บทที่ 3

วิธีการวิจัย

งานวิจัยนี้มีวัตถุประสงค์เพื่อออกแบบและสร้างระบบเช็คพอยน์เวอร์ชวลแมชชีนแบบใหม่ที่สามารถทำการเช็คพอยน์ได้อย่างมีประสิทธิภาพโดยใช้เทคนิคการแยกเทรดการประมวลผลของเวอร์ชวลแมชชีนและการสร้างไฟล์เช็คพอยน์ออกจากกัน ซึ่งเทรดที่ถูกสร้างขึ้นมาจะช่วยทำเช็คพอยน์ขณะที่เวอร์ชวลแมชชีนกำลังทำงานอยู่โดยใช้เทคนิคไลฟ์ไมเกรชัน นั่นคือเวอร์ชวลแมชชีนจะไม่ถูกหยุดการทำงานในขณะที่มีการคัดลอกหน่วยความจำจนกระทั่งถึงเวลาที่มันต้องบันทึกสถานะลงในอิมเมจไฟล์ และเพื่อใช้ประโยชน์จากซีพียูมัลติคอร์ (Akhter, & Roberts, 2006, p. 12; Fruehe, 2005, p. 71) เทรดจะถูกกำหนดให้ทำงานบนคอร์อื่นที่ไม่ใช่คอร์ที่เวอร์ชวลแมชชีนรันอยู่

โดยทั่วไป ความสามารถในการเช็คพอยน์ไม่ได้ถูกสร้างให้เป็นส่วนหนึ่งของเวอร์ชวลแมชชีนมอนิเตอร์ อย่างไรก็ตาม ความสามารถของเวอร์ชวลแมชชีนมอนิเตอร์ที่สนับสนุนวิธีการเช็คพอยน์คือความสามารถที่จะเก็บสถานะและเรียกสถานะให้กลับมาทำงานต่อ และความสามารถสำหรับทำไลฟ์ไมเกรชัน

ความสามารถในการไมเกรทสถานะลงไฟล์ของเควีเอ็มอาศัยวิธีการสร้างสแนปช็อตของเวอร์ชวลแมชชีน (savevm) และการเรียกสถานะให้กลับมาทำงานต่ออาศัยวิธีการเรียกสแนปช็อตของเวอร์ชวลแมชชีนขึ้นมา (loadvm) ซึ่งเป็นความสามารถของคีมู โดยการใช้ savevm สแนปช็อตของเวอร์ชวลแมชชีนจะถูกสร้างขึ้นมาจากข้อมูลสถานะของเวอร์ชวลแมชชีน ซึ่งประกอบด้วยสถานะของซีพียู หน่วยความจำ และสถานะของดีไวซ์ และสแนปช็อตของดิสก์อิมเมจที่เขียนได้ทั้งหมด การเก็บสแนปช็อตของเวอร์ชวลแมชชีนจะต้องเก็บในดิสก์อิมเมจที่มีรูปแบบเป็น qcow2 ส่วนการใช้ loadvm คือการเซ็ตเวอร์ชวลแมชชีนทั้งเครื่องให้กลับมาอยู่ในสถานะที่ได้สร้างสแนปช็อตไว้

ไลฟ์ไมเกรชัน (live migration) คือความสามารถในการย้ายเวอร์ชวลแมชชีนจากเครื่องหนึ่งไปอีกเครื่องหนึ่งได้โดยเกือบจะไม่ต้องหยุดการทำงานของเวอร์ชวลแมชชีน สำหรับการไมเกรท หน่วยความจำและสถานะของเวอร์ชวลแมชชีนจะถูกส่งจากโฮสแมชชีนต้นทางไปยังปลายทางโดยใช้ทีซีพีซ็อกเก็ต ในเควีเอ็ม ไลฟ์ไมเกรชันโปรโตคอลสามารถใช้ในการบันทึกสถานะของเวอร์ชวลแมชชีนลงไฟล์ได้

การใช้ไมเกรชันโปรโตคอลสำหรับเก็บสถานะของเวอร์ชวลแมชชีนลงไฟล์นี้มีประโยชน์ในกรณีที่ผู้ใช้ต้องการไมเกรทเวอร์ชวลแมชชีนโดยส่งข้อมูลผ่านไฟล์ซิสเต็มส์หรือการไมเกรทไม่สามารถทำผ่านเน็ตเวิร์คโดยตรงได้

อย่างไรก็ตาม การเขียนสถานะของเวอร์ชวลแมชชีนลงไฟล์นั้นยังไม่ใช่การเช็คพอยน์ เนื่องจากไม่มีการเก็บสถานะของดิสก์อิมเมจที่สอดคล้องกับสถานะของเวอร์ชวลแมชชีน ดังนั้น การสร้างเช็คพอยน์ไฟล์ก็คือการเก็บสถานะของเวอร์ชวลแมชชีนและสถานะของดิสก์อิมเมจที่สอดคล้องกัน งานวิจัยนี้ได้สร้างความสามารถในการเช็คพอยน์ของเวอร์ชวลแมชชีนมอนิเตอร์ขึ้นมาจากพื้นฐานของระบบไลฟ์ไมเกรชันเดิมสามวิธีดังที่จะได้อธิบายต่อไป โดยวิธีที่สามเป็นประโยชน์ที่คาดว่าจะได้รับ (contribution) หลักในงานวิจัยนี้

1. การเช็คพอยน์เวอร์ชวลแมชชีนในเควีเอ็มโดยใช้ไมเกรชันโปรโตคอล

ไมเกรชันโปรโตคอลสามารถใช้เช็คพอยน์เวอร์ชวลแมชชีนในเควีเอ็มได้โดยใช้การบันทึกข้อมูลสถานะของเวอร์ชวลแมชชีนลงไฟล์ ก่อนที่จะส่งไมเกรชันรีเควสท์เพื่อไมเกรทลงไฟล์ ผู้ใช้จะต้องสั่งให้เวอร์ชวลแมชชีนหยุดทำงานก่อน โดยใช้คำสั่งในคีมูมอนิเตอร์ซึ่งเป็นคอนโซลที่รับคำสั่งจากคอมมานไลน์ ในงานวิจัยนี้ได้เพิ่มความสามารถในการหยุดทำงานและการทำงานต่อหลังจากเช็คพอยน์อัตโนมัติโดยที่ผู้ใช้ไม่จำเป็นต้องสั่งให้หยุด/ทำงานต่อในคีมูมอนิเตอร์ และการเก็บสถานะของดิสก์อิมเมจที่สอดคล้องกัน

2. ไลฟ์เช็คพอยน์โปรโตคอล (Live checkpoint protocol)

การหยุดเวอร์ชวลแมชชีนเพื่อไมเกรทสถานะลงไฟล์ของเควีเอ็มทำให้ไม่ได้ใช้ประโยชน์จากไลฟ์ไมเกรชันโปรโตคอลที่มีอยู่ การยอมให้เวอร์ชวลแมชชีนทำงานต่อในขณะที่ทำไมเกรทสถานะของเวอร์ชวลแมชชีนลงไฟล์อาจช่วยให้แอพพลิเคชันทำงานเสร็จเร็วขึ้น เพราะทำให้เกิดการโอเวอร์แลปของการประมวลผลของเวอร์ชวลแมชชีนกับการไมเกรทข้อมูลลงไฟล์ แต่อย่างไรก็ตามวิธีการนี้ก็ยังมีผลกระทบต่อประสิทธิภาพของเวอร์ชวลแมชชีนเนื่องจากมีการขัดจังหวะการทำงานของเวอร์ชวลแมชชีนเพื่อไมเกรทเพจหน่วยความจำเป็นระยะๆ งานวิจัยนี้ได้เพิ่มความสามารถในการไมเกรทสถานะลงไฟล์ของเควีเอ็มให้เป็นแบบไลฟ์ไมเกรชันและเพิ่มการเก็บสถานะของดิสก์อิมเมจที่สอดคล้องกัน

3. เทรดเบสไลฟ์เช็คพอยน์โปรโตคอล (Thread-based live checkpoint protocol)

เมื่อเวอร์ชวลแมชชีนต้องทำงานปกติพร้อมกับงานเช็คพอยน์มีข้อเสียคือในกรณีที่แอพพลิเคชันต้องการใช้หน่วยความจำมาก การเขียนหน่วยความจำที่เกิดขึ้นจากการทำงานตามปกติในปริมาณมากจะขยายเวลาของการเช็คพอยน์ออกไปเพราะการเปลี่ยนแปลง

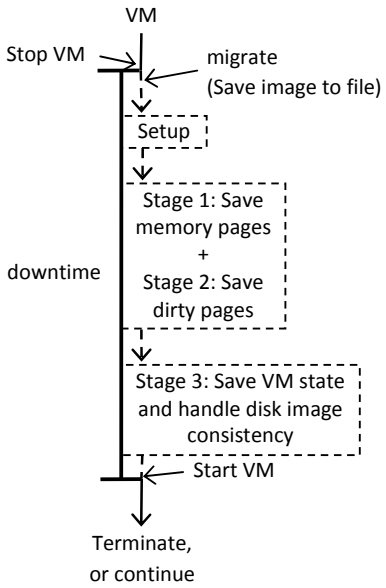
หน่วยความจำที่เพิ่มขึ้นหมายถึงงานเช็คพอยน์ที่เพิ่มขึ้นด้วย เวลาของการเช็คพอยน์ที่ถูกขยายออกไปเรื่อยๆนี้จะทำให้ประสิทธิภาพการทำงานของแอพพลิเคชันลดลงได้ เพราะมีโอเวอร์เฮดที่เกิดจากการขัดจังหวะการทำงานของเวอร์ชวลแมชชีน งานวิจัยนี้จึงได้เพิ่มความสามารถในการเช็คพอยน์ที่เวอร์ชวลแมชชีนสามารถทำงานต่อไปได้โดยมีเทรดช่วยทำงานเช็คพอยน์ส่วนใหญ่ให้ด้วยวิธีการนี้ เวอร์ชวลแมชชีนและเทรดจะทำงานขนานกันไปซึ่งจะช่วยปรับปรุงประสิทธิภาพให้ดีขึ้นเพราะ 1) เทรดสำหรับไมเกรชันทำให้การสร้างเช็คพอยน์ไฟล์เป็นไปอย่างรวดเร็วและมีเช็คพอยน์ไฟล์ที่เล็ก และ 2) ลดการขัดจังหวะการทำงานของเวอร์ชวลแมชชีนได้เป็นอย่างมาก

ในส่วนต่อไปคือรายละเอียดของอัลกอริทึมของวิธีการเช็คพอยน์ทั้ง 3 แบบ รวมถึงการสร้างไฟล์สถานะของเวอร์ชวลแมชชีนและดิสก์อิมเมจด้วย

การเช็คพอยน์เวอร์ชวลแมชชีนในเควีเอ็ม

ไมเกรชันโปรโตคอล exec เป็นโปรโตคอลที่ใช้บันทึกสถานะของเวอร์ชวลแมชชีนลงไฟล์ (การเช็คพอยน์เวอร์ชวลแมชชีน) โดยใช้อัลกอริทึมเดียวกับไมเกรชันโปรโตคอล tcp ที่ใช้ไมเกรทเวอร์ชวลแมชชีนระหว่างเครื่อง ดังภาพที่ 3.1

ภาพที่ 3.1
ขั้นตอนการเช็คพอยน์เวอร์ชวลแมชชีนตามปกติในเควีเอ็ม



อย่างไรก็ตาม ก่อนที่จะเริ่มต้นการบันทึกสถานะของเวอร์ชวลแมชชีน จะต้องหยุดเวอร์ชวลแมชชีนก่อนโดยใช้คำสั่ง “stop” ในคีมูมอนิเตอร์ และหลังจากใช้คำสั่ง “migrate” จะเกิดการทำไมเกรชันลงสู่ไฟล์ตามกระบวนการใน stage 1, stage 2, และ stage 3 ในภาพ 3.1 สุดท้ายหลังจากเสร็จสิ้นการบันทึกอิมเมจไฟล์จึงใช้คำสั่ง “cont” ในคีมูมอนิเตอร์เพื่อให้เวอร์ชวลแมชชีนทำงานต่อ นั่นคือ เวอร์ชวลแมชชีนจะต้องหยุดทำงานในขณะที่ทำการเช็คพอยน์

ขั้นตอนการไมเกรทสถานะลงไฟล์ตามปกติของเควีเอ็มจึงมีดังนี้

1. หยุดเวอร์ชวลแมชชีน
2. เมื่อเวอร์ชวลแมชชีนได้รับไมเกรชันรีเควสท์ (ไมเกรทลงไฟล์) มันจะเริ่มต้นทำการเซ็ตอัพสถานะไมเกรชัน
3. คัดลอกหน่วยความจำ
   - คัดลอกเพจของหน่วยความจำสำหรับรอบแรกลงไฟล์ (stage 1)
   - ในรอบถัดไป คัดลอกหน่วยความจำเพจที่เหลือจนกว่าจะเป็นไปตามเงื่อนไขการหยุดคัดลอกหน่วยความจำ (stage 2)
   - เนื่องจากเวอร์ชวลแมชชีนถูกหยุดก่อนที่จะคัดลอกหน่วยความจำ ดังนั้นการคัดลอกเพจของหน่วยความจำอาจเสร็จสิ้นได้ใน stage 2
4. คัดลอกสถานะของเวอร์ชวลแมชชีน (stage 3)
   - แต่ละดีไวซ์คัดลอกสถานะของมันลงไฟล์
   - ถ้าการคัดลอกหน่วยความจำยังไม่เสร็จสิ้น หน่วยความจำเพจที่เหลือทั้งหมดจะถูกคัดลอกต่อใน stage 3 นี้
5. สตาร์ทเวอร์ชวลแมชชีนให้ทำงานต่อหรือเลิกการทำงาน

เนื่องจากก่อนที่จะบันทึกอิมเมจของเวอร์ชวลแมชชีน จะต้องทำการหยุดเวอร์ชวลแมชชีนที่กำลังรันอยู่ก่อน ดังนั้น สำหรับการเช็คพอยน์แบบปกติของเควีเอ็มที่ใช้ในงานวิจัยนี้ จะถูกเพิ่มฟังก์ชัน vm_stop() เพื่อหยุดเวอร์ชวลแมชชีนเมื่อได้รับไมเกรชันรีเควสท์ และเวอร์ชวลแมชชีนจะต้องทำงานต่อหลังจากทำเช็คพอยน์ ฟังก์ชัน vm_start() และการบันทึกสถานะของดิสก์ที่สอดคล้องกับสถานะของเวอร์ชวลแมชชีน จึงถูกเพิ่มเข้าไปเพื่อให้เวอร์ชวลแมชชีนทำงานต่อหลังจากเสร็จสิ้นการบันทึกสถานะของเวอร์ชวลแมชชีนใน stage 3

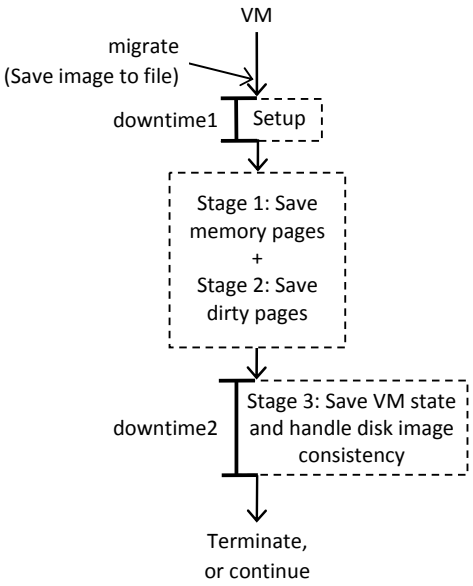
เวลาที่ใช้ในการเช็คพอยน์ของวิธีการนี้จะเท่ากับเวลาที่เวอร์ชวลแมชชีนหยุดทำงาน ซึ่งมีผลต่อการทำงานของแอพพลิเคชัน ดังนั้น ถ้าการเช็คพอยน์สามารถทำได้โดยที่เวอร์ชวลแมชชีนไม่ต้องหยุดทำงาน ประสิทธิภาพการทำงานของแอพพลิเคชันก็ควรจะดีขึ้น แต่สำหรับ

โปรโตคอลเดิมของเควีเอ็มนี้ การไมเกรทสถานะลงไฟล์ไม่สามารถทำได้โดยที่ไม่หยุดเวอร์ชวลแมชชีนก่อน เพราะจะทำให้เกิดความผิดพลาดขึ้น คือหลังจากเสร็จสิ้นการเช็คพอยน์แล้ว เวอร์ชวลแมชชีนจะไม่สามารถทำงานต่อได้ถึงแม้จะใช้คำสั่ง “cont” ในคีมูมอนิเตอร์ นอกจากนั้น การรีสตาร์ทเวอร์ชวลแมชชีนจากไฟล์สถานะก็ไม่สามารถทำได้เช่นกัน ดังนั้น การเช็คพอยน์แบบไลฟ์ไมเกรชันจึงไม่สามารถทำได้เพียงแค่ไม่ทำการหยุดเวอร์ชวลแมชชีน

ไลฟ์เช็คพอยน์โปรโตคอล (Live checkpoint protocol)

การบันทึกอิมเมจของเวอร์ชวลแมชชีนลงไฟล์สามารถทำในรูปแบบไลฟ์ไมเกรชันได้ เนื่องจากอัลกอริทึมของมันคืออัลกอริทึมที่ทำไลฟ์ไมเกรชัน วิธีการที่ใช้ในงานวิจัยนี้คือการหยุดเวอร์ชวลแมชชีนในขั้นตอนการเซ็ตอัพ และให้เวอร์ชวลแมชชีนทำงานต่อก่อนที่จะเริ่มต้นการคัดลอกหน่วยความจำใน stage 1 ดังภาพที่ 3.2 ด้วยวิธีการนี้ กลไกการเช็คพอยน์จะเป็นแบบไลฟ์ไมเกรชัน คือสามารถทำเช็คพอยน์ได้โดยที่เวอร์ชวลแมชชีนไม่ต้องหยุดทำงาน และเมื่อเสร็จสิ้นการเช็คพอยน์แล้ว เวอร์ชวลแมชชีนจะสามารถกลับมาทำงานต่อได้ นอกจากนั้นวิธีการนี้ยังทำให้สามารถเรียกเวอร์ชวลแมชชีนขึ้นมาทำงานต่อจากไฟล์สถานะได้อีกด้วย

ภาพที่ 3.2
ขั้นตอนการเช็คพอยน์เวอร์ชวลแมชชีนของไลฟ์เช็คพอยน์โปรโตคอล



เช่นเดียวกับไลฟ์ไมเกรชันในโปรโตคอล tcp คือหลังจากที่คัดลอกหน่วยความจำจนกระทั่งตรงตามเงื่อนไขในการหยุดคัดลอกหน่วยความจำแล้ว จึงหยุดเวอร์ชวลแมชชีนเพื่อบันทึกสถานะของเวอร์ชวลแมชชีนและหน่วยความจำที่เหลือ อย่างไรก็ตาม ในโปรโตคอล tcp เวอร์ชวลแมชชีนที่เครื่องเดิมจะไม่ทำงานต่อในกรณีที่ทำไมเกรชันสำเร็จ ในงานวิจัยนี้ต้องการให้เวอร์ชวลแมชชีนทำงานต่อ จึงต้องเพิ่มฟังก์ชัน vm_start() เข้าไปหลังจากที่เสร็จสิ้นการบันทึกสถานะใน stage 3

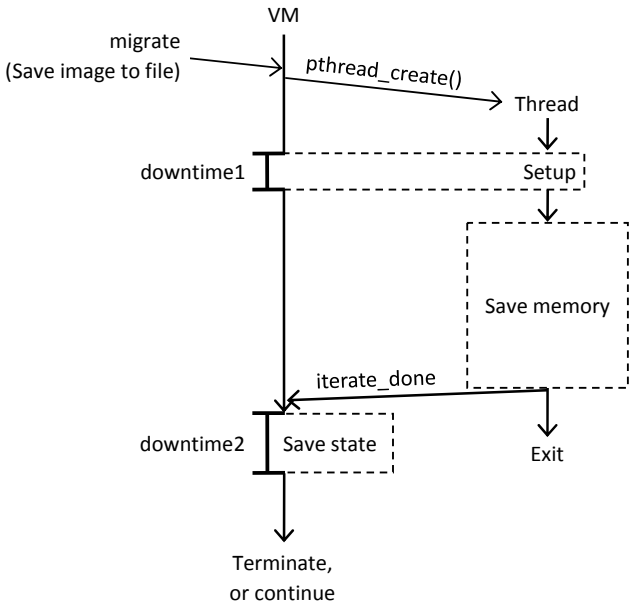
เงื่อนไขการหยุดคัดลอกหน่วยความจำในเควีเอ็ม คือใน stage 2 เวลาที่คาดว่าจะต้องใช้ในการบันทึกหน่วยความจำเพจที่เหลือต้องไม่มากกว่าเวลาที่ตั้งค่าไว้ ซึ่งเป็นเวลาที่เวอร์ชวลแมชชีนหยุดทำงาน (downtime) โดยเวลาที่เหลือคำนวณจากขนาดของเพจหน่วยความจำที่เหลือคูณแบนด์วิดท์ และค่าดาวน์ไทม์ที่ถูกเซ็ตในเควีเอ็มคือ 30ms นั่นคือ การคัดลอก dirty page จะทำเป็นระยะๆ (ทุกๆ 100ms ตามค่าดีฟอลต์ของเควีเอ็ม) จนกว่าเวลาที่เหลือมีค่าไม่เกิน 30ms อย่างไรก็ตาม เมื่อเวอร์ชวลแมชชีนรันแอพพลิเคชันที่มีข้อมูลขนาดใหญ่อาจทำให้เวลาที่ใช้ในการบันทึกอิมเมจไฟล์นานขึ้นได้เนื่องจากการคำนวณเวลาที่เหลือมีค่ามากกว่าดาวน์ไทม์ที่ตั้งไว้มาก ซึ่งมีผลต่อประสิทธิภาพการทำงานของแอพพลิเคชันในเวอร์ชวลแมชชีนด้วย เนื่องจากเวอร์ชวลแมชชีนทำงานไปพร้อมกับการบันทึกอิมเมจไฟล์ ดังนั้น งานวิจัยนี้จึงเสนอวิธีการที่ใช้เทรดเพื่อช่วยทำงานเช็คพอยน์โดยส่วนใหญ่ของไลฟ์เช็คพอยน์โปรโตคอล

เทรดเบสไลฟ์เช็คพอยน์โปรโตคอล (Thread-based live checkpoint protocol)

1. การออกแบบ

แทนที่เวอร์ชวลแมชชีนโปรเซสที่กำลังทำงานอยู่จะต้องเสียเวลาทำเช็คพอยน์เอง ซึ่งทำให้เวลาการทำงานของแอพพลิเคชันและเวลาการทำเช็คพอยน์เพิ่มขึ้น ในงานวิจัยนี้ เวอร์ชวลแมชชีนโปรเซสจะสร้างเทรดขึ้นมาให้ทำเช็คพอยน์เมื่อได้รับไมเกรชันรีเควสท์ของโปรโตคอล exec ดังนั้น เวอร์ชวลแมชชีนและเทรดที่ช่วยทำเช็คพอยน์จะทำงานขนานกันไป โดยเงื่อนไขการหยุดคัดลอกหน่วยความจำใช้ค่าดาวน์ไทม์เดิมที่ถูกเซ็ตตามปกติในเควีเอ็ม (30ms) อย่างไรก็ตาม เพื่อให้แน่ใจว่าเทรดจะไม่ใช้เวลาทำงานนานเกินไป งานวิจัยนี้จึงได้กำหนดจำนวนรอบสูงสุดของการคัดลอกหน่วยความจำที่ 200 รอบเป็นเงื่อนไขที่สอง ซึ่งผู้ใช้สามารถกำหนดค่านี้ได้ผ่านคอนฟิกูเรชันไฟล์ นอกจากนี้ เพื่อให้ได้ประโยชน์จากซีพียูมัลติคอร์ เทรดที่ทำเช็คพอยน์จะถูกกำหนดให้รันบนคอร์อื่นที่ไม่ใช่คอร์ที่เวอร์ชวลแมชชีนกำลังรันอยู่

ภาพที่ 3.3
ขั้นตอนการเช็คพอยน์เวอร์ชวลแมชชีนของเทรดเบสไลฟ์เช็คพอยน์โปรโตคอล



จากภาพที่ 3.3 เมื่อเวอร์ชวลแมชชีน (VM) ได้รับคำสั่งให้บันทึกสถานะลงไฟล์ มันจะสร้างเทรดขึ้นมาเทรดหนึ่ง (Thread) ให้ทำงานคัดลอกหน่วยความจำในขณะที่มันกลับไปทำงานตามปกติ และ VM จะทำการตรวจสอบสถานะการทำงานของเทรดเป็นระยะๆ (ทุกๆ 100ms ตามค่าดีฟอลต์ของเควีเอ็ม) เมื่อมันพบว่าเทรดได้ทำการคัดลอกหน่วยความจำเสร็จแล้ว จึงเริ่มบันทึกสถานะรวมทั้งหน่วยความจำเพจที่เหลือหลังจากที่เทรดหยุดทำงาน แล้วจึงกลับไปทำงานต่อ เวลาที่หยุดการทำงานของเวอร์ชวลแมชชีนหรือดาวน์ไทม์คือระยะเวลาเซ็ตอัพ (downtime1 ในภาพที่ 3.3) และระยะเวลาในการบันทึกสถานะของเวอร์ชวลแมชชีนในรอบสุดท้าย (downtime2 ในภาพที่ 3.3) ถึงแม้ว่าขั้นตอนการบันทึกสถานะในขั้นสุดท้ายจะไม่ได้ทำทันทีที่เทรดคัดลอกหน่วยความจำเสร็จ แต่ระยะเวลาตั้งแต่เทรดจบการทำงานจนถึงเวลาที่เวอร์ชวลแมชชีนเริ่มบันทึกสถานะจะมีค่าไม่เกิน 100ms ดังนั้น ค่าดาวน์ไทม์ของวิธีการนี้ควรจะเปรียบเทียบได้กับดาวน์ไทม์ของวิธีการไลฟ์เช็คพอยน์ปกติ เนื่องจาก VM ทำการเช็คการทำงานเสร็จของ Thread ว่า Thread ทำงานเสร็จหรือไม่ทุกๆ 100ms ดังนั้น ช่วงเวลาที่อาจมีการเปลี่ยนแปลงค่าในหน่วยความจำหลังจากที่ Thread ทำงานเสร็จก็จะไม่มากกว่า 100ms

ภาพที่ 3.4

อัลกอริทึมของเทรดเบสไลฟ์เช็คพอยน์โปรโตคอล

Algorithm: VM

```
1  create Thread
   (Wait for thread to complete setup)
2  create_timer(clock, callback_thread_state)
   (Continue VM execution and periodically
   check thread's state)
3  IF thread_exit THEN
      vm_stop()
      Write device state, CPU state, and
   remaining dirty pages
      vm_start()
   ENDIF
```

Algorithm: Thread

```
1  vm_stop()
2  setup()
3  thread_exit ← false
4  vm_start()
5  Write memory pages for 1st iteration
6  REPEAT
      Write all dirty pages
   UNTIL expected_time <= max_downtime  OR
   iteration >= 200
7  thread_exit ← true
8  EXIT
```

โปรโตคอลสำหรับเช็คพอยน์แบบเทรดประกอบด้วยอัลกอริทึม 2 อัลกอริทึม ดังภาพที่ 3.4 ซึ่งมีรายละเอียดในการทำงานดังนี้ (ในคำอธิบายต่อไปนี้ กำหนดให้ข้อความหลัง “VM: “ เป็นเหตุการณ์ที่เกิดขึ้นในการทำงานของ Algorithm VM และ “Thread: “ เป็นเหตุการณ์ที่เกิดขึ้นในการทำงานของ Algorithm Thread)

จากภาพที่ 3.4 การเช็คพอยน์เวอร์ชวลแมชชีนของเทรดเบสไลฟ์เช็คพอยน์โปรโตคอลมีขั้นตอนดังนี้

1. VM: หลังจากได้รับไมเกรชันรีเควสท์ให้ไมเกรทสถานะลงไฟล์ VM จะสร้างเทรดขึ้นมาเทรดหนึ่ง (Thread) เพื่อให้ช่วยทำงานเช็คพอยน์ แล้วจึงกลับไปทำงานต่อ (step 1 ของ Algorithm VM)

2. Thread: หยุดการทำงานของ VM ชั่วคราวเพื่อกำหนดค่าเริ่มต้นของสถานะไมเกรชัน และมาร์คสถานะของมันว่ากำลังทำงาน (step 1-3 ของ Algorithm Thread)

3. Thread: หลังจากนั้นจึงสตาร์ท VM ให้ทำงานต่อ (step 4 ของ Algorithm Thread)

4. VM: เมื่อกลับมาทำงานต่อ VM จะกำหนดค่า interrupt timer เพื่อให้ทำการตรวจสอบเป็นระยะว่า Thread ทำงานเสร็จหรือยัง (step 2 ของ Algorithm VM)

5. Thread: คัดลอกหน่วยความจำและเขียนลงไฟล์

   - คัดลอกเพจของหน่วยความจำลงไฟล์ในรอบแรก (step 5 ของ Algorithm Thread)

- สำหรับทุกๆรอบถัดไป คัดลอก dirty page ทั้งหมดของรอบก่อนหน้านั้น จนกว่าจะเป็นไปตามเงื่อนไขการหยุดคัดลอกหน่วยความจำ (step 6 ของ Algorithm Thread)

6. Thread: มาร์คสถานะการทำงานว่าเสร็จสิ้น แล้วจบการทำงาน (step 7-8 ของ Algorithm Thread)

7. VM: ถ้าพบว่า Thread ทำงานเสร็จแล้ว VM จะหยุดการทำงานเพื่อทำการคัดลอกสถานะ รวมทั้ง dirty page จากรอบสุดท้าย และเมื่อสิ้นสุดการเช็คพอยน์แล้ว VM จะกลับไปทำงานต่อ (step 3 ของ Algorithm VM)

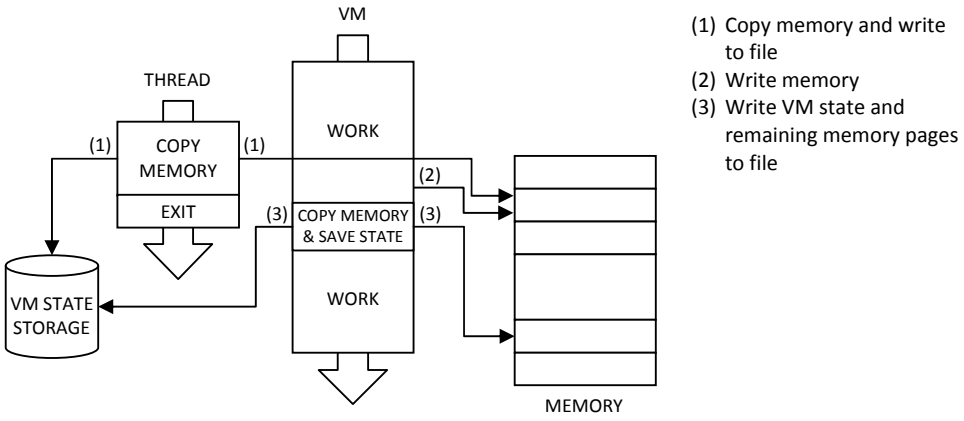
การทำงานเช็คพอยน์โดยส่วนใหญ่เป็นการทำงานของเทรด และเมื่อเวลาที่จะต้องหยุดเวอร์ชวลแมชชีนน้อยเพียงพอแล้ว เทรดจึงจบการทำงานและให้เวอร์ชวลแมชชีนทำงานเช็คพอยน์ต่อในขั้นสุดท้าย

2. การวิเคราะห์ความถูกต้องของโปรโตคอล

การคัดลอกหน่วยความจำของเทรดสามารถทำงานไปพร้อมกับเวอร์ชวลแมชชีนได้ ดังนั้นในขณะที่เทรดกำลังคัดลอกหน่วยความจำ เวอร์ชวลแมชชีนสามารถเข้าถึงหน่วยความจำได้เช่นกัน สถานการณ์ที่เวอร์ชวลแมชชีนและเทรดต้องการเข้าถึงหน่วยความจำบล็อกเดียวกันอาจเกิดขึ้นได้ ดังภาพที่ 3.5

ภาพที่ 3.5

การทำงานเช็คพอยน์ของเทรดขณะที่เวอร์ชวลแมชชีนกำลังทำงานตามปกติ



อย่างไรก็ตาม สถานการณ์นี้จะไม่ทำให้เกิดความไม่สอดคล้องระหว่างหน่วยความจำจริงและการอัพเดท dirty page log ของเทรด เนื่องจากเทรดทำงานเพียงแค่คัดลอกหน่วยความจำ ซึ่งอธิบายได้ดังนี้

1. การคัดลอกหน่วยความจำของเทรดจะทำทีละบล็อก โดยค้นหาบล็อกที่ dirty สำหรับในขั้นที่ 2 (iterate) การคัดลอกหน่วยความจำ 1 รอบอาจไม่ได้ทำจนครบทุกบล็อก เนื่องจากเงื่อนไขของการคัดลอก ดังนี้

- เมื่อจำนวนไบต์ที่เขียนลงไฟล์ในรอบนั้นมีค่ามากกว่าขอบเขตที่กำหนด (จำนวนไบต์มากที่สุดที่สามารถส่งได้ใน 1 รอบ) ซึ่งคำนวณจากการเซ็ตค่าแบนด์วิดท์ ในงานวิจัยนี้ใช้ค่าแบนด์วิดท์เดิมของเควีเอ็ม ทำให้ขอบเขตที่กำหนดมีค่าเท่ากับ 3.2MB หลังจากที่เทรดคัดลอกหน่วยความจำเสร็จ 1 รอบแล้ว ในรอบถัดไปของการค้นหาบล็อกที่ dirty มันจะค้นหาที่บล็อกถัดไปต่อจากรอบก่อนหน้านี้
- เมื่อค้นหาจนครบทุกบล็อกแล้ว

2. เมื่อเจอบล็อกที่ dirty เทรดจะทำการรีเซ็ตหน่วยความจำบล็อกนั้นให้เป็น not dirty ก่อน แล้วจึงคัดลอกหน่วยความจำบล็อกนั้น

3. ในกรณีที่เวอร์ชวลแมชชีนทำการเปลี่ยนแปลงหน่วยความจำบล็อกเดิมอีกครั้ง หลังจากที่เทรดได้คัดลอกบล็อกนั้นไปแล้ว บล็อกนั้นจะถูกเซ็ตให้เป็น dirty และถูกคัดลอกใหม่อีกครั้งเมื่อเทรดทำการค้นหาบล็อกที่ dirty ในรอบถัดไป

4. การคัดลอกหน่วยความจำในขั้นสุดท้าย เทรดจะค้นหาหน่วยความจำบล็อกที่ dirty โดยที่เวอร์ชวลแมชชีนหยุดทำงาน ดังนั้น dirty page log ที่อัพเดทในรอบสุดท้ายจึงสอดคล้องกับหน่วยความจำจริงในขณะที่เขียนข้อมูลสถานะลงไฟล์

เมื่อเวอร์ชวลแมชชีน (VM) ทำการเขียนหน่วยความจำ (write) มันจะเซ็ต dirty bit (set_dirty) และเมื่อเทรด (Thread) จะทำการคัดลอกหน่วยความจำ (copy) มันจะรีเซ็ต dirty bit (reset_dirty) ก่อน ดังนั้นเหตุการณ์การเขียนหน่วยความจำของ VM คือ write → set_dirty และเหตุการณ์การคัดลอกหน่วยความจำของ Thread คือ reset_dirty → copy ภาพที่ 3.6 แสดงสถานการณ์ที่อาจเกิดขึ้นได้เมื่อเวอร์ชวลแมชชีนและเทรดทำงานพร้อมกัน

ภาพที่ 3.6

สถานการณ์การเขียนหน่วยความจำของเวอร์ชวลแมชชีนและการคัดลอกหน่วยความจำของเทรด

step	VM	Thread
1		reset_dirty
2	write	
3	set_dirty	
4		copy

(a)

step	VM	Thread
1		reset_dirty
2	write	
3		copy
4	set_dirty	

(b)

step	VM	Thread
1	write	
2		reset_dirty
3		copy
4	set_dirty	

(c)

step	VM	Thread
1	write	
2		reset_dirty
3	set_dirty	
4		copy

(d)

step	VM	Thread
1		reset_dirty
2	write	copy
3	set_dirty	

(e)

step	VM	Thread
1	write	
2	set_dirty	reset_dirty
3		copy

(f)

สถานการณ์ (a) – (d) ในภาพ 3.6 จะทำให้เกิดการคัดลอกหน่วยความจำเพจเดิมซ้ำในรอบถัดไปเพราะการรีเซ็ต dirty bit ของเทรดมาก่อนการเซ็ต dirty bit ของเวอร์ชวลแมชชีน แต่อย่างไรก็ตาม สถานการณ์เหล่านี้จะไม่ทำให้เกิดความผิดพลาดในการคัดลอกหน่วยความจำของเทรด เนื่องจากจะไม่มีเหตุการณ์การรีเซ็ต dirty bit ของเทรดหลังจากการเซ็ต dirty bit ของเวอร์ชวลแมชชีนที่ไม่มีการคัดลอกหน่วยความจำตามมา

กรณี (a) Thread ทำการรีเซ็ต dirty bit แต่ก่อนที่มันจะเริ่มต้นคัดลอกข้อมูล VM ได้ทำการเขียนหน่วยความจำเพจนั้นและเซ็ต dirty bit ดังนั้น หลังจากที่ Thread ได้คัดลอกเพจนั้นไปแล้ว ในรอบถัดไป มันจะต้องทำการคัดลอกเพจนั้นซ้ำอีกครั้ง เช่นเดียวกับกรณี (b) Thread คัดลอกเพจของหน่วยความจำไปก่อนที่ VM จะเซ็ต dirty bit ในรอบถัดไป Thread จึงต้องคัดลอก

ซ้ำอีกครั้ง ส่วนกรณี (c) และ (d) คือกรณีที่ VM เขียนเพจของหน่วยความจำก่อนที่ Thread จะรีเซ็ต dirty bit แต่ไม่ว่าการคัดลอกหน่วยความจำของ Thread จะเกิดก่อนหรือหลังการเซ็ต dirty bit ของ VM ในรอบถัดไป Thread ก็ต้องคัดลอกหน่วยความจำซ้ำเช่นกัน การคัดลอกหน่วยความจำเพจเดิมซ้ำในสถานการณ์เหล่านี้เกิดขึ้นเพราะเหตุการณ์ reset_dirty → copy ของ Thread และเหตุการณ์ write → set_dirty ของ VM ไม่เป็น atomic แต่ถึงแม้ว่าการทำงานของ Thread และ VM จะมี race condition กลไกการคัดลอกหน่วยความจำของ Thread ก็จะไม่มีความผิดพลาดที่เกิดจากการไม่ได้คัดลอกหน่วยความจำเพจที่ควรจะต้องคัดลอก เพราะเหตุการณ์ที่จะทำให้เกิดความผิดพลาดคือ set_dirty → reset_dirty ↛ copy จะไม่เกิดขึ้นในโปรโตคอลนี้

กรณี (e) VM และ Thread เข้าถึงหน่วยความจำบล็อกเดียวกัน กรณีนี้จะทำให้การคัดลอกหน่วยความจำของ Thread เกิดความผิดพลาดได้ อย่างไรก็ตาม VM จะต้องทำการเซ็ต dirty bit หลังจากที่มันเขียนเสร็จ ดังนั้นในรอบถัดไป Thread จะต้องคัดลอกเพจที่ผิดพลาดนี้ซ้ำอีก และถ้าเกิดเหตุการณ์นี้ขึ้นอีก การคัดลอกหน่วยความจำเพจนั้นก็จะยังคงผิดพลาด แต่เมื่อ Thread จบการทำงาน VM จะหยุดการทำงานชั่วคราวเพื่อบันทึกสถานะและคัดลอกหน่วยความจำเพจที่ถูกเซ็ตว่า dirty ดังนั้นเมื่อไม่มีการเปลี่ยนแปลงหน่วยความจำเพจที่กำลังถูกคัดลอก ทำให้การคัดลอกในรอบสุดท้ายของเพจนั้นทำได้อย่างถูกต้อง

กรณี (f) เมื่อ VM และ Thread ต้องการเซ็ต dirty bit เดียวกัน กรณีนี้อาจทำให้การเซ็ต dirty bit ผิดพลาดได้ ซึ่งถ้าเวอร์ชวลแมชชีนทำการเขียนหน่วยความจำเพจเดิมอีกครั้ง การเซ็ต dirty bit ในรอบนี้ก็จะผิดพลาดด้วย และอาจทำให้การค้นหาเพจที่ dirty ของเทรดเกิดความผิดพลาดไปด้วย ดังนั้นจึงต้องมีการซิงโครไนซ์ระหว่างเวอร์ชวลแมชชีนและเทรด เพื่อให้แน่ใจว่าโปรโตคอลทำงานได้อย่างถูกต้อง

นอกจากเรื่องของซิงโครไนเซชัน สิ่งที่จะต้องพิจารณาเพื่อพัฒนาความถูกต้องของโปรโตคอลคือการสลับลำดับคำสั่ง เนื่องจากในขั้นตอนการคัดลอกหน่วยความจำของเทรด การรีเซ็ต dirty bit จำเป็นต้องทำก่อนการคัดลอกหน่วยความจำ ถ้ามีการสลับลำดับคำสั่งโดยคอมไพเลอร์ที่ทำให้เกิดการคัดลอกหน่วยความจำก่อนรีเซ็ต dirty bit แล้ว จะเป็นไปได้ที่เทรดจะคัดลอกหน่วยความจำผิดพลาด นั่นคือในกรณีที่เทรดคัดลอกหน่วยความจำผิดพลาดในกรณี (e) หรือคัดลอกก่อนการอัพเดทของเวอร์ชวลแมชชีน แต่ทำการรีเซ็ต dirty bit หลังจากที่เวอร์ชวลแมชชีนทำการเซ็ต dirty bit นั้นไปก่อนหน้านี้

4. การอิมพลีเมนต์

การอิมพลีเมนต์โปรโตคอลไลฟ์เช็คพอยน์แบบเทรดนี้ประกอบด้วยการอิมพลีเมนต์หลักของโปรโตคอล และการอิมพลีเมนต์เพื่อพัฒนาความถูกต้องของโปรโตคอล ได้แก่ การทำซิงโครไนเซชันและการป้องกันการสลับลำดับคำสั่ง

4.1 การอิมพลีเมนต์หลักของโปรโตคอล

ในโปรโตคอล exec เมื่อเวอร์ชวลแมชชีนได้รับไมเกรชันรีเควสท์ให้บันทึกอิมเมจลงไฟล์ ฟังก์ชัน do_migrate() จะสร้างเทรดขึ้นมาในกรณีที่พบว่าจำนวนซีพียูของโฮสมีมากกว่าหนึ่งซีพียู จากนั้น เทรดที่ถูกสร้างขึ้นมาจะเริ่มต้นทำงานเพื่อเช็คพอยน์เวอร์ชวลแมชชีน โดยในขั้นที่สองของการบันทึกเพจของหน่วยความจำเป็นขั้นที่ใช้เวลามากที่สุด ดังนั้นก่อนที่จะทำขั้นนี้ ถ้าเทรดรันอยู่บนซีพียูเดียวกับเวอร์ชวลแมชชีน มันจะต้องถูกเซ็ตให้ไปรันบนซีพียูอื่น และเมื่อถึงเวลาที่เทรดหยุดการบันทึกหน่วยความจำ มันจะจบการทำงาน จากนั้น เวอร์ชวลแมชชีนเทรดเดิมจะหยุดการทำงานเพื่อบันทึกสถานะของมัน รวมทั้งหน่วยความจำเพจที่เหลือ แล้วจึงทำงานต่อหรือจบการทำงาน

แทนที่เวอร์ชวลแมชชีนจะเริ่มต้นการทำงานเพื่อบันทึกข้อมูลสถานะในฟังก์ชัน do_migrate() มันจะทำเพียงแค่สร้างเทรดขึ้นมา ซึ่งเทรดจะเรียก do_migrate_state() เพื่อเริ่มต้นงานเช็คพอยน์ โดยที่งานใน do_migrate() จะถูกย้ายไปที่ do_migrate_state() ดังภาพที่ 3.7 เทรดเริ่มต้นด้วยการเรียก exec_start_outgoing_migration() เพื่อทำการเซ็ตอัพและคัดลอกเพจของหน่วยความจำทั้งหมดในรอบแรก สำหรับการทำงานตามปกติของการบันทึกเพจของหน่วยความจำ เวอร์ชวลแมชชีนโปรเซสจะตั้งเวลาให้ทำการบันทึกหน่วยความจำเป็นระยะๆในฟังก์ชัน qemu_fopen_ops_buffered() ส่วนในเวอร์ชันที่ใช้เทรด มาสเตอร์เทรด (VM) จะตั้งเวลาให้ตรวจสอบว่าเทรดที่มันสร้างขึ้นมาทำงานเสร็จหรือยัง ถ้าทำเสร็จแล้ว มันจะเริ่มต้นบันทึกสถานะ ส่วนเทรดที่ถูกสร้างขึ้นมาจะบันทึกหน่วยความจำไปเรื่อยๆจนกว่าค่าดาวน์ไทม์ที่คำนวณได้จะมีค่าไม่เกินค่าที่ตั้งไว้ หรือทำงานจนถึงขอบเขตจำนวนรอบสูงสุดที่สามารถทำได้ ความแตกต่างของการคัดลอกหน่วยความจำคือเทรดจะคัดลอกหน่วยความจำโดยไม่มีการตั้งเวลา ดังนั้น งานคัดลอกหน่วยความจำของเทรดจึงทำได้เร็วกว่าวิธีการของไลฟ์ไมเกรชันเดิม หลังจากคัดลอกหน่วยความจำเสร็จในรอบแรกแล้ว ก่อนที่เทรดจะเริ่มคัดลอก dirty page มันจะต้องเขียนข้อมูลลงไฟล์ก่อนโดยเรียก buffered_rate_tick() แล้วจึงคัดลอกหน่วยความจำ ดังนั้น ฟังก์ชันนี้จึงถูกเรียกซ้ำดังภาพที่ 3.7 ซึ่งตามปกติการบันทึกหน่วยความจำและการบันทึกสถานะจะทำในฟังก์ชัน migrate_fd_put_ready() ดังนั้นในเวอร์ชันที่ใช้เทรด ฟังก์ชันนี้จะแบ่งการทำงานเป็นสองส่วน คือเทรดทำส่วนที่บันทึกหน่วยความจำ และมาสเตอร์เทรดทำส่วนที่บันทึกสถานะ ดังภาพที่ 3.8

ภาพที่ 3.7

การอิมพลีเมนต์ do_migrate_state()

```
do_migrate_state()

  exec_start_outgoing_migration()
  ...

  WHILE NOT iterate_done
    buffered_rate_tick()
  ENDWHILE

RETURN
```

ภาพที่ 3.8

การอิมพลีเมนต์ migrate_fd_put_ready() ในเวอร์ชันที่ใช้เทรด

```
INIT max_downtime to 30 (ms)
INIT iteration to 0
INIT iterate_done to 0
...

migrate_fd_put_ready()

  IF first_iterate THEN
    Remove vl_cpu From cpuset
    pthread_setaffinity_np(self, cpu_set_t_size, cpuset)
  ENDIF

  IF thread THEN
    qemu_savevm_state_iterate(QEMUFile)
    IF expected_time <= max_downtime OR iteration >= 200 THEN
      SET iterate_done to 1
      Exit
    ELSE
      Add 1 to iteration
    ENDIF
  ENDIF

  IF master THEN
    vm_stop()
    bdrv_flush_all()
    qemu_savevm_state_complete(QEMUFile)
    vm_start()
  ENDIF

RETURN
```

ภาพที่ 3.9

การคำนวณเงื่อนไขการหยุดคัดลอกหน่วยความจำใน ram_save_live()

```
ram_save_live()
  ...

  memory_set_dirty()
  ram_save_block()

  bwidth = bytes_transferred / time_transferred

  expected_time = ram_remaining() * PAGE_SIZE / bwidth

RETURN
```

เงื่อนไขการหยุดคัดลอกหน่วยความจำคือเมื่อเวลาที่คาดว่าจะต้องใช้ในการคัดลอกหน่วยความจำที่เหลือ (expected_time) มีค่าไม่เกินค่าดาวน์ไทม์ ซึ่งเป็นการคำนวณตามปกติใน ram_save_live() ดังภาพที่ 3.9 ในงานวิจัยนี้เพิ่มเงื่อนไขจำนวนรอบสูงสุดที่เทรดสามารถทำได้ เพื่อที่เทรดจะไม่ทำงานนานเกินไป ซึ่งแสดงในภาพที่ 3.8

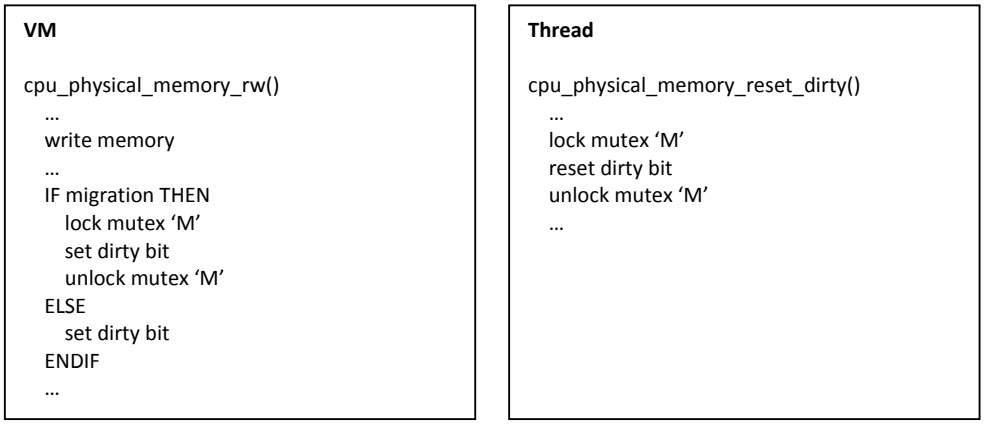
แต่ละครั้งที่เทรดเรียก buffered_rate_tick() มันจะทำ migrate_fd_put_ready() เพื่อคัดลอกหน่วยความจำเพจที่ถูกแก้ไขในรอบก่อนหน้านี้ โดยเรียก qemu_savevm_state_iterate() และเมื่อถึงเวลาที่เทรดหยุดคัดลอกหน่วยความจำ มันจะเซ็ตค่า iterate_done เพื่อให้มาสเตอร์เทรดทำขั้นตอนต่อไปคือหยุดเวอร์ชวลแมชชีน และเรียก bdrv_flush_all() เพื่อบันทึกสถานะของดีไวซ์ และ qemu_savevm_state_complete() เพื่อบันทึกสถานะของซีพียู รวมทั้งหน่วยความจำเพจที่เหลือทั้งหมด จากนั้นจึงให้เวอร์ชวลแมชชีนทำงานต่อหลังจากเสร็จสิ้นการเช็คพอยน์ ซึ่งวิธีการเช็คพอยน์ที่ใช้เทรดนี้ จะทำให้เวอร์ชวลแมชชีนสามารถทำงานไปได้โดยไม่ต้องเสียเวลาทำเช็คพอยน์เอง และเพื่อให้แน่ใจว่าเทรดที่ทำเช็คพอยน์จะไม่รันบนซีพียูเดียวกับเวอร์ชวลแมชชีน ในรอบแรก ซีพียูที่เวอร์ชวลแมชชีนรันอยู่จะถูกเอาออกไปจากกลุ่มของซีพียูที่เทรดสามารถรันได้ ทำให้เทรดสามารถรันบนซีพียูใดก็ได้ยกเว้นซีพียูของเวอร์ชวลแมชชีน

4.2 การอิมพลีเมนต์วิธีการซิงโครไนซ์ระหว่างเวอร์ชวลแมชชีนและเทรด (VM-Thread synchronization)

เนื่องจากในระหว่างที่มีการคัดลอก dirty page จากหน่วยความจำจริงของเวอร์ชวลแมชชีนโดยเทรด เวอร์ชวลแมชชีนและเทรดอาจทำการเปลี่ยนแปลงค่า dirty bit ของเพจเดียวกันในเวลาเดียวกันได้ เพื่อป้องกันความผิดพลาดที่เกิดจากเหตุการณ์นี้ จึงต้องมีการซิงโครไนซ์

ระหว่างเวอร์ชวลแมชชีนและเทรดโดยการใส่ mutual exclusion เข้าไปที่ฟังก์ชันที่มีการเปลี่ยนแปลงค่า dirty bit ของเวอร์ชวลแมชชีนและเทรด ดังภาพที่ 3.10

ภาพที่ 3.10
การซิงโครไนซ์เวอร์ชวลแมชชีนและเทรดด้วย mutex



สำหรับเวอร์ชวลแมชชีน การซิงโครไนซ์เพื่อเซ็ต dirty bit จะเกิดขึ้นเฉพาะในช่วงเวลาที่เทรดกำลังทำงานอยู่ด้วย

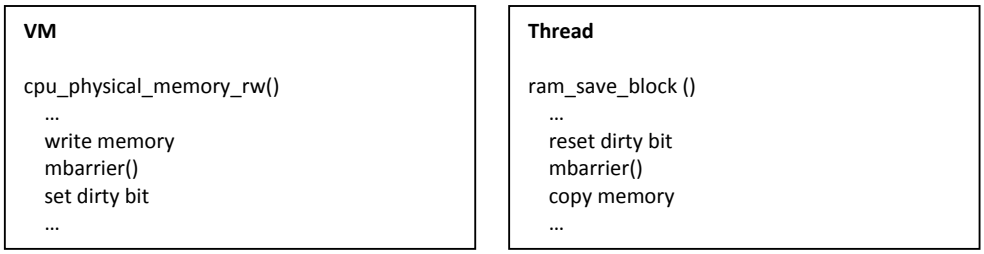
4.3 การอิมพลีเมนต์วิธีการป้องกันการสลับลำดับคำสั่ง

เมื่อเทรดทำการคัดลอกหน่วยความจำ มันจะต้องทำการรีเซ็ต dirty bit ก่อนที่จะคัดลอกเพจนั้น ซึ่งลำดับของเหตุการณ์นี้จำเป็นสำหรับการทำงานที่ถูกต้องของโปรโตคอล และเนื่องจากไม่มีความเกี่ยวเนื่องกัน (dependency) ระหว่างเหตุการณ์ทั้งสอง จึงเป็นไปได้ที่คอมไพเลอร์จะสลับลำดับคำสั่งนี้ ดังนั้นจึงต้องมีการเพิ่ม memory barrier เข้าไปเพื่อให้แน่ใจว่าเทรดจะต้องรีเซ็ต dirty bit ก่อนที่จะคัดลอกข้อมูล ดังภาพที่ 3.11 และ 3.12

ภาพที่ 3.11
การป้องกันการสลับลำดับคำสั่งด้วย memory barrier

```
#define mbarrier() __asm__ __volatile__("": : :"memory")
```

ภาพที่ 3.12
การป้องกันการสลับลำดับคำสั่งในการคัดลอกหน่วยความจำของเทรดและ
การเขียนหน่วยความจำของเวอร์ชวลแมชชีน



mbarrier() คือ memory barrier ที่จะจัดลำดับคำสั่งทั้ง load และ store หมายความว่าคำสั่ง load และ store ที่มาก่อน memory barrier จะต้องทำให้เสร็จก่อนคำสั่ง load และ store ใดๆที่อยู่หลัง memory barrier (McKenney, 2005) ทำให้การทำโอเปอเรชันเป็นไปตามลำดับ (Butenhof, 2007, p. 93) ด้วยวิธีการนี้ เทรดจึงทำการรีเซ็ต dirty bit ก่อนที่จะคัดลอกข้อมูลในหน่วยความจำเสมอ

การสร้างไฟล์สถานะของเวอร์ชวลแมชชีน

การสร้างไฟล์สถานะของเวอร์ชวลแมชชีนในงานวิจัยนี้ใช้คำสั่ง “dd” ในโปรโตคอล exec ซึ่งจะรับคำสั่งที่ใช้ในการบันทึกสถานะของเวอร์ชวลแมชชีนลงไฟล์ โดยอาจใช้คำสั่ง “gzip” เพื่อให้ไฟล์สถานะอยู่ในรูปแบบที่มีการบีบอัด หรือคำสั่ง gpg เพื่อให้ไฟล์สถานะอยู่ในรูปแบบที่มีการเข้ารหัส หลังจากไฟล์นี้ถูกสร้างขึ้นมา ข้อมูลสถานะของเวอร์ชวลแมชชีนได้แก่ หน่วยความจำและสถานะของซีพียู จะถูกเขียนลงในไฟล์นี้ ส่วนการทำงานของไอโอ (IO operation) ทั้งหมดจะถูกฟลัชลงดิสก์หลังจากที่หยุดเวอร์ชวลแมชชีนเพื่อบันทึกสถานะในขั้นที่ 3 โดยการเรียก bdrv_flush_all() บนดิสก์อิมเมจไฟล์เดสคริปเตอร์ ซึ่งจะทำให้ไฟล์ซิสเต็มส์ฟลัชข้อมูลทั้งหมดลงดิสก์ เพื่อให้สถานะของดิสก์อิมเมจสอดคล้องกับไฟล์สถานะของเวอร์ชวลแมชชีน

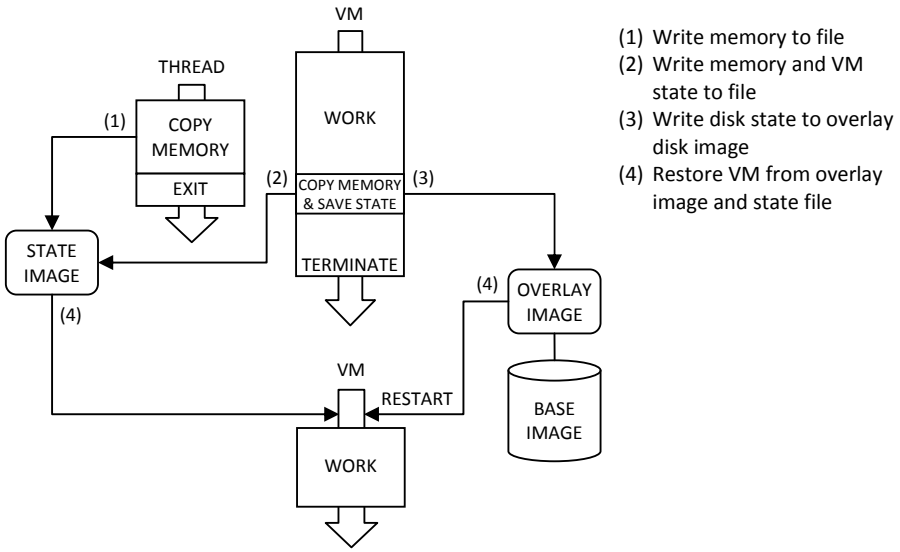
ดิสก์อิมเมจ

ในงานวิจัยนี้ ดิสก์อิมเมจถูกสร้างขึ้นในรูปแบบ qcow2 ของคีมูซึ่งเป็นรูปแบบที่ลดขนาดอิมเมจเพื่อให้ไม่มีพื้นที่ว่างในไฟล์ นั่นคือ ขนาดไฟล์จะโตขึ้นตามการเปลี่ยนแปลงที่ดิสก์อิมเมจจริงเท่านั้น (copy-on-write) และเมื่อได้ติดตั้งระบบปฏิบัติการลงไปแล้ว จะสร้างโอเวอร์เลย์ (overlay image) ขึ้นมาซึ่งมีขนาดเล็กกว่าอิมเมจเดิม (base image) มาก การรันระบบด้วยโอเวอร์เลย์อิมเมจจะทำให้การเปลี่ยนแปลงเกิดขึ้นเฉพาะบนโอเวอร์เลย์ ส่วนเบสอิมเมจจะยังคงมีสภาพเดิม ซึ่งทำให้ง่ายที่จะย้อนกลับสู่สถานะเดิมของระบบ

การรีสตาร์ทเวอร์ชวลแมชชีน

การรีสตาร์ทเวอร์ชวลแมชชีนจากไฟล์สถานะต้องใช้คำสั่งเดียวกับคำสั่งที่ใช้รันเวอร์ชวลแมชชีนนั้น ตามด้วย “-incoming” ที่ระบุคำสั่ง “dd” ในโปรโตคอล exec ให้อ่านไฟล์สถานะที่ได้บันทึกไว้ ดังนั้น ข้อมูลในการรีสตาร์ทจึงประกอบด้วยเบสอิมเมจเดิม โอเวอร์เลย์อิมเมจที่ถูกเปลี่ยนแปลงซึ่งรวมทั้งการทำไอโอ และไฟล์ข้อมูลสถานะของเวอร์ชวลแมชชีน ทำให้เวอร์ชวลแมชชีนสามารถย้อนการทำงานกลับมาอยู่ในจุดที่ทำเช็คพอยน์ได้ ดังภาพที่ 3.13

ภาพที่ 3.13
การรีสตาร์ทเวอร์ชวลแมชชีนจากไฟล์สถานะ



วิธีการทดลอง

1. เครื่องมือที่ใช้ในการทดลอง

1. โฮสแมชชีน มีรายละเอียดดังนี้
   - Architecture: x86_64
   - Processor: Intel Core2 Quad CPU 2.40GHz
   - Memory capacity: 4GB
   - OS: Ubuntu 9.04
   - Kernel: 2.6.28-13-generic
2. เวอร์ชวลแมชชีน มีรายละเอียดดังนี้
   - Processor: 1 CPU ของ Intel Core2 Quad CPU 2.40GHz
   - Memory capacity: 512MB
   - OS: Fedora 11
   - Kernel: 2.6.29.4-167.fc11.x86_64
3. ไฮเปอร์ไวเซอร์ kvm-88
4. เบนช์มาร์ก NPB3.3-SER คือ การอิมพลีเมนต์แบบซีเรียลของเบนช์มาร์ก NPB (NAS Parallel Benchmarks) ซึ่งเป็นชุดโปรแกรมที่ถูกออกแบบมาเพื่อช่วยประเมินประสิทธิภาพของซูเปอร์คอมพิวเตอร์แบบขนาน (Baily et al., 1994, p. 1) โปรแกรมเบนช์มาร์กมาจากแอพพลิเคชัน Computational fluid dynamics (CFD) ประกอบด้วย เคอเนลเบนช์มาร์ก 5 เบนช์มาร์ก ได้แก่ EP (Embarrassingly Parallel), MG (MultiGrid), CG (Conjugate Gradient), FT (3-D fast-Fourier Transform Partial Differential Equation), และ IS (Integer Sort) และแอพพลิเคชันจำลอง 6 แอพพลิเคชัน ได้แก่ แอพพลิเคชัน CFD BT (Block Tridiagonal solver), LU (LU solver), และ SP (Pentadiagonal solver) แอพพลิเคชันด้านข้อมูล DT (Data Traffic), และ DC (Data Cube) และแอพพลิเคชันคำนวณทางเคมี UA (Unstructured Adaptive) ในซีเรียลแพ็คเก็จประกอบด้วย 10 เบนช์มาร์ก ได้แก่ BT, CG, DC, EP, FT, IS, LU, MG, SP, และ UA

2. การวัดประสิทธิภาพ

การประเมินประสิทธิภาพของการเช็คพอยน์เวอร์ชวลแมชชีนโดยใช้เทรด ทำโดยการเปรียบเทียบเวลาที่ใช้ในการเอ็กซิคิวต์ของโปรแกรม NPB เวลาที่ใช้ในการเช็คพอยน์ และขนาดโอเวอร์เฮดของการทำเช็คพอยน์ ระหว่างเวอร์ชวลแมชชีนที่มีการทำเช็คพอยน์แบบหยุดการทำงานตามปกติ แบบไลฟ์เช็คพอยน์ และแบบไลฟ์เช็คพอยน์ที่ใช้เทรด ซึ่งได้จากการจับเวลาดังนี้

1. เวลาที่ใช้ในการเอ็กซิคิวต์ของโปรแกรม NPB (execution time) คือเวลาการทำงานของโปรแกรมในเวอร์ชวลแมชชีนที่มีการจับเวลาภายในโปรแกรมแบบเรียลไทม์ คือสามารถจับเวลาได้ถึงแม้ว่าเวอร์ชวลแมชชีนจะหยุดทำงาน

2. เวลาที่ใช้ในการเช็คพอยน์ (checkpoint latency) คือเวลาที่ใช้ในการทำงานเช็คพอยน์ซึ่งเริ่มตั้งแต่เวอร์ชวลแมชชีนได้รับคำสั่งให้ทำเช็คพอยน์ (ไมเกรชันรีเควสท์แบบไมเกรทลงไฟล์) จนถึงเวลาที่เวอร์ชวลแมชชีนเสร็จสิ้นการบันทึกสถานะ นั่นคือเวลาทั้งหมดที่ใช้ในการทำหนึ่งเช็คพอยน์

3. ขนาดเช็คพอยน์โอเวอร์เฮด (checkpoint overhead) คือเวลาที่ใช้ในการเอ็กซิคิวต์ของโปรแกรม NPB ที่เพิ่มขึ้นเนื่องจากการเช็คพอยน์เวอร์ชวลแมชชีน ค่าโอเวอร์เฮดของแต่ละโปรโตคอลคำนวณจากการหาค่าความแตกต่างระหว่างค่าเฉลี่ยของเวลาการทำงานในแต่ละเบนช์มาร์กที่มีการเช็คพอยน์ด้วยแต่ละโปรโตคอลและค่าเฉลี่ยของเวลาการทำงานในแต่ละเบนช์มาร์กที่ไม่มีการเช็คพอยน์

นอกจากการวัดประสิทธิภาพเหล่านี้ ข้อมูลที่สำคัญที่ควรนำมาพิจารณาร่วมกับผลการประเมินประสิทธิภาพคือเวลาที่เวอร์ชวลแมชชีนหยุดทำงานหรือดาวน์ไทม์ สำหรับการเช็คพอยน์แบบหยุดการทำงาน ค่าดาวน์ไทม์จะเท่ากับเวลาที่ใช้ในการเช็คพอยน์ และสำหรับไลฟ์เช็คพอยน์และไลฟ์เช็คพอยน์ที่ใช้เทรด ค่าดาวน์ไทม์คือเวลาที่หยุดเวอร์ชวลแมชชีนในอัลกอริทึม นั่นคือเวลาที่ทำการเซ็ตอัพรวมกับเวลาที่ทำการบันทึกสถานะในขั้นสุดท้าย

3. เบนช์มาร์ก

การเลือกเบนช์มาร์กเพื่อใช้ในการทดลองพิจารณาจากลักษณะที่แตกต่างกันของโปรแกรมในซีเรียลแพ็คเก็จของ NPB3.3 ตารางที่ 3.1 และ 3.2 แสดงขนาดของปัญหา ขนาดหน่วยความจำที่ต้องการ (Mw) เวลาที่ใช้รันโปรแกรมใน NPB1.0 ที่อิมพลีเมนต์บนหนึ่ง

โปรเซสเซอร์ของ Cray Y-MP (Baily et al., 1994, pp. 8-10) และจำนวน floating-point operation (flop) (Saini & Bailey, 1996, p.7) สำหรับ 8 เบนช์มาร์ก คลาส A และ B

ตารางที่ 3.1

NAS Parallel Benchmarks คลาส A

Benchmark Code	Problem size	Memory (Mw)	Time (sec)	Operation count ($x10^9$)
EP (Embarrassingly Parallel)	2^{28}	1	151	26.68
MG (MultiGrid)	256^3	57	54	3.905
CG (Conjugate Gradient)	14000	10	22	1.508
FT (3-D FFT PDE)	256^2x128	59	39	5.631
IS (Integer Sort)	2^{23}	26	21	0.7812
LU (Lower-Upper Symmetric Gauss-Seidel)	64^3	30	344	64.57
SP (Scalar-Pentadiagonal)	64^3	6	806	102.0
BT (Block-Tridiagonal)	64^3	24	923	181.3

ตารางที่ 3.2

NAS Parallel Benchmarks คลาส B

Benchmark Code	Problem size	Memory (Mw)	Time (sec)	Operation count ($x10^9$)
EP (Embarrassingly Parallel)	2^{30}	18	512	100.9
MG (MultiGrid)	256^3	59	114	18.81
CG (Conjugate Gradient)	75000	97	998	54.89
FT (3-D FFT PDE)	512x256^2	162	366	71.37
IS (Integer Sort)	2^{25}	114	126	3.150
LU (Lower-Upper Symmetric Gauss-Seidel)	102^3	122	1973	319.6
SP (Scalar-Pentadiagonal)	102^3	22	2160	447.1
BT (Block-Tridiagonal)	102^3	96	3554	721.5

นอกจากเบนช์มาร์กที่แสดงในตารางที่ 3.1 และ 3.2 แล้ว NPB3.3 ยังมีเบนช์มาร์กที่ควรถูกพิจารณาเช่นกันคือเบนช์มาร์ก DC เนื่องจากลักษณะที่แตกต่างจากเบนช์มาร์กอื่นคือเป็นเบนช์มาร์กที่จัดการข้อมูลขนาดใหญ่ มีการทำงานที่เรียกว่า Data Cube Operator (DCO) บน Arithmetic Data Set (ADC) โดยซีเรียลเวอร์ชันของ DC ถูกเพิ่มเข้ามาตั้งแต่ NPB3.1 ขนาดของ

ADC และรายละเอียดต่างๆของเบนช์มาร์ก DC แสดงในตารางที่ 3.3 การเลือกคลาสที่เหมาะสมสำหรับการทดลองพิจารณาจากขนาดไฟล์ผลลัพธ์

ตารางที่ 3.3

ขนาดข้อมูล, ขนาด ADC, ขนาดผลลัพธ์ และรายละเอียดอื่นๆของเบนช์มาร์ก DC

	U	S	W	A	B
Dimensions	-	5	10	15	20
Generator period	-	88200	9699690	653119005	306037160385
Number of tuples	-	10^3	10^5	10^6	10^7
ADC size	-	28 KB	4.8 MB	68 MB	880 MB
Views generated	all	all	all	$0{:}2^{15} - 1{:}2^6$	$0{:}2^{20} - 1{:}2^{14}$
Output size	-	547 KB	2.594 GB	17.84 GB	30.84 GB
Number of generated tuples	-	29232	89297411	454765673	595267023
Verification checksum	-	464620213	1401796434318	7141688178042	9348365700453

ที่มา: “Benchmarking Memory Performance with the Data Cube Operator,” โดย Frumkin, M. A., & Shabanov, L. V., 2004, *NAS Technical Report NAS-04-013*, น. 14.

ดังนั้น เบนช์มาร์กที่ถูกเลือกมาใช้ในการทดลองได้แก่ เบนช์มาร์ก EP.B, BT.A, CG.B, และ DC.W การรันเบนช์มาร์กเหล่านี้เพื่อค้นหาการใช้หน่วยความจำของแต่ละเบนช์มาร์กและขนาดหน่วยความจำของเวอร์ชวลแมชชีนที่มีการใช้งานขณะที่รันแต่ละเบนช์มาร์กในงานวิจัยนี้แสดงในตารางที่ 3.4 เบนช์มาร์กถูกรันบนโฮสแมชชีนที่ใช้ในการทดลอง ขนาดหน่วยความจำที่ถูกใช้โดยแต่ละโปรแกรมได้มาจากการใช้คำสั่ง pmap ขนาดหน่วยความจำของเวอร์ชวลแมชชีนที่ใช้ในขณะรันโปรแกรม (active memory) ได้มาจากการใช้คำสั่ง vmstat

ตารางที่ 3.4

การใช้หน่วยความจำของเบนช์มาร์กที่ใช้ในการทดลองและขนาด active memory ของเวอร์ชวลแมชชีน

Benchmark Code	Memory usage (Kbytes)	VM active memory (Kbytes)
EP.B	3752	170792
BT.A	47200	220288
CG.B	191112	197056
DC.W	11632	191848

เบนช์มาร์ก EP (Embarrassingly Parallel) คือเคอเนลเบนช์มาร์กที่เก็บสถิติจากตัวเลขสุ่มแบบเกาส์เซียน (Gaussian) ขนาดใหญ่ที่ถูกสร้างขึ้นตามรูปแบบที่เหมาะสมสำหรับการคำนวณแบบขนาน ปัญหานี้คือปัญหาปกติของแอพพลิเคชันมอนติคาร์โล (Monte Carlo) (Saini & Bailey, 1996, p.3) ด้วยอัลกอริทึมที่มีประสิทธิภาพสูงของเบนช์มาร์ก EP มันจึงเป็นเบนช์มาร์กที่ใช้วัดประสิทธิภาพในทางคำนวณและมีอัตราการเข้าถึงหน่วยความจำต่ำ เบนช์มาร์กนี้จึงถูกใช้เป็นตัวแทนของแอพพลิเคชันที่ต้องการใช้หน่วยความจำและมีการเขียนหน่วยความจำน้อย

เบนช์มาร์ก CG (Conjugate Gradient) ใช้วิธีคอนจูเกตเกรเดียนต์ในการคำนวณการประมาณค่าลักษณะเฉพาะ (eigenvalue) ที่น้อยที่สุดของเมทริกซ์มากเลขศูนย์สมมาตรขนาดใหญ่ที่มีค่าลักษณะเฉพาะทั้งหมดเป็นบวก เคอเนลเบนช์มาร์กนี้คือการคำนวณกริดแบบไร้โครงสร้างและใช้การคูณเมทริกซ์เวกเตอร์ (Saini & Bailey, 1996, p.4) เบนช์มาร์กนี้ถูกใช้เป็นตัวแทนของแอพพลิเคชันที่ต้องการใช้หน่วยความจำมาก

เบนช์มาร์ก BT (Block Tridiagonal) คือแอพพลิเคชัน CFD จำลองที่แก้ระบบสมการสามแนวทแยงแบบบล็อกที่ไม่มีลักษณะเด่นแนวทแยงด้วยบล็อกขนาด 5x5 ที่เป็นอิสระจากกันหลายระบบ (Saini & Bailey, 1996, p. 5) โดยมีลักษณะการอ้างถึงหน่วยความจำแบบปกติ (data locality) ที่ดีกว่า CG เบนช์มาร์กนี้ถูกใช้เป็นตัวแทนของแอพพลิเคชันที่มีการเขียนหน่วยความจำมาก

เบนช์มาร์ก DC (Data Cube) คือแอพพลิเคชันด้านข้อมูลที่สร้างข้อมูลขนาดใหญ่ขึ้นมาและสามารถใช้วัดประสิทธิภาพของหน่วยความจำระดับใดก็ได้ ตั้งแต่ระดับ L1 จนถึงที่เก็บข้อมูลแบบกระจายและ I/O โดยเบนช์มาร์กนี้จะรับชุดข้อมูลสังเคราะห์ที่เป็นพารามิเตอร์ขนาดเล็กและสร้างมุมมองหลายๆแบบของข้อมูลชุดนี้ขึ้นมา ซึ่งสามารถจัดอยู่ในกลุ่มการจัดเรียงหลายมิติได้ สิ่งที่ใช้วัดประสิทธิภาพของเบนช์มาร์กนี้คือจำนวนข้อมูลประกอบที่ถูกสร้างขึ้นในหนึ่งวินาที หรือ TUPS (**TU**ples generated **P**er **S**econd) ซึ่งเป็นตัวแทนของอัตราความเร็วในการสร้างข้อมูล (Frumkin & Shabanov, 2004, p. 2-3) เนื่องจากเบนช์มาร์กนี้สร้างข้อมูลขนาดใหญ่ โดยขนาดไฟล์ข้อมูลที่สร้างขึ้นมาจะขึ้นอยู่กับขนาดข้อมูลนำเข้า (input data) เบนช์มาร์กนี้จึงเป็นตัวแทนของแอพพลิเคชันที่มีข้อมูลปริมาณมาก (data-intensive) ในงานวิจัยนี้ใช้คลาส W ของเบนช์มาร์กนี้ซึ่งจะสร้างไฟล์ขนาด 2.4GB

4. การออกแบบการทดลอง

เบนช์มาร์กที่ถูกเลือกมาใช้ในการทดลองทั้งหมดจะรันในเวอร์ชวลแมชชีนโดยมีการทำเช็คพอยน์เวอร์ชวลแมชชีนโดยใช้เช็คพอยน์โปรโตคอลที่มีวิธีการแตกต่างกัน ดังนี้

1. เช็คพอยน์โปรโตคอลแบบหยุดเวอร์ชวลแมชชีน (freeze checkpoint)
2. ไลฟ์เช็คพอยน์โปรโตคอล (live checkpoint)
3. ไลฟ์เช็คพอยน์โปรโตคอลแบบเทรด (thread-based live checkpoint)

ข้อมูลจากการทดลองประกอบด้วยเวลาการทำงาน (execution time) ของแต่ละเบนช์มาร์ก เวลาที่ใช้ในการเช็คพอยน์ (checkpoint latency) เวลาที่เวอร์ชวลแมชชีนหยุดทำงาน (downtime) และขนาดของไฟล์สถานะ การวิเคราะห์ข้อมูลคิดจากค่าเฉลี่ยของการรัน 10 ครั้ง

การวิเคราะห์ข้อมูล

1. เปรียบเทียบเวลาที่ใช้ในการทำงานของแต่ละเบนช์มาร์กที่รันในเวอร์ชวลแมชชีนที่มีการทำเช็คพอยน์ระหว่างวิธีการที่แตกต่างกัน 3 แบบโดยใช้การทดสอบทางสถิติด้วย One-Way ANOVA
2. เปรียบเทียบเวลาที่ใช้ในการเช็คพอยน์ของแต่ละเบนช์มาร์กที่รันในเวอร์ชวลแมชชีนที่มีการทำเช็คพอยน์ระหว่างวิธีการที่แตกต่างกัน 3 แบบโดยใช้การทดสอบทางสถิติด้วย One-Way ANOVA
3. วิเคราะห์ขนาดโอเวอร์เฮดของแต่ละวิธีการเช็คพอยน์ โดยเปรียบเทียบกับเวลาการทำงานของแต่ละเบนช์มาร์กที่รันโดยไม่มีการเช็คพอยน์

การทดลองเบื้องต้น

ในการทดลองเบื้องต้น ทำการเปรียบเทียบเวลาในการทำงานของเบนช์มาร์ก CG คลาส B และ LU คลาส A ที่รันในเวอร์ชวลแมชชีนและวิเคราะห์ขนาดโอเวอร์เฮดของการเช็คพอยน์เวอร์ชวลแมชชีนระหว่างวิธีการที่แตกต่างกัน 3 แบบ การวิเคราะห์ข้อมูลคิดจากค่าเฉลี่ยของการรัน 5 ครั้งในแต่ละกรณี การใช้หน่วยความจำของทั้งสองเบนช์มาร์กและขนาดหน่วยความจำของเวอร์ชวลแมชชีนที่ใช้ในขณะรันแต่ละเบนช์มาร์ก (active memory) แสดงในตารางที่ 3.5

ตารางที่ 3.5
การใช้หน่วยความจำของเบนช์มาร์กที่ใช้ในการทดลองเบื้องต้นและ
ขนาด active memory ของเวอร์ชวลแมชชีน

Benchmark Code	Memory usage (Kbytes)	VM active memory (Kbytes)
CG.B	191112	197056
LU.A	42168	208340

เบนช์มาร์ก LU (lower-upper diagonal) คือแอพพลิเคชัน CFD จำลองที่ไม่ได้ใช้วิธีแยกตัวประกอบแอล-ยู (LU factorization) แต่ใช้วิธีการเชิงตัวเลข SSOR (symmetric successive over-relaxation) เพื่อแก้ปัญหาระบบสามเหลี่ยมบนและล่างของเมทริกซ์มากเลขศูนย์ที่เป็นบล็อกขนาด 5x5 (Saini & Bailey, 1996, p.4) โดยมีลักษณะคล้ายกับ BT คือมีการเขียนหน่วยความจำมาก ส่วนเบนช์มาร์ก CG เป็นเบนช์มาร์กที่มีความต้องการใช้หน่วยความจำมาก

1. ผลการทดลองเบื้องต้น

ผลการทดลองประกอบด้วยค่าเฉลี่ยและส่วนเบี่ยงเบนมาตรฐานของเวลาในการทำงานของเบนช์มาร์ก CG คลาส B และ LU คลาส A เวลาในการเช็คพอยน์เวอร์ชวลแมชชีน และเวลาที่เครื่องหยุดทำงาน และการทดสอบความแตกต่างระหว่างวิธีการเช็คพอยน์ที่แตกต่างกันด้วย One-Way ANOVA รวมทั้งการประเมินโอเวอร์เฮดของแต่ละวิธีการเช็คพอยน์

ตารางที่ 3.6
ค่าเฉลี่ยและส่วนเบี่ยงเบนมาตรฐานของเวลาการทำงาน เวลาเช็คพอยน์ และเวลาที่เวอร์ชวลแมชชีนหยุดทำงาน ของเบนช์มาร์ก CG คลาส B สำหรับการทดลองเบื้องต้น

Checkpoint protocol	Execution time (s)		Checkpoint latency (s)		Downtime (s)	
	Mean	S.D.	Mean	S.D.	Mean	S.D.
Freeze checkpoint	116.08	1.43	13.99	.04	13.99	.04
Live checkpoint	110.99	2.38	14.61	.28	1.06	1.31
Thread-based live checkpoint	102.23	1.43	6.11	.29	.11	.02

จากตารางที่ 3.6 พบว่าเวลาในการทำงานของเบนช์มาร์ก CG คลาส B ที่รันในเวอร์ชวลแมชชีนที่มีการทำเช็คพอยน์ด้วยวิธีการที่ใช้เทรดในไลฟ์เช็คพอยน์ใช้เวลาน้อยที่สุด ($\bar{X}$=102.23; S.D.=1.43) และการรันเบนช์มาร์กที่มีการเช็คพอยน์แบบหยุดการทำงานของเครื่องใช้เวลามากที่สุด ($\bar{X}$=116.08; S.D.=1.43) สำหรับเวลาในการเช็คพอยน์ วิธีการที่ใช้เทรดในไลฟ์เช็คพอยน์ใช้เวลาน้อยที่สุด ($\bar{X}$=6.11; S.D.=.29) และวิธีการไลฟ์เช็คพอยน์ปกติใช้เวลาเช็คพอยน์มากที่สุด ($\bar{X}$=14.61; S.D.=.28)

ตารางที่ 3.7

ค่าเฉลี่ยและส่วนเบี่ยงเบนมาตรฐานของเวลาการทำงาน เวลาเช็คพอยน์ และเวลาที่เวอร์ชวลแมชชีนหยุดทำงาน ของเบนช์มาร์ก LU คลาส A สำหรับการทดลองเบื้องต้น

Checkpoint protocol	Execution time (s)		Checkpoint latency (s)		Downtime (s)	
	Mean	S.D.	Mean	S.D.	Mean	S.D.
Freeze checkpoint	103.25	1.69	12.36	.10	12.36	.10
Live checkpoint	107.89	1.84	102.11	.86	.17	.22
Thread-based live checkpoint	94.06	2.03	6.52	.93	.15	.07

จากตารางที่ 3.7 พบว่าเวลาในการทำงานของเบนช์มาร์ก LU คลาส A ที่รันในเวอร์ชวลแมชชีนที่มีการทำเช็คพอยน์ด้วยวิธีการที่ใช้เทรดในไลฟ์เช็คพอยน์ใช้เวลาน้อยที่สุด ($\bar{X}$=94.06; S.D.=2.03) และการรันเบนช์มาร์กที่มีการเช็คพอยน์แบบไลฟ์เช็คพอยน์ปกติใช้เวลามากที่สุด ($\bar{X}$=107.89; S.D.=1.84) สำหรับเวลาในการเช็คพอยน์ วิธีการที่ใช้เทรดในไลฟ์เช็คพอยน์ใช้เวลาน้อยที่สุด ($\bar{X}$=6.52; S.D.=.93) และวิธีการไลฟ์เช็คพอยน์ปกติใช้เวลาเช็คพอยน์มากที่สุด ($\bar{X}$=102.11; S.D.=.86)

ตารางที่ 3.8

การทดสอบความเท่ากันของความแปรปรวนของเวลาการทำงานและเวลาเช็คพอยน์ของเบนช์มาร์ก CG คลาส B และ LU คลาส A ระหว่างวิธีการเช็คพอยน์ที่แตกต่างกัน

Benchmark		Levene Statistic	df1	df2	Sig.
CG.B	Execution time	.546	2	12	.593
	Checkpoint latency	3.652	2	12	.058
LU.A	Execution time	.532	2	12	.600
	Checkpoint latency	3.094	2	12	.083

จากตารางที่ 3.8 พบว่าความแปรปรวนของเวลาการทำงานและเวลาที่ใช้ในการเช็คพอยน์ของเบนช์มาร์ก CG คลาส B และ LU คลาส A ระหว่างวิธีการเช็คพอยน์ที่ต่างกัน 3 แบบ ไม่แตกต่างกันอย่างมีนัยสำคัญ

ตารางที่ 3.9

การทดสอบความแตกต่างของเวลาการทำงานและเวลาเช็คพอยน์ของเบนช์มาร์ก CG คลาส B ระหว่างวิธีการเช็คพอยน์ที่แตกต่างกันด้วย F-test

CG.B		Sum of Squares	df	Mean Square	F	Sig.
Execution time	Between Groups	490.503	2	245.252	75.494	.000
	Within Groups	38.983	12	3.249		
	Total	529.487	14			
Checkpoint latency	Between Groups	224.657	2	112.329	2099.862	.000
	Within Groups	.642	12	.053		
	Total	225.299	14			

จากตารางที่ 3.9 พบว่าเวลาที่ใช้ในการทำงานและเวลาเช็คพอยน์ของเบนช์มาร์ก CG คลาส B ที่มีการเช็คพอยน์เวอร์ชวลแมชชีนด้วยวิธีการที่ต่างกัน 3 แบบ มีความแตกต่างกันอย่างมีนัยสำคัญ

ตารางที่ 3.10

การทดสอบความแตกต่างของเวลาการทำงานและเวลาเช็คพอยน์ของเบนช์มาร์ก CG คลาส B ระหว่างวิธีการเช็คพอยน์ที่แตกต่างกันเป็นรายคู่ด้วย Scheffe

CG.B	(I) protocol	(J) protocol	Mean Difference (I-J)	Sig.
Execution time	freeze	live	5.08800*	.003
		thread-based	13.84600*	.000
	live	freeze	-5.08800*	.003
		thread-based	8.75800*	.000
	thread-based	freeze	-13.84600*	.000
		live	-8.75800*	.000
Checkpoint latency	freeze	live	-.62000*	.004
		thread-based	7.88200*	.000
	live	freeze	.62000*	.004
		thread-based	8.50200*	.000
	thread-based	freeze	-7.88200*	.000
		live	-8.50200*	.000

*p<.05

จากตารางที่ 3.10 พบว่าเวลาการทำงานของเบนช์มาร์ก CG คลาส B ที่มีการเช็คพอยน์เวอร์ชวลแมชชีนด้วยวิธีการไลฟ์เช็คพอยน์ปกติใช้เวลาน้อยกว่าการเช็คพอยน์แบบหยุดการทำงานของเครื่องอย่างมีนัยสำคัญ (p<.05) และการเช็คพอยน์ด้วยวิธีการที่ใช้เทรดในไลฟ์เช็คพอยน์ใช้เวลาน้อยกว่าการทำงานที่มีการเช็คพอยน์แบบหยุดการทำงานของเครื่องและแบบไลฟ์เช็คพอยน์ปกติอย่างมีนัยสำคัญ (p<.05)

ส่วนเวลาที่ใช้ในการเช็คพอยน์ของเบนช์มาร์ก CG คลาส B พบว่าการเช็คพอยน์ด้วยวิธีการที่ใช้เทรดในไลฟ์เช็คพอยน์ใช้เวลาน้อยกว่าเวลาเช็คพอยน์ที่หยุดการทำงานของเครื่องและแบบไลฟ์เช็คพอยน์ปกติอย่างมีนัยสำคัญ (p<.05)

ตารางที่ 3.11

การทดสอบความแตกต่างของเวลาการทำงานและเวลาเช็คพอยน์ของเบนช์มาร์ก LU คลาส A ระหว่างวิธีการเช็คพอยน์ที่แตกต่างกันด้วย F-test

LU.A		Sum of Squares	df	Mean Square	F	Sig.
Execution time	Between Groups	495.178	2	247.589	71.445	.000
	Within Groups	41.585	12	3.465		
	Total	536.763	14			
Checkpoint latency	Between Groups	28707.880	2	14353.940	26483.772	.000
	Within Groups	6.504	12	.542		
	Total	28714.384	14			

จากตารางที่ 3.11 พบว่าเวลาที่ใช้ในการทำงานและเวลาเช็คพอยน์ของเบนช์มาร์ก LU คลาส A ที่มีการเช็คพอยน์เวอร์ชวลแมชชีนด้วยวิธีการที่ต่างกัน 3 แบบ มีความแตกต่างกันอย่างมีนัยสำคัญ

ตารางที่ 3.12

การทดสอบความแตกต่างของเวลาการทำงานและเวลาเช็คพอยน์ของเบนช์มาร์ก LU คลาส A ระหว่างวิธีการเช็คพอยน์ที่แตกต่างกันเป็นรายคู่ด้วย Scheffe

LU.A	(I) protocol	(J) protocol	Mean Difference (I-J)	Sig.
Execution time	freeze	live	-4.63600*	.007
		thread-based	9.19000*	.000
	live	freeze	4.63600*	.007
		thread-based	13.82600*	.000
	thread-based	freeze	-9.19000*	.000
		live	-13.82600*	.000
Checkpoint latency	freeze	live	-89.74600*	.000
		thread-based	5.83800*	.000
	live	freeze	89.74600*	.000
		thread-based	95.58400*	.000
	thread-based	freeze	-5.83800*	.000
		live	-95.58400*	.000

*p<.05

จากตารางที่ 3.12 พบว่าเวลาการทำงานของเบนช์มาร์ก LU คลาส A ที่มีการเช็คพอยน์เวอร์ชวลแมชชีนด้วยวิธีการไลฟ์เช็คพอยน์ปกติใช้เวลามากกว่าการเช็คพอยน์แบบหยุดการทำงานของเครื่องอย่างมีนัยสำคัญ ($p<.05$) และการเช็คพอยน์ด้วยวิธีการที่ใช้เทรดในไลฟ์เช็คพอยน์ใช้เวลาน้อยกว่าการทำงานที่มีการเช็คพอยน์แบบหยุดการทำงานของเครื่องและแบบไลฟ์เช็คพอยน์ปกติอย่างมีนัยสำคัญ ($p<.05$)

ส่วนเวลาที่ใช้ในการเช็คพอยน์ของเบนช์มาร์ก LU คลาส A พบว่าการเช็คพอยน์ด้วยวิธีการที่ใช้เทรดในไลฟ์เช็คพอยน์ใช้เวลาเช็คพอยน์น้อยกว่าการเช็คพอยน์แบบหยุดการทำงานของเครื่องและแบบไลฟ์เช็คพอยน์ปกติอย่างมีนัยสำคัญ ($p<.05$)

ตารางที่ 3.13

ค่าเฉลี่ยของขนาดของไฟล์สถานะเวอร์ชวลแมชชีนที่รันเบนช์มาร์ก CG คลาส B และ LU คลาส A

Benchmark	Checkpoint protocol	Size of state file (MB)	Restart time (s)
CG.B	Freeze checkpoint	456.18	5.17
	Live checkpoint	469.74	5.33
	Thread-based live checkpoint	528.08	6.15
LU.A	Freeze checkpoint	400.28	4.08
	Live checkpoint	3256.32	51.53
	Thread-based live checkpoint	566.40	5.64

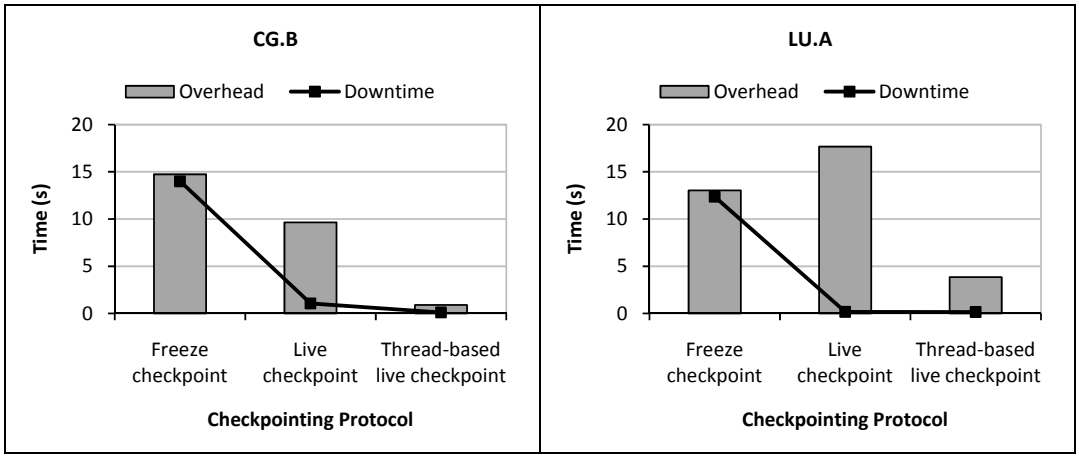
จากตารางที่ 3.13 พบว่าโดยปกติ การเช็คพอยน์ด้วยวิธีการไลฟ์เช็คพอยน์จะทำให้ขนาดของไฟล์สถานะใหญ่กว่าการเช็คพอยน์แบบหยุดการทำงานของเครื่อง เนื่องจากหน่วยความจำถูกเปลี่ยนแปลงได้ในขณะที่ทำเช็คพอยน์ ในกรณีของเบนช์มาร์ก LU คลาส A การเช็คพอยน์ด้วยวิธีการไลฟ์เช็คพอยน์ปกติจะสร้างไฟล์สถานะที่มีขนาดใหญ่กว่าการเช็คพอยน์ด้วยวิธีการอื่นอย่างเห็นได้ชัดซึ่งเกิดจากการใช้เวลาเช็คพอยน์ที่นานมากเกินไป

การอิมพลีเมนต์โปรโตคอลในการทดลองเบื้องต้นนี้สามารถสร้างเช็คพอยน์ไฟล์และรีสตาร์ทเวอร์ชวลแมชชีนให้ทำงานต่อได้อย่างถูกต้องถึงแม้ว่าจะไม่มีการทำซิงโครไนเซชันและการทำเมมโมรีแบเรียร์ จากตารางที่ 3.13 เวลาในการรีสตาร์ทเวอร์ชวลแมชชีนจะขึ้นอยู่กับขนาดของเช็คพอยน์ไฟล์

ตารางที่ 3.14
โอเวอร์เฮดของวิธีการเช็คพอยน์แต่ละแบบในการทดลองเบื้องต้น

Benchmark	Checkpoint protocol	Overhead time	
		(s)	(%)
CG.B	Freeze checkpoint	14.74	14.55%
	Live checkpoint	9.65	9.52%
	Thread-based live checkpoint	0.89	0.88%
LU.A	Freeze checkpoint	13.03	14.44%
	Live checkpoint	17.67	19.58%
	Thread-based live checkpoint	3.84	4.25%

ภาพที่ 3.14
เช็คพอยน์โอเวอร์เฮดและดาวน์ไทม์ของวิธีการเช็คพอยน์แต่ละแบบในการทดลองเบื้องต้น



ตารางที่ 3.14 และภาพที่ 3.14 แสดงเช็คพอยน์โอเวอร์เฮดของวิธีการเช็คพอยน์แต่ละแบบ โดยเปรียบเทียบกับเวลาการทำงานของเบนช์มาร์ก CG คลาส B ($\bar{X}$=101.34; S.D.=.96) และ LU คลาส A ($\bar{X}$=90.22; S.D.=.95) ในเวอร์ชวลแมชชีนที่ไม่มีการทำเช็คพอยน์ และเวลาที่เวอร์ชวลแมชชีนหยุดทำงาน (ดาวน์ไทม์) ซึ่งพบว่าวิธีการที่ใช้เทรดในไลฟ์เช็คพอยน์มีโอเวอร์เฮดน้อยที่สุดทั้งเบนช์มาร์ก CG คลาส B และ LU คลาส A (0.88% และ 4.25% ตามลำดับ) วิธีการแบบหยุดการทำงานของเครื่องมีโอเวอร์เฮดมากที่สุด (14.55%) ใน CG คลาส B และวิธีการไลฟ์เช็คพอยน์ปกติมีโอเวอร์เฮดมากที่สุด (19.58%) ใน LU คลาส A

2. การอภิปรายและสรุปผลการทดลองเบื้องต้น

จากผลการทดลองเบื้องต้น วิธีการเช็คพอยน์ที่ใช้เทรดในไลฟ์เช็คพอยน์เป็นเทคนิคที่มีประสิทธิภาพดีที่สุด นั่นคือ เวลาการทำงานของแอพพลิเคชันที่เพิ่มขึ้นจากการเช็คพอยน์เวอร์ชวลแมชชีนมีน้อยมาก เนื่องจากเวอร์ชวลแมชชีนสามารถทำงานไปโดยที่เทรดช่วยทำงานเช็คพอยน์โดยส่วนใหญ่ให้ รวมทั้งการใช้เวลาเช็คพอยน์ที่น้อย ทำให้ประสิทธิภาพดีกว่าวิธีการหยุดเวอร์ชวลแมชชีนซึ่งเป็นวิธีการตามปกติของเควีเอ็ม และดีกว่าวิธีการที่ใช้ไลฟ์เช็คพอยน์โดยตรง ที่ทำให้เวลาการทำงานของแอพพลิเคชันเพิ่มขึ้นมากกว่า และใช้เวลาทำเช็คพอยน์มากกว่า

สำหรับแอพพลิเคชันที่ใช้หน่วยความจำมากแต่มีการเขียนหน่วยความจำไม่มากเกินไป เช่น เบนช์มาร์ก CG วิธีการไลฟ์เช็คพอยน์สามารถช่วยปรับปรุงประสิทธิภาพการเช็คพอยน์ให้ดีขึ้นได้ โดยทำให้เวลาการทำงานของแอพพลิเคชันลดลงจากการเช็คพอยน์ด้วยวิธีการเดิม อย่างไรก็ตาม ในกรณีของแอพพลิเคชันที่มีการเขียนหน่วยความจำมาก เช่น เบนช์มาร์ก LU วิธีการไลฟ์เช็คพอยน์จะทำให้เวลาการทำงานของแอพพลิเคชันเพิ่มขึ้น ซึ่งเวลาการเช็คพอยน์ที่ยาวนานทำให้โอเวอร์เฮดเพิ่มขึ้นไปด้วย ส่วนการใช้เทรดเพื่อช่วยทำเช็คพอยน์ในวิธีการของไลฟ์เช็คพอยน์ เวลาการเช็คพอยน์ยังคงน้อยเช่นเดิมทั้งในกรณีของแอพพลิเคชันที่ใช้หน่วยความจำมากและแอพพลิเคชันที่มีการเขียนหน่วยความจำมาก การเช็คพอยน์ที่ใช้เวลานานของวิธีการไลฟ์เช็คพอยน์ปกติเกิดขึ้นเนื่องจากในระหว่างการคัดลอกหน่วยความจำแต่ละรอบ เพจของหน่วยความจำจะถูกเปลี่ยนแปลงมากจนทำให้การหยุดคัดลอกหน่วยความจำตามเงื่อนไขต้องล่าช้าออกไปเรื่อยๆ ซึ่งเกิดจากการที่เวอร์ชวลแมชชีนจะต้องทำงานตามปกติและทำเช็คพอยน์สลับกันไป ในขณะที่การใช้เทรดเพื่อช่วยทำการคัดลอกหน่วยความจำแทน แต่ละรอบของการคัดลอกหน่วยความจำจะมีจำนวนเพจของหน่วยความจำที่ถูกเปลี่ยนแปลงน้อยกว่า เพราะเทรดทำงานคัดลอกหน่วยความจำได้เร็วกว่าวิธีการไลฟ์เช็คพอยน์ปกติ ทำให้การหยุดคัดลอกหน่วยความจำตามเงื่อนไขเกิดขึ้นได้เร็วกว่า ดังนั้นปัญหาของการใช้เวลาเช็คพอยน์ที่นานเกินไปจึงเป็นข้อเสียที่สำคัญของวิธีการไลฟ์เช็คพอยน์ปกติ ในกรณีที่การหยุดคัดลอกหน่วยความจำเกิดขึ้นเมื่อแอพพลิเคชันทำงานเสร็จ การเช็คพอยน์นั้นก็จะไม่มีประโยชน์

วิธีการไลฟ์เช็คพอยน์สามารถช่วยให้ประสิทธิภาพของการบันทึกสถานะของเวอร์ชวลแมชชีนดีขึ้นได้ แต่มีข้อเสียคือไม่เหมาะกับแอพพลิเคชันที่มีการเขียนหน่วยความจำมาก เพราะถ้าแอพพลิเคชันมีการเปลี่ยนแปลงหน่วยความจำมาก จะทำให้จำนวนหน่วยความจำที่จะต้องคัดลอกในแต่ละรอบมากตามไปด้วย การใช้เทรดเพื่อช่วยทำเช็คพอยน์สามารถแก้ปัญหานี้

ของไลฟ์เช็คพอยน์ได้ เพราะเทรดสามารถทำงานคัดลอกได้อย่างรวดเร็วโดยอิสระ รวมทั้งยังทำให้โอเวอร์เฮดลดลง แอพพลิเคชันทำงานเสร็จเร็วขึ้น ในขณะที่เวลาที่เครื่องหยุดทำงานยังคงน้อยมากเช่นเดิม