รหัสโครงการ SUT7-705-55-12-39

รายงานการวิจัย

คอมไพเลอร์เชิงพลวัตสำหรับจาวา
(A Dynamic Compiler for Java)

ได้รับทุนอุดหนุนการวิจัยจาก
มหาวิทยาลัยเทคโนโลยีสุรนารี

ผลงานวิจัยเป็นความรับผิดชอบของหัวหน้าโครงการวิจัยแต่เพียงผู้เดียว

มหาวิทยาลัยเทคโนโลยีสุรนารี

รหัสโครงการ SUT7-705-55-12-39

รายงานการวิจัย

คอมไพเลอร์เชิงพลวัตสำหรับจาวา
(A Dynamic Compiler for Java)

คณะผู้วิจัย

หัวหน้าโครงการ

ผู้ช่วยศาสตราจารย์ ดร.ชาญวิทย์ แก้วกสิ

มหาวิทยาลัยเทคโนโลยีสุรนารี

ได้รับทุนอุดหนุนการวิจัยจากมหาวิทยาลัยเทคโนโลยีสุรนารี ปีงบประมาณ พ.ศ. 2555

ผลงานวิจัยเป็นความรับผิดชอบของหัวหน้าโครงการวิจัยแต่เพียงผู้เดียว

กันยายน 2558

กิตติกรรมประกาศ

งานวิจัยชิ้นนี้จะไม่สามารถสำเร็จลุล่วงไปได้ด้วยดีหากไม่ได้รับการสนับสนุนจากหน่วยงานในมหาวิทยาลัย ทั้งในระดับสาขาวิชา สำนักวิชา สถานวิจัยสำนักวิชาวิศวกรรมศาสตร์ และสถาบันวิจัยและพัฒนา และทีมพัฒนา Java Development Kit และผู้ช่วยวิจัยทุกท่าน

ผู้วิจัยขอขอบคุณ John Rose และ Remi Forax จากทีมพัฒนา invokedynamic เป็นพิเศษ ที่สนับสนุนข้อมูลทางเทคนิคในประเด็นเชิงลึกที่เกี่ยวข้องกับ invokedynamic และโครงสร้างอื่น ๆ ของ Java 1.7

มหาวิทยาลัยเทคโนโลยีสุรนารี

บทคัดย่อ

งานวิจัยนี้ศึกษาการพัฒนาคอมไพเลอร์สำหรับแปลงโปรแกรมภาษาจาวาให้เป็นโปรแกรมที่ใช้ชุดคำสั่ง invokedynamic เพื่อเพิ่มคุณสมบัติความพลวัตให้กับโปรแกรม โดยทำการดัดแปลงคอมไพเลอร์ของจาวาใน OpenJDK 1.7 ให้สร้างเมธอดเริ่มต้นที่ทำการโยงไปหาตัวจัดการเมธอดและวัตถุคอลไซต์ซึ่งเชื่อมต่อกับเมธอดจริง แทนการเชื่อมต่อกับเมธอดจริงโดยตรง โดยเสนอกฎการแปลงในรูปแบบตามแคลลูลัสของ Featureweight Java

จากผลงานวิจัย พบว่าสามารถพัฒนาคอมไพเลอร์ที่แปลงโปรแกรมให้ใช้ชุดคำสั่ง invokedynamic ได้สำเร็จโดยทดสอบกับชุดตัววัด SciMark 2.0 และได้ผลการทำงานถูกต้อง 100% ด้วยชุดตัววัดดังกล่าว สำหรับประเด็นด้านประสิทธิภาพพบว่าโปรแกรมที่ใช้ชุดคำสั่ง invokedynamic ส่วนใหญ่มีประสิทธิภาพอยู่ในระดับเกิน 20% ของโปรแกรมดั้งเดิม โดยพบว่าโปรแกรมที่มีการคำนวณซับซ้อนจะทำให้ JVM ไม่สามารถดึงประสิทธิภาพของโปรแกรมที่ใช้ชุดคำสั่ง invokedynamic ได้เต็มที่ ซึ่งเป็นประเด็นที่สามารถปรับปรุง JVM ได้ในรุ่นต่อ ๆ ไป

มหาวิทยาลัยเทคโนโลยีสุรนารี

Abstract

The research work described in this report studied a development of a compiler for translating a Java binary program into a program that utilizes invokedynamic instructions. The benefit of adding invokedynamic instructions is for increasing the dynamic property for the compiled program. The work tweaked a compiler in OpenJDK 1.7 to make it generate Bootstrap Method that allow linking Method Handles with Callsite Objects which connected with real methods. This work explain transformation rules using Featherweight Java calculus.

From the experimental results, the compiler can successfully translate programs from the SciMark 2.0 suite to be invokedynamic implanted and they work 100% correctly. From the performance perspective, the translated programs have performance around 20% compared to the original benchmark programs. There is an observation that the JVM could not optimize a complex invokedynamic program well enough, which is a for improving JVM in the future.

มหาวิทยาลัยเทคโนโลยีสุรนารี

สารบัญ

มหาวิทยาลัยเทคโนโลยีสุรนารี

สารบัญรูป

มหาวิทยาลัยเทคโนโลยีสุรนารี

บทที่ 1
บทนำ

ภาษา Java Virtual Machine Language (JVML) (Gosling, 2013) เป็นภาษากึ่งกลาง (intermediate language) ที่นิยามไว้ในข้อกำหนด (specification) ของ Java Virtual Machine ซึ่งโดยทั่วไปเรียกรหัสในภาษานี้ว่าไบต์โค้ด (Bytecode) โดยภาษา JVML นี้ได้ถูกออกแบบให้มีความสามารถทวนสอบ (verify) ได้ก่อนการรัน และรูปแบบของข้อมูลในภาษานี้มีข้อมูลเชิงชนิด (type information) ที่เพียงพอต่อการโยงกันเชิงสัญลักษณ์ (symbolic linking) ระหว่างหน่วยของคลาส ซึ่งในภาษา JVML มีการนิยามชุดคำสั่งสำหรับการเรียกใช้ (invoke) เมธอดของภาษา Java ไว้ 4 แบบตามชนิดของเมธอดที่แตกต่างกัน

อย่างไรก็ตามเพื่อสนับสนุนการทำงานของภาษากลุ่มพลวัต (dynamic language) นั้น ได้มีการนิยามชุดคำสั่ง invoke ตัวใหม่ที่มีชื่อว่า invokedynamic (Da Vinci Machine project, 2009; Rose, 2008; Rose, 2009) ขึ้นมาเพื่อสนับสนุนกลุ่มภาษาพลวัต เช่นภาษา JavaScript (Mozilla Corp., 2015), Jython (Hugunin et al., 2015), JRuby (Nutter et al., 2015) หรือ Clojure (Hickey et al., 2015) เป็นต้น

กลุ่มภาษาพลวัตที่ไม่ใช่ภาษา Java นี้ได้รับการตอบรับในระดับหนึ่ง อย่างไรก็ตาม Java ยังเป็นภาษาที่มีความนิยมมากที่สุดจากการสำรวจ Programming Community Index (TIOBE Software, 2015) การทำให้ Java มีความสามารถเชิงพลวัตจึงเป็นสิ่งที่จำเป็นมากขึ้นสำหรับในปัจจุบัน เนื่องจากความสามารถในกลุ่มภาษาพลวัตนั้นมีประโยชน์ในหลายกรณี เช่นการสร้างระบบเชิงลักษณะแบบพลวัต เป็นต้น (Kaewkasi, 2009)

รายงานวิจัยฉบับนี้อธิบายถึงการสร้างคอมไพเลอร์และ Software Development Kit (SDK) ที่แปลงภาษา Java เป็น JVML ซึ่งใช้ invokedynamic เพื่อให้โปรแกรมที่คอมไพล์ออกมานั้นมีพฤติกรรมเหมือนโปรแกรมจากกลุ่มภาษาพลวัต แต่รูปแบบไวยากรณ์ยังเป็นภาษา Java อยู่ โดยจะทำการศึกษาโครงสร้างภายในของ OpenJDK 1.7 (Oracle Corp., 2010) และทำการแก้ไขคอมไพเลอร์ javac ให้สร้างไบต์โค้ด (bytecode) แบบ invokedynamic ซึ่งจะเป็นไปตามกฎการแปลงที่ทำการคิดค้นขึ้น

วัตถุประสงค์ของโครงการวิจัย

1. เพื่อคิดค้นกฎการแปลงแบบฟอร์มอลจากการเรียกเมธอดปกติเป็นการเรียกเมธอดเชิงพลวัตในโปรแกรมสำหรับเครื่องจักรเสมือนจาวา
2. เพื่อพัฒนาคอมไพเลอร์ต้นแบบสำหรับภาษา Java ที่สร้างรหัสการเรียกเมธอดเชิงพลวัต

ขอบเขตของโครงการวิจัย

คอมไพเลอร์ที่พัฒนาขึ้นเป็นต้นแบบที่ทำงานได้บนระบบปฏิบัติการ Linux ชนิด 32 บิต เท่านั้น

ทฤษฎี สมมุติฐานและกรอบแนวความคิดของโครงการวิจัย

งานวิจัยนี้สร้างอยู่บนพื้นฐานของชุดคำสั่ง invokedynamic ของ Java Virtual Machine (JVM) สำหรับ OpenJDK 1.7 เป็นต้นไปนั้น ประกอบด้วยไบต์โค้ดใหม่ นั่นคือ invokedynamic และส่วนประกอบอื่นๆ เช่น เมธอดเริ่มต้น (bootstrap method) และตัวจัดการเมธอด (method handles) สำหรับตัวไบต์โค้ดของ invokedynamic นั้นเป็นคำสั่งขนาด 5 ไบต์ และไม่มีข้อมูลชนิดอยู่กับตัว invokedynamic เพราะได้รับการออกแบบไว้ให้สามารถแทนความหมายของคำสั่ง invoke ของ JVM ได้ทุกแบบ (Rose, 2008)

โครงสร้างที่ใช้ร่วมกับ invokedynamic เรียกว่า name-and-type เป็นโครงสร้างที่ระบุ *ชื่อ*ของ **จุดเรียกเมธอด** (call site) รวมทั้งข้อมูล*ชนิดของพารามิเตอร์ที่รับเข้า* และ *ชนิดของการคืนค่า* โดย **เมธอดเริ่มต้น** (bootstrap method) เป็นเมธอดที่ทำงานในกระบวนการแรกสุดของขั้นตอนการเรียกใช้ invokedynamic และเมื่อทำงานเสร็จสิ้น **เมธอดเริ่มต้น**จะคืนค่าเป็นวัตถุของคลาส `java.dyn.CallSite` เมื่อมีการร้องขอจาก JVM โดยที่ JVM จะทำการส่งค่าข้อมูลของจุดเรียกพลวัตที่กำลังทำงานอยู่ไปให้**เมธอดเริ่มต้น**

คอมไพเลอร์ที่ต้องการสนับสนุนการทำงานแบบพลวัตที่ใช้คำสั่ง invokedynamic นั้นจำเป็นต้องทำการเตรียมการในการสร้างเมธอดที่เหมาะสมให้แต่ละคลาสโดยทำการลงทะเบียนตัวเมธอดให้กับ JVM ในส่วนการตั้งค่าเริ่มต้นของคลาส สำหรับในส่วนของการสร้างวัตถุของจุดเรียกนั้นจำเป็นต้องตั้งค่าเป้าหมายของวัตถุจุดเรียกด้วย**ตัวจัดการเมธอด** (method handle) ซึ่ง**ตัวจัดการเมธอด**เป็นโครงสร้างขนาดเล็กใน JVM ที่ออกแบบให้สามารถเรียกเมธอดได้ สำหรับโครงสร้างในลักษณะที่คล้ายกันนี้ ใน JDK 6 คือการประมวลผลด้วยกลไก Reflection (Bodden,

2010) เป็นการทำงานที่ได้ผลลัพธ์แบบเดียวกัน แต่จะมีประสิทธิภาพต่ำกว่าในระบบตามมาตรฐาน JSR-292 (Rose, 2008) ซึ่งใน JSR-292 มีการนิยามตัวจัดการเมธอดไว้หลายชนิด โดยชนิดที่จะใช้ในงานวิจัยนี้ คือ ตัวจัดการเมธอดโดยตรง (direct method handle) ซึ่งสามารถใช้กระบวนการของคลาส Lookup สร้างขึ้นมาได้

คำอธิบายศัพท์

เมธอดเริ่มต้น (Bootstrap Method) หมายถึง เมธอดของภาษาจาวาที่ถูกเรียกใช้งานเมื่อคำสั่ง invokedynamic ถูกเรียกใช้งานเป็นครั้งแรก

คอลไซต์ (Call site) หมายถึง ตำแหน่งการเรียกใช้งานเมธอดในโปรแกรม โดยเป็นตัวแทนของตำแหน่งดังกล่าวคือคลาส CallSite ซึ่งอยู่ในระบบของ Java Development Kit หรือ JDK

วัตถุคอลไซต์ (Call site object) หมายถึง วัตถุที่แทนตำแหน่งการใช้งานเมธอดซึ่งเป็นวัตถุของคลาส CallSite ที่ถูกส่งออกมาจากเมธอดเริ่มต้นสำหรับใช้งานกับคำสั่ง invokedynamic

ไบต์โค้ด (Bytecode) หมายถึงรหัสที่ใช้ทำงานกับเครื่องจักรเสมือนโดยในภาษาจาวาก็คือไฟล์ .class ที่ได้จากการคอมไพล์ไฟล์ .java

เครื่องจักรเสมือน (Virtual machine) หมายถึง เครื่องจักรที่ทำหน้าที่เป็นตัวกลางของการทำงานระหว่างไบต์โค้ดของโปรแกรมกับเครื่องจักร ในภาษาจาวาเครื่องจักรเสมือนช่วยให้ภาษาจาวาสามารถทำงานกับสถาปัตยกรรมของเครื่องจักรที่มีสภาพแวดล้อมที่แตกต่างกันได้

สำหรับส่วนถัด ๆ ไปของเอกสารรายงานวิจัยฉบับนี้ประกอบไปด้วย การทบทวนวรรณกรรมในบทที่ 2 จากนั้นจะเป็นการอธิบายกฎการแปลงเป็น invokedynamic ในบทที่ 3 และตามด้วยการทดสอบในบทที่ 4 และบทสรุปในบทที่ 5

มหาวิทยาลัยเทคโนโลยีสุรนารี

บทที่ 2
การทบทวนวรรณกรรม

คอมไพเลอร์และ SDK ที่จะพัฒนาขึ้นมานี้สร้างอยู่บน OpenJDK ที่ประกอบไปด้วย คอมไพเลอร์ javac ซึ่งทำการแปลงภาษา Java เป็นไบต์โค้ดและ Java Runtime Environment (JRE) โดยที่ระบบเดิมนั้นมีข้อจำกัดเชิงพลวัต คือ

1. คอมไพเลอร์ javac นั้นจะสามารถสร้างชุดคำสั่งแบบพลวัตได้ก็ต่อเมื่อมีการระบุให้คำสั่งดังกล่าวเป็นชนิดพลวัต ทำให้ไม่สามารถคอมไพล์ต้นรหัส (source code) ที่มีอยู่ในปัจจุบันให้ได้รับประโยชน์จากความเป็นพลวัตของโหมด invokedynamic ได้

2. JRE นั้นถูกสร้างขึ้นเพื่อให้ทำงานในโหมดธรรมดา ไม่ใช่ในโหมด invokedynamic

ในปัจจุบันภาษาที่สนับสนุน invokedynamic ได้แก่ JRuby (Nutter et al., 2015) ซึ่งเป็น อิมพลีเมนเตอชันของภาษา Ruby (Matsumoto et al., 2015) โดยตัวคอมไพเลอร์ JRuby จะแปลงต้นรหัสภาษา Ruby ให้เป็นไบต์โค้ดตามรูปแบบของภาษา JVML และจึงทำการรันบน JVM แม้ว่า JRuby จะใช้ความสามารถของโหมด invokedynamic แล้วก็ตาม แต่ข้อจำกัดของการปรับใช้นั้นยังมีอยู่ที่ตัวภาษา อีกทั้งภาษา Java ก็เป็นภาษาที่ยังได้รับความนิยมอยู่มากในปัจจุบัน (TIOBE Software, 2015) ดังนั้นการสร้างคอมไพเลอร์ภาษาจาวาที่สนับสนุนความเป็นพลวัตของโหมด invokedynamic จึงมีประโยชน์ต่อผู้ใช้ทั่วไปที่จะสามารถนำไปปรับใช้ต่อไปได้ นอกจาก JRuby แล้วกลุ่มภาษาพลวัต เช่น Groovy หรือ Clojure ก็มีแนวโน้มที่จะสนับสนุนโหมด invokedynamic โดย Groovy จะสนับสนุนโหมดนี้ในรุ่น 2.0 (Koenig et al., 2007)

ระบบ JSR-292 Backport (Forax, 2009) เป็นอีกงานวิจัยหนึ่งที่มีความเกี่ยวข้องกับโหมด invokedynamic โดยระบบนี้จะทำงานตรงข้ามกับงานวิจัยนี้ ซึ่ง Backport จะทำการอ่านไบต์โค้ดแบบ invokedynamic แล้วแปลงกลับเป็นไบต์โค้ดกลุ่มเดิมที่อยู่ในโหมดธรรมดา ซึ่งทำงานได้บน JDK 6 สำหรับกฎการแปลงที่ได้จากงานวิจัยนี้จะมีประโยชน์ในการทวนสอบความถูกต้องของการทำงานของ Backport นี้ด้วย

ในปี Ponge และ Mouel (2012) เสนอโครงการ JooFlux ซึ่งเป็นเครื่องมือสำหรับสร้างระบบการโปรแกรมเชิงลักษณะ (aspect-oriented programming) แบบพลวัต โดยใช้คำสั่ง

invokedynamic โดยใช้กลไกของเมธอด filterArguments แและเมธอด filterReturnValue ในแพ็คเกจ java.lang.invoke ในการรวมไบต์โค้ด (Bytecode combination) JooFlux ใช้เทคนิคการเชื่อมต่อเมธอดของ Bootstrap Method ร่วมกับ Call site object และพัฒนาระบบในลักษณะของ Java Agent ทำให้เกิดการแปลงไบต์โค้ดได้ในขณะเวลารันไทม์ ทำให้ระบบการโปรแกรมเชิงลักษณะของ JooFlux สามารถมีคุณสมบัติเชิงพลวัตได้

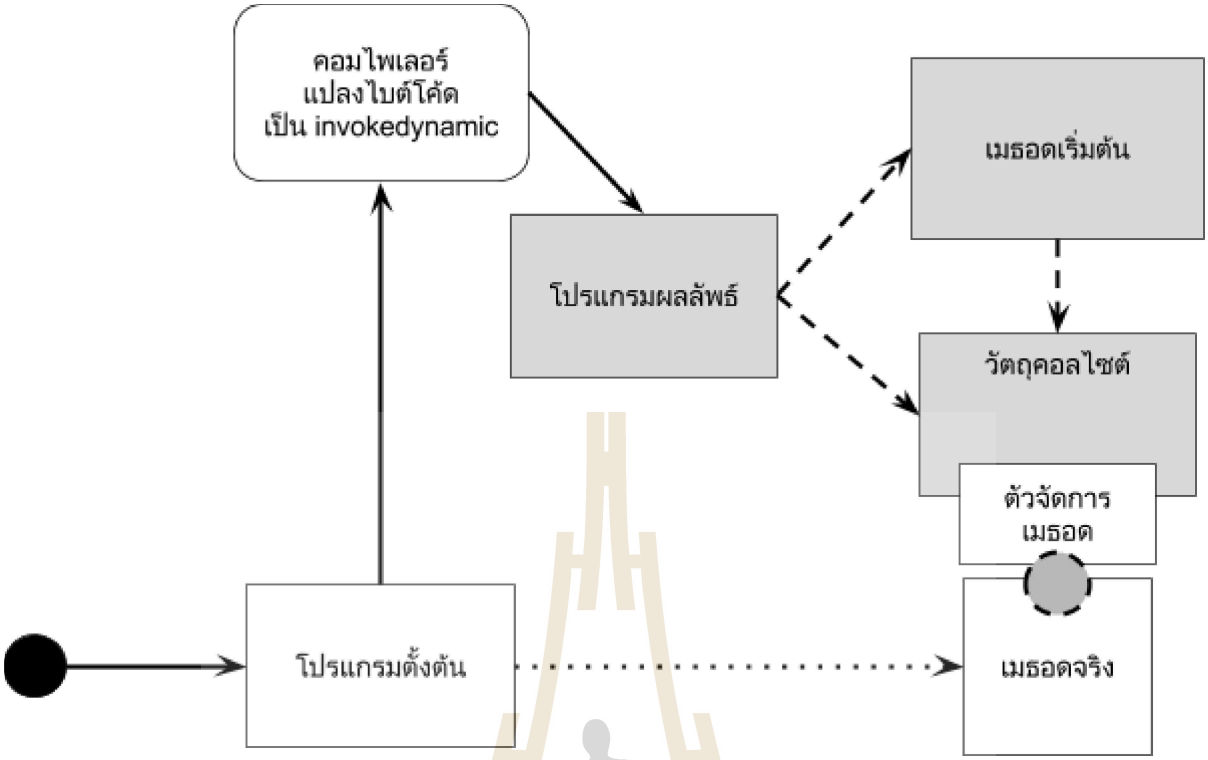
การทำงานของระบบคอมไพเลอร์ invokedynamic ที่นำเสนอในงานวิจัยนี้ รับตัวป้อน (input) เป็นไฟล์ .class และมีกระบวนการทำการแปลงไฟล์ .class ปกติให้สามารถทำงานด้วยชุดคำสั่ง invokedynamic โดยมีขั้นตอนดังต่อไปนี้

1. ตรวจสอบชุดคำสั่ง invoke ทั้งหมดใน .class
2. สร้างอะเรย์ของตัวจัดการเมธอด (Method Handle)
3. สร้างเมธอดเริ่มต้น (Bootstrap Method)
4. แปลงชุดคำสั่ง invoke ทั้งหมดให้อยู่ในรูปแบบของ invokedynamic

เมื่อโปรแกรมรับไฟล์ .class ของภาษาจาวาเข้ามา ไฟล์ .class จะผ่านตัวแปลงไบต์โค้ด ผลลัพธ์ที่ได้จะเป็นโปรแกรม .class ของภาษาจาวาตัวใหม่ ที่ภายในมีการใช้งานคำสั่ง invokedynamic แทนที่คำสั่ง invoke อื่น ๆ เมื่อโปรแกรมที่ได้รับการแปลงเริ่มทำงาน จะเกิดกลไกการทำงานครั้งแรกที่ผ่านการทำงานของเมธอดเริ่มต้น (Bootstrap Method) จากนั้นเมธอดเริ่มต้นจะทำการสร้างวัตถุคอลไซต์ (Call site object) ที่เชื่อมโยงอยู่กับเมธอดจริง เพื่อทำงานต่อไป

2.1 เมธอดเริ่มต้น (Bootstrap Method)

เมธอดเริ่มต้น เป็นเมธอดที่ถูกสร้างขึ้นในขั้นตอนของการแปลงคำสั่งไบต์โค้ดที่ใช้เรียกคอนสตรักเตอร์หรือเมธอดแต่ละประเภทไปเป็นคำสั่ง invokedynamic โดยเมธอดเริ่มต้นจะถูกเรียกใช้งานก็ต่อเมื่อตำแหน่งที่ถูกแปลงไปเป็นคำสั่ง invokedynamic ถูกเรียกเป็นครั้งแรก ซึ่งวัตถุของคลาส MethodHandle และวัตถุคอลไซต์เป็นองค์ประกอบสำคัญสำหรับใช้สร้างเมธอดเริ่มต้น



รูปที่ 2.1 แผนผังแสดงการทำงานของคอมไพเลอร์สำหรับแปลงโปรแกรม
ให้เป็นชุดคำสั่ง invokedynamic

รูปที่ 2.1 แสดงองค์ประกอบและแผนผังการทำงานของคอมไพเลอร์สำหรับแปลงโปรแกรให้เป็นชุดคำสั่ง invokedynamic โดย *โปรแกรมตั้งต้น* จะโยง (ลูกศรเส้นประ) เข้ากับเมธอดจริงด้วยคำสั่ง invoke ต่าง ๆ หลังจากผ่านโปรแกรมคอมไพเลอร์จะได้ *โปรแกรมผลลัพธ์* ที่มี เมธอดเริ่มต้น วัตถุคอลไซต์ และใช้ตัวจัดการเมธอดโยงหาเมธอดจริง แทนการโยงเข้ากับเมธอดจริงโดยตรง โดยโปรแกรมผลลัพธ์จะใช้ invokedynamic แทนคำสั่ง invoke อื่น ๆ

มหาวิทยาลัยเทคโนโลยีสุรนารี

2.2 ตัวจัดการเมธอด (Method Handle)

ตัวจัดการเมธอด (Method Handle) เป็นวัตถุของคลาส MethodHandle ซึ่งทำหน้าที่เป็นตัวอ้างอิงไปยังเมธอดหรือคอนสตรัคเตอร์โดยตรง ในบริบทของคำสั่ง invokedynamic นั้น หน้าที่ของตัวจัดการเมธอดเทียบได้กับตัวชี้ที่ทำการชี้ไปยังเมธอดหรือคอนสตรัคเตอร์ที่ต้องการเรียกใช้งานจริง

การค้นหาเมธอดหรือคอนสตรัคเตอร์ที่ต้องการอ้างอิงถึงนั้น สามารถแบ่งออกได้ตามประเภทของการเรียกใช้งาน โดยมีกลุ่มคำสั่งของการค้นหาเมธอดหรือคอนสตรักเตอร์เพื่อให้ได้ตัวจัดการเมธอดของการเรียกแต่ละแบบสำหรับใช้ร่วมกับคำสั่ง invokedynamic ดังนี้

1) เมธอด findStatic ใช้สำหรับค้นหาเมธอดที่เป็นแบบสถิตย์ (Static method)
2) เมธอด findVirtual ใช้สำหรับค้นหาเมธอดทั่วไป หรือ เมธอดที่เป็นแบบอินเตอร์เฟส
3) เมธอด findSpecial ใช้สำหรับค้นหาเมธอดที่ในการสืบทอด (Inherited method)
4) เมธอด findConstructor ใช้สำหรับค้นหาคอนสตรักเตอร์

2.3 วัตถุคอลไซต์

วัตถุคอลไซต์เป็นวัตถุที่แทนตำแหน่งการใช้งานเมธอดซึ่งเป็นวัตถุของคลาส CallSite ที่ถูกส่งออกมาจากเมธอดเริ่มต้นในครั้งแรกที่คำสั่ง invokedynamic ถูกเรียกใช้งาน โดยถ้าคำสั่ง invokedynamic ถูกเรียกขึ้นอีกครั้งก็จะมาใช้วัตถุคอลไซต์โดยตรงซึ่งไม่ต้องเข้ามาทำงานในเมธอดเริ่มต้นอีก

ในบทถัดไปจะกล่าวถึงกฎการแปลงโปรแกรมปกติ ให้เป็นโปรแกรมที่บรรจุชุดคำสั่ง invokedynamic เพื่อแสดงให้เห็นความหมายของการทำงานของคำสั่ง invoke แต่ละประเภทที่เทียบเท่าความหมายของ invokedynamic

มหาวิทยาลัยเทคโนโลยีสุรนารี

บทที่ 3
กฎสำหรับแปลงโปรแกรมเป็น invokedynamic

ในบทนี้จะกล่าวถึงกฎการแปลงสำหรับ invokedynamic สำหรับการเรียกใช้คำสั่ง invoke ชนิดต่าง ๆ ของ Java Virtual Machine โดยกฎการแปลงจะแบ่งออกเป็น 5 กลุ่ม ตามชนิดของการ invoke นั่นคือ Static Method Call, Virtual Method Call, Interface Method Call, Inherited Method Call และ Constructor Call

3.1 การเรียกใช้เมธอดเชิงสถิตย์

การเรียกใช้เมธอดเชิงสถิตย์ (Static Method Call) คือการเรียกใช้เมธอดระดับคลาสในภาษาจาวา ในส่วนนี้จะเริ่มต้นด้วยการอธิบายความหมายของ invokestatic โดยใช้ Featherweight Java (Igarachi, Pierce and Wadler, 2001) และตามด้วยการแปลงการเรียกใช้เมธอดเชิงสถิตย์ ไปเป็น invokedynamic

ความหมายของ invokestatic สามารถอธิบายได้ดังนี้ กำหนดให้มีโปรแกรม ตามรูปแบบ Featherweight Java

```
class C extends Object {
  static D m(Object arg1) {
    return new D()
  }
}


class D extends Object {
}


C.m(new Object())
```

มหาวิทยาลัยเทคโนโลยีสุรนารี

โปรแกรมสำหรับเรียกใช้เมธอด m ของคลาส C ซึ่งคืนค่าเป็นวัตถุ D สามารถประมวลผลได้ตามขั้นตอนดังต่อไปนี้

$$\rightarrow C.m(new\ Object())\ \{new\ Object()/arg1\}$$
$$\rightarrow new\ D()$$

ในภาษา JVML การเรียกใช้เมธอด C.m(new Object()) สามารถคอมไฟล์เป็น invokestatic ได้ด้วยกฎดังนี้

กฎการคอมไพล์เมธอดเชิงสถิตย์

$$\frac{C.m(\bar{e}): D}{invokestatic(C, m, \bar{e}): Object}$$

กฎการแปลง invokestatic ไปเป็น invokedynamic

$$\frac{mh = findStatic(C, m, D, typesof(\bar{e}))}{invokestatic(C, m, \bar{e}): Object \rightarrow invokedynamic(mh, \bar{e}): D}$$

โดย mh เป็นตัวจัดการเมธอด
C และ D เป็นคลาส
m เป็นเมธอด
$\bar{e}$ เป็น expression

ในขั้นตอนของคอมไพเลอร์ javac ปกติ จะเป็นการแปลงการเรียกใช้ภาษาจาวา เป็นภาษา JVML โดยการเรียกใช้เมธอดแบบ static ของ

$$C.m(new\ Object())$$

ได้ด้วย

$$invokestatic(C,\ m,\ new\ Object()) : Object$$

มหาวิทยาลัยเทคโนโลยีสุรนารี

ในขั้นตอนของคอมไพเลอร์ภาษาจาวาที่เสนอในเอกสารฉบับนี้จะทำงานตามกฎนี้ ซึ่งด้วยกฎการแปลง invokestatic เราสามารถแทนที่การเรียกใช้

$$invokestatic(C,\ m,\ new\ Object()) : Object$$

ได้ด้วย

$$invokedynamic((name \Rightarrow m,\ type \Rightarrow (Object) : D),\ new\ Object()) : Object$$

จะสังเกตได้ว่า invokestatic คืนค่าเป็น Object เนื่องจากเป็นสิ่งที่ระบุไว้ในข้อกำหนดภาษา JVML

สำหรับขั้นตอนของการประเมินโปรแกรมจะทำให้โปรแกรมที่แปลงแล้วด้วยคอมไพเลอร์ซึ่งอยู่ในรูปแบบ

```
class C extends Object {
  static D m(Object arg1) {
    return new D()
  }
}


class D extends Object {
}


invokedynamic((name ⇒ m, type ⇒ (Object): D),
              new Object()): Object
```

โดยคู่ลำดับ (name ⇒ m, type ⇒ (Object): D) คือโครงสร้าง NameAndType ตามข้อกำหนดใน JVML และกำหนดให้ตาราง BM เป็นดังต่อไปนี้

(name ⇒ m, type ⇒ (Object): D) **match** { (name, type) → **new** MH(type, C.m) }

มหาวิทยาลัยเทคโนโลยีสุรนารี

โดยคู่ลำดับ (name ⇒ m, type ⇒ (Object): D) จะแปลงเป็นวัตถุของคลาส MH ซึ่งเป็นผลลัพท์จากการ findStatic บนตาราง BM จะสามารถประเมิน invokedynamic ได้ด้วยขั้นตอนต่อไปนี้

$$
\begin{aligned}
&invokedynamic((name \Rightarrow m, type \Rightarrow (Object) : D),\ new\ Object()) :\ Object \\
&\rightarrow invokedynamic(new\ MH((Object) : D,\ C.m),\ new\ Object()) :\ Object \\
&\rightarrow invokedynamic(C.m,\ new\ Object()) :\ D \\
&\rightarrow C.m(new\ Object()) : D \\
&\rightarrow new\ D()
\end{aligned}
$$

เมื่อแปลงกลับไปเป็นโปรแกรมที่เทียบเท่าการทำงานได้ปกติ นั่นคือโปรแกรมจะสามารถทำงานได้ถูกต้อง

3.2 การเรียกใช้เมธอดเชิงเสมือน

เมธอดเชิงเสมือน (Virtual Method Call) คือการเรียกใช้เมธอดระดับวัตถุในภาษาจาวา ในส่วนนี้จะเริ่มต้นด้วยการอธิบายความหมายของ invokevirtual โดยใช้ไวยากรณ์ของแคลคูลัสสำหรับจาวา FeatherweightJava เช่นเดียวกับ Static Method Call และตามด้วยการแปลง Virtual Method Call ไปเป็น invokedynamic

ความหมายของ invokevirtual มีดังต่อไปนี้ กำหนดให้มีโปรแกรม ตามรูปแบบ Featherweight Java

```
class C extends Object {
  D m(Object arg1) {
    return this.getD();
  }
  D getD() {
    return new D();
  }
}


class D extends Object {
}


new C().m(new Object());
```

มหาวิทยาลัยเทคโนโลยีสุรนารี

โปรแกรมสำหรับเรียกใช้เมธอด m ของวัตถุคลาส C ซึ่งคืนค่าเป็นการเรียกใช้เมธอด getD() ของตัวมันเอง และในเมธอด getD() นั้นคืนค่าเป็นวัตถุคลาส D โดยจะสามารถประมวลผลได้ตามขั้นตอนดังต่อไปนี้

$$new\ C().m(new\ Object()) \,/\, [this :\ new\ C(),\ arg1 :\ new\ Obect()]$$
$$\rightarrow new\ C().getD() \,/\, [this :\ new\ C()]$$
$$\rightarrow new\ D()$$

ในภาษา JVML การเรียกใช้เมธอด new C().m(new Object()) สามารถคอมไฟล์เป็น invokevirtual ได้ด้วยกฎดังนี้

กฎการคอมไพล์เมธอดเชิงเสมือน

$$\frac{new\ C().m(\bar{e}) : D \quad this = new\ C()}{invokevirtual(C, m, this, \bar{e}) : Object}$$

กฎการแปลง invokevirtual ไปเป็น invokedynamic

$$\frac{mh = findV\,irtual(C, m, D, typesof(\bar{e})) \quad this = new\ C()}{invokevirtual(C, m, this, \bar{e}) : Object \rightarrow invokedynamic(mh, this, \bar{e}) : D}$$

โดย mh เป็นตัวจัดการเมธอด
C และ D เป็นคลาส
m เป็นเมธอด
$this$ เป็นนิพจน์ this
$\bar{e}$ เป็นรายการนิพจน์ของอากิวเมนต์

ในขั้นตอนของคอมไพเลอร์ javac ปกติ จะเป็นการแปลงการเรียกใช้ภาษาจาวา เป็นภาษา JVML โดยการเรียกใช้เมธอดเชิงเสมือนของ

$$new\ C().m(new\ Object())$$

ได้ด้วย

มหาวิทยาลัยเทคโนโลยีสุรนารี

$$invokevirtual(C,\ m,\ new\ C(), new\ Object()) : Object$$

ในขั้นตอนของคอมไพเลอร์ภาษาจาวาที่เสนอในเอกสารฉบับนี้จะทำงานตามกฎนี้ ซึ่งด้วยกฎการแปลง invokevirtual เราสามารถแทนที่การเรียกใช้

$$invokevirtual(C,\ m,\ new\ C(),\ new\ Object()) : Object$$

ได้ด้วย

$$invokedynamic((name \Rightarrow m,\ type \Rightarrow (C,\ Object) :\ D),\ new\ C(),\ new\ Object()) : Object$$

ในลักษณะเดียวกันกับ invokestatic จะสังเกตได้ว่า invokevirtual คืนค่าเป็น Object เนื่องจากเป็นสิ่งที่ระบุไว้ในข้อกำหนดภาษา JVML สำหรับขั้นตอนของการประเมินโปรแกรมจะทำให้โปรแกรมที่แปลงแล้วด้วยคอมไพเลอร์ซึ่งอยู่ในรูปแบบ

```
class C extends Object {
  D m(Object arg1) {
    return this.getD();
  }
  D getD() {
    return new D();
  }
}


class D extends Object {
}


invokedynamic((name ⇒m, type ⇒(C, Object): D),
               new C(), new Object()): Object
```

โดยคู่ลำดับ (name ⇒ m, type ⇒ (C, Object): D) คือโครงสร้าง NameAndType ตามข้อกำหนดใน JVML และทำการกำหนดให้มีตารางการแปลง NameAndType ไปเป็นตัวจัดการเมธอดดังนี้

มหาวิทยาลัยเทคโนโลยีสุรนารี

(name ⇒ m, type ⇒ (C, Object): D) **match** { (name, type) -> **new** MH(type, C.m) }

คู่ลำดับ (name => m, type => (C, Object): D) จะแปลงเป็นวัตถุของคลาส MH ซึ่งเป็นตัวจัดการเมธอด โดยเป็นผลลัพท์จากการ findVirtual

จากการเตรียมการดังกล่าว จะสามารถประเมินโปรแกรมได้ด้วยขั้นตอนต่อไปนี้

$$
\begin{aligned}
&invokedynamic((name \Rightarrow m, type \Rightarrow (C,\ Object) : D), new\ C(), new\ Object()) :\ Object \\
&\rightarrow invokedynamic(new\ MH((C, Object) : D,\ C.m), new\ C(), new\ Object()) :\ Object \\
&\rightarrow invokedynamic(C.m, new\ C(), new\ Object()) :\ D\ \{new\ C()/this\} \\
&\rightarrow new\ C().m(new\ Object()) : D\ \{new\ C()/this,\ new\ Object()/arg1\} \\
&\rightarrow new\ C().getD()\ \{new\ C()/this\} \\
&\rightarrow new\ D()
\end{aligned}
$$

การเพิ่มความหมายของ this ลงไปใน invokedynamic เป็นเพราะในการสร้างจริงนั้นตัวจัดการเมธอดสำหรับ invokevirtual จะมีลักษณะการโยงอาร์กิวเมนต์ตัวแรก (binding) ให้เป็นวัตถุ this ตามการเรียกใช้เมธอดปกติ

เมื่อแปลงกลับไปเป็นโปรแกรมที่เทียบเท่าการทำงานได้ปกติ นั่นคือโปรแกรมจะสามารถทำงานได้ถูกต้อง

3.3 การเรียกใช้เมธอดเชิงอินเตอร์เฟส

การเรียกใช้เมธอดเชิงอินเตอร์เฟส (Interface Method Call) คือการเรียกใช้เมธอดบนอินเตอร์เฟสในภาษาจาวา ในส่วนนี้จะเริ่มต้นด้วยการอธิบายความหมายของ invokeinterface โดยใช้ไวยากรณ์ของแคลคูลัสสำหรับจาวา Featherweight Java เช่นเดียวกับ Static Method Call และ Virtual Method Call และตามด้วยการแปลง Interface Method Call ไปเป็น invokedynamic

ความหมายของ invokeinterface สามารถอธิบายได้ดังต่อไปนี้ กำหนดให้มีโปรแกรม ตามรูปแบบ Featherweight Java

```
interface I {
  D m(Object arg1);
}


class C extends Object implements I {
  D m(Object arg1) {
    return this.getD();
  }
  D getD() {
    return new D();
  }
}


class D extends Object {
}


(I)new C().getD();
```

ในตัวอย่างแรกจะเป็นการเรียกใช้เมธอดไม่สำเร็จ ผลการทำงานเป็น stuck โดยโปรแกรมตัวอย่างจะพยายามเรียกใช้เมธอด getD ของคลาส C ผ่านการแคสท์เป็นอินเตอร์เฟส I ซึ่งสามารถแสดงขั้นตอนการประมวลผลที่เกิด stuck ได้ดังต่อไปนี้

→ *(I)new C().getD() {C implements I ok}*
→ *(new C() as I).getD()*
→ *stuck*

```
interface I {
  D m(Object arg1);
}


class C extends Object implements I {
  D m(Object arg1) {
    return this.getD();
  }
  D getD() {
    return new D();
  }
}
```

มหาวิทยาลัยเทคโนโลยีสุรนารี

```
class D extends Object {
}


(I)new C().m(new Object());
```

ในตัวอย่างถัดมา โปรแกรมสำหรับเรียกใช้เมธอด m ของอินเตอร์เฟส I ซึ่งคืนค่าเป็นวัตถุ D โดยมีขั้นตอนการแคสท์วัตถุ new C() ให้เป็นอินเตอร์เฟส I ก่อนการเรียกใช้เมธอด m ในตัวอย่างนี้เป็นการประมวลผลที่สำเร็จ เนื่องจากการเรียกใช้เมธอด m สามารถทำได้ผ่านอินเตอร์เฟส I โดยสามารถประมวลผลได้ตามขั้นตอนดังต่อไปนี้

$\rightarrow (I)new\ C().m(new\ Object())\ \{C\ implements\ I\ ok\}$
$\rightarrow (new\ C()\ as\ I).m(new\ Object())\ \{new\ C()/this,\ new\ Object()/args1\}$
$\rightarrow new\ D()$

จากตัวอย่างที่ผ่านมาเทียบกับตัวอย่างของการเรียกใช้เมธอดแบบ Virtual จะเห็นได้ว่ามีการอ้างอิงถึงวัตถุคลาส C ในสองลักษณะที่ต่างกันคือ

new C()

และ

(new C() as I)

โดย new C() เป็นวัตถุคลาส C ที่มองจากภายนอกจะเห็นเมธอด m และเมธอด getD ตามนิยามของคลาส C ในขณะที่ (new C() as I) จะเป็นวัตถุคลาส C ที่มองจากภายนอกเห็นเพียงเมธอด m ตามนิยามของอินเตอร์เฟส I

วัตถุ (new C() as I) สามารถสร้างได้ด้วย *กฎการแคสท์ชนิดวัตถุเป็นอินเตอร์เฟส* โดยมีเงื่อนไขการแคสท์ตามนิพจน์ (I)new C() และมีการนิยามคลาส C ไว้ในตารางคลาส CT โดยที่ C จะต้อง อิมพลีเมนต์อินเตอร์เฟส I แล้วสมเหตุสมผล

ในภาษา JVML การเรียกใช้เมธอด (new C() as I).m(new Object()) สามารถคอมไฟล์เป็น invokeinterface ได้ด้วยกฎดังนี้

กฎการคอมไพล์การเรียกใช้อินเตอร์เฟส

$$\frac{(new\ C()\ as\ I).m(\bar{e}): D \quad this = (new\ C()\ as\ I)}{invokeinterface(I, m, this, \bar{e}): Object}$$

กฎการแปลง invokeinterface ไปเป็น invokedynamic

$$\frac{mh = findVirtual(I, m, D, typesof(\bar{e})) \quad this = (new\ C()\ as\ I)}{invokeinterface(I, m, this, \bar{e}): Object \rightarrow invokedynamic(mh, this, \bar{e}): D}$$

โดย mh เป็นตัวจัดการเมธอด

C และ D เป็นคลาส

I เป็นอินเตอร์เฟส

m เป็นเมธอดที่ประกาศไว้ใน I

$this$ เป็นนิพจน์ this ที่แคสท์ชนิดเป็น I

$\bar{e}$ เป็นรายการนิพจน์ของอากิวเมนต์

ในขั้นตอนของคอมไพเลอร์ javac ปกติ จะเป็นการแปลงการเรียกใช้ภาษาจาวา เป็นภาษา JVML โดยการเรียกใช้เมธอดแบบ interface ของ

$$(new\ C()\ as\ I).m(new\ Object())$$

ได้ด้วย

$$invokeinterface(I,\ m,\ (new\ C()\ as\ I), new\ Object()) : Object$$

ในขั้นตอนของคอมไพเลอร์ภาษาจาวาที่เสนอในเอกสารฉบับนี้จะทำงานตามกฎนี้ ซึ่งด้วยกฎการแปลง invokeinterface เราสามารถแทนที่การเรียกใช้

$$invokeinterface(I,\ m,\ (new\ C()\ as\ I), new\ Object()) : Object$$

มหาวิทยาลัยเทคโนโลยีสุรนารี

ได้ด้วย

$$invokedynamic((name \Rightarrow m,\ type \Rightarrow (I,\ Object) :\ D),\\ (new\ C()\ as\ I),\ new\ Object()) : Object$$

ในลักษณะเดียวกันกับ invokestatic และ invokevirtual การเรียกใช้เมธอดแบบ invokeinterface คืนค่าเป็น Object ตามข้อกำหนดในภาษา JVML แต่จะต่างออกไปตรงที่ invokeinterface ทำงานกับอินเตอร์เฟส ในขณะที่ invokestatic และ invokevirtual ทำงานกับคลาส

สำหรับขั้นตอนของการประเมินโปรแกรมจะทำให้โปรแกรมที่แปลงแล้วด้วยคอมไพเลอร์ซึ่งอยู่ในรูปแบบ

```
interface I {
  D m(Object arg1);
}


class C extends Object implements I {
  D m(Object arg1) {
    return this.getD();
  }
  D getD() {
    return new D();
  }
}


class D extends Object {
}


invokedynamic((name⇒m, type⇒(I, Object): D),
              (new C() as I), new Object()): Object
```

โดยคู่ลำดับ (name => m, type => (I, Object): D) คือโครงสร้าง NameAndType ตามข้อกำหนดใน JVML และทำการกำหนดให้มีตารางการแปลง NameAndType ไปเป็นตัวจัดการเมธอดดังนี้

(name => m, type => (I, Object): D) **match** { (name, type) -> **new** MH(type, I.m) }

คู่ลำดับ (name => m, type => (I, Object): D) จะแปลงเป็นวัตถุของคลาส MH ซึ่งเป็นตัวจัดการเมธอด โดยเป็นผลลัพท์จากการ findInterface

จากการเตรียมการดังกล่าว จะสามารถประเมินโปรแกรมได้ด้วยขั้นตอนต่อไปนี้

$$
\begin{aligned}
&invokedynamic((name \Rightarrow m, type \Rightarrow (I,\ Object) : D),\\
&\qquad\qquad (new\ C()\ as\ I), new\ Object()) :\ Object\\
&\rightarrow invokedynamic(new\ MH((I, Object) : D,\ I.m), (new\ C()\ as\ I), new\ Object()) :\ Object\\
&\rightarrow invokedynamic(I.m, (new\ C()\ as\ I), new\ Object()) :\ D\ \{(new\ C()\ as\ I)/this\}\\
&\rightarrow (new\ C()\ as\ I).m(new\ Object()) : D\ \{new\ C()/this\ ,\ new\ Object()/args1\}\\
&\rightarrow new\ C().getD()\ \{new\ C()/this\}\\
&\rightarrow new\ D()
\end{aligned}
$$

ความจำเป็นในการเพิ่มความหมายของ this ลงไปใน invokedynamic ในประโยค

$$\rightarrow invokedynamic(I.m, (new\ C()\ as\ I), new\ Object()) :\ D\ \{(new\ C()\ as\ I)/this\}$$

เนื่องจากในการอิมพลีเมนต์จริงนั้นตัวจัดการเมธอด (method handle) สำหรับ invokevirtual จะมีลักษณะการ bind argument ตัวแรกให้เป็นวัตถุ this ตามการเรียกใช้เมธอดปกติ

เมื่อแปลงกลับไปเป็นโปรแกรมที่เทียบเท่าการทำงานได้ปกติ นั่นคือโปรแกรมจะสามารถทำงานได้ถูกต้อง

3.4 การเรียกใช้เมธอดสืบทอด

การเรียกใช้เมธอดสืบทอด (Inherited Method Call) คือการเรียกใช้เมธอดของซูเปอร์คลาสในภาษาจาวา จะเป็นการใช้ invokespecial เพื่อเรียกใช้เมธอดของคลาสแม่ ในส่วนนี้จะเริ่มต้นด้วยการอธิบายความหมายของ invokespecial โดยใช้ไวยากรณ์ของแคลคูลัสสำหรับจาวา Featherweight Java เช่นเดียวกับ Static Method Call และ Virtual Method Call และ Interface Method Call จากนั้นจะเป็นการอธิบายการแปลงการเรียกใช้ Inherited Method Call เป็น invokedynamic

มหาวิทยาลัยเทคโนโลยีสุรนารี

ความหมายของ invokespecial มีดังต่อไปนี้ กำหนดให้มีโปรแกรม ตามรูปแบบ Featherweight Java

```
class A extends Object {
  D m(Object args) {
    return new D();
  }
}

class C extends A {
  D m(Object arg1) {
    return super.m(arg1);
  }
}


class D extends Object {
}


new C().m(new Object());
```

ในตัวอย่างแรกจะเป็นการเรียกใช้เมธอดดังต่อไปนี้

$$\rightarrow new\ C().m(new\ Object())\ \{(new\ C()\ as\ A)/super, new\ Object()/arg1\}$$
$$\rightarrow (new\ C()\ as\ A).m(new\ Object())\ \{(new\ C()\ as\ A)/this, new\ Object()/arg1\}$$
$$\rightarrow new\ D()$$

มหาวิทยาลัยเทคโนโลยีสุรนารี

ในภาษา JVML การเรียกใช้เมธอด super.m(args) สามารถคอมไฟล์เป็น invokespcial ได้ด้วยกฎดังนี้

กฎการคอมไพล์การเรียกใช้เมธอดสืบทอด

$$\frac{super.m(\bar{e})\colon D \quad this{=}new\ C() \quad class\ C\ extends\ A\ \{...\}\ ok}{invokespecial(A, m, (this\ as\ A), \bar{e})\colon Object}$$

กฎการแปลง invokespecial ไปเป็น invokedynamic

$$\frac{mh = findSpecial(A, m, D, typesof(\bar{e})) \quad this = new\ C()}{invokespecial(A, m, (this\ as\ A), \bar{e})\colon Object \rightarrow invokedynamic(mh, this, \bar{e})\colon D}$$

โดย mh เป็นตัวจัดการเมธอด

A, C และ D เป็นคลาส

m เป็นเมธอด

$this$ เป็นนิพจน์ this

$super$ เป็นนิพจน์ super

$\bar{e}$ เป็นรายการนิพจน์ของอากิวเมนต์

ในขั้นตอนของคอมไพเลอร์ javac ปกติ จะเป็นการแปลงการเรียกใช้ภาษาจาวา เป็นภาษา JVML โดยการเรียกใช้เมธอดของการสืบทอด

$$super.m(new\ Object())$$

ในคลาส C ที่สืบทอดจาก A ได้ด้วย

$$invokespecial(A,\ m,\ new\ C(), new\ Object()) : Object$$

ในขั้นตอนของคอมไพเลอร์ภาษาจาวาที่เสนอในเอกสารฉบับนี้จะทำงานตามกฎนี้ ซึ่งด้วยกฎการแปลง invokespecial เราสามารถแทนที่การเรียกใช้

$$invokespecial(A,\ m,\ new\ C(), new\ Object()) : Object$$

มหาวิทยาลัยเทคโนโลยีสุรนารี

ได้ด้วย

$$invokedynamic((name \Rightarrow m,\ type \Rightarrow (A,\ Object) :\ D),\\ new\ C(),\ new\ Object()) : Object$$

สังเกตได้ว่า invokespecial จะต้องเรียกใช้ภายในคลาส และเป็นนิพจน์อิสระไม่ได้ นั่นคือการเรียกใช้จะต้องเริ่มจากนิพจน์อื่นที่เข้าไปทำงานภายในเมธอด สำหรับขั้นตอนของการ evaluate จะทำให้โปรแกรมที่แปลงแล้วด้วยคอมไพเลอร์ซึ่งอยู่ในรูปแบบ

```
class A extends Object {
  D m(Object args) {
    return new D();
  }
}

class C extends A {
  D m(Object arg1) {
    return invokedynamic((name⇒m, type⇒(A, Object): D),
                new C(), new Object()): Object
  }
}


class D extends Object {
}


new C().m(new Object());
```

โดยคู่ลำดับ (name => m, type => (A, Object): D) คือโครงสร้าง NameAndType ตามข้อกำหนดใน JVML และทำการกำหนดให้มีตารางการแปลง NameAndType ไปเป็นตัวจัดการเมธอดดังนี้

(name => m, type => (A, Object): D) **match** { (name, type) -> **new** MH(type, A.m) }

คู่ลำดับ (name => m, type => (A, Object): D) จะแปลงเป็นวัตถุของคลาส MH ซึ่งเป็นตัวจัดการเมธอด โดยเป็นผลลัพท์จากการ findVirtual

จากการเตรียมการดังกล่าว จะสามารถประเมินโปรแกรมได้ด้วยขั้นตอนต่อไปนี้

```
invokedynamic((name⇒m, type⇒(A, Object): D),
                new C(), new Object()): Object
```

$$
\begin{aligned}
&new\ C().m(new\ Object()) \\
&\rightarrow invokedynamic((name \Rightarrow m,\ type \Rightarrow (A, Object) : D),\ new\ C(),\ new\ Object()) : Object \\
&\rightarrow invokedynamic(new\ MH((A, Object) : D, A.m), new\ C(),\ new\ Object()) : Object \\
&\rightarrow invokedynamic(A.m,\ new\ C(),\ new\ Object()) :\ D\ \{(new\ C()\ as\ A)/this\} \\
&\rightarrow (new\ C()\ as\ A).m(new\ Object()) :\ D\ \{(new\ C()\ as\ A)/this,\ new\ Object()/arg1\} \\
&\rightarrow new\ D()
\end{aligned}
$$

การแทนค่า invokedynamic(A.m, new C(), ...): D ของนิพจน์ this ที่คลาสของนิพจน์ this (ในที่นี้คือ C) ต่างกับคลาสเจ้าของเมธอด (ในที่นี้คือ A) จะเป็นการแทนที่ด้วยวัตถุที่แคสท์แล้ว ตามตัวอย่างจะได้ (new C() as A) เนื่องจากมีความจำเป็นที่ต้องให้วัตถุ this มีข้อมูลของซูเปอร์คลาส เพื่อให้สามารถเลือกเมธอดของคลาสแม่มาเรียกใช้ได้

ในลักษณะเดียวกันกับการเรียกใช้เมธอดสำหรับ invokevirtual การเพิ่มความหมายของ this ลงไปใน invokedynamic เป็นเพราะในการสร้างจริงนั้นตัวจัดการเมธอดสำหรับ invokespecial จะมีลักษณะการโยงอาร์กิวเมนต์ตัวแรก (binding) ให้เป็นวัตถุ super / this ตามการเรียกใช้เมธอดปกติ

เมื่อแปลงกลับไปเป็นโปรแกรมที่เทียบเท่าการทำงานได้ปกติ นั่นคือโปรแกรมจะสามารถทำงานได้ถูกต้อง

3.5 การเรียกใช้คอนสตรักเตอร์

การเรียกใช้คอนสตรักเตอร์ (Constructor Call) คือการเรียกใช้คอนสตรักเตอร์ในภาษาจาวา การเรียกใช้คอนสตรักเตอร์ในภาษาจาวาจะเป็นการเรียกใช้ด้วย invokespecial เช่นเดียวกับ Inherited Method Call แต่มีข้อแตกต่างตรงที่การเรียกใช้คอนสตรักเตอร์เป็นการ invokespecial บนเมธอดพิเศษชื่อ <init>

มหาวิทยาลัยเทคโนโลยีสุรนารี

ความหมายของคอนสตรักเตอร์ มีดังต่อไปนี้ คอนสตรักเตอร์มีหน้าที่ตั้งค่าให้ฟิลด์ของวัตถุแต่ละฟิลด์ โดนกำหนดให้มีโปรแกรม

```
class C extends Object {
  D d;
  C() {
    this.d = new D();
  }
}


class D extends Object {
}


new C().d
```

$$
\begin{aligned}
&new\ C().d \\
&\rightarrow new\ C(d : new\ D()).d \\
&\rightarrow new\ D()
\end{aligned}
$$

สัญลักษณ์ new C(d: new D()) หมายความถึง วัตถุ new C() ที่มีฟิลด์ d และฟิลด์ d มีค่าเป็น new D() ในการอิมพลีเมนต์จริงของ Java Virtual Machine นั้น การเรียกใช้ invokespecial ของคอนสตรักเตอร์จะคืนค่าเป็นชนิด void (V) เนื่องจากลำดับของชุดคำสั่ง JVML มีลักษณะดังต่อไปนี้

```
aload 0                          # ทำการโหลดตัวแปร this เข้าสู่ stack
invokespecial(C, <init>): V      # เรียกใช้คอนสตรักเตอร์ของคลาส C
aload 0                          # ทำการโหลดตัวแปร this เข้าสู่ stack
return 0                         # คืนค่าตัวแปร this ที่เป็นวัตถุ
```

แต่ในแคลคูลัสที่ใช้จะทำการลดรูปการเรียกใช้ invokespecial สำหรับคอนสตรักเตอร์ให้สามารถคืนค่าได้ เพื่อให้ความหมายเทียบเท่าชุดคำสั่งดังกล่าว

มหาวิทยาลัยเทคโนโลยีสุรนารี

```
class C extends Object {
  D d;
  C() {
    this.d = new D();
  }
}


class D extends Object {
}


invokespecial(C, <init>, new C).d
```

นิพจน์ในแคลลูลัสที่เทียบเท่ากับชุดคำสั่งข้างต้นคือ

$$invokespecial(C, \ < init >, \ new \ C) : \ C$$

โดย new C หมายถึงนิพจน์การสร้างวัตถุแต่ยังไม่ถึงขั้นตอนการเรียกใช้คอนสตรักเตอร์ ซึ่งแตกต่างกับนิพจน์ new C() ที่แสดงถึงวัตถุสมบูรณ์ที่ผ่านการเรียกใช้ตอนสตรักเตอร์แล้ว

ในภาษา JVML การเรียกใช้เมธอด new C() สามารถคอมไฟล์เป็น invokespecial ได้ด้วยกฎดังนี้

กฎการคอมไพล์การเรียกใช้คอนสตรักเตอร์

$$\frac{new \ C(\bar{e})}{invokespecial(C, <init>, new \ C, \bar{e}): C}$$

กฎการแปลง invokespecial ไปเป็น invokedynamic

$$\frac{mh = findConstructor(C, typesof(\bar{e}))}{invokespecial(C, <init>, new \ C, \bar{e}): C \rightarrow invokedynamic(mh, new \ C, \bar{e}): C}$$

มหาวิทยาลัยเทคโนโลยีสุรนารี

โดย mh เป็นตัวจัดการเมธอด

C เป็นคลาส

$new\ C$ เป็นนิพจน์วัตถุที่ยังไม่ผ่านการสร้างของคอนสตรักเตอร์

$\bar{e}$ เป็นรายการนิพจน์ของอากิวเมนต์

ในขั้นตอนของคอมไพเลอร์ภาษาจาวาที่เสนอในเอกสารฉบับนี้จะทำงานตามกฎนี้ ซึ่งด้วยกฎการแปลง invokespecial สำหรับคอนสตรักเตอร์สามารถแทนที่การเรียกใช้

$$invokespecial(C,\ < init >,\ new\ C) : C$$

ได้ด้วย

$$invokedynamic((name \Rightarrow < init >,\ type \Rightarrow () :\ C), new\ C) : C$$

โดยคู่ลำดับ (name => <init>, type => (): C) คือโครงสร้าง NameAndType ตามข้อกำหนดใน JVML โดยมีการระบุ name เป็น <init> เนื่องจาก <init> เป็นชื่อเฉพาะของเมธอดที่แทนคอนสตรักเตอร์ในภาษา JVML และทำการกำหนดให้มีตารางการแปลง NameAndType ไปเป็นตัวจัดการเมธอดดังนี้

(name => <init>, type => (): C) **match** { (name, type) -> **new** MH(type, C.<init>) }

คู่ลำดับ (name => <init>, type => (): C) จะแปลงเป็นวัตถุของคลาส MH ซึ่งเป็นตัวจัดการเมธอด โดยเป็นผลลัพท์จากการ findConstructor

จากการเตรียมการดังกล่าว จะสามารถประเมินโปรแกรมได้ด้วยขั้นตอนต่อไปนี้

มหาวิทยาลัยเทคโนโลยีสุรนารี

```
class C extends Object {
  D d;
  C() {
    this.d = new D();
  }
}


class D extends Object {
}


invokedynamic((name⇒<init>, type⇒(): C), new C).d
```

$$invokedynamic((name \Rightarrow < init >, type \Rightarrow () : C),\ new\ C).d$$
$$\rightarrow invokedynamic(new\ MH(() : C,\ C.< init >),\ new\ C).d$$
$$\rightarrow new\ C(d :\ new\ D()).d$$
$$\rightarrow new\ D()$$

การเรียกใช้คอนสตรักเตอร์นั่นมีโอกาสทำให้เกิดผลข้างเคียง (side-effect) ซึ่งทำให้สถานะของวัตถุเปลี่ยน โดยกำหนดให้ new C(d: new D()) แทนวัตถุคลาส C ที่เกิดผลข้างเคียงมีค่าฟิลด์ d เป็นวัตถุ new D()

ดังนั้นเมื่อมีการเรียกดึงค่าฟิลด์ด้วย .d บนวัตถุดังกล่าวจะเกิดการ evaluate ให้คืนค่าเป็นค่าวัตถุ new D() ของฟิลด์ d

เมื่อแปลงกลับไปเป็นโปรแกรมที่เทียบเท่าการทำงานได้ปกติ นั่นคือโปรแกรมจะสามารถทำงานได้ถูกต้อง

มหาวิทยาลัยเทคโนโลยีสุรนารี

3.6 ฟังก์ชันช่วยเหลือ (Helper Functions)

ฟังก์ชันช่วยเหลือ คือฟังก์ชันสำหรับใช้ร่วมกับกฎการประเมินเพื่อให้กฎการประเมินสามารถใช้อธิบายการประเมินของโปรแกรมได้อย่างกระชับและรัดกุมมากขึ้น

3.6.1 การแมชต์และกำหนดตัวจัดการเมธอด

กำหนดให้

mh = (name => m, type => (Object): D)

(name => m, type => (Object): D) **match** { (name, type) -> **new** MH(type, C.m) }

3.6.2 ฟังก์ชัน lookup

lookup(CT, m, t) {

คืนค่า CT[C] ที่บรรจุเมธอด m ซึ่งมี signature เป็นชนิด t

}

3.6.3 ฟังก์ชัน typeof

typeof(e) {

คืนค่า C เมื่อ $\frac{e = new\ C(\bar{x})}{C}$

}

3.6.4 ฟังก์ชัน typesof

typesof(e) {

คืนค่าเป็น เซ็ตว่าง ถ้า e == null

คืนค่าเป็น typeof(head(e)) + typesof(tail(e))

}

มหาวิทยาลัยเทคโนโลยีสุรนารี

3.6.4 ฟังก์ชัน body

body(m, C) {
    คืนค่าเป็นร่างเมธอด m ของคลาส C
}

3.6.5 ฟังก์ชัน field

field(f, C) {
    คืนค่าเป็นคู่ลำดับนิยามของฟิลด์ f ของคลาส C
}

มหาวิทยาลัยเทคโนโลยีสุรนารี

3.7 กฎการประเมิน (Evaluation Rules)

กฎการประเมิน คือกฎที่ใช้ ...

กฎการ lookup ตัวจัดการเมธอด:

$$\frac{(name \Rightarrow m, type \Rightarrow (Object):D) \quad C = lookup(CT, m, type)}{new\ MH(type, C.m)}$$

กฎการลบตัวจัดการเมธอด (Method Handle Erasure) แบบ static:

$$\frac{invokedynamic(new\ MH(t, C.m), \bar{e}): Object \quad D{=}rettype(t) \quad typesof(\bar{e}){=}argtypes(t)}{invokedynamic(C.m, \bar{e}): D}$$

กฎการลบตัวจัดการเมธอด (Method Handle Erasure) แบบ virtual:

$$\frac{invokedynamic(new\ MH(t, C.m), this, \bar{e}): Object \quad D{=}rettype(t) \quad typesof(this \oplus \bar{e}){=}argtypes(t)}{invokedynamic(C.m, this, \bar{e}): D}$$

กฎการ bind this สำหรับ invokedynamic:

$$\frac{invokedynamic(C.m, this, \bar{e}): D \quad typeof(this){=}C}{this.m(\bar{e}){:}D}$$

กฎการเรียกใช้เมธอดจริงของ invokestatic:

$$\frac{invokedynamic(C.m, \bar{e}): D}{C.m(\bar{e}): D}$$

กฎการแคสท์ชนิดของวัตถุเป็นอินเตอร์เฟส:

$$\frac{(I)new\ C() \quad class\ C\ extends\ D\ implements\ I\ \{...\}\ ok}{(new\ C()\ as\ I)}$$

มหาวิทยาลัยเทคโนโลยีสุรนารี

กฎการเรียกใช้เมธอดแบบ virtual:

$$\frac{body(m, C) = x \Rightarrow e_0 \qquad method(m, I)\ ok}{new\ C().m(\bar{e}) \rightarrow \{\bar{e}/x, new\ C()/this\}e_0}$$

กฎการเรียกใช้เมธอดปกติ:

$$\frac{body(m, C) = x \Rightarrow e_0}{C.m(\bar{e}) \rightarrow \{\bar{e}/x\}e_0}$$

กฎการเรียกใช้เมธอดบนอินเตอร์เฟส:

$$\frac{body(m, C) = x \Rightarrow e_0}{(new\ C()\ as\ I).m(\bar{e}) \rightarrow \{\bar{e}/x, new\ C()/this\}e_0}$$

กฎการเรียกใช้เมธอดของซูเปอร์คลาส:

$$\frac{body(m, C) = x \Rightarrow e_0 \qquad class\ C\ extends\ A\ \{...\}\ ok}{new\ C().m(\bar{e}) \rightarrow \{\bar{e}/x, (new\ C()\ as\ A)/super, new\ C()/this\}e_0}$$

กฎการดึงค่าฟิลด์

$$\frac{field(f, C) = D\ f}{new\ C(\bar{f}: \bar{e}).d \rightarrow e}$$

กฎการเรียกใช้ invokedynamic สำหรับคอนสตรักเตอร์:

$$\frac{(name \Rightarrow <init>, type \Rightarrow (\overline{C}):C) \qquad C = lookup(CT, <init>, type)}{invokedynamic(new\ MH(():C, C.<init>), new\ C, \bar{e}) \rightarrow new\ C(\bar{f}: \bar{e})}$$

มหาวิทยาลัยเทคโนโลยีสุรนารี

ในบทนี้ได้กล่าวถึงความหมายของการเรียกใช้เมธอดในรูปแบบต่าง ๆ ของภาษาจาวา รวมทั้งกฎการแปลงการเรียกใช้เหล่านั้นให้อยู่ในรูปของการเรียกใช้ด้วยชุดคำสั่ง invokedynamic สำหรับในบทถัดไปจะกล่าวถึงการทดสอบทั้งในเชิงความถูกต้องและเชิงประสิทธิภาพของโปรแกรมที่ได้รับการแปลงเป็น invokedynamic เรียบร้อยแล้ว

บทที่ 4
การทดสอบ

การทดสอบมีสองแนวทางคือ การทดสอบความถูกต้องและการทดสอบประสิทธิภาพ โดยมีการทดสอบโดยใช้ชุดตัววัด (Benchmark) ซึ่งตัววัดในชุดดังกล่าวเป็นโปรแกรมภาษาจาวาที่ทำการแปลงเป็นด้วยคอมไพเลอร์ที่พัฒนาขึ้น และมีการเรียกใช้คำสั่ง invoke ในรูปแบบต่าง ๆ

การทดสอบความถูกต้องของโปรแกรมกำหนดโดยให้สามารถแปลง และสั่งให้โปรแกรมตัววัด ที่เตรียมขึ้นทำงานได้ถูกต้องทั้งหมด ในขณะเดียวกันก็วัดประสิทธิภาพของโปรแกรมที่แปลงแล้วเทียบกับโปรแกรมที่ยังไม่ได้แปลง ตัววัด ที่นำมาทดสอบเป็นระบบตัววัดประสิทธิภาพของ SciMark 2.0 (Pozo and Miller, 2011) ซึ่งเป็นที่ยอมรับและใช้กันแพร่หลายในการวัดประสิทธิภาพของคอมไพเลอร์และระบบรันไทม์สำหรับภาษาจาวา

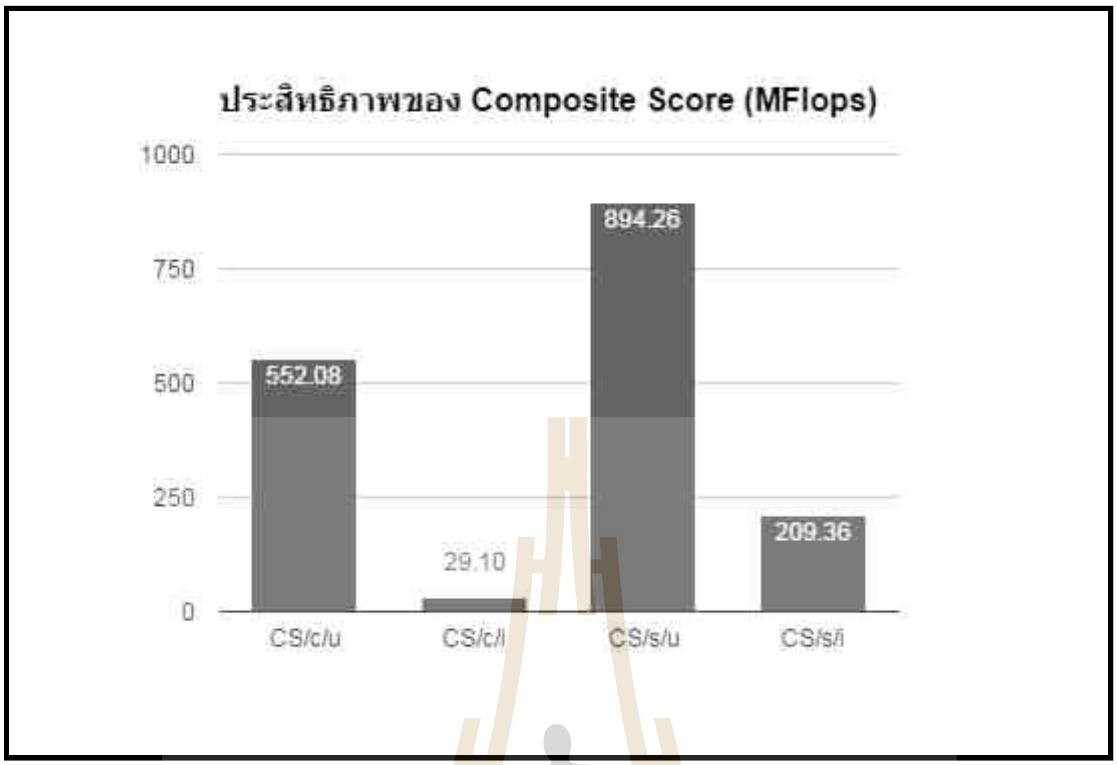
ในการทดสอบมีการตั้งสมมติฐานไว้ว่า การทดสอบความถูกต้องของโปรแกรมควรเป็น 100% และ ประสิทธิภาพของแต่ละตัววัดเมื่อโปรแกรมใช้ชุดคำสั่ง invokedynamic จะอยู่ในระดับเกิน 20% ของโปรแกรมที่ไม่ได้คอมไพล์เป็น invokedynamic

4.1 ผลการทดลองการวัดประสิทธิภาพของ Composite Score

ตัววัด Composite Score เป็นตัววัดในชุด SciMark 2 โดยมี 4 คอนฟิกกูเรชันในการวัดประสิทธิภาพดังนี้

CS/c/u	คือ Composite Score ที่รันด้วย JVM ใน client โหมดและไม่ได้ผ่านการแปลง
CS/c/i	คือ Composite Score ที่รันด้วย JVM ใน client โหมดและผ่านการคอมไพล์ให้เป็น invokedynamic แล้ว
CS/s/u	คือ Composite Score ที่รันด้วย JVM ใน server โหมดและไม่ได้ผ่านการแปลง
CS/s/i	คือ Composite Score ที่รันด้วย JVM ใน server โหมดและผ่านการคอมไพล์ให้เป็น invokedynamic แล้ว

มหาวิทยาลัยเทคโนโลยีสุรนารี



รูปที่ 4.1 กราฟแสดงการวัดประสิทธิภาพของ Composite Score ใน SciMark 2.0

ในการทดลองที่ 4.1 นี้วัดหน่วยของประสิทธิภาพเป็น MFlops โดยค่าสูงจะแสดงถึงประสิทธิภาพมาก จะเห็นได้ว่า client โหมดของ JVM ที่ทำงานด้วยชุดคำสั่ง invokedynamic (CS/c/i) นั้นประสิทธิภาพยังไม่สูงเมื่อเทียบกับโปรแกรมที่ไม่ได้ผ่านการแปลง (CS/c/u) ในขณะที่ server โหมดของ JVM ที่ทำงานด้วยชุดคำสั่ง invokedynamic (CS/s/i) มีประสิทธิภาพสูงถึง 1 ใน 5 ของโปรแกรมที่ไม่ได้ผ่านการแปลง (CS/s/u)

4.2 ผลการทดลองการวัดประสิทธิภาพของ FFT

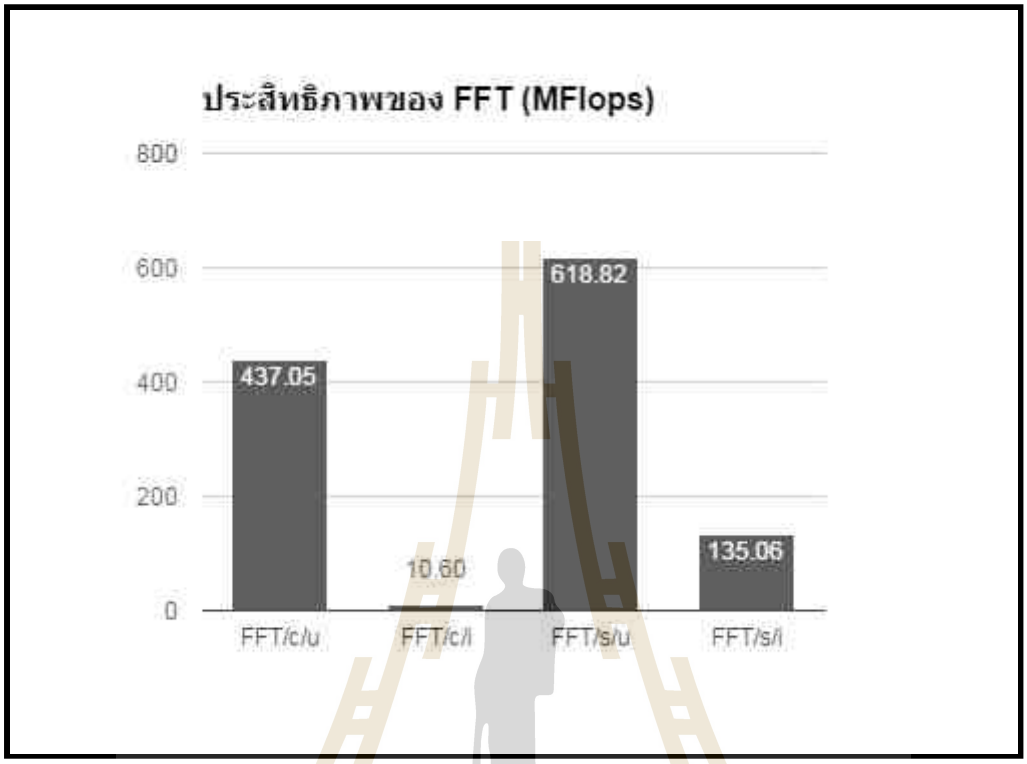
ตัววัด FFT เป็นตัววัดที่อยู่ในชุด SciMark 2.0 เช่นเดียวกับ Composite Score โดย FFT มีลักษณะที่เน้นการคูณเมตริกซ์ที่เป็นจำนวนทศนิยม ตัววัดมีคอนฟิกกูเรชัน 4 แบบดังนี้

FFT/c/u คือ FFT ที่รันด้วย JVM ใน client โหมดและไม่ได้ผ่านการแปลง

FFT/c/i คือ FFT ที่รันด้วย JVM ใน client โหมดและผ่านการคอมไพล์ให้เป็น invokedynamic แล้ว

FFT/s/u คือ FFT ที่รันด้วย JVM ใน server โหมดและยังไม่ผ่านการแปลง

FFT/s/i คือ FFT ที่รันด้วย JVM ใน server โหมดและผ่านการคอมไพล์ให้เป็น invokedynamic แล้ว



รูปที่ 4.2 กราฟแสดงการวัดประสิทธิภาพของ FFT ใน SciMark 2.0

เช่นเดียวกับการทดลองที่ 4.1 ในการทดลองที่ 4.2 นี้ประสิทธิภาพของโปรแกรมมีหน่วยวัดเป็น MFlops โดยค่ามากหมายถึงมีประสิทธิภาพสูง จะเห็นได้ว่าคอนฟิกกูเรชัน FFT/s/i มีความเร็วถึง 1 ใน 5 ของคอนฟิกกูเรชัน FFT/s/u ซึ่งสอดคล้องกับการทดลอง 4.1

4.3 ผลการทดลองการวัดประสิทธิภาพของ Monte Carlo

ตัววัด Monte Carlo เป็นอีกหนึ่งตัววัดที่อยู่ใน SciMark 2.0 ซึ่งเป็นการอิมพลีเมนต์อัลกอริทึม Monte Carlo ตามชื่อตัววัด ในลักษณะเดียวกันกับตัววัดอื่นของชุด SciMark 2.0 การทดลองมี 4 คอนฟิกกูเรชันดังต่อไปนี้

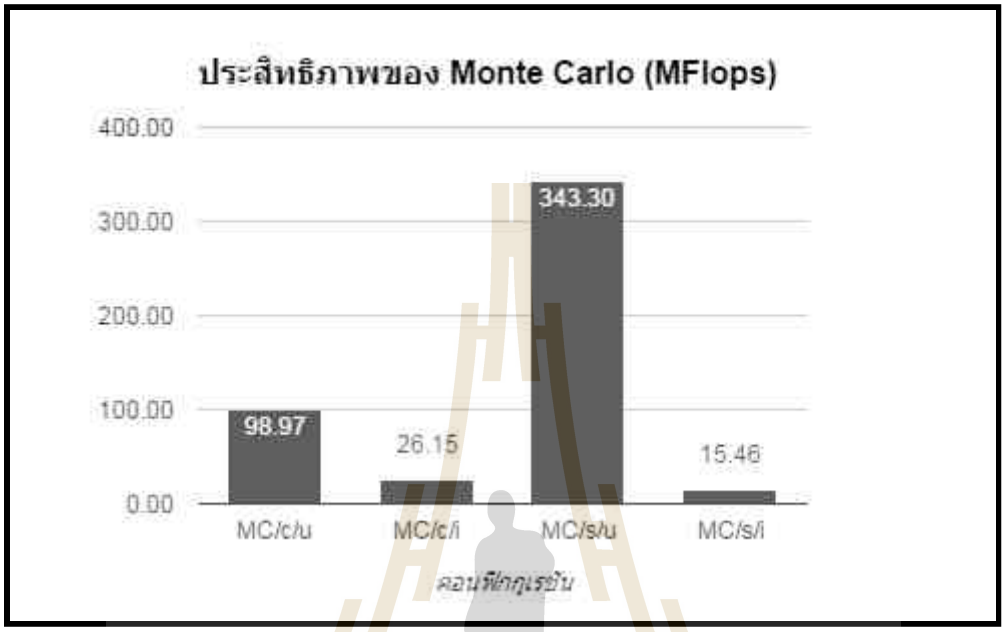
MC/c/u คือ Monte Carlo ที่รันด้วย JVM ใน client โหมดและยังไม่ผ่านการคอมไพล์

MC/c/i คือ Monte Carlo ที่รันด้วย JVM ใน client โหมดและผ่านการคอมไพล์เป็น invokedynamic แล้ว

มหาวิทยาลัยเทคโนโลยีสุรนารี

MC/s/u คือ Monte Carlo ที่รันด้วย JVM ใน server โหมดและยังไม่ผ่านการคอมไพล์

MC/s/i คือ Monte Carlo ที่รันด้วย JVM ใน server โหมดและผ่านการคอมไพล์เป็น invokedynamic แล้ว



รูปที่ 4.3 กราฟแสดงการวัดประสิทธิภาพของ Monte Carlo ใน SciMark 2.0

4.4 ผลการทดลองการวัดประสิทธิภาพของ SOR

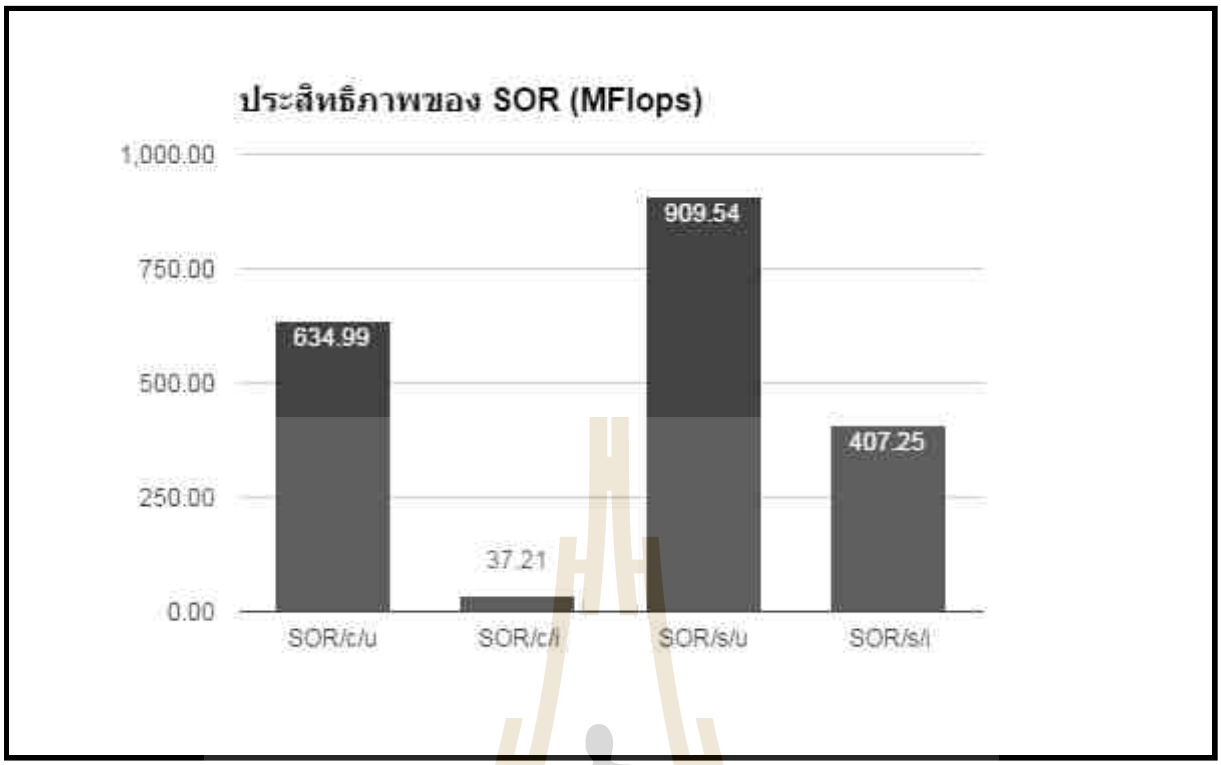
ตัววัด SOR เป็นตัววัดในชุด SciMark ที่เน้นการประมวลผลที่ใช้การวนรอบและเลขจำนวนเต็ม ในลักษณะเดียวกันกับตัววัดอื่นของชุด SciMark 2.0 การทดลองมี 4 คอนฟิกกูเรชันดังต่อไปนี้

SOR/c/u คือ SOR ที่รันด้วย JVM ใน client โหมดและยังไม่ผ่านการคอมไพล์

SOR/c/i คือ SOR ที่รันด้วย JVM ใน client โหมดและผ่านการคอมไพล์เป็น invokedynamic แล้ว

SOR/s/u คือ SOR ที่รันด้วย JVM ใน server โหมดและยังไม่ผ่านการคอมไพล์

SOR/s/i คือ SOR ที่รันด้วย JVM ใน server โหมดและผ่านการคอมไพล์เป็น invokedynamic แล้ว



รูปที่ 4.4 กราฟแสดงการวัดประสิทธิภาพของ SOR ใน SciMark 2.0

จากรูปที่ 4.4 ประสิทธิภาพของ SOR ใน server โหมดที่ผ่านการคอมไพล์เป็น invokedynamic (SOR/s/i) แล้วนั้นมีประสิทธิภาพเกือบ 45% ของโปรแกรมที่ยังไม่ผ่านการคอมไพล์ (SOR/s/u) แสดงให้เห็นว่า JVM สามารถประมวลผลประสิทธิภาพของชุดคำสั่ง invokedynamic สำหรับการคำนวณแบบจำนวนเต็มได้ดีกว่าการคำนวณแบบทศนิยม

4.5 ผลการทดลองการวัดประสิทธิภาพของ Sparse Matmul

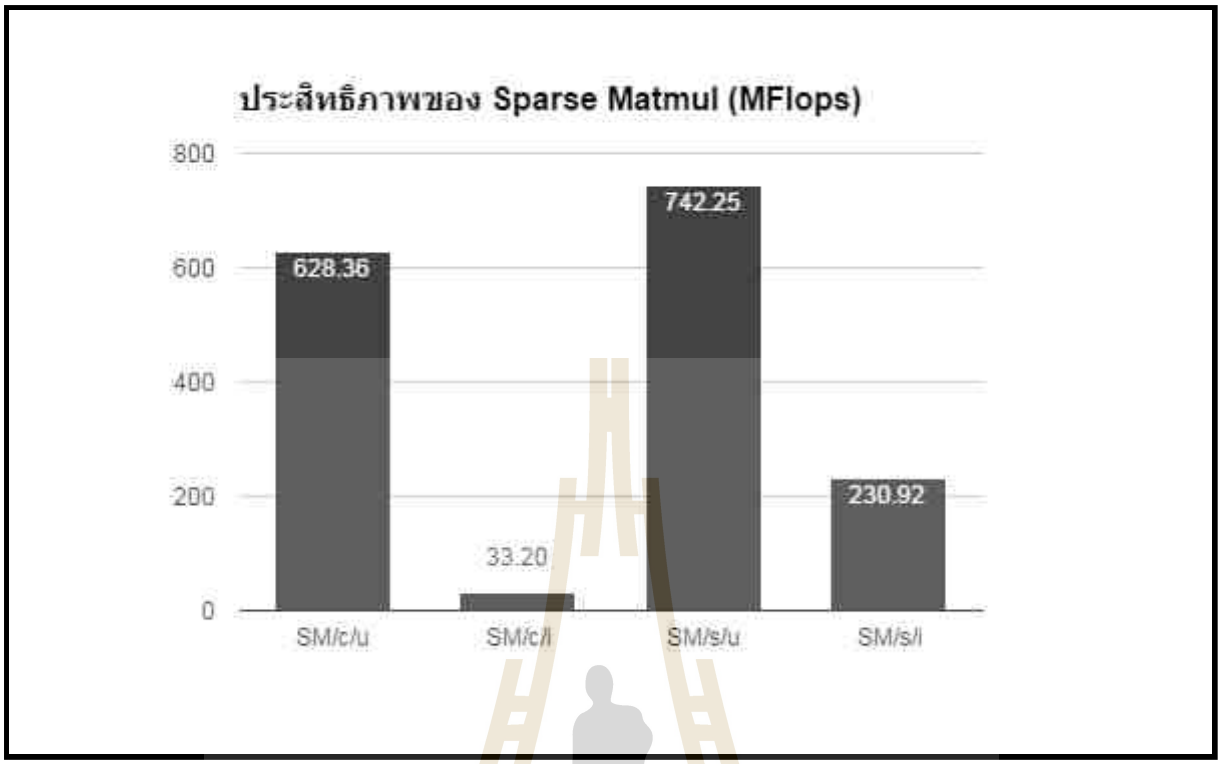
ตัววัด Sparse Matmul เป็นตัววัดในชุด SciMark ที่เน้นการประมวลผลอะเรย์สองมิติขนาดใหญ่ ในลักษณะเดียวกันกับตัววัดอื่น ๆ ในชุด SciMark คอนฟิกกูเรชันในการทดลองนี้มีจำนวน 4 คอนฟิกกูเรชันดังต่อไปนี้

SM/c/u คือ Sparse Matmul ที่รันด้วย JVM ใน client โหมดและยังไม่ผ่านการคอมไพล์

SM/c/i คือ Sparse Matmul ที่รันด้วย JVM ใน client โหมดและผ่านการคอมไพล์เป็น invokedynamic แล้ว

SM/s/u คือ Sparse Matmul ที่รันด้วย JVM ใน server โหมดและยังไม่ผ่านการคอมไพล์

SM/s/i คือ Sparse Matmul ที่รันด้วย JVM ใน server โหมดและผ่านการคอมไพล์เป็น invokedynamic แล้ว



รูปที่ 4.5 กราฟแสดงการวัดประสิทธิภาพของ Sparse Matmul ใน SciMark 2.0

จากรูปที่ 4.5 คอนฟิกกูเรชัน SM/s/i มีประสิทธิภาพเกือบ 1 ใน 3 ของ SM/s/u แสดงให้เห็นว่า JVM สามารถประมวลผลประสิทธิภาพของชุดคำสั่ง invokedynamic สำหรับการคำนวณอะเรย์สองมิติได้มีประสิทธิภาพกว่าสมมติฐาน

4.6 ผลการทดลองการวัดประสิทธิภาพของ LU

ตัววัด LU เป็นตัววัดในชุด SciMark ที่เน้นการแยกตัวประกอบ LU ซึ่งเป็นลักษณะเช่นเดียวกันกับตัววัดอื่น ๆ ในชุด SciMark คอนฟิกกูเรชันในการทดลองนี้มีจำนวน 4 คอนฟิกกูเรชัน ดังต่อไปนี้

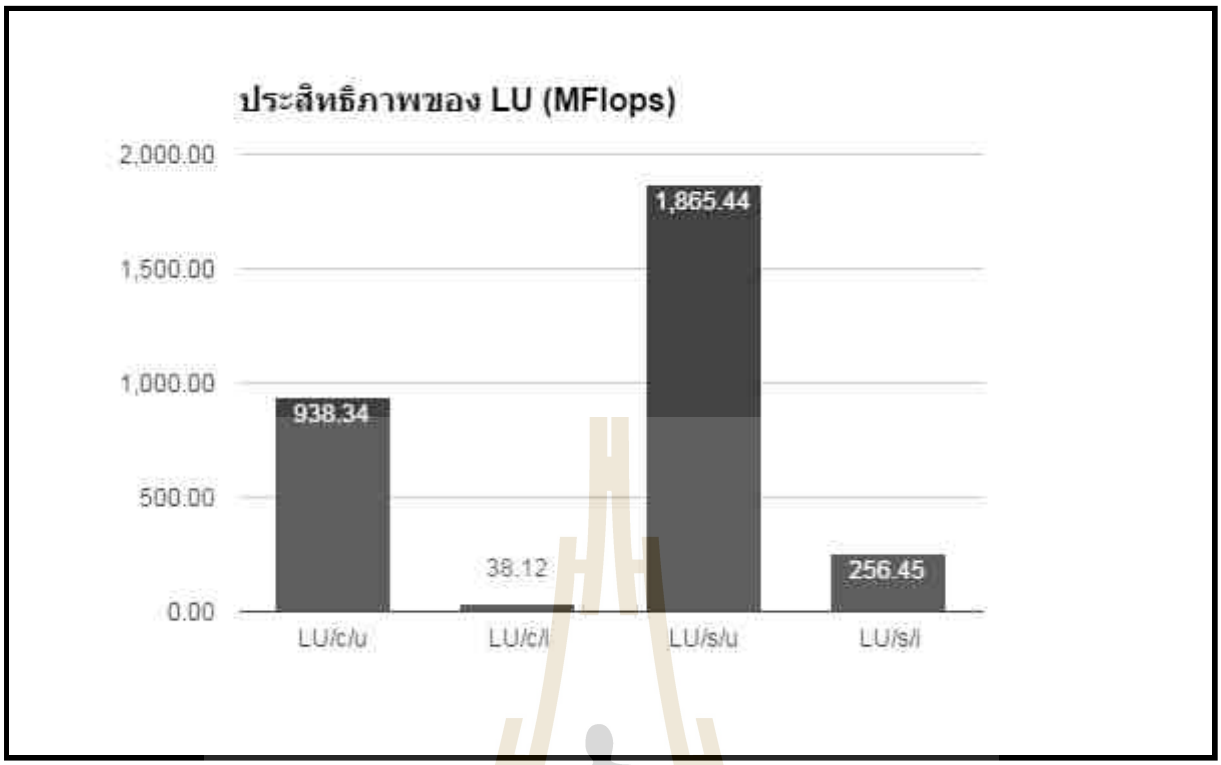
LU/c/u คือ LU ที่รันด้วย JVM ใน client โหมดและยังไม่ผ่านการคอมไพล์

LU/c/i คือ LU ที่รันด้วย JVM ใน client โหมดและผ่านการคอมไพล์เป็น invokedynamic แล้ว

LU/s/u คือ LU ที่รันด้วย JVM ใน server โหมดและยังไม่ผ่านการคอมไพล์

LU/s/i คือ LU ที่รันด้วย JVM ใน server โหมดและผ่านการคอมไพล์เป็น invokedynamic แล้ว

มหาวิทยาลัยเทคโนโลยีสุรนารี



รูปที่ 4.6 กราฟแสดงการวัดประสิทธิภาพของการแยกตัวประกอบ LU ใน SciMark 2.0

จากผลการทดสอบในรูปที่ 4.6 แสดงให้เห็นว่า ประสิทธิภาพของโปรแกรมในคอนฟิกกูเรชัน LU/s/i ซึ่งใช้ชุดคำสั่ง invokedynamic นั้นมีประสิทธิภาพเพียง 13% ของ LU/s/u ซึ่งเป็นโปรแกรมที่ไม่ได้คอมไพล์เป็นชุดคำสั่ง invokedynamic และไม่เป็นไปตามสมมติฐาน โดยการประมวลผลการแยกตัวประกอบ LU น่าจะมีลักษณะการคำนวณที่ซับซ้อนในลักษณะที่ JVM ไม่สามารถปรับปรุงคุณภาพ invokedynamic ของโปรแกรมนี้ได้เต็มที่เมื่อเทียบกับตัววัดในการทดลองที่ 4.1-4.5

ในบทนี้ได้กล่าวถึงการทดสอบความถูกต้องและประสิทธิภาพของโปรแกรมที่แปลงชุดคำสั่ง invokedynamic โดยพบว่าการแปลงโปรแกรมตัววัดทั้งหมดทำงานได้ถูกต้อง 100% และประสิทธิภาพของโปรแกรมที่รันด้วยคำสั่ง invokedynamic นั้นมีประสิทธิภาพเกิน 20% ของโปรแกรมที่ยังไม่ได้แปลง ซึ่งส่วนใหญ่เป็นไปตามสมมติฐาน ในบทถัดไปจะเป็นบทสรุปของเอกสารงานวิจัยนี้

บทที่ 5
บทสรุป

โปรแกรมที่สนับสนุน invokedynamic มีลักษณะพิเศษที่ยืดหยุ่น สามารถใช้พัฒนาแอพพลิเคชันที่ต้องการความเป็นพลวัตได้ ในงานวิจัยชิ้นนี้ได้ทำการพัฒนาคอมไพเลอร์สำหรับแปลงโปรแกรมในภาษาจาวาที่ผ่านการคอมไพล์ด้วย javac แล้วอีกครั้งหนึ่ง ให้มีการใช้งานชุดคำสั่ง invokedynamic เพื่อสนับสนุนคุณสมบัติความเป็นพลวัตดังกล่าว

เพื่ออธิบายถึงความหมายของกฎการแปลงโปรแกรมปกติให้เป็นโปรแกรมในรูปแบบ invokedynamic ในบทที่ 3 ได้มีการอธิบายกฎที่ใช้สร้างคอมไพเลอร์ตามรูปแบบแคลคูลัสภาษา Featureweight Java โดยกฎของการแปลงนั้นครอบคลุมการเรียกใช้เมธอดเชิงสถิตย์ การเรียกใช้เมธอดเชิงเสมือน การเรียกใช้เมธอดบนอินเตอร์เฟส การเรียกใช้เมธอดสืบทอด และการเรียกใช้คอนสตรักเตอร์ พร้อมทั้งยกตัวอย่างโปรแกรมที่สามารถแปลงได้อย่างถูกต้องด้วยกฎการแปลงโปรแกรม

ในบทที่ 4 ได้นำเสนอการวัดความถูกต้องและประสิทธิภาพของคอมไพเลอร์ที่พัฒนาขึ้นโดนชุดตัววัดนำมาจาก SciMark 2.0 Benchmark (Pozo and Miller, 2011) โดยนำตัววัดแต่ละตัวมาผ่านการแปลงด้วยคอมไพเลอร์ที่พัฒนาขึ้นและวัดประสิทธิภาพเทียบกับโปรแกรมที่ยังไม่ได้ทำการแปลง โดยตั้งสมมติฐานว่าความถูกต้องควรเป็น 100% และประสิทธิภาพควรเกิน 20% ของโปรแกรมเดิม จากการทดสอบสรุปได้ว่า ความถูกต้องของการแปลงเป็นไปตามสมมติฐาน ในขณะที่ประสิทธิภาพของโปรแกรมตัววัดที่แปลงเป็น invokedynamic แล้วนั้นส่วนใหญ่จะได้ประสิทธิภาพเกิน 20% มีเพียงตัววัดการแยกตัวประกอบ LU เท่านั้นที่ประสิทธิภาพต่ำ ทั้งนี้เกิดจากความซับซ้อนของอัลกอริทึมในโปรแกรมตัววัดที่ทำให้ JVM ไม่สามารถปรับปรุงประสิทธิภาพได้เต็มที่ในขณะรัน

จากผลการทดสอบพบว่า โหมด server ของ JVM ทำงานได้ดีกว่าโหมด client สำหรับการปรับปรุงประสิทธิภาพของชุดคำสั่ง invokedynamic อย่างชัดเจน และ JVM สามารถปรับปรุงประสิทธิภาพของการคำนวณเลขจำนวนเต็มที่ใช้ชุดคำสั่ง invokedynamic ได้ดีกว่าการประมวลผลทศนิยม ซึ่งเป็นพฤติกรรมที่ปกติ

มหาวิทยาลัยเทคโนโลยีสุรนารี

การศึกษาในระยะถัดไปนั้นสามารถทำได้โดยการนำ JVM รุ่น 8 และ 9 มาทดลองเทียบประสิทธิภาพ เนื่องจากมีแนวโน้มว่าระบบ invokedynamic ภายใน JVM ทั้งสองรุ่นจะได้รับการปรับปรุงประสิทธิภาพที่สูงขึ้น รวมทั้งการนำระบบที่พัฒนาขึ้นมาประยุกต์สร้างเป็นระบบเชิงพลวัตชนิดต่าง ๆ เช่น ระบบเชิงลักษณะ ระบบโมดูลเชิงวัตถุที่ทำงานเป็นอิสระต่อกันได้เป็นต้น

บรรณานุกรม

1. Bodden, E. Invokedynamic slower than reflection? (ca. 2010) URL: http://mail.openjdk.java.net/pipermail/mlvm-dev/2010-June/001769.html
2. Da Vinci Machine project, the. (ca. 2009). URL: http://openjdk.java.net/projects/mlvm.
3. Forax, R. The JSR-292 Backport Project (ca. 2009). URL: http://wiki.jvmlangsummit.com/images/b/b1/Jsr292-backport.pdf
4. Gosling, J., Joy, B., Steele, G., Barcha, G. and Buckley, A. 2013. The Java Language Specification. Java SE 7 Edition.
5. Hickey, R. et al. Clojure project (ca. 2015). URL: http://clojure.org.
6. Hugunin, J. et al. Jython project (ca. 2015). URL: http://www.jython.org.
7. Igarachi, A., Pierce, B. C., and Wadler, P. Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. v.23 n.3., pp. 396-450, May 2001.
8. Kaewkasi, C. An Aspect-Oriented Approach to Productivity Improvement for A Dynamic Language using Typing Concerns. PhD thesis, The University of Manchester, 2009.
9. Kiczales, G., Lamping, J., Mendhekar A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J. Aspect-Oriented Programming, In Proceedings of ECOOP'97, Lecture Notes in Computer Science, Springer, vol.1241, pp.220-242, 1997.
10. Koenig, D., Glover, A., King, P., Laforge, G., and Skeet, J. 2007. Groovy in Action. Manning Publications Co.
11. Lindholm, T. and Yellin, F. 1999. Java Virtual Machine Specification. 2nd. Addison-Wesley Longman Publishing Co., Inc.
12. Matsumoto, Y. et al. Ruby Documentation (ca. 2015). URL: https://www.ruby-lang.org/en/documentation/
13. Mozilla Corp. Rhino: JavaScript for Java (ca. 2015). URL: http://www.mozilla.org/rhino.
14. Nutter, C. et al. JRuby project (ca. 2015). URL: http://kenai.com/projects/jruby.

มหาวิทยาลัยเทคโนโลยีสุรนารี

15. Oracle Corp. 2010. The OpenJDK Project (ca. 2010). URL: http://openjdk.java.net/projects/
16. Ponge, J., and Mouël, F. L. (2012). JooFlux: Hijacking Java 7 InvokeDynamic To Support Live Code Modifications. arXiv preprint arXiv:1210.1039.
17. Pozo, R. and Miller, B. Java SciMark 2.0 (ca. 2011). URL: http://math.nist.gov/scimark2.
18. Rose, J. 2008. JSR 292: Supporting dynamically typed languages on the Java platform. http://jcp.org/en/jsr/detail?id=292.
19. Rose, J. 2009. 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. DOI=http://doi.acm.org/10.1145/1711506.1711508.
20. TIOBE Software. 2015. Programming Community Index (ca. 2015). URL: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

มหาวิทยาลัยเทคโนโลยีสุรนารี

ประวัติคณะผู้วิจัย

1. ชื่อ (ภาษาไทย) ดร. ชาญวิทย์ แก้วกสิ
   (ภาษาอังกฤษ) Dr. Chanwit Kaewkasi
2. เลขหมายบัตรประจำตัวประชาชน 3840100382000
3. ตำแหน่งปัจจุบัน ผู้ช่วยศาสตราจารย์
4. หน่วยงาน
   สาขาวิชาวิศวกรรมคอมพิวเตอร์
   สำนักวิชาวิศวกรรมศาสตร์
   มหาวิทยาลัยเทคโนโลยีสุรนารี
   044-224224
   chanwit@sut.ac.th
5. ประวัติการศึกษา

ปี	ระดับการศึกษา	อักษรย่อปริญญาและชื่อเต็ม	สาขาวิชา	สถาบัน
2539-2542	ปริญญาตรี	วศ.บ. / วิศวกรรมศาสตรบัณฑิต (เกียรตินิยมอันดับ 1)	วิศวกรรมคอมพิวเตอร์	มหาวิทยาลัยเทคโนโลยีสุรนารี
2544-2546	ปริญญาโท	วศ.ม. / วิศวกรรมศาสตรมหาบัณฑิต	วิศวกรรมคอมพิวเตอร์	จุฬาลงกรณ์มหาวิทยาลัย
2549-2553	ปริญญาเอก	Ph.D. / Computer Science	Computer Science	The University of Manchester, UK

6. สาขาวิชาการที่มีความชำนาญพิเศษ
   เทคโนโลยีเชิงวัตถุ, เทคโนโลยีเชิงลักษณะ, วิศวกรรมซอฟต์แวร์
7. ประสบการณ์ที่เกี่ยวข้องกับการบริหารงานวิจัยทั้งภายในและภายนอกประเทศ :
   7.1. หัวหน้าโครงการวิจัย: โครงการวิจัยการออกแบบระบบคลัสเตอร์สำหรับกลุ่มเครื่องแม่ข่ายโปรแกรมประยุกต์ ทุนอุดหนุนการวิจัยเพื่อสนับสนุนการสร้างและพัฒนานักวิจัยรุ่นใหม่ ปี พ.ศ. 2547

7.2. หัวหน้าโครงการวิจัย: โครงการวิจัยการออกแบบและพัฒนาเฟรมเวิร์คการเชื่อมต่อข้อมูลสำหรับลูกข่ายแบบบางของระบบกลุ่มแม่ข่ายโปรแกรมประยุกต์ ทุนอุดหนุนการวิจัย มหาวิทยาลัยเทคโนโลยีสุรนารี ปี พ.ศ. 2548

7.3. ผู้ร่วมวิจัย: โครงการวิจัยการพัฒนาซอฟต์แวร์ตัวอย่างด้านความปลอดภัยอาหาร ทุนนวัต-กรรมและสิ่งประดิษฐ์ สมเด็จพระเทพรัตนราชสุดาฯ สยามบรมราชกุมารี ปีพ.ศ.2547

8. งานวิจัยที่ดำเนินการเสร็จแล้ว :

8.1. โครงการวิจัยการออกแบบระบบคลัสเตอร์สำหรับกลุ่มเครื่องแม่ข่ายโปรแกรมประยุกต์ ปีที่พิมพ์ 2548
แหล่งทุน มหาวิทยาลัยเทคโนโลยีสุรนารี

8.2. โครงการวิจัยการพัฒนาซอฟต์แวร์ตัวอย่างด้านความปลอดภัยอาหาร ปีที่พิมพ์ 2548
แหล่งทุน กองทุนนวัตกรรมและสิ่งประดิษฐ์ สมเด็จพระเทพรัตนราชสุดาฯ สยามบรมราชกุมารี มหาวิทยาลัยเทคโนโลยีสุรนารี

8.3. โครงการวิจัยการออกแบบและพัฒนาเฟรมเวิร์คการเชื่อมต่อข้อมูลสำหรับลูกข่ายแบบบางของระบบกลุ่มแม่ข่ายโปรแกรมประยุกต์ ปีที่พิมพ์ 2553
แหล่งทุน มหาวิทยาลัยเทคโนโลยีสุรนารี

8.4. โครงการการพัฒนาซอฟตแวร์สำหรับจัดการและรายงานตำแหน่งพิกัดยานพาหนะที่สามารถปรับแต่งได้ด้วยภาษาเฉพาะทางภาษาไทย
แหล่งทุน กองทุนนวัตกรรมและสิ่งประดิษฐ์ สมเด็จพระเทพรัตนราชสุดาฯ สยามบรมราชกุมารี มหาวิทยาลัยเทคโนโลยีสุรนารี

9. งานวิจัยระหว่างดำเนินการ:

9.1. โครงการพัฒนาระบบปฏิบัติการเฉพาะทางสำหรับการศึกษาโครงสร้างผลึกโปรตีน สถานภาพ อยู่ระหว่างดำเนินการ คิดเป็นร้อยละ 90
แหล่งทุน สำนักงานพัฒนาวิทยาศาสตร์และเทคโนโลยีแห่งชาติ