)	aj (%)
ตัวแนะนำก่อน (บนเซิร์ฟเวอร์)	95.11	-79.00
ตัวแนะนำก่อน (บนไคลเอนต์)	91.56	-79.11
ตัวแนะนำหลัง (บนเซิร์ฟเวอร์)	-	-80.11
ตัวแนะนำหลัง (บนไคลเอนต์)	-	-80.22
ตัวแนะนำหลังการคืนค่า (บนเซิร์ฟเวอร์)	1.22	-22.22
ตัวแนะนำหลังการคืนค่า (บนไคลเอนต์)	0.44	-21.56
ตัวแนะนำครอบ (บนเซิร์ฟเวอร์)	-	-463.78
ตัวแนะนำครอบ (บนไคลเอนต์)	-	-462.67

สำหรับผลการทดสอบประสิทธิภาพการทำงานของตัวรวมไบต์โค้ด AABC ในส่วนตัวแนะนำหลังพบว่าการทำงานของระบบเชิงลักษณะแบบพลวัตโดยใช้คำสั่ง invokedynamic มีการทำงานที่ช้ากว่าระบบที่ทำงานร่วมกับ AspectJ บนเซิร์ฟเวอร์เฉลี่ย 80.11% บนไคลเอนต์เฉลี่ย 80.22% เพราะระบบที่ทำงานร่วมกับ AspectJ ต้องเตรียมการสานตัวแนะนำแต่ละประเภทที่ถูกเขียนขึ้นในคลาส Aspect ตอนช่วงเวลาแปลโปรแกรม ทำให้การทำงานของระบบในช่วงเวลาโปรแกรมกำลังทำงานมีการทำงานที่เร็วกว่าระบบเชิงลักษณะแบบพลวัตโดยใช้คำสั่ง invokedynamic ที่มีการรวมไบต์โค้ดโดยใช้ตัวรวมไบต์โค้ด AABC ซึ่งจะทำการสานไบต์โค้ดของตัวแนะนำต่าง ๆ ในช่วงเวลาโปรแกรมกำลังทำงาน แต่เมื่อเทียบกับความสามารถแบบพลวัตที่สามารถทำการปรับปรุงแก้ไขระบบได้ขณะช่วงเวลาโปรแกรมกำลังทำงาน ถือว่าคุ้มค่ามากกว่าระบบที่มีการทำงานแบบคงที่ ที่ไม่สามารถทำการปรับเปลี่ยนแก้ไขระบบได้ในช่วงเวลาโปรแกรมกำลังทำงาน

สำหรับประสิทธิภาพของการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนของตัวแนะนำหลังการคืนค่ามีประสิทธิภาพการทำงานที่ใกล้เคียงกับตัวรวมไบต์โค้ดที่ภาษาจาวามีมาให้โดยมีความเร็วเฉลี่ยบนเซิร์ฟเวอร์ 1.22% และบนไคลเอนต์ 0.44% เนื่องจากตัวรวมไบต์โค้ด AABC ในส่วนตัวแนะนำหลังการคืนค่ามีการพัฒนาต่อยอดมาจากตัวรวมไบต์โค้ดที่ภาษาจาวามีมาให้ ซึ่งทำการตัดส่วนที่ไม่เกี่ยวข้องกับการทำงานตามหลักการของตัวแนะนำหลังการคืนค่าออกไปเพียงเล็กน้อยเท่านั้น ส่งผลให้ประสิทธิภาพการทำงานในส่วนนี้ไม่ต่างกันมาก และเมื่อเทียบประสิทธิภาพกับ AspectJ มีการทำงานที่ช้ากว่าในการทดสอบบนเซิร์ฟเวอร์เฉลี่ยเพียง 22.56% และบนไคลเอนต์เฉลี่ยเพียง 21.56% เท่านั้น

สำหรับประสิทธิภาพของตัวแนะนำครอบสามารถสรุปได้ว่า การทำงานร่วมกับตัวรวมไบต์โค้ด AABC มีการทำงานที่ช้ากว่าระบบที่ทำงานร่วมกับ AspectJ บนเซิร์ฟเวอร์ 463.78% และบนไคลเอนต์ 462.67% ซึ่งได้อธิบายสาเหตุในส่วนอภิปรายผลแล้วในส่วนที่ 4.3.1 ในหัวข้อที่ 4 พร้อมกับการเสนอแนะวิธีการใช้งานตัวรวมไบต์โค้ด AABC ในส่วนตัวแนะนำครอบให้สามารถใช้งานในส่วนนี้เร็วขึ้นได้หากมีการเรียกโปรซีดที่มีการระบุจำนวนอาร์กิวเมนต์ซึ่งจะทำให้มีการทำงานที่เร็วขึ้นเพราะจะทำให้ไม่ต้องเสียเวลาในช่วงเวลาโปรแกรมกำลังทำงานในการค้นหาตัวที่มีจำนวนอาร์กิวเมนต์ที่เข้ากับตัวที่ต้องการเรียกใช้งาน

5.1.2 ประสิทธิภาพการทำงานของตัวแจงส่วนการตัดจุด

สำหรับประสิทธิภาพการทำงานของตัวแจงส่วนการตัดจุดสำหรับใช้งานกับระบบเชิงลักษณะแบบพลวัตโดยใช้คำสั่ง invokedynamic สามารถสรุปได้ว่าการทำงานของตัวแจงส่วนการตัดจุดมีผลกระทบต่อประสิทธิภาพการทำงานของระบบน้อยมากเนื่องจากกระบวนการเรียกใช้งานตัวแจงส่วนการตัดจุดนี้จะถูกเรียกใช้งานเมื่อคำสั่ง invokedynamic ถูกเรียกใช้เพียงครั้งแรกและครั้งเดียวเท่านั้นตามหลักการทำงานของคำสั่ง invokedynamic ทำให้ประสิทธิภาพโดยรวมของระบบเสียไปกับกระบวนการทำงานของตัวแจงส่วนการตัดจุดน้อยมาก ดังนั้นตัวแจงส่วนการตัดจุดที่ได้ทำการพัฒนาขึ้นนี้จึงเป็นประโยชน์ต่อผู้พัฒนาซอฟต์แวร์เป็นอย่างมาก เพราะจะทำให้สามารถกำหนดเงื่อนไขของลักษณะจุดที่ต้องการตัดได้ง่ายขึ้นและประสิทธิภาพการทำงานของระบบโดยรวมก็ไม่เสียไปมากด้วย

5.2 ข้อเสนอแนะ

ในการพัฒนาตัวรวมไบต์โค้ด AABC สำหรับใช้งานร่วมกับระบบเชิงลักษณะแบบพลวัตโดยใช้คำสั่ง invokedynamic เป็นการพัฒนาเบื้องต้นเท่านั้น ซึ่งในอนาคตสามารถทำการพัฒนาต่อยอดเพื่อเพิ่มประสิทธิภาพการทำงานของตัวรวมไบต์โค้ด AABC ตามตัวแนะนำแต่ละประเภทให้ดีขึ้นได้ และสามารถทำการพัฒนาต่อเพื่อนำไปใช้งานจริงทางภาคอุตสาหกรรมได้

สำหรับการเปลี่ยนตัวแนะนำที่ถูกสานเข้าไปในไบต์โค้ดแล้วสามารถทำการเปลี่ยนแปลงได้โดยการลบวัตถุคอลไซต์ที่คำสั่ง invokedynamic เรียกใช้งาน ซึ่งเมื่อวัตถุคอลไซต์ถูกลบไปแล้วคำสั่ง invokedynamic ก็จะเข้าสู่กระบวนการสานไบต์โค้ดใหม่อีกครั้งโดยตัวแนะนำใหม่ที่ต้องการเปลี่ยนเข้าไปสามารถกระทำผ่านกระบวนการนี้ได้

และจากการพัฒนาสร้างตัวแจงส่วนการตัดจุดเพื่อใช้สำหรับเลือกจุดที่ต้องการตัดสำหรับผู้พัฒนาซอฟต์แวร์สามารถใช้งานได้ง่ายขึ้น ซึ่งมีการพัฒนาพีซีดีที่ใช้สำหรับกำหนดเงื่อนไขการตัดจุดเพียงบางส่วนที่มีอยู่ใน AspectJ เท่านั้นซึ่งเพียงพอต่อการใช้งานสำหรับทดสอบประสิทธิภาพใน

งานวิจัยนี้ แต่ในอนาคตสามารถทำการพัฒนาต่อให้สามารถใช้งานพีซีดีที่รองรับเงื่อนไขการตัดจุดที่ครอบคลุมมากขึ้นตามแบบอย่างที่มีอยู่ใน AspectJ

รายการอ้างอิง

Blackburn, M. S., Garner, R., Hoffmann, C., Khang, M. A., McKinley, S. K., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, Z. S., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., Dincklage, V. D., and Wiedermann, B. (2006). The DaCapo benchmarks: java benchmarking development and analysis. **SIGPLAN Not. 41**, 169-190. DOI=10.1145/1167515.1167488.

Bockisch, C.,Haupt, M., Mezini, M. and Ostermann, K. (2004). Virtual machine support for dynamic join points. **In Proceedings of the 3rd international conference on Aspect-oriented software development (AOSD '04)**. ACM, New York, NY, USA, 83-92.

Hirschfeld, R. (2001). AspectS - Aspect-Oriented Programming with Squeak. **In Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World (NODe '02)**, Mehmet Aksit, Mira Mezini, and Rainer Unland (Eds.). Springer-Verlag, London, UK, UK, 216-232.

Java™ Platform, Standard Edition 7. (2012). Available: http://docs.oracle.com/ javase/7/docs/api

Kaewkasi, C. (2010). Towards performance measurements for the Java Virtual Machine's invokedynamic. **In Proceedings on The 4th workshop on Virtual Machines and Intermediate Languages**, Nevada, USA.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, Lopes, V. C., Loingtier, J. and Irwin, J. (1997). Aspect-Oriented Programming. **In Proceedings of the European Conference on Object-Oriented Programming (ECOOP)**. Vol. LNCS 1241.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G. (2001). An Overview of AspectJ. **In Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)**, Jorgen Lindskov Knudsen (Ed.). Springer-Verlag, London, UK, UK, 327-353.

Loskutov, A. (2005). **Bytecode Outline** [On-line]. Available: http://andrei.gmxhome.de/bytecode/index.html/.

Nicoară, A., Alonso, G. (2005). Dynamic AOP with PROSE. **In Proceedings of 1st International Workshop on Adaptive and Self-Managing Enterprise Applications**.

Parr, T. (2010). Language implementation patterns. Available: http:// www.antlr.org/.

Pawlak, R., Seinturier, L., Duchien, L. and Florin, G. (2001). JAC:A flexible solution for aspect-oriented programming in Java. **In Proceedings of the 3rd International Conference on Reflection**. Kyoto Japan.

Ponge, J. and Mouël, Le F. (2012). JooFlux: Hijacking Java 7 InvokeDynamic To Support Live Code Modifications. **Research Report**, INRIA CITI Lab, INSA Lyon.

Popovici, A., Alonso, G. and Gross, T. (2003). Just-in-time aspects: efficient dynamic weaving for Java. **In Proceedings of the 2nd international conference on Aspect-oriented software development**, Boston, USA.

Popovici, A., Gross, T. and Alonso, G. (2002). Dynamic weaving for aspect-oriented programming. **In Proceedings of the 1st international conference on Aspect-oriented software development (AOSD '02)**. ACM, New York, NY, USA, 141-147. Available: http://doi.acm.org/10.1145/508386.508404

Pozo, R. and Miller, B. (2010). Java SciMark 2.0. Available: http://math.nist.gov/ SciMark2

Rose, J. (2009). Bytecodes meet combinators: invokedynamic on the JVM. **In Proceeding VMIL '09 Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages**, Orlando, FL.

Rose, J. (2008). JSR 292: Supporting dynamically typed languages on the Java platform. Available: http://jcp.org/en/jsr/detail?id=292

Sato, Y., Chiba, S. and Tatsubori, M. (2003). A selective, just-in-time aspect weaver. **In Proceedings of the 2nd international conference on Generative programming and component engineering (GPCE '03)**. Springer-Verlag New York, Inc., New York, NY, USA, 189-208.

Suvée, D., Vanderperren, W. and Jonckers, V. (2003). JAsCo: an aspect-oriented approach tailored for component based software development. **In AOSD'03: Proceedings of the 2nd international conference on Aspect-Oriented software development**, 21-29.

Tim, R. (1982). Object oriented programming. **SIGPLAN Not. 17**, 9 (September 1982), 51-57. DOI=10.1145/947955.947961

Villazon, A. Binder, W., Ansaloni, D., and Moret, P. (2009). Advanced runtime adaptation for Java. **SIGPLAN Not. 45**, 2 (October 2009), 85-94. Available: http://doi.acm.org/-10.1145/1837852.1621621.

ภาคผนวก ก

บทความทางวิชาการที่ได้รับการตีพิมพ์เผยแพร่ในระหว่างการศึกษา

รายชื่อบทความที่ได้รับการตีพิมพ์เผยแพร่ในระหว่างการศึกษา

Nopnipa, S. and Kaewkasi, C. (2013). **Aspect-aware bytecode combinators for a dynamic AOP system with invokedynamic**. In Proceedings of The 10 th International Joint Conference on Computer Sciences and Software Engineering (JCSSE 2013), pp. 258-263.

สันติ นภนิภา และ ชาญวิทย์ แก้วกสิ (2557). **การสร้างตัวแจงส่วนการตัดจุดสำหรับระบบเชิงลักษณะแบบพลวัต**. การประชุมวิชาการระดับประเทศด้านเทคโนโลยีสารสนเทศ (National Conference on Information Technology: NCIT) ครั้งที่ 6, หน้า 491-496.

Aspect-aware Bytecode Combinators for a Dynamic AOP System with Invokedynamic

Santi Nopnipa
School of Computer Engineering
Suranaree University of Technology
Nakhon Ratchasima, Thailand 30000
M5540075@g.sut.ac.th

Chanwit Kaewkasi
School of Computer Engineering
Suranaree University of Technology
Nakhon Ratchasima, Thailand 30000
chanwit@sut.ac.th

Abstract— **This paper presents a weaving implementation named Aspect-Aware Bytecode Combinators (AABC) to help optimization of the dynamic AOP using `invokedynamic`, which has been included in the JVM since Java 7. The main contribution of this paper is to demonstrate how a dynamic AOP system could be developed with `invokedynamic`. This paper discusses 4 common kinds of advice, which are supported by AABC. They are before, after, around and after-return advice. This paper argues that `filterArguments` and `filterReturnValue` natively supported by the JVM are not suitable to use as the weaving mechanism as they have incorrect semantic to implement AOP advice. It is also found that they have some performance overheads to serve as the weaving mechanism. Thus, this paper presents a new set of bytecode combinators, named AABC, which are carefully designed to match the semantic of the pointcut-advice model of AspectJ. The experimental results showed that performance of AABC in the before advice experiment is faster than the standard bytecode combinators. In addition, the paper demonstrated that the around advice can be successfully implemented using `MethodHandle` supported in Java 7.**

Keywords—dynamic AOP; runtime; weaving mechanism; invokedynamic;

I. INTRODUCTION

An ability to modify a system during runtime is a vital requirement for business applications that must run all the time. For this reason, there are several efforts to solve this problem. It has gained attention by several research works [15], [14], [2], [16] as one of the important software engineering properties for the middleware. One of these concepts is hot swapping. Hot swapping is supported by several dynamic languages such as Lisp, Erlang, Pike and Smalltalk. In Java, this ability has been found in the JVM Tool Interface [18]. It can replace system components during runtime without interruption or shutting the system down.

For decades, middleware systems have been implemented using object-oriented programming (OOP) because it has been proven to be useful for software maintenance and modification. However, there are some concerns that cut across the system and cannot be separated into modules using OOP. These problems can be solved by aspect-oriented programming (AOP) [10] with modular units that encapsulate crosscutting concerns. One of the most well-known programming languages in the AOP area undeniably is AspectJ [11], an AOP extension for the Java programming language. AOP copes with crosscutting concerns by allowing separation of similar sets of behavior out of a system's points, called *join points*, into a module. These join point locations are expressed by *pointcut designators*, and the module is called an *aspect*. Later, the system can be recomposed the aspect back to the system. This mechanism is called *weaving*. In AspectJ, the weaving process is usually done at the compile-time or the load-time. It is categorized as static AOP, whereas static AOP cannot weave during runtime. It is difficult for administrators to maintain system during runtime. Thus, how aspects are woven during runtime has been investigated. One of popular mechanism to build dynamic AOP is hot swapping which had been contributed by several research works such as PROSE [15, 14], Steamloom [2], Wool [19], JBossAOP [7], AspectWerkz [22], JAsCo [21], HotWave [23] etc. These systems are categorized as dynamic AOP systems. However, the performance of dynamic AOP is slow when compared to static AOP. This limitation has motivated several research works to speeding the weaving mechanism up.

Recently, the Java Development Kit (JDK) has introduced a new feature to support dynamically typed language by including the `invokedynamic` instruction into the JVM [17]. The mechanism of `invokedynamic` which will be discussed in Section II (*A*) can be implemented to build dynamic AOP to help optimization of the dynamic AOP.

There is a previous work that has investigated techniques to translate method invocations to `invokedynamic` [8]. Surprisingly, this notion to build dynamic AOP by `invokedynamic` has also been implemented independently in JooFlux [16] but some semantics of AOP advice are incorrect and incomplete which will be discussed in Section II (*C*). Hence, this paper aims to present new combinators, that are preferable to build dynamic AOP advice, named Aspect-Aware Bytecode Combinators (AABC).

This paper is organized as follows. An overview of AABC is discussed in Section II. Section III describes the experimental design and experimental results. Section IV

discusses related work in the area of dynamic AOP. The paper ends with conclusion and future work in Section V.

II. Aspect-Aware Bytecode Combinator

This Section describes Aspect-Aware Bytecode Combinator (AABC) and its related components. AABC is the main contribution of this paper to demonstrate how a dynamic AOP system could be developed with `invokedynamic`. The block diagram of AABC is shown in Fig. 1. In Fig. 1, the diagram consists of a set of JVM instructions as the input. They are .class files which will be 1) programs running in the dynamic AOP system will be transformed by the bytecode transformer to change invoke instructions from other kinds to `invokedynamic` and 2) advice codes to be woven into the programs. The transformation rules that use with the *Invocation Transformer* are adopted from [8]. The overall system that uses AABC must be comprised from the following components.

A. invokedynamic

Java SE 7 has been extended to support other languages beside the Java programming language. To this end, a new instruction, `invokedynamic`, has been added into the JVM [17]. The role of `invokedynamic` in a dynamic AOP system is that it allows altering semantics of method invocations during runtime. The *Invocation Transformer* component in Fig. 1 performs replacing method invocations, such as, `invokestatic`, `invokevirtual`, `invokeinterface` and `invokespecial` to `invokedynamic` as discussed in [8]. After transformation of method invocations is completed, *Invocation Transformer* will generate a bootstrap method, which is responsible for creating a `MethodHandle` for each invocation. After an `invokedynamic` instruction is dispatched for the first time, its associated bootstrap method will be called once. An instance of `MethodHandle` will be created and assigned to a `CallSite` object, which is returned from the bootstrap method to the JVM. A `MethodHandle` here can be combined from standard bytecode combinators, or AABC proposed in this paper to support the features of AOP. Obviously, in this category of dynamic AOP systems, the pointcut matching and the dynamic weaving process are designed to perform inside the bootstrap method.

Fig. 4 shows an example of a bootstrap method for weaving (a) a before advice, (b) an around advice, (c) an after-return advice and (d) an after advice to the `MethodHandle` "target" using AABC. The overview structure of advice declaration will be discussed next, in Section II (*B*).

B. Advice codes

AOP is a paradigm to manage crosscutting concerns to make a system has better modularity. In the pointcut-advice model, an aspect contains a set of pointcut expressions for quantifying join points, which can be method calls or constructor calls, for example. After selecting, these join points can be advised by new behaviors, so called advice codes. There are several kinds of advice in AOP. Although, the advice model supported by AABC is designed to follow AspectJ, they are slightly different due to the limitation of bytecode combinators in Java 7.

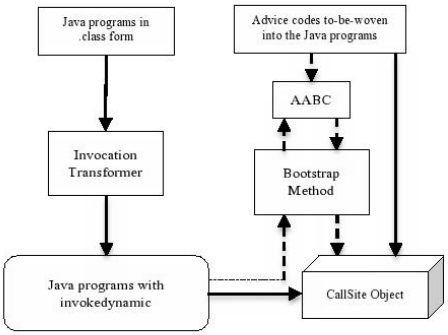


Fig. 1. The concept of a dynamic AOP system with AABC.

This paper discusses 4 common kinds of advice, which are supported by AABC. They are `before`, `after`, `around` and `after-return` advice. Firstly, a `before` advice is an advice code that will be woven to the pre-processing position of the join points. At this position, the advice code can observe values of parameters or performing some actions before that of the join point. Secondly, an `after` advice is an advice code that will be woven to the post-processing position of the join points. Similar to a `before` advice, it can observe values of parameters, and performing some actions after the action of the join point is taken. Thirdly, an `after-return` advice acts similarly to an `after` advice except that this kind of advice allows modification of the return value of the joint point. Finally, an `around` advice is an advice that can modify or change the semantic of the original join points. An `around` advice can perform the action of the original join point by invoking "`proceed`". In a dynamic AOP system supported by AABC the semantic of "`proceed`" is implemented using a `MethodHandle` as motivated by [1]. Advice codes to be woven into the programs are shown in Fig. 3. Their corresponding advice codes written in AspectJ are shown in Fig. 2.2.

Fig. 2.1 shows an example of a `pointcut` expression written in AspectJ. It is used by the Fibonacci benchmark in the experimental Section. The `pointcut` expression select calls to every method, `call(* *(..))`, within class Fib, `within(Fib)`, except calls inside the method `main`, `!withincode(public static void main())`. This `pointcut` expression has an equivalent semantic to the bootstrap method illustrated in Fig. 4. `MethodHandle` "`target`" inside the bootstrap method is the current join point, prepared by the Invocation Transformer and stored inside array "`mh`". Of course, this excludes method `main`, as it is not used to measure in the experiments. Next, a `Methodhandle` of an advice code will be looked up. In the example, (a) a `before` advice, (b) an `around` advice, (c) an `after-return` advice and (d) an `after` advice will be used. Both current join point `target` and the `MethodHandle` of method `before` advice will be passed into class `CallBeforeAdvice`, `AroundAdvice`, `CallAfterReturnAdvice` and `CallAfterAdvice` in sequence to weave there. Each class is responsible for creating AABC that returns a woven method handle, "`mha`".

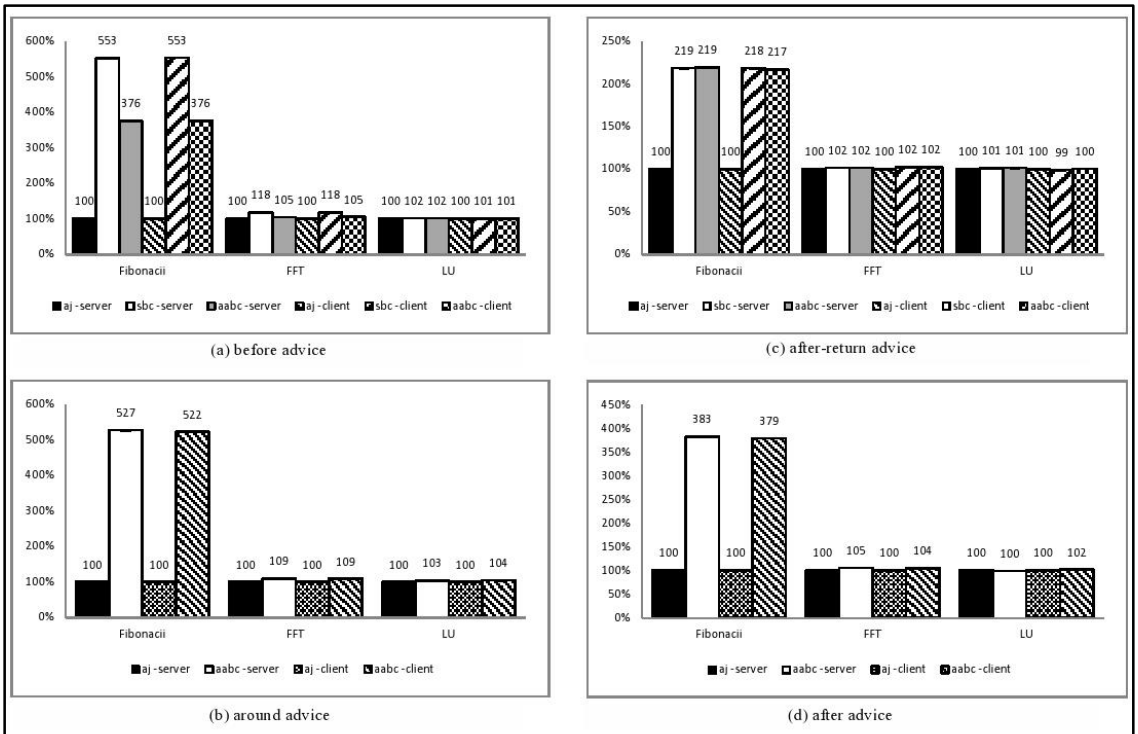


Fig. 5. Bar charts illustrating performance results for each kind of AOP advice (lower is better).

Table I. EXPERIMENTAL RESULT OF BEFORE ADVICE.

Exec. Plat.	Mean (s)			Overhead (%)		
	Fib	FFT	LU	Fib	FFT	LU
aj -server	0.53	1.34E-04	1.04E-02	100	100	100
sbc -serve	2.91	1.59E-04	1.06E-02	553	118	102
aabc -server	1.98	1.41E-04	1.06E-02	376	105	102
aj -client	0.53	1.34E-04	1.04E-02	100	100	100
sbc -client	2.91	1.58E-04	1.05E-02	553	118	101
aabc -client	1.98	1.41E-04	1.05E-02	376	105	101

Table II. EXPERIMENTAL RESULT OF AROUND ADVICE.

Exec. Plat.	Mean (s)			Overhead (%)		
	Fib	FFT	LU	Fib	FFT	LU
aj -server	0.82	1.44E-04	1.04E-02	100	100	100
aabc -server	4.34	1.57E-04	1.06E-02	553	118	101
aj -client	0.82	1.44E-04	1.06E-02	376	105	101
aabc -client	4.31	1.57E-04	1.04E-02	100	100	100

Table III. EXPERIMENTAL RESULT OF AFTER-RETURN ADVICE.

Exec. Plat.	Mean (s)			Overhead (%)		
	Fib	FFT	LU	Fib	FFT	LU
aj -server	0.52	1.34E-04	1.05E-02	100	100	100
sbc-server	1.14	1.36E-04	1.06E-02	219	102	101
aabc -server	1.15	1.36E-04	1.06E-02	219	102	101
aj -client	0.53	1.33E-04	1.06E-02	100	100	100
sbc -client	1.15	1.36E-04	1.05E-02	218	102	99
aabc -client	1.14	1.36E-04	1.06E-02	217	102	100

Table IV. EXPERIMENTAL RESULT OF AFTER ADVICE.

Exec. Plat.	Mean (s)			Overhead (%)		
	Fib	FFT	LU	Fib	FFT	LU
aj -server	0.52	1.34E-04	1.05E-02	100	100	100
aabc -server	1.98	1.41E-04	1.05E-02	383	105	100
aj -client	0.52	1.35E-04	1.07E-02	100	100	100
aabc -client	1.98	1.41E-04	1.09E-02	379	104	102

transformed from an original invocation instruction is formatted as below.

```
INVOKESTATIC
th/ac/sut/Scimark2/LU.new_copy([I)[I
```

is transformed to

```
INVOKEDYNAMIC
1:th/ac/sut/scimark2/LU:new_copy([I)[I
```

The format uses colons as delimiters. The name and type information is split as the array names in Fig. 4. Element `names[0]`, "1" in this example, is the index of the array of `Methodhandle` that point to the target method of this call site. Element `names[1]`, "th/ac/sut/scimark2/LU", is the owner class name. Element `names[2]` is the method name of this call site, in this example "`new_copy`".

AABC supports four common kinds of advice. The implementation is shown in Fig. 4. Its have the following components.

- Class `CallBeforeAdvice` in Fig. 4 (a) is used for weaving the before advice code in the Fig. 3 (a). It combines two `MethodHandles`, the `MethodHandle`

of method `beforeAdvice` with a `MethodHandle` of `target` method. It is also optimized to remove some addition overheads found in the standard bytecode combinators, mentioned above.

- Class `AroundAdvice` in Fig. 4 (b) is used for weaving the around advice code in the Fig. 3 (b). It combines two `MethodHandles` between the `MethodHandle` of method `aroundAdvice` with a `MethodHandle` of `target` method. It supports invocation of the original join point via an efficient use of `MethodHandle` as the `proceed` statement.
- Class `CallAfterReturnAdvice` in Fig. 4 (c) is used for weaving the after-return advice code in the Fig. 3 (c). It combines two `MethodHandles`, the `MethodHandle` of method `afterReturnAdvice` and the `MethodHandle` of target.
- And class `CallAfterAdvice` in Fig. 4 (d) is used for weaving the after advice code in the Fig. 3 (d). It combines two `MethodHandles` between the `MethodHandle` of method `afterroundAdvice` with a `MethodHandle` of `target` method.

AABC supports `Context` to supplement each advice to help pass the arguments of `target` to an advice code.

III. Experiment

To evaluate the performance of the newly developed set of AABC, the experiments have been conducted on an Acer ASPIRE laptop 4830G, CPU Intel Core i5-2410M at 2.3 GHz, RAM 8 GB running Windows 7 Ultimate 64-bit. The Java Development Kit 1.7.0_09 for 64 bit has been used as the JVM in these experiments. The *Invocation Transformer* is implemented using ASM 4.0 [4] as this version supports the `invokedynamic` instruction.

There are 4 experiments of `before`, `after`, `around` and `after-return` advice, according to the common kinds of supported advice in AspectJ. Benchmarks of each experiment running on AspectJ version 1.7.1 were used as the performance baseline. These AOP advice codes can be written in plain Java, as shown in Fig. 4. Method bodies of all advice codes are empty, except the around advice code that contains only a simple statement to call proceed.

Each experiment employed 4 benchmarks from SciMark 2.0 [20], i.e. Fibonacci, Fast Fouier Transformations (FFT), and LU matrix factorization (LU). In case of before advice and after-return advice, AspectJ (aj), standard bytecode combinators (sbc) and AABC (aabc) were compared. The scb implementation is derived from the implementation of JooFlux [16]. Other two experiments, namely `around` advice and `after` advice, does not contain results from the scb implementation because JooFlux simply does not support them. Each implementation was running on both server and client VM configurations, and `bb.util.Benchmark` was used as the measurement tool [3].

Fig. 5 shows the experimental results that consist of (a) before advice, (b) around advice, (c) after-return advice and (d) after advice. Table 1 show to 4 numerical Fig.5 from the experiments. Note that, CI is the 95% confidence interval values. Not surprisingly, the experimental results indicated that performance of the dynamic AOP system is slower than the static AOP implementation. However, the result also indicated that AABC is faster than the standard bytecode combinators in Java 7 because the current set of bytecode combinators were not designed to support AOP.

In the case of Fibonacci, it is much more slower than FFT and LU because the overhead occurred from recursive calls. It is obvious that there are still rooms of optimization for the `invokedynamic` subsystem in the JVM.

IV. Related work

This paper explores a feasibility to develop a dynamic AOP system using `invokedynamic` by introducing a new set of bytecode combinators. There are several research works that focused on exploring dynamic AOP Systems, such as PROSE [15, 14], Steamloom [2], AspectS [5], HotWave [23], Wool [19], JAsCo [21], DJAsCo [13], DandyJ [6], Groovy AOP [9] and JooFlux [16].

PROSE is one of the first known works to create a dynamic AOP, it explored a possibility of modification of JVM application codes at runtime.

Steamloom is the work seeking on how to hot weaving aspects and unweaving aspects out of the system. Weaving mechanism in Steamloom is done by dynamically subclassing and overriding target methods.

AspectS took another approach to build a dynamic AOP using meta-object protocol (MOP), which is a mechanism that shares the same concept of invokedynamic. `Callsite` dispatching in MOP is much slower than `invokedynamic` implementation in a virtual machine. The performance problems one of the main motivations that drove the implementation of `invokedynamic` in Java 7. Implementation of a dynamic AOP over MOP is done by intercepting method calls and weave an AOP advice into the layer of MOP.

Groovy AOP is a work to develop a dynamic AOP for the Groovy language. Beside MOP implementation, Groovy AOP supports Groovy JIT which allows recompilation of the woven code into bytecodes.

Recently, JooFlux has illustrated how to develop a basic dynamic AOP system with `invokedynamic` using the standard bytecode combinators of Java 7. JooFlux has a bytecode transformation agent to convert invocation instructions into invokedynamic before perform weaving. It supports two kinds of advice, namely before and after. However, the after advice of JooFlux actually implements after-return advice of the poincut-advice model of AspectJ. In addition, JooFlux lacks of around advice which is presented in this paper.

V. Conclusion and Future Work

This paper has presented a new set of bytecode combinators, AABC, for `invokedynamic`. Motivated by in other previous work [8], this paper also showed that it is possible to develop a dynamic AOP system by utilizing the `invokedynamic` instructions in Java 7. The work presented in this paper took several concepts from the class `MethodHandle` in package `java.lang.invoke`, standard bytecode combinators, `filterArguments` and `filterReturnValue`, to develop AABC. AABC has been designed to have the close semantics to the advice model in AspectJ.

The experimental results showed that performance of AABC in the before advice experiment is faster than the standard bytecode combinators, as illustrated in Fig. 5 (a). In addition, the paper demonstrated that the around advice can be successfully implemented using `MethodHandle` supported in Java 7.

```
pointcut fin() : call(* *(..))
     && within(Fib) &&
          ! withincode(public static void main());
```

Fig. 2.1. A simple pointcut declaration written in AspectJ.

(a). before advice

```
before() : fin() {

}
```

(b). around advice

```
Object around() : fib() {
    return proceed();
}
```

(c). after-return advice

```
after() returning (Object o): fib() {

}
```

(d). after advice

```
after() : fib() {

}
```

Fig. 2.2. Examples of AOP advice declaration written in AspectJ.

Weaving mechanism inside each classes will be discussed next in Section II (*C*). Finally the woven method handle is assigned to the call site. Fig. 4 lists all weavable advice, written in Java, supported by AABC.

C. Bytecode Combinator

From the study of JooFlux [16], the author suggested that an AOP advice can be developed using the concept of bytecode combinator from the class `MethodHandles` in package `java.lang.invoke`. To develop a mechanism to weave *before advice*, the combinator named `filterArguments` is used. In the similar way, a mechanism to weave *after advice* can be done using the combinator named `filterReturnValue`.

In addition, this technique is generalized to support arguments of any type and size by applying `asSpreader` and `asCollector`. Combinator asSpreader is responsible for making a `MethodHandle` to accept parameters in the form of a single `Object[]`. For example, `MethodHandle` with signature `(String,String):void` will be converted to a new one as `(Object[]):void`. In contrast, Combinator `asCollector` is responsible for converting a `MethodHandle` with the parameter type of `Object[]` into a specific set of types.

JooFlux uses `asSpreader` inconjuction with `filterArguments` to achieve the behavior of before advice. It also uses `asCollector` and `filterReturnValue` to implement `after` advice. It is important to note that `after` advice mentioned in JooFlux actually contains the semantic of `after-return` advice as it allows to alter the return value of the join point.

This paper argues that `filterArguments` and `filterReturnValue` natively supported by the JVM are not suitable to use as the weaving mechanism as they have incorrect semantic to implement AOP advice. It is also found that they have some performance overheads to serve as the weaving mechanism. Thus, this paper presents AABC, which are carefully designed to match the semantic of the pointcut-advice model of AspectJ.

Used by code listings shown in Fig. 4, the name and type information used by an `invokedynamic` instruction

(a). before advice

```
public static void before(Context ct){
  // write some action here
}
```

(b). around advice

```
public static Object around(Context ct,MethodHandle proceed)
        throws Throwable{
  // write some action here
  return proceed.invokeExact();
}
```

(c). after-return advice

```
public static Object afterReturn(Context ct, Object returnValue){
  // write some action here
  return returnValue;
}
```

(d). after advice

```
public static void after(Context ct, Object returnValue){
  // write some action here
}
```

Fig. 3. Example of AOP advices use with AABC.

(a). Bootstrap method for weaving a before advice.

```
public static CallSite bootstrap(MethodHandles.Lookup caller,
        String name, MethodType type)throws...{
   String[] names = name.split(":");
   MethodHandle target = mh[Integer.valueOf(names[0])];
   MethodHandle beforeAdvice = MethodHandles.lookup().findStatic(
        FFTBefore.class, "before",
        MethodType.methodType(void.class, Context.class));
   MethodHandle mha = CallBeforeAdvice.beforeAd(
        beforeAdvice,target.asType(target.type().generic()),
        names[1], names[2],type.parameterCount()).asType(type);
   return new ConstantCallSite(mha);
}
```

(b). Bootstrap method for weaving an around advice.

```
public static CallSite bootstrap(MethodHandles.Lookup caller,
        String name, MethodType type)throws...{
   String[] names = name.split(":");
   MethodHandle target = mh[Integer.valueOf(names[0])];
   MethodHandle aroundAdvice = MethodHandles.lookup().findStatic(
        FibonacAround.class, "around",
        MethodType.methodType(Object.class,
        Context.class, MethodHandle.class));
   MethodHandle mha = AroundAdvice.aroundAd(
        target.asType(target.type().generic()), aroundAdvice,
        type.parameterCount()).asType(type);
   return new ConstantCallSite(mha);
}
```

(c). Bootstrap method for weaving an after-return advice.

```
public static CallSite bootstrap(MethodHandles.Lookup caller,
        String name, MethodType type)throws...{
   String[] names = name.split(":");
   MethodHandle target = mh[Integer.valueOf(names[0])];
   MethodHandle afterReturnAdvice = MethodHandles.lookup().
        findStatic(AfterReturn.class, "after",
        MethodType.methodType(Object.class, Context.class,
        Object.class));
   MethodHandle asObject = target.asType(
        type.changeReturnType(Object.class));
   MethodHandle mha =
   CallAfterReturnAdvice.afterAd(afterReturnAdvice,
        asObject);
   return new ConstantCallSite(mha.asType(type));
}
```

(d). Bootstrap method for weaving an after advice.

```
public static CallSite bootstrap(MethodHandles.Lookup caller,
        String name, MethodType type)throws...{
   String[] names = name.split(":");
   MethodHandle target = mh[Integer.valueOf(names[0])];
   MethodHandle afterAdvice = MethodHandles.lookup().findStatic(
        FFTAfter.class, "after", MethodType.methodType(void.class,
        Context.class, Object.class));
   MethodHandle mha = CallAfterAdvice.afterAd(afterAdvice,
        target.asType(target.type().generic()),names[1], names[2],
        type.parameterCount()).asType(type);
   return new ConstantCallSite(mha);
}
```

Fig. 4. Bootstrap methods for weaving each kind of AOP advice used by AABC.

From the work presented in this paper, it is interesting to find that implementation of `filterArguments` is slower than `filterReturnValue`. This investigation of how `filterReturnValue` is implemented should be studied in details. Of course, another optimization that can boost the current weaving mechanism will be investigated.

Although the performance of AABC-based dynamic AOP is slower than a static AOP system such as AspectJ, the AABC approach still provides an important property, which allows the system's behaviour to be altered without shutting or restarting the system. This is an advantage provided by the use of the new `invokedynamic` instruction of java SE 7. AABC is the work-in-progress, so it is not that easy to use at the moment. Proper pointcut matching facilities are being implemented. Also, the *Invocation Transformer* part used in this paper is still limited. It is being improved for generality.

References

[1] M. Appeltauer, M. Haupt and R. Hirschfeld, "Layered method dispatch with invokedynamic", Proceedings of the the 24th European Conference on Object-Oriented Programming (Maribor, Slovenia, June 21 – 25, 2010).

[2] C. Bockisch, M. Haupt, M. Mezini and K. Ostermann, "Virtual Machine Support for Dynamic Join Points", In AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development, pages 83-92. ACM, 2004.

[3] B. Boyer, "Java benchmarking article website". http://www.ellipticgroup.com/html/benchmarkingArticle.html.

[4] E. Bruneton, R. Lenglet and T. Coupaye, "ASM: a code manipulation tool to implement adaptable systems", Adaptable and extensible component systems, November 2002, Grenoble, France.

[5] R. Hirschfeld, "AspectS - aspect-oriented programming with Squeak", In Revised Papers from NODe '02, pages 216–232, London, UK, 2003.

[6] M. Horie, S. Morita and S. Chiba, "Distributed dynamic weaving is a crosscutting concern", In SAC'11: Proceedings of the 2011 ACM Symposium on Applied Computing, Pages 1353-1360.

[7] JBoss. Open source middleware software. website. http://jboss.com/jbossaop/.

[8] C. Kaewkasi, "Towards performance measurements for the Java Virtual Machine's invokedynamic", In Proceedings on The 4th workshop on Virtual Machines and Intermediate Languages (Reno, Nevada, United States, October 17, 2010). ACM New York, NY, USA.

[9] C. Kaewkasi and J. R. Gurd, "Groovy AOP: a dynamic AOP system for a JVM-based language", Proceedings of the 2008 AOSD workshop on Software engineering properties of languages and aspect technologies. ACM New York, NY, USA.

[10] G. Kiczales, et al, "Aspect-Oriented Programming". Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Vol. LNSC 1241.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ", In ECOOP'01: Proceedings of the 15th European Conference on Object-Oriented Programming, LNCS2027, pages 237-353.

[12] T. Lindholm, F. Yellin, G. Bracha and A. Buckley, "The Java virtual machine specification", final release. Addition-Wesley Longman Publishing Co., Inc.

[13] L. D. B. Navarro, M. Sudholt, W. Vanderperren, B. D. Fraine, and D. Suvee, "Explicitly distributed AOP using AWED", In AOSD'06: Proceedings of the 5th international conference on Aspect-oriented software development, pages 51-62.

[14] A. Nicoara, and G. Alonso, "Dynamic AOP with PROSE", In Proceedings of ASMEA '05 in conjunction with CAISE '05, 2005.

[15] A. Nicoara, G. Alonso and T. Roscoe, "Controlled, Systematic, and Efficient Code Replacement for Running Java Programs", In Proceedings on the European professional society in Systems (Glassgow, Scotland, March 31 – April 4, 2008).

[16] J. Ponge and F. Le Mouël, "JooFlux: Hijacking Java 7 InvokeDynamic To Support Live Code Modifications", Research Report, INRIA CITI Lab, INSA Lyon, 2012.

[17] J. Rose, "Bytecodes meet combinators: invokedynamic on the JVM", In Proceedings of the Third Workshop on Virtual Machines and intermediate Languages (Orlando, Florida, October 25-29, 2009). VMIL '09. ACM, New York, NY, 1-11.

[18] Sun Microsystems, Inc. JVM Tool Interface (JVMTI) version 1.1. website. http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html.

[19] Y. Sato, S. Chiba and M. Tatsubori, "A selective, just-in-time aspect weaver", In GPCE'03: Proceedings of the 2nd international conference on Generative programming and component engineering, pages 189-208.

[20] SciMark2.0 website. http://math.nist.gov/scimark2/.

[21] D. Suvee, W. Vanderperren and V. Jonckers, "JAsCo: an Aspect-Oriented approach tailored for component based software development", In AOSD'03: Proceedings of the 2nd international conference on Aspect-Oriented software development, pages 21-29.

[22] A. Vasseur, "Dynamic AOP and Runtime Weaving for Java-How does AspectWerkz address it?", In Dynamic Aspects Workshop (DAW04). Lancaster, England.

[23] A. Villazon, B. Binder, D. Ansaloni and P. Moret, "Advanced runtime adaptation for Java", In GPCE'09: Proceedings of the eighth international conference on Generative programming and component engineering, pages 85-94.

การประชุมวิชาการระดับประเทศด้านเทคโนโลยีสารสนเทศ (National Conference on Information Technology: NCIT) ครั้งที่ 6

การสร้างตัวแจงส่วนการตัดจุดสำหรับระบบเชิงลักษณะแบบพลวัต

สันติ นภนิภา และ ชาญวิทย์ แก้วกสิ

สาขาวิชาวิศวกรรมคอมพิวเตอร์ สำนักวิชาวิศวกรรมศาสตร์ มหาวิทยาลัยเทคโนโลยีสุรนารี นครราชสีมา
Emails: M5540075@g.sut.ac.th, chanwit@sut.ac.th

บทคัดย่อ

การพัฒนาซอฟต์แวร์ที่สามารถทำการปรับปรุงแก้ไขระบบได้ขณะระบบกำลังทำงานอยู่จะเป็นประโยชน์ต่อการบำรุงรักษาและการปรับปรุงระบบในอนาคตเป็นอย่างมาก และจากกระบวนการทำงานของ Java 7 คำสั่ง `invokedynmamic` สามารถนำมาสร้างระบบเชิงลักษณะแบบพลวัตได้ซึ่งได้ทำการทดสอบและพัฒนาแล้วในการใช้ตัวสานไบต์โค้ดที่เรียกว่า AABC

บทความนี้เป็นการต่อยอดโดยการสร้างตัวแจงส่วนการตัดจุด (pointcut parser) ขึ้นมาเพื่ออำนวยความสะดวกในการเลือกจุดร่วม (join point) ในการดำเนินการตามหลักการโปรแกรมเชิงลักษณะ โดยจากการทดสอบประสิทธิภาพพบว่าการใช้งานตัวแจงส่วนการตัดจุดกระทบต่อการทำงานของระบบเพียงเล็กน้อยเนื่องจากถูกเรียกใช้งานเพียงครั้งเดียวตอนคำสั่ง `invokedynamic` ถูกเรียกใช้เป็นครั้งแรกในเวอร์ชวลแมชีนเท่านั้น

คำสำคัญ— pointcut; parser; invokedynamic; dynamic AOP

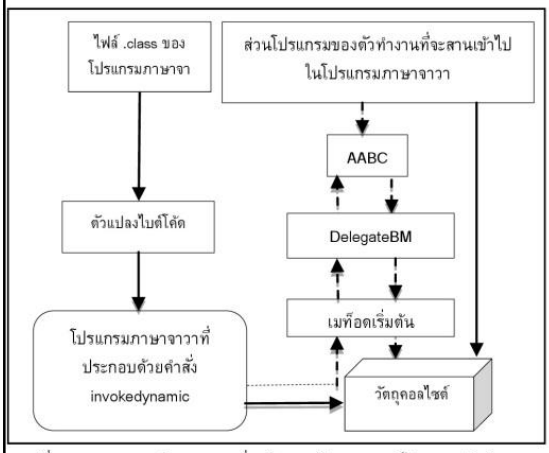
1. บทนำ

ความจำเป็นในการใช้งานซอฟต์แวร์ในปัจจุบันได้เข้ามามีบทบาทต่อการทำงานของมนุษย์เป็นอย่างมาก ซอฟต์แวร์ที่มีประสิทธิภาพและสามารถลดค่าใช้จ่ายในการทำงานขององค์กรได้ก็เป็นสิ่งสำคัญเช่นกัน ส่งผลให้การพัฒนาซอฟต์แวร์เกิดขึ้นเป็นจำนวนมาก สำหรับการเลือกใช้เครื่องมือสำหรับการพัฒนาและบำรุงรักษาซอฟต์แวร์ก็มีความจำเป็นโดยเฉพาะอย่างยิ่งระบบที่จำเป็นต้องทำงานตลอดเวลาและถ้าหากต้องทำการปรับปรุงระบบ ระบบก็จะต้องหยุดการทำงานซึ่งจะนำความยุ่งยากและค่าใช้จ่ายเป็นจำนวนมากแก่ธุรกิจ ดังนั้นจึงเกิดแนวคิดที่จะให้การทำการปรับปรุงแก้ไขระบบสามารถกระทำได้โดยที่ระบบกำลังทำงานอยู่ (Run time) โดยแนวคิดดังกล่าวนี้เรียกว่า การปรับปรุงแก้ไขแบบพลวัต [1], [2], [3], [4], [5], [6]

ภาษาจาวาได้รับความนิยมเพราะใช้หลักการการโปรแกรมเชิงวัตถุ (object-oriented programming) หรือ OOP ทำให้การสร้างซอฟต์แวร์สามารถสร้างโดยการแยกส่วนการทำงานของโปรแกรมออกเป็นโมดูลได้ ทำให้สามารถสร้างและดูแลซอฟต์แวร์สะดวกและรวดเร็วขึ้น อย่างไรก็ตามยังมีข้อจำกัดบางประการเกิดขึ้นคือ การทำงานของโปรแกรมที่สร้างขึ้นบางส่วนยังมีลักษณะการทำงานที่เหมือนกันแต่ไม่สามารถแยกออกมาเป็นโมดูลได้ด้วยหลักการการโปรแกรมเชิงวัตถุ แต่ปัญหาดังกล่าวสามารถจัดการได้ด้วยหลักการการโปรแกรมเชิงลักษณะ (aspect-oriented programming) หรือ AOP [7] ซึ่งจะมีการพิจารณาจากลักษณะการทำงานที่เหมือนกันและกระจายตัวอยู่ในโปรแกรมที่มีอยู่โดยทำการตัดขวาง (cross-cutting) ออกมาเป็นโมดูลผ่านทางเงื่อนไขของการกำหนดการตัดจุด (pointcut) สำหรับแนวคิดการโปรแกรมเชิงลักษณะนี้สามารถเพิ่มโมดูลให้กับระบบที่มีอยู่ได้ โดยภาษาที่ใช้สำหรับการสร้างซอฟต์แวร์ตามหลักการของการโปรแกรมเชิงลักษณะที่ได้รับความนิยมคือ AspectJ [8] ซึ่งถูกพัฒนาขึ้นใช้งานกับภาษาจาวา ทำให้ระบบที่ถูกเขียนขึ้นด้วยภาษาจาวาสามารถเพิ่มโมดูลให้กับระบบผ่าน AspectJ แต่การทำงานของ AspectJ จะทำตอนช่วงคอมไพล์ไทม์ (compile time) หรือช่วงโหลดไทม์ (load time) เท่านั้น ซึ่งเป็นลักษณะการทำงานเชิงสถิตย์ (static AOP) ที่ไม่สามารถทำการปรับปรุงแก้ไขแบบพลวัตได้ ดังนั้นจึงมีการพัฒนาต่อให้การโปรแกรมเชิงลักษณะสามารถทำงานแบบพลวัต (dynamic AOP) เกิดขึ้นแต่ประสิทธิภาพการทำงานของการโปรแกรมเชิงลักษณะแบบพลวัตพบว่าช้ากว่าการทำงานเชิงสถิตย์เป็นอย่างมากทำให้เกิดการวิจัยและพัฒนาเป็นจำนวนมากซึ่งจะกล่าวต่อไปในส่วนของงานวิจัยที่เกี่ยวข้อง

เมื่อเร็ว ๆ นี้ Java SE 7 [11] ได้ปรับปรุงการทำงานของจาวาเวอร์ชวลแมชีน (JVM) โดยการเพิ่มคำสั่ง `invokedynamic` [12] สำหรับรองรับการทำงานของภาษาไดนามิกส์ที่ทำงานบนจาวาเวอร์ชวลแมชีนให้สามารถทำงานได้ง่ายและมีประสิทธิภาพขึ้น โดยกระบวนการทำงานของ `invokedynamic` สามารถนำมาสร้างระบบการโปรแกรมเชิงลักษณะแบบพลวัต (dynamic AOP) ได้ [3], [4] ดังนั้นบทความนี้ได้ทำการพัฒนาต่อยอดมาจากการสร้าง Aspect-Aware Bytecode Combinator หรือ AABC [3] ซึ่งเป็นตัวรวมไบต์โค้ดที่ใช้สำหรับการสร้างระบบการโปรแกรมเชิงลักษณะแบบพลวัต โดยได้ทำการพัฒนาสร้างตัวแจงส่วนการตัดจุด (pointcut parser) เพื่อการใช้งานที่ง่ายขึ้นโดยเทียบไวยากรณ์กับการประกาศการตัดจุดใน AspectJ

สำหรับในส่วนที่ 2 ของบทความนี้จะอธิบายรายละเอียดของตัวแจงส่วนการตัดจุด จากนั้นในส่วนที่ 3 จะกล่าวถึงการทดลองและอภิปรายผล ส่วนที่ 4 จะกล่าวถึงงานวิจัยที่เกี่ยวข้อง และส่วนที่ 5 จะสรุปผลและกล่าวถึงแนวทางการพัฒนาต่อในอนาคต



รูปที่ 1. แผนภาพการทำงานของเครื่องมือการสร้างระบบการโปรแกรมเชิงลักษณะแบบพลวัตโดยใช้ invokedynamic

2. ตัวแจงส่วนการตัดจุด (pointcut parser)

เหตุผลที่ทำการพัฒนาตัวแจงส่วนการตัดจุดเพิ่มขึ้นมาเพื่ออำนวยความสะดวกให้กับผู้พัฒนาระบบซอฟต์แวร์ให้สามารถใช้งาน AABC ได้ง่ายขึ้น ซึ่งไวยากรณ์ตัวแจงส่วนการตัดจุดที่ได้พัฒนาขึ้นนี้ทำการเทียบความหมายและรูปแบบการเขียนใน AspectJ [8] โดยเครื่องมือที่เลือกใช้สำหรับการสร้างตัวแจงส่วนการตัดจุดก็คือ ANTLR [9] ในการทำความเข้าใจกระบวนการทำงานทั้งหมดของ AABC [3] ได้แสดงในรูปที่ 1 ซึ่งได้ทำการเพิ่ม `DelegateBM` หรือ Delegate Bootstrap Method เข้ามาเพื่อใช้สำหรับตรวจสอบไฟล์ Aspect โดยประกอบไปด้วยตัวทำงานของการโปรแกรมเชิงลักษณะ (AOP advice) แต่ละประเภทตามความต้องการของผู้พัฒนาซอฟต์แวร์สำหรับใช้งานกับระบบ โดยรูปแบบของการประกาศการใช้งานตัวแจงส่วนการตัดจุดได้ใช้ประโยชน์ผ่านทางการกำหนด annotation ดังได้แสดงตัวอย่างในรูปที่ 4 โดยเทียบความหมายและรูปแบบการเขียนจาก AspectJ ตามรูปที่ 3

สำหรับไวยากรณ์ที่ใช้สำหรับตรวจสอบค่าของ annotation ที่รับมาได้แสดงในรูปที่ 2 ซึ่งเป็นไวยากรณ์ที่ทำการพัฒนาขึ้นจาก pointcut designators หรือ พีซีดี ซึ่งเป็นเงื่อนไขในการเลือกจุดสำหรับการตัดจุด (pointcut) เพื่อง่ายต่อการใช้งานในการเลือกจุดที่ผู้ใช้งานต้องการสานตัวทำงาน (advice) เข้าไปโดยการพัฒนาสำหรับใช้งานร่วมกับตัวรวมไบต์โค้ด AABC ซึ่งจะทำการพัฒนาเพียงบางส่วนของเงื่อนไขทั้งหมดของพีซีดีที่ที่ประกอบกันขึ้นเป็นเงื่อนไขใน pointcut ของ AspectJ ได้แก่ `call`, `execution` และ `withincode` โดยตัวอย่างการใช้งานจริงดังต่อไปนี้

```
@Before("call(*  *(..)) && !withincode(public
static void main())")
```

เป็นตัวอย่างของการกำหนด annotation สำหรับการสานของตัวทำงานก่อนหน้า (Before advice) โดยที่พีซีดีแต่ละตัวจะประกอบกันขึ้นเป็นเงื่อนไขอยู่ในวงเล็บของ `@Before` โดยส่วนนี้คือส่วนของการตัดจุด (pointcut) ซึ่งเป็นเงื่อนไขที่จะนำไปแจงส่วนตามไวยากรณ์ในรูปที่ 2

```
unit
        :       expression EOF
        ;
expression
        :       '!' '(' expression ')'
        |       '(' expression ')'
        |       expression '&&' expression
        |       expression '||' expression
        |       pcd | '!' pcd
        ;
pcd
        :       call | execution | withincode
        ;
withincode
        :       'withincode' '(' access* returntype '
'method'
                ('arguments*'))'
        ;
call
        :       'call' '('returntype'
                'pkc'('checkargument'))'
        ;
execution
        :       'execution' '('returntype'
                'pkc'('checkargument'))'
        ;
access
        :       'public ' | 'static '
        ;
pkc
        :       pk* classname method | '*'
        ;
returntype
        :       '*'|Identifier2|Identifier
        ;
pk
        :       Identifier'.'|'*.'
        ;
classname
        :       Identifier2 '.'|'*.'
        ;
method
        :       Identifier|'*'
        ;
checkargument
        :       arguments | '..'|''
        ;
arguments
        :       argument','arguments
        |       argument
        ;
argument
        :       Identifier2 | Identifier
        ;
Identifier
        :       Smallletter Letter*
        ;
Identifier2
        :       Bigletter Letter*
        ;
```

รูปที่ 2. ไวยากรณ์ตัวแจงส่วนการตัดจุดสำหรับ AABC

เพื่อตรวจสอบว่าตำแหน่งใดบ้างที่ถูกเรียก โดยตามกระบวนการทำงานของคำสั่ง `invokedynamic` จะต้องเข้ามาทำงานในเมท็อดเริ่มต้น (Bootstrap method) ก่อนในครั้งแรกของการถูกเรียกใช้งาน โดยเมท็อดเริ่มต้นจะทำการเรียก `DelegateBM` สำหรับแจงส่วนโดยตรวจสอบว่าตรงตามเงื่อนไขดังกล่าวหรือไม่ จากตัวอย่างคือ `call(*  *(..))` หมายความว่าจะเลือกส่วนของการเรียกใช้งานทั้งหมดโดยที่ `*` ตัวแรกหมายถึงเลือกชนิดของค่าคืนกลับ (return type) ทุกชนิด `*` ตัวที่สองหมายถึงเลือกการเรียกใช้งานทุกคลาส ทุกแพ็คเกจ และ `(..)` หมายถึงเลือกพารามิเตอร์ที่เข้ามาทุกรูปแบบ และสำหรับ

```
!withincode(public static void main())
```

หมายความว่ายกเว้นในเมท็อด `main` ทั้งหมด จากนั้นถ้าตรงตามเงื่อนไขที่ได้ทำการแจงส่วนแล้วจะทำการสานตัวทำงานของการโปรแกรมเชิงลักษณะแต่ละประเภทที่ถูกเขียนใน Aspect เข้ากับเมท็อดเป้าหมายที่ถูกเรียก โดยในขั้นตอนการผูกเมท็อดเข้าด้วยกันจะทำการในรูปของ MethodHandle สำหรับขั้นตอนการสานทั้งหมดนี้จะถูก

```
public aspect Aspect{

1)  before() : call(* *(..)) &&
          !withincode(public static void main()){
     //write some action here
    }

2)  after() : call(* *(..)) &&
          !withincode(public static void main()){
     //write some action here
    }

3)  after() returning (Object o): call(* *(..)) &&
          !withincode(public static void main()){
     //write some action here
    }

4)  Object around() : call(* *(..)) &&
          !withincode(public static void main()){
     //write some action here
     return proceed();
    }
}
```

รูปที่ 3. ตัวอย่างการสร้าง Aspect ของ AspectJ

```
public class Aspect{

1) @Before("call(* *(..)) &&
          !withincode(public static void main())")
   public static void before(Context ct){
          //write some action here
   }

2) @After("call(* *(..)) &&
          !withincode(public static void main())")
   public static void after(Object returnTarget,
                            Context ct){
          //write some action here
   }

3) @AfterReturn("call(* *(..)) &&
          !withincode(public static void main())")
   public static Object afterReturn(
                            Object returnTarget){
          //write some action here
          return returnTarget;
   }

4) @Around("call(* *(..)) &&
          !withincode(public static void main())")
   public static Object around(Context ct)
                            throws Throwable{
          //write some action here
          return ct.proceed.invokeWithArguments(
                                    ct.arg);
   }
}
```

รูปที่ 4. ตัวอย่างการสร้าง Aspect สำหรับใช้งานร่วมกับ AABC

กระทำในช่วงโปรแกรมกำลังทำงานเท่านั้น (Runtime) ซึ่งหมายความว่าผู้สร้างซอฟต์แวร์หรือผู้ดูแลระบบสามารถแทรกการทำงานหรือทำการปรับปรุงระบบใด ๆ ก็ได้ในขณะที่ระบบกำลังทำงานอยู่

3. การทดลองและอภิปรายผล

การทดลองจะทำการทดลองบนเครื่องคอมพิวเตอร์แบบพกพา Acer ASPIRE laptop 4830G หน่วยประมวลผล Intel Core i5-2410M 2.3 GHz หน่วยความจำหลัก 8 GB ระบบปฏิบัติการ Windows 7 Ultimate ระบบประมวลผลแบบ 64-bit ใช้ Java Development Kit เวอร์ชัน 1.7.0_09 สำหรับระบบประมวลผลแบบ 64 bit ใช้ ASM 4.0 สำหรับจัดการไบต์โค้ด ใช้ AspectJ เวอร์ชัน 1.7.1 สำหรับใช้เป็นเกณฑ์วัดผลประสิทธิภาพการทำงาน โดยจะทำการทดลองชุดการทดสอบของ SciMark 2.0 [10] และใช้ bb.util.Benchmark สำหรับจับเวลาในการทดลอง โดยการทดลองจะทำการทดสอบประสิทธิภาพโดยแบ่งการวัดผลของประสิทธิภาพออกเป็น 2 ประเภทหลักดังนี้

การทดลองที่ 1 จะทำการวัดประสิทธิภาพการใช้งาน AABC ที่ทำงานผ่านตัวแจงส่วนการตัดจุดซึ่งในการทดลองได้ทำการกำหนดเงื่อนไขการแปลงตามเงื่อนไขนี้

```
call(* *(..)) &&
!withincode(public static void main())
```

ซึ่งหมายความว่าทำการเลือกการเรียกทั้งหมดยกเว้นการเรียกในเมท็อด `main` ซึ่งได้อธิบายรายละเอียดแล้วในส่วนที่ 2 และในตัวอย่างการใช้งานของการทดสอบจริงที่ถูกเขียนอยู่ใน Aspect แสดงในรูปที่ 4 ซึ่งการทดลองย่อยได้แบ่งการทดสอบออกเป็น 4 ประเภทตามตัวทำงานดังนี้ ตัวทำงานก่อนหน้า (Before advice), ตัวทำงานตามหลัง (After advice), ตัวทำงานตามหลังแบบเปลี่ยนค่าคืนกลับ (After-return advice) และตัวทำงานแบบครอบ (Around advice) โดยในส่วนของตัวทำงานก่อนหน้าและตัวทำงานตามหลังแบบเปลี่ยนค่าคืนกลับได้ทำการทดสอบเทียบกับ AspectJ (aj), standard bytecode combinators (sbc) และ งานที่ได้พัฒนาขึ้นชื่อว่า Aspect-aware bytecode combinators (aabc) สำหรับในส่วนของตัวทำงานตามหลังและตัวทำงานแบบครอบจะทำการทดสอบเพียง aj และ aabc เท่านั้น โดยการทดลองทำการวัดผลบน server และ client ของจาวาเวอร์ชวลแมชีน

ส่วนการทดลองที่สองจะทำการทดสอบประสิทธิภาพการใช้งานตัวแจงส่วนการตัดจุด (pointcut parser) โดยทำการควบคุมตัวแปรของการทดสอบในส่วนนี้จะทำการตั้งค่าตัวแปลงการเรียกแต่ละประเภทไปเป็น `invokedynamic` โดยจะไม่ให้ทำการแปลงการเรียกภายในเมท็อด `main` ทั้งหมด จากนั้นจะทำการทดสอบโดยตั้งค่าเงื่อนไขการตัดจุดตามเงื่อนไขที่ได้ยกตัวอย่างไว้ข้างต้น และทำการเทียบผลกับการทดสอบโดยไม่ใช้ตัวแจงส่วนการตัดจุดโดยทั้งสองจะทำการเลือกสานเฉพาะตัวทำงานก่อนหน้าเพียงตัวเดียวเท่านั้นเพราะในส่วนนี้ต้องการวัดผลการใช้งานของตัวแจงส่วนการตัดจุด

จากผลการทดลองในส่วนแรกได้แสดงผลในรูปที่ 5(a) ถึงรูปที่ 5(d) และตารางผลการทดลองที่ 1 และ 2 พบว่าประสิทธิภาพโดยรวมไม่ต่างจากผลการทดลองที่ได้เสนอไปแล้วในบทความก่อนหน้า [3] ในการทดสอบส่วนของตัวแนะนำแบบครอบพบว่าการกำหนดเรียกใช้งานโปรซีด (proceed) ในรูปแบบที่รองรับอาร์กิวเมนต์ได้ทุกจำนวนดังตัวอย่างการเรียกใช้งานดังนี้

```
ct.proceed.invokeWithArguments(ct.arg);
```

ส่งผลให้ประสิทธิภาพการทำงานช้าลงเนื่องจากจะต้องมีการผูกไบต์โค้ดให้ตรงตามจำนวนอาร์กิวเมนต์ที่รับมา แต่ถ้ามีการใช้โปรซีดโดยระบุอาร์กิวเมนต์ดังตัวอย่างนี้

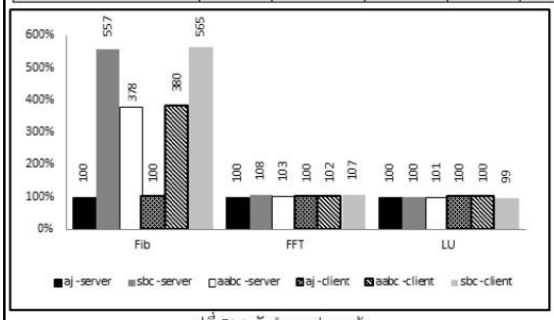
```
ct.proceed.invokeWithArguments(ct.arg[0]);
```

ตาราง 1. ผลการทดลองตัวทำงานก่อนหน้าและตัวทำงานตามหลังแบบเปลี่ยนค่าคืนกลับ

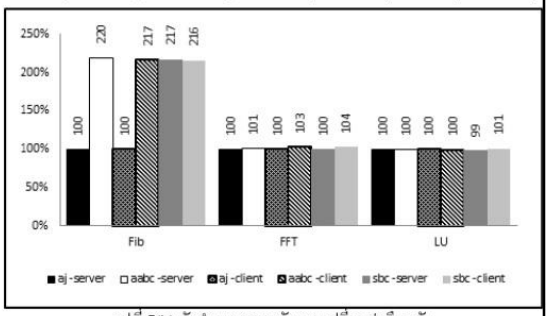
แพลตฟอร์ม การทำงาน	ผลการทดลองตัวทำงานก่อนหน้า						ผลการทดลองตัวทำงานตามหลังแบบเปลี่ยนค่าคืนกลับ					
	เวลาเฉลี่ย			โอเวอร์เฮด (%)			เวลาเฉลี่ย			โอเวอร์เฮด (%)		
	Fib (s)	FFT (us)	LU (ms)	Fib	FFT	LU	Fib (s)	FFT (us)	LU (ms)	Fib	FFT	LU
aj -server	0.519	133.589	10.231	100	100	100	0.518	134.184	10.269	100	100	100
sbc -server	2.895	143.723	10.280	557	108	100	1.127	134.264	10.166	217	100	99
aabc -server	1.966	138.195	10.292	378	103	101	1.139	135.844	10.251	220	101	100
aj -client	0.519	135.418	10.284	100	100	100	0.520	132.591	10.171	100	100	100
sbc -client	2.935	144.458	10.150	565	107	99	1.124	137.256	10.269	216	104	101
aabc -client	1.973	137.633	10.254	380	102	100	1.127	136.546	10.149	217	103	100

ตาราง 2. ผลการทดลองตัวทำงานหลังและตัวทำงานแบบครอบ

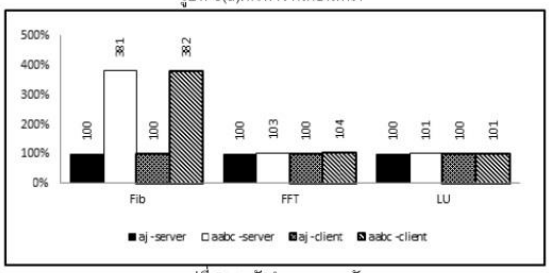
แพลตฟอร์ม การทำงาน	ผลการทดลองตัวทำงานตามหลัง						ผลการทดลองตัวทำงานแบบครอบ					
	เวลาเฉลี่ย			โอเวอร์เฮด (%)			เวลาเฉลี่ย			โอเวอร์เฮด (%)		
	Fib (s)	FFT (us)	LU (ms)	Fib	FFT	LU	Fib (s)	FFT (us)	LU (ms)	Fib	FFT	LU
aj -server	0.519	134.231	10.183	100	100	100	0.821	133.032	10.266	100	100	100
aabc -server	1.976	138.163	10.285	381	103	101	1.953	136.824	10.253	238	103	100
aj -client	0.517	132.624	10.249	100	100	100	0.818	134.279	10.154	100	100	100
aabc -client	1.977	137.966	10.37	382	104	101	1.951	139.474	10.162	238	104	100



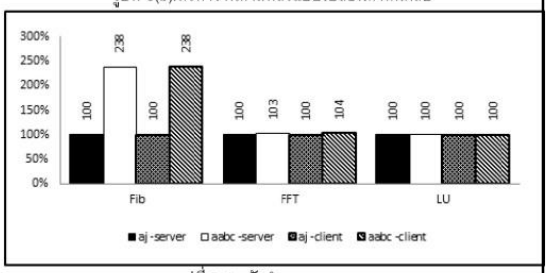
รูปที่ 5(a).ตัวทำงานก่อนหน้า



รูปที่ 5(b).ตัวทำงานตามหลังแบบเปลี่ยนค่าคืนกลับ



รูปที่ 5(c). ตัวทำงานตามหลัง



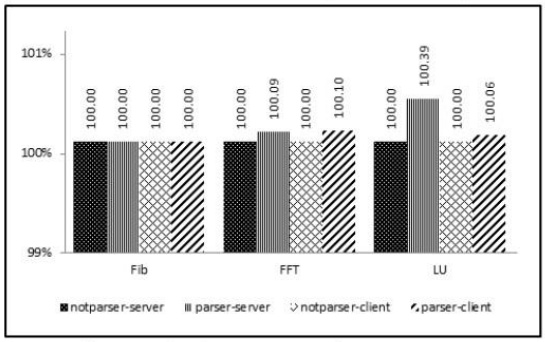
รูปที่ 5(d). ตัวทำงานแบบครอบ

จะส่งผลให้ประสิทธิภาพการทำงานที่เร็วกว่าเพราะไม่ต้องเข้าไปทำการผูกไบต์โค้ดที่เข้ากับจำนวนอาร์กิวเมนต์ที่รับเข้ามา โดยการเรียกใช้งานดังกล่าวส่งผลทำให้ประสิทธิภาพการทำงานของตัวทำงานแบบครอบที่ได้แสดงในรูปที่ 5(d) ในส่วนของการทดสอบ Fibonacii ดีขึ้นเมื่อเทียบกับผลของการทดลองของตัวทำงานแบบครอบใน [3] ที่มีการเรียกแบบรองรับทุกอาร์กิวเมนต์

สำหรับการทดลองในส่วนที่ 2 ได้ทำการทดสอบวัดประสิทธิภาพการทำงานของตัวแจงส่วนการตัดจุดได้แสดงในรูปที่ 6 และตารางที่ 3 พบว่าประสิทธิภาพการทำงานแทบจะไม่เปลี่ยนไปเท่าไรหรือเปลี่ยนไปเพียงเล็กน้อยเท่านั้น สาเหตุที่ทำให้ประสิทธิภาพการทำงานแทบไม่เปลี่ยนไปเพราะว่ากระบวนการทำงานของตัวแจงส่วนการตัดจุดจะทำงานเมื่อมีการเรียกใช้คำสั่ง `invokedynamic` เป็น

ตารางที่ 3 ผลประสิทธิภาพการใช้ตัวแจงส่วนแบ่งจุด

แพลตฟอร์ม / การทำงาน	เวลาเฉลี่ย			โอเวอร์เฮด (%)		
	Fib (s)	FFT (us)	LU (ms)	Fib	FFT	LU
notparser-server	1.951	139.741	10.364	100.00	100.00	100.00
parser-server	1.951	139.869	10.404	100.00	100.09	100.39
notparser-client	1.951	139.961	10.404	100.00	100.00	100.00
parser-client	1.951	140.104	10.410	100.00	100.10	100.06



รูปที่ 6. กราฟเปรียบเทียบประสิทธิภาพการใช้ตัวแจงส่วนการตัดจุด

ครั้งแรกโดยหลักการทำงานของคำสั่ง `invokedynamic` เมื่อถูกเรียกเป็นครั้งแรกจะเข้ามาทำงานที่เมท็อดเริ่มต้น (Bootstrap method) ก่อนซึ่งในได้เมท็อดเริ่มต้นได้ทำการออกแบบให้ทำการเรียกใช้ `DelegateBM` สำหรับเรียกใช้งานในการแจงส่วนการตัดจุดแล้วถ้าตำแหน่งที่คำสั่ง `invokedynamic` ถูกเรียกขึ้นสอดคล้องกับเงื่อนไขของการตัดจุดที่ได้กำหนดไว้ก็จะทำการผูกไบต์โค้ดโดยใช้ AABC ในการผูกไบต์โค้ดตามที่ได้สร้างไว้ในไฟล์ Aspect จากนั้นจะทำการคืนค่าเป็นวัตถุคอลไซต์ออกมาให้เมท็อดเริ่มต้นแล้วเมท็อดเริ่มต้นก็ส่งต่อไปให้คำสั่ง `invokedynamic` เรียกใช้งานต่อไปแล้วเมื่อมีการเรียกคำสั่ง `invokedynamic` ซ้ำเป็นครั้งที่สองก็สามารถเรียกใช้งานวัตถุคอลไซต์ได้โดยตรงซึ่งจะไม่เข้ามาทำในเมท็อดเริ่มต้นอีก ดังนั้นกระบวนการที่ได้กล่าวมาทั้งหมดนี้ทำให้การเรียกใช้งานของตัวแจงส่วนการตัดจุดถูกเรียกใช้งานเพียงครั้งแรกและครั้งเดียวเท่านั้น ส่งผลให้ประสิทธิภาพการทำงานของชุดทดสอบเมื่อทดสอบแล้วปรากฏว่าเปลี่ยนแปลงเพียงเล็กน้อยเท่านั้น

4. งานวิจัยที่เกี่ยวข้อง

ในปี 2001 ได้มีงานวิจัยซึ่งนำเสนอแพลตฟอร์มที่มีชื่อว่า AspectS [13] ซึ่งเป็นส่วนขยายของ Smalltalk ให้สามารถใช้งานการโปรแกรมเชิงลักษณะได้ โดยอาศัยตัวห่อบล็อกของเมท็อด (block method wrapper) ช่วยสำหรับสาน advice แต่ละประเภทที่เหมาะสม

ในปีเดียวกัน Pawlak และคณะ ได้เสนอเฟรมเวิร์กชื่อ JAC [2] ซึ่งเป็นเฟรมเวิร์กสำหรับการโปรแกรมเชิงลักษณะใช้งานกับภาษาจาวาถูกพัฒนาตามไวยากรณ์ภาษาจาวา โดย JAC ต่างจาก AspectJ ตรงที่ AspectJ เป็น class-based แต่ JAC เป็น object-based โดยการสร้าง aspect แบ่งออกเป็น 3 วิธีคือ wrapping methods, role methods และ exception handlers โดย JAC ใช้แนวคิดของตัวควบคุมการห่อ (wrapping controller) เพื่อใช้งาน aspect สำหรับจุดสนใจของ JAC ตั้งอยู่บนการนิยามสถาปัตยกรรมทั่วไปสำหรับ AOP ที่พัฒนาให้เข้ากับนิพจน์แบบเดิมของภาษาจาวา

PROSE (PROgrammable extenSions of sErvices) เป็นแพลตฟอร์มที่ใช้สำหรับสร้างระบบเชิงลักษณะแบบพลวัต [15], [16], [17] ได้สนับสนุนการปรับเปลี่ยนการทำงานของระบบให้มีความยืดหยุ่นโดยสามารถสานและไม่สานตัวทำงานขณะโปรแกรมกำลังทำงานอยู่ได้ ซึ่ง Aspect ที่ใช้ใน PROSE ถูกเขียนขึ้นมาจากภาษาจาวา โดยประยุกต์จาก debugger interface ซึ่งเป็นส่วนหลักของเครื่องจักรเสมือน แต่ประสิทธิภาพการทำงานที่ได้พบว่า PROSE มีการทำงานที่ช้า

ถึงแม้ว่าประสิทธิภาพของ PROSE มีความช้าแต่วิธีการของ PROSE ก็มีประโยชน์ในการพัฒนาต่อโดยใช้ชื่อแพลตฟอร์มที่พัฒนาขึ้นว่า Wool [5] โดยได้พัฒนาโดยใช้สองวิธีคือ วิธีแรกทำการแทรกตัวเกี่ยว (hook) ไปยังเบรกพอยต์ซึ่ง และอีกวิธีจะทำการสร้างโปรแกรมในที่ซึ่งตัวเกี่ยวถูกฝังไว้กับการเรียกใช้เมท็อดแล้วทำการเริ่มต้นโปรแกรมไปยังจาวาเวอร์ชวลแมชีนใหม่อีกครั้ง วิธีดังกล่าวนี้จะหลีกเลี่ยงการแทรกตัวเกี่ยวตรงที่ไม่มีความจำเป็นต้องทำการแทรกส่งผลให้ประสิทธิภาพของ Wool ทำงานได้เร็วกว่า PROSE

ต่อมา JAsCo [6] ได้พัฒนาโมเดลองค์ประกอบใหม่รวม trap ซึ่งเป็นกับดักสำหรับการใช้งานการโปรแกรมเชิงลักษณะแบบพลวัตซึ่งสามารถถอดออกได้ อย่างไรก็ตามประสิทธิภาพการทำงานของ JAsCo มีการทำงานที่ช้า

ในปี 2004 ได้มีการดัดแปลง Jikes Research Virtual Machine (RVM) ซึ่งเป็นเครื่องจักรเสมือนสำหรับทดลองในทางวิจัยโดยเครื่องมือที่พัฒนาขึ้นนี้ชื่อว่า Steamloom [14] โดยได้นำเสนอการทดสอบการตัดจุดแบบพลวัต (dynamic pointcut) ซึ่งการกำหนดเงื่อนไขใน pointcut จะสามารถค้นหาและพบเมท็อดหรือคอนสตรัคเตอร์ที่ต้องการได้ในช่วงโปรแกรมกำลังทำงาน (Run time) เท่านั้น โดยประสิทธิภาพของ Steamloom ที่ทำการดัดแปลง Jikes Research Virtual Machine (RVM) ใหม่นั้นมีความเร็วกว่าการตัดจุดแบบพลวัตโดยใช้ AspectJ ซึ่งทำงานบน RVM ตัวดั่งเดิมแต่งานนี้ยังมีข้อผิดพลาดที่เกิดขึ้นอีกมาก

เมื่อไม่นานมานี้ JooFlux [4] ได้ถูกนำเสนอขึ้นซึ่งเป็นเครื่องมือสำหรับสร้างระบบการโปรแกรมเชิงลักษณะแบบพลวัตโดยใช้คำสั่ง `invokedynamic` โดยใช้ตัวรวมไบต์โค้ดที่มีอยู่ในภาษาจาวานำมาประยุกต์ใช้สร้างตัวทำงานก่อนหน้าและตัวทำงานตามหลัง แต่

พบว่าตัวรวมไบต์โค้ดที่ JooFlux นำมาใช้ยังมีการทำงานบางส่วนไม่ตรงตามนิยามของตัวทำงาน และมีบางส่วนของการทำงานที่ไม่เกี่ยวข้องต่อการทำงานของตัวทำงานทำให้ประสิทธิภาพของ JooFlux ยังช้าอยู่

จากปัญหาที่พบใน JooFlux งานวิจัยนี้ได้เกิดขึ้นโดยใช้ชื่อของตัวรวมไบต์โค้ดใหม่ว่า AABC [3] โดยใช้ประโยชน์จากคำสั่ง `invokedynamic` เช่นกันโดยทำการพัฒนาตัวรวมไบต์โค้ดสำหรับตัวทำงานแต่ละประเภทขึ้นมา และได้ทำการดัดแปลงตัวรวมไบต์โค้ดบางตัวให้ทำงานเฉพาะตามนิยามของตัวทำงานส่งผลให้ประสิทธิภาพการทำงานดีขึ้น

5. สรุปผล

บทความนี้ได้นำเสนอตัวแจงส่วนการดัดจุดซึ่งต่อยอดมาจากตัวรวมไบต์โค้ด AABC เพื่ออำนวยความสะดวกในการใช้งานการเลือกจุดร่วม โดยประสิทธิภาพในการใช้งานพบว่าไม่ทำให้ระบบทำงานช้าลงซึ่งเป็นผลมาจากขั้นตอนการแจงส่วนการดัดจุดและการรวมไบต์โค้ดซึ่งถูกกระทำตอนคำสั่ง `invokedynamic` ถูกเรียกเป็นครั้งแรกเท่านั้น และสำหรับพีซีดีของการดัดจุดที่นำมาสร้างเป็นเพียงสับเซตของพีซีดีทั้งหมดที่ใช้สำหรับการดัดจุดใน AspectJ เท่านั้น

ดังนั้นจึงสรุปได้ว่าการใช้งานตัวแจงส่วนการดัดจุดสามารถช่วยอำนวยความสะดวกในการใช้งานสำหรับการเลือกจุดตัด และแทบจะไม่มีผลต่อการทำงานของระบบ และยิ่งไปกว่านั้นคือผู้ใช้งานสามารถทำการกำหนดเงื่อนไขการดัดจุดและขียนตัวทำงานเพื่อสานเข้าไปโดยผ่านกระบวนการเหล่านี้ได้ง่ายขึ้นพร้อมทั้งกระบวนการเหล่านี้จะกระทำการในขณะที่ระบบกำลังทำงานอยู่ซึ่งสะดวกต่อผู้ดูแลระบบ และเป็นประโยชน์ต่อการนำไปใช้งานกับระบบที่มีความสำคัญที่จำเป็นต้องเปิดทำงานตลอดเวลา และสำหรับอนาคตงานวิจัยนี้จะทำการพัฒนาต่อยอดให้สามารถใช้งานเงื่อนไขของการดัดจุดได้มากและครอบคลุมขึ้น และจะทำการพัฒนาการทำงานของตัวทำงานแต่ละประเภทให้เร็วขึ้น

เอกสารอ้างอิง

[1] Elliptic Group, "Java benchmarking article", website. http://www.ellipticgroup.com/html/benchmarkingArticle.html.

[2] R. Pawlak, L. Seinturier, L. Duchien and G. Florin, "JAC:A flexible solution for aspect-oriented programming in Java", In Proceedings of the 3rd International Conference on Reflection. Kyoto Japan, 2001.

[3] S. Nopnitp and C. Kaewkasi, "Aspect-aware bytecode combinators for a dynamic AOP system with invokedynamic", Computer Science and Software Engineering (JCSSE2013), May. 2013.

[4] J. Ponge and F. Le Mouël, "JooFlux: Hijacking Java 7 InvokeDynamic To Support Live Code Modifications", Research Report, INRIA CITI Lab, INSA Lyon, 2012.

[5] Y. Sato, S. Chiba and M. Tatsubori, "A selective, just-in-time aspect weaver", In GPCE'03: Proceedings of the 2nd international conference on Generative programming and component engineering, pages 189-208, 2003.

[6] D. Suvee, W. Vanderperren and V. Jonckers, "JAsCo: an Aspect-Oriented approach tailored for component based software development", In AOSD'03: Proceedings of the 2nd international conference on Aspect-Oriented software development, pp. 21-29, Mar. 2003.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier and J. Irwin, "Aspect-Oriented Programming", In Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Vol. LNCS 1241.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold. "An Overview of AspectJ", In Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01), Jørgen Lindskov Knudsen (Ed.). Springer-Verlag, London, UK, UK, 327-353.

[9] ANTLR, (2013). Website. http://www.antlr.org/.

[10] SciMark2.0 website. http://math.nist.gov/scimark2/.

[11] Java™ Platform, Standard Edition 7, (2012). Website. http://docs.oracle.com/javase/7/docs/api/.

[12] J. Rose, "Bytecodes meet combinators: invokedynamic on the JVM", In Proceedings of the Third Workshop on Virtual Machines and intermediate Languages (Orlando, Florida, October 25-29, 2009). VMIL '09. ACM, New York, NY, 1-11.

[13] R. Hirschfeld, "AspectS - aspect-oriented programming with Squeak", In Revised Papers from NODe '02, pages 216–232, London, UK, 2003.

[14] C. Bockisch, M. Haupt, M. Mezini and K. Ostermann, "Virtual Machine Support for Dynamic Join Points", In AOSD '04: Proceedings of the 3rd international conference on Aspect - oriented software development, pages 83-92. ACM, 2004.

[15] Popovici, A., Alonso, G. and Gross, T., (2003). Just-in-time aspects: efficient dynamic weaving for Java. In Proceedings of the 2nd international conference on Aspect-oriented software development, Boston, USA.

[16] A. Popovici, T. Gross and G. Alonso, "Dynamic weaving for aspect-oriented programming". In Proceedings of the 1st international conference on Aspect-oriented software development (AOSD '02). ACM, New York, NY, USA, 141-147, 2002.

[17] A. Nicoara, and G. Alonso, "Dynamic AOP with PROSE", In Proceedings of ASMEA '05 in conjunction with CAISE '05, 2005.

ประวัติผู้เขียน

นายสันติ นภนิภา เกิดเมื่อวันที่ 20 พฤศจิกายน พ.ศ. 2531 ที่ อำเภอพนมสารคาม จังหวัดฉะเชิงเทรา ได้รับการศึกษาในระดับมัธยมศึกษาปีที่ 1 ถึงชั้นมัธยมศึกษาปีที่ 6 โรงเรียนพนมสารคามพนมอดุลวิทยา อำเภอพนมสารคาม จังหวัดฉะเชิงเทรา ในปีการศึกษา 2551 ได้เข้าศึกษาต่อในระดับปริญญาตรีสำนักวิชาวิศวกรรมศาสตร์ หลังจากจบปีการศึกษาที่ 1 ได้ถูกคัดเลือกศึกษาต่อในสาขาวิชาวิศวกรรมคอมพิวเตอร์ สำนักวิชาวิศวกรรมศาสตร์ มหาวิทยาลัยเทคโนโลยีสุรนารี และสำเร็จการศึกษาในปีการศึกษา 2554 ภายหลังสำเร็จการศึกษาในระดับปริญญาตรี ได้เข้าศึกษาในระดับปริญญาโท สาขาวิชาวิศวกรรมคอมพิวเตอร์ สำนักวิชาวิศวกรรมศาสตร์ มหาวิทยาลัยเทคโนโลยีสุรนารี ในปีการศึกษา 2555

ในระหว่างการศึกษาได้รับความอนุเคราะห์อย่างดีจากอาจารย์ประจำวิชาเทคโนโลยีเชิงวัตถุ (Object-Oriented Technology) ให้เป็นผู้ช่วยสอนปฏิบัติการ และได้รับการตีพิมพ์เผยแพร่บทความวิชาการโดยรายละเอียดสามารถดูได้ที่ภาคผนวก ก