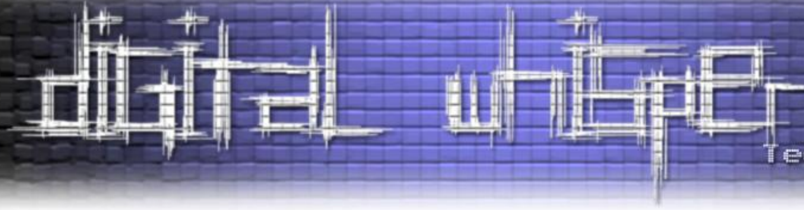



# טכניקות אופטימיזציה של קוד

מאת מיכאל שליטין

## הקדמה

בעולם היום, אנו רגלים לכתוב קוד ולא תמיד מייחסים חשיבות ליעלות שלו. זאת, בעיקר מפני שאנו יכולים להניח כי הקומפיילר יבצע את האופטימיזציות הדרושות. אך במקרים שבהם נדרש להקפיד על ביצועים גבוהים במיוחד, אנו נדרש לפעמים להבין בדיוק מה קורה בפועל בעת הרצת הקוד שכתבנו.

בהמשך למאמר הקודם על קוהרנטיות cache, אנו נצלול לתוך טכניקות אופטימיזציות שונות שבהן מעבדים וקומפיילרים משתמשים על מנת לייעל ולשפר את ביצועי התוכניות שלנו.

במאמר הקרוב אנו נתחיל מסקירה קצרה יחסית של ה-store buffer, משם נמשיך ונראה איך המעבד מבצע שליפת מידע מהזיכרון ולאחר מכן נלמד על ספקולציות שהמעבד עושה שהיא טכניקה חשובה שבה המעבדים המודרנים משתמשים. אחרי כל אלה נעבור להסתכל מזווית קצת שונה ונבדוק מספר טכניקות ודרכים שבהם הקומפיילרים היום מנסים לבצע אופטימיזציות.

**הערה:** את כל החומר הזה למדתי מהאינטרנט ללא כל הכוונה ואין לי רקע אקדמי. כך שיכולות להיות טעויות במאמר. אני מניח שיש למגזין קוראים שמבינים טוב ממני את הנושא, אז אם אתם רואים טעויות - אשמח שתתקנו אותי!

## Store buffer

### מבוא

המעבדים היום הרבה יותר מהירים מהזיכרון הראשי, ולכן כל גישה לזיכרון יכולה לגרום לעיכוב משמעותי בביצועי המעבד. לכן מעבדים מנסים לצמצם ככל הניתן את העיכוב הזה, ובשל כך פותחו טכניקות המנסות לנתק כמה שניתן את ביצועי המעבד מהזיכרון עצמו, כמו cache-ים וכל מיני סוגי buffer-ים יעודיים.

אחד ה-buffer-ים המרכזיים הוא ה-store buffer (ידוע גם כ-write buffer) והמטרה שלו היא לנתק את ביצוע הכתיבה של המעבד. ה-store buffer מאפשר למעבד לשגר את הוראת הכתיבה ואז הוא יכול "לשכוח" מהכתיבה וכל האחריות של ביצוע הכתיבה וכל מה שכרוך בה עוברת ל-store buffer.

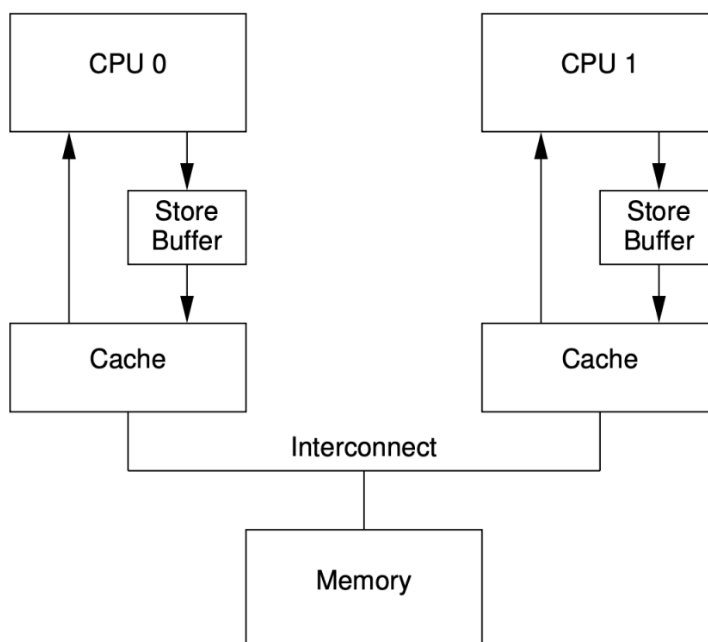
ה-store buffer מספק טכניקה ידועה שנעשה בה שימוש ברוב המעבדים והארכיטקטורות (כמו x86, ARM, AMD64, PPC ועוד רבות) ולכן יש ל-store buffer כל מיני מימושים והתנהגויות שונות בין הארכיטקטורות, במאמר הזה אני אתאר בצורה כללית ה-store buffer ואגע קצת בגרסאות ספציפיות.

### Store buffer

ה-store buffer הוא סוג של זיכרון ביניים המשמש לאחסון נתונים הנכתבים מהמעבד ל-cache. במערכות מרובות ליבות, השימוש ב-store buffers יכול לפגוע בעקביות הרצף של הזיכרון.

ה-store buffer מאפשר ל-cache להמשיך לטפל בבקשות קריאה בזמן שהכתיבה מתבצעת ברקע, מה שמועיל במיוחד כשמדובר בזיכרון ראשי איטי. במצב זה, קריאות נוספות יכולות להתבצע מבלי להמתין לזיכרון הראשי. אם ה-store buffer מתמלא (כלומר, כל ה-slot-ים תפוסים), הכתיבה הבאה תיאלץ להמתין עד ש-slot-ים יתפנו.

בדומה ל-cache, הוא ממוקם בין הליבה לזיכרון הראשי, ומאפשר לליבה להמשיך בביצוע ההוראה הבאה מבלי להמתין לסיום פעולת הכתיבה בזיכרון הראשי:




[With Store Buffers from Caches With Store Buffers (www.kernel.org)]

גם במעבדים שפועלים בסדר עיבוד פשוט, קיימת תועלת בשימוש ב-store buffer. השימוש ב-store buffer מאפשר למעבד להימנע מהשהיות שנגרמות כתוצאה מ-cache miss במהלך פעולות כתיבה.

בפרקטיקה, מעבדים אינם נדרשים לשמור על סדר רציף של פעולות כתיבה, בניגוד לפעולות קריאה (load), ולכן אין צורך שהכתיבות יהיו זמינות מיד. ה-store buffer מאפשר למעבד להמשיך בביצוע הוראות נוספות, מבלי להמתין לכך שהכתיבה ל-cache תושלם.

עוד דרך להבין את ההתנהגות של ה-store buffer היא להסתכל עליו כאל סוג של register renaming.

כדי לשפר את היעילות ולהפחית את העומס על המערכת, ניתן ליישם אופטימיזציה בשם "store combining buffer", שמאחד כתיבות בעלות כתובות יעד עוקבות ל-entry אחד ב-buffer. פעולה זו מונעת מהכתיבות לתפוס מקומות נפרדים ב-buffer, ובכך מפחיתה את הסיכוי לעיכובים ב-pipeline של המעבד.

ה-store buffer מקבל את פרטי הכתובת, והנתונים הקשורים לפעולת הכתיבה של הליבה לזיכרון. כאשר הליבה מבצעת הוראת store, היא יכולה להכניס ל-buffer את הפרטים הרלוונטיים, כמו מיקום הכתיבה, הנתונים וגודל הטרנזקציה.

כך, הליבה לא צריכה להמתין לסיום הכתיבה בזיכרון הראשי ויכולה להמשיך בביצוע ההוראה הבאה, בעוד שה-store buffer דואג לניקוז הכתיבה אל מערכת הזיכרון.

ה-store buffer יכול לשפר את ביצועי המערכת, כיוון שהליבה לא נאלצת להמתין לסיום פעולת ה-store. למעשה, כל עוד יש מקום ב-store buffer, הוא מספק דרך להסתיר את השהיית הכתיבה. אם מספר פעולות הכתיבה נמוך או מפוזר היטב, ה-store buffer לא יתמלא. אבל, אם הליבה מייצרת כתיבות בקצב מהיר יותר מכפי שה-buffer מסוגל לנקז, הוא יתמלא והביצועים ירדו.

ה-store buffer-ים חייבים להיות קטנים יחסית, מה שעלול להוביל למצב שבו מעבד שמבצע רצף כתיבות ממלא את ה-store buffer שלו (למשל, אם כל הכתיבות גורמות ל-cache miss). במצב כזה, המעבד נאלץ להמתין עד להשלמת פעולת ה-invalidate שתפנה מקום ב-store buffer לפני שהוא יוכל להמשיך בפעולתו. תרחיש דומה יכול להתרחש גם לאחר מחסום זיכרון, כאשר כל הכתיבות הבאות חייבות להמתין להשלמת פעולת ה-invalidate, בין אם הן גורמות להחמצת cache ובין אם לא.

ה-cache-ים וה-store buffers נתפסים כיתרון כיוון שהם מאיצים את ביצוע התוכנית. עם זאת, הם גם מציבים אתגרים שאין בליבה ללא cache. אחת הבעיות היא שזמן הביצוע של התוכנית יכול להפוך ללא דטרמיניסטי.

המשמעות היא שזמן הביצוע של קטע קוד מסוים עשוי להשתנות משמעותית, דבר שעלול להוות בעיה במערכות hard real-time שבהן נדרשת התנהגות דטרמיניסטית. כתוצאה מכך, ייתכן שתזדקק לדרך לשלוט בגישה לחלקים שונים של הזיכרון באמצעות ה-cache וה-store buffer.

ליבות מעבד השתמשו במשך זמן רב ב-store buffers כדי להחזיק stores מחויבים (committed) עד ששאר מערכת הזיכרון יכולה לעבד את ה-stores. ה-store נכנס ל-store buffer כשהוא מתחייב, ויוצא מה-buffer כאשר הבלוק שייכתב נמצא ב-cache במצב קוהרנטיות קריאה-כתיבה. store יכולה להיכנס ל-store buffer לפני שה-cache השיג הרשאות קוהרנטיות קריאה-כתיבה עבור הבלוק שייכתב; ה-store buffer מסתיר את ההשהיה של שירות store miss . מכיוון ש-stores הן תדירות, היכולת להימנע מעיכובים ברוב המקרים היא יתרון חשוב. כמו כן, אין סיבה לעכב את הליבה מכיוון שהליבה אינה תלויה בכלום; ה-store מבקשת לעדכן את הזיכרון אך לא את מצב הליבה.

כתיבה לזיכרון (זיכרון ראשי ו-cache-ים) ניתנת לאיחסון פנימי על ידי המעבד ב-store buffer לפני כתיבת הנתונים בפועל למיקום זיכרון. ניתן לשפר את ביצועי המערכת על ידי buffering ל-stores, כפי שמוצג במקרים הבאים:

- כשטרנזקציות זיכרון בעדיפות גבוהה יותר, כמו קריאה, מתחרות על גישה לזיכרון עם כתיבה, כתיבה יכולה להתעכב לטובת קריאה, מה שממזער או מבטל עיכוב ביצוע הוראות עקב קריאה של אופרנד זיכרון.
- כשהזיכרון תפוס, buffering ל-stores, בזמן שהזיכרון תפוס מסיר את הכתיבה מ-pipeline ביצוע הפקודות, מה שמשחרר משאבי ביצוע פקודות.

המעבד מנהל את ה-store buffer כך שיהיה שקוף לתוכנה. גישה לזיכרון בודקת את ה-store buffer, והמעבד משלים את הכתיבה לזיכרון מה-buffer לפי סדר התוכנית. בארכיטקטורת x86 המעבד מרוקן את ה- store buffer על ידי כתיבת התוכן לזיכרון כתוצאה מביצוע כל אחת מהפעולות הבאות: פעולות: io, Serializing, fence, Locked, Interrupts ו-Exceptions.

### Combining & Store buffer

חלק מה-store buffers תומכים במיזוג כתיבה, שנקרא גם שילוב כתיבה. הם יכולים למזג מספר פעולות כתיבה, למשל, זרם של כתיבות לבייטים סמוכים, לאחת, ובכך להפחית את תעבורת הכתיבה לזיכרון החיצוני ולהגביר את הביצועים.

ה-store buffer לא מטפל בהכרח ב-cache lines מלאות. הסיבה לכך היא שהנתון שנשמר ב-store buffer מכיל רק את הערך שצריך לאחסן, ולא את שאר הנתונים שמצויים בשורת ה-cache המתאימה. זהו יתרון משמעותי, כי למעבד שמבצע את פעולת ה-store אין גישה למידע על הנתונים האחרים שנמצאים ב- cache line. עם זאת, כש-cache line המתאימה מגיעה ליעדה, ניתן למזג לתוכה את כל הערכים שמאוחסנים ב-

store buffer ומתאימים ל-cache line הזו. לאחר המיזוג, ניתן להסיר את הערכים המתאימים מ-store buffer, בעוד ששאר המידע ב-cache line נותר ללא שינוי.

ה-store buffers יכולים להתנהג באופן דומה ל-store-combining buffers מכיוון שמספר כתיבות עשוי להיאסף באופן פנימי לפני העברת הנתונים ל-cache-ים או לזיכרון הראשי.

בכמה ארכיטקטורות , במיוחד באלה עם סידור חלש, מבוצע מיזוג של ערכי store buffer כדי ליצור התחייבות אחת ל-L1d. לדוגמה, ניתן למזג זוג של stores בגודל 32 ביטים לכל אחד, ולהפוך אותם לכתיבה אחת בגודל 8 בתים ל-cache L1d.

בביצוע של הוראות שמשנות זיכרון לרוב צריך לבצע קריאה של הזיכרון לפני הכתיבה עצמה בגלל שהתוכן ב-cache נשמר כ-cache line שלמות (לרוב cache line צריכות להיות שלמות אבל יש מימושים שתומכים גם ב-cache line עם תוכן חלקי למשל רכיבים שמשתמשים ב-AMBA CHI) לכן יש לטעון את התוכן של ה-cache line לפני פעולת הכתיבה עצמה.

על מנת להשיג את ה-cache line עם הרשאות מתאימות המעבד עצמו צריך לשלוח הודעות מתאימות בפרוטוקול הקוהרנטיות, למשל במקרה של כתיבה שהתוכן של הזיכרון לא קיים ב-cache אז צריך לשלוח בקשה מתאימה (ב-MESI בקשת read-invalidate וב-AMBA CHI בקשת ReadUnique לדוגמה). במימושים שונים האחריות הזאת נמצאת ברכיבים שונים למשל ב-x86 של intel קיים ה-line fill buffer ויש מימושים שמעברים את האחריות ל-store buffer עצמו (או לפחות רכיב שקוף לאחר ה-store buffer).

### Forward & Store buffer

תוצאה ישירה של השימוש ב-store buffer היא שהכתיבה שביצע המעבד לא נכתבת מיד ל-cache. כתוצאה מכך, בכל פעם שהמעבד מבצע קריאה לשורת cache, הוא בודק קודם כל את ה-store buffer שלו כדי לראות אם השורה המבוקשת כבר נמצאת שם. ייתכן שכתיבה לשורה זו כבר התבצעה על ידי המעבד, אך השורה עדיין לא נכתבה ל-cache כי הכתיבה ממתינה ב-store buffer. חשוב לציין שמעבד יכול לקרוא את הכתיבה הקודמת שלו מה-store buffer שלו, אך מעבדים אחרים לא יכולים לראות את הכתיבות הללו עד שהן נשטפות ל-cache. כלומר, מעבד לא יכול לסרוק את ה-store buffer של מעבדים אחרים.

התנהגות הזאת נקראת store forwarding (ידוע גם כ-bypassing) והיא מתייחסת למצב שבו הליבה יכולה לקרוא ערכים מתוך ה-store buffer שלה גם אם המידע עדיין לא נכתב לזיכרון הראשי ול-cache.

ה-store forwarding מאפשר לרצף פעולות כתיבה לקריאה עם store buffer עשוי להיות נמוך יותר מהסכום של זמני הביצוע עבור פעולות load ו-store בנפרד בגלל שה-store forwarding מאפשר ל-load לקבל את הנתונים ישירות מה-store buffer, במקום לעבור דרך ה-cache.

### Ordering & Store buffer

באופן כללי, מעבדים מודרניים אינם משתמשים במודל זיכרון עם עקביות רציפה (Sequential Consistency), מה שמאפשר סידור מחדש של פעולות store-load. תכונה זו נכונה גם עבור ארכיטקטורות כמו x86 ו--SPARC TSO.

שני stores יכולים להיות מסודרים מחדש אם לליבה יש store buffer שאינו פועל לפי סדר FIFO, כלומר הוא מאפשר ל-stores לצאת בסדר שונה מהסדר שבו הם נכנסו. מצב כזה יכול להתרחש כשה-store הראשון מקבל cache miss בזמן שהשני מקבל hit, או כאשר ה-store השני יכול להתמזג עם store מוקדם יותר. יש לשים לב שהסידורים מחדש הללו אפשריים גם אם הליבה מבצעת את ההוראות לפי סדר התוכנית שלה. סידור מחדש של stores לכתובות זיכרון שונות אינו משפיע על ביצוע thread יחיד. אבל, בליבה מרובת thread-ים, סידור מחדש כזה מאפשר לליבות שונות לראות ערכי זיכרון ישנים לפני שהן רואות את הכתיבה המעודכנת (כלומר, את הערך החדש). חשוב לציין שבעיה זו אינה נפתרת גם אם ה-store buffer מתנקז להיררכיית זיכרון קוהרנטית לחלוטין. קוהרנטיות עשויה להפוך את ה-cache-ים לבלתי נראים, אבל ה-stores כבר מסודרות מחדש.

ליבה בודדת של מעבד עושה שימוש ב-store buffer כדי להסתיר את זמן ההשהיה של store misses. כאשר store פורש מה-pipeline של המעבד, הוא נכנס ל-store buffer ומשם מתנקז למערכת ה-cache או הזיכרון. זה בטוח עבור ליבה בודדת כל עוד loads בודקים את ה-store buffer עבור stores בביצוע (outstanding) עם אותה כתובת. עם זאת, במערכות עם מספר ליבות, כללי הסדר של מודל עקביות רציפה (SC) מונעים שימוש ישיר ב-store buffer. ליבות המתוזמנות באופן דינמי יכולות להסתיר חלק מהשהייה של ה-store, אך לא את כולה. לשם כך הוצעו פתרונות אגרסיביים, כגון ספקולציות מעבר לחלון ההוראות, אשר כוללים פרישה ספקולטיבית של loads ו-stores אשר עברו misses, תוך שמירה על מצב ההוראות שנעשה בו שימוש ספקולטיבית בנפרד.

בעיות היישום של הוראות RMW אטומיות במודל TSO (Total Store Order) דומות לאלו של הוראות אטומיות במודל SC. ההבדל העיקרי הוא ש-TSO מאפשר ל-loads לעבור, כלומר להופיע לפני, stores קודמות שנכתבו ל-store buffer.

### RMW & Store buffer

הוראות אטומיות מיוחדות במידה מסוימת מכיוון שהן עוקפות את ה-store buffer (או לפחות מתנהגות כאילו הן עוקפות אותו).

בביצוע הוראות RMW הכתיבה (כלומר, ה-store) עשויה להיכתב ל-store buffer. כדי להבין את היישום של RMW אטומיים ב-TSO, הוראת RMW נחשבת כהוראת load שאחריה מיד באה הוראת store. חלק ה-load של ה-RMW לא יכול לעבור loads מוקדמים יותר עקב כללי הסידור של TSO. ייתכן שבתחילה נראה שחלק

ה-load של ה-RMW יכול לעבור stores מוקדמות יותר ב-store buffer, אבל זה לא חוקי. אם חלק ה-load של ה-RMW עובר store מוקדמת יותר, אז חלק ה-store של ה-RMW יצטרך לעבור גם את ה-store המוקדמת יותר, שכן ה-RMW הוא זוג אטומי. אבל מכיוון של-stores אסור לעבור זו בזו ב-TSO, חלק ה-load של ה-RMW לא יכול לעבור גם store מוקדמת יותר.

אילוצי הסדר האלה על RMW משפיעים על היישום. מכיוון שלא ניתן לבצע את חלק ה-load של ה-RMW עד שסודרו stores מוקדמים יותר (כלומר, יצאו מ-store buffer), ה-RMW האטומי מנקז את ה-store buffer לפני שהוא יכול לבצע את חלק ה-load של ה-RMW. יתר על כן, כדי להבטיח שניתן לסדר את חלק ה-store מיד לאחר חלק ה-load, חלק ה-load דורש הרשאות קוהרנטיות של קריאה-כתיבה, ולא רק הרשאות קריאה מספיקות ל-loads רגילים. לבסוף, כדי להבטיח אטומיות ל-RMW, ייתכן שבקר ה-cache לא יוותר על הרשאת קוהרנטיות לבלוק בין ה-load ל-store. יישום אופטימלי יותר של RMW אפשרי. לדוגמה, אין צורך לנקז את store buffer כל עוד:

1. לכל ערך שכבר נמצא ב-store buffer יש הרשאת קריאה-כתיבה ב-cache ושומר על הרשאת הקריאה-כתיבה ב-cache עד שה-RMW מתחייב.
2. הליבה מבצעת בדיקה של ספקולציות load. באופן לוגי, כל ה-stores וה-loads הקודמים יתחייבו כיחידה (לפעמים נקראת "chunk") מיד לפני ה-RMW.

לפני ביצוע הוראה אטומית, הליבה מנקזת את ה-store buffer, משיגה את הבלוק עם הרשאות קוהרנטיות קריאה-כתיבה, ולאחר מכן מבצעת את חלק ה-load ואת חלק ה-store. מכיוון שהבלוק נמצא במצב קריאה-כתיבה, חלק ה-store מבוצע ישירות ל-cache, עוקף את ה-store buffer. בין ביצוע חלק ה-load לחלק ה-store, אם קיים חלון כזה, אסור לבקר ה-cache לפנות את הבלוק; אם תגיע בקשת קוהרנטיות נכנסת, יש לדחות אותה עד לביצוע חלק ה-store של ה-RMW.

השיטה של TSO להטמעת RMW פשוטה, אך היא שמרנית מדי ומקריבה ביצועים מסוימים. יש לציין, ניקוז ה-store buffer אינו נדרש מכיוון שמודל relaxed מאפשר גם לחלק ה-load וגם לחלק ה-store של ה-RMW לעבור stores מוקדמות יותר. לפיכך, ניתן להשיג הרשאות קוהרנטיות של קריאה-כתיבה לבלוק ולאחר מכן לבצע את חלק ה-load וחלק ה-store מבלי לוותר על הבלוק בין שתי הפעולות הללו. ישנם יישומים נוספים של RMW אטומיים, אך הם אינם נידונים כאן.

## Fetch

### מבוא

עכשיו אחרי שראינו טכניקה שהמעבד משתמש בה כדי לגרום לכתיבות לעבוד בצורה מהירה יותר נעבור לראות איך המעבד מנסה לשפר את הביצועים בביצוע של קריאה (מידע והוראות).

### שליפה מוקדמת

מטרת השליפה המוקדמת היא להפחית את זמן ההשהיה של גישה לזיכרון. למרות שה-pipeline של ההוראות ויכולת הביצוע מחוץ לסדר (out-of-order) של מעבדים מודרניים יכולים להפחית חלק מהשהיית הזיכרון, זה מוגבל בעיקר לגישות שהן cache hit. כדי לכסות את כל זמן השליפה של גישה לזיכרון הראשי, ה-pipeline היה צריך להיות ארוך מאוד, מה שלא פרקטי. חלק מהמעבדים שאינם תומכים בביצוע מחוץ לסדר מנסים לפצות על כך באמצעות הגדלת מספר הליבות, אך זה יעיל רק אם כל הקוד יכול לפעול במקביל.

מנגנון ה-Prefetching יכול להסתיר את ה-latency בצורה נוספת. המעבד יכול לבצע Prefetching בעצמו כשהוא מופעל על ידי אירועים מסוימים (שליפה מוקדמת בחומרה) או כשהתוכנית מבקשת זאת במפורש (שליפה מוקדמת בתוכנה).

זרימת התוכנית נוטה להיות הרבה יותר צפויה מאשר דפוסי הגישה לנתונים. המעבדים המודרניים מצטיינים בזיהוי דפוסים, מה שמסייע בשיפור היעילות של prefetching.

#### שליפה מוקדמת בחומרה

שליפה מוקדמת בחומרה מופעלת בדרך כלל על ידי רצף של שתי cache miss או יותר בדפוס מסוים. החמצות אלה יכולות להתרחש בשורות cache עוקבות. בעבר, מנגנוני השליפה המוקדמת זיהו רק החמצות cache בשורות סמוכות, אך בחומרה עכשווית, הם מזהים גם דפוסים של דילוג על מספר קבוע של שורות cache ומתאימים את עצמם לכך.

אחת החולשות המרכזיות של השליפה המוקדמת היא שאינה יכולה לחצות גבולות page-ים, בגלל תמיכת המעבדים ב-demand paging. אם השליפה הייתה יכולה לחצות גבולות page-ים, היא הייתה עלולה להפעיל אירוע מערכת הפעלה כדי להעלות page לזיכרון, וזה עשוי לפגוע בביצועים.

לכן, גם אם השליפה המוקדמת מצליחה לנבא את הדפוסים בצורה מדויקת, התוכנית עדיין תחווה החמצות cache בגבולות page-ים, אלא אם תבצע שליפה מוקדמת או קריאה מפורשת מה-page החדש.

**שליפה מוקדמת - ספקולציה**

היכולת של מעבד לבצע הוראות מחוץ לסדר מאפשרת להעביר הוראות קדימה אם אין התנגשויות ביניהן. אמנם לא ניתן להשתמש בערך ישירות, אך המעבד יכול להתחיל לעבד אותו. ה-load הספקולטיבי עושה את עבודתו והשהיית ה-load מוסתרת.

ל-loads ספקולטיביים יש יתרון נוסף בכך שהם טוענים את הערך ישירות לתוך הרגיסטר ולא לשורת ה-cache, שם הוא עלול להימחק שוב, למשל התיזמון (scheduling) של ה-thread מופסק.

בהמשך המאמר נדון בהרחבה בספקולציות שמעבדים עושים.

## תיאור טכניקת ה-Prefetching

שליפה מוקדמת (prefetching) יכולה להיות מסווגת לפי האם היא מחייבת או לא, והאם היא מבוצעת על ידי חומרה או תוכנה. בשליפה מחייבת, הערך שמועבר קשור בזמן שבו נעשתה השליפה. לכן, שליפה מחייבת יכולה להיתקל במגבלות אם מעבד אחר משנה את המיקום בזיכרון בזמן שבין השליפה המוקדמת לקריאה בפועל, מה שעלול לגרום לערך שנשמר להתיישן. לעומת זאת, בשליפה לא מחייבת, הנתונים מועברים למיקום קרוב למעבד, כמו ל-cache, ומוחזקים שם עד לשימושם בפועל, תוך שמירה על קוהרנטיות. בשליפה לא מחייבת מבוססת חומרה, אין השפעה על מודלי עקביות הזיכרון, והיא יכולה לשמש לשיפור הביצועים בלבד.

שליפה אוטומטית מראש יכולה לשפר ביצועים על ידי קיצור זמן ההמתנה של פעולות קריאה שמתעכבות עקב סדר ההוראות בתוכנית. במקרים כאלה, הנתונים נכנסים ל-cache במצב נקי בזמן שהקריאה מתעכבת. מכיוון שהשליפה המוקדמת אינה מחייבת, מובטח שהקריאה תשיב את הערך הנכון ברגע שהפעולה תורשה להסתיים.

במקרה של כתיבה, ניתן להשתמש בשליפה בלעדית לקריאה כדי להשיג בעלות על השורה בזיכרון, מה שמאפשר לפעולת הכתיבה להתרחש במהירות ברגע שהיא מורשית. שליפה כזו מתאימה לפרוטוקולי invalidate, אך אינה מתאימה לפרוטוקולי עדכון שבהם קשה לשמור על עקביות בזמן פעולת הכתיבה. כמו בשליפה המוקדמת לקריאה, השורה שנשלפה מתבטלת או מתעדכנת אם מתבצעת כתיבה למיקום זה על ידי מעבד אחר בזמן שבין השליפה לכתיבה בפועל.

## יישום טכניקת ה-Prefetching

הדרך הנפוצה לאכוף סדר בתוכנית היא לעכב את הנפקת הפעולה עד להשלמת פעולות קודמות. שליפה מוקדמת יכולה להשתלב במסגרת זו כאשר החומרה מנפיקה באופן אוטומטי שליפות מוקדמות (לקריאה או לכתיבה) עבור פעולות שמתעכבות עקב אילוצי סדר.

בקשת שליפה מוקדמת פועלת בדומה לבקשת קריאה או כתיבה רגילה מהזיכרון. בתחילה, נבדק אם השורה קיימת כבר ב-cache. אם כן, הבקשה מתבטלת, ואם לא, הבקשה מועברת לזיכרון. התשובה מהזיכרון ממוקמת ב-cache. אם יש בקשות נוספות לאותה שורה, הן משולבות בבקשת השליפה המוקדמת כדי למנוע כפילות.

אחת הבעיות בשליפה מוקדמת היא שה-cache עשוי להיות עמוס יותר, שכן הנתונים נגישים פעמיים – פעם אחת בשליפה המוקדמת ופעם נוספת בקריאה בפועל. עם זאת, מכיוון ששליפות מוקדמות מתבצעות רק כשפעולות רגילות מתעכבות, לא צפויה בעיה משמעותית בעומס ה-cache.

בנוסף, מבט קדימה בזרם ההוראות יכול לסייע בטכניקות של שליפה מוקדמת הנשלטות על ידי חומרה, במיוחד במעבדים התומכים בתזמון דינמי, חיזוי branch-ים וביצוע ספקולטיבי. מבט קדימה זה מאפשר לשליפה מראש לחפוף עם פעולות זיכרון המתעכבות, ולשפר את ביצועי המערכת.

יש כל מיני פרמטרים שמבדילים בין סוגי שליפה מקודמת:

- פעולות השליפה המוקדמת מתבצעות באופן אוטומטי על ידי החומרה או שהקומפיילר או המתכנת חייבים להפעיל אותן דרך תוכנה.
- האם הנתונים שנשלפים מראש הם עותק משותף או עותק בלעדי, בהתאמה לקריאה רגילה או לשליפה מוקדמת בלעדית לקריאה.
- ישנם שני סוגים של שליפה מוקדמת: ביוזמת הצרכן, שבו המעבד שמבצע את השליפה הוא הצרכן של הנתונים, וביוזמת היצרן, שבו מעבד אחד דוחף את הנתונים לכיוון מעבד אחר שצפוי לצרוך אותם.

## ביטול שליפות מוקדמות מיותרות

כדי למנוע שליפות מוקדמות מיותרות, יש לבדוק את הנתונים בהיררכיית ה-cache של המעבד לפני שהן נשלחות לזיכרון. בנוסף, חשוב לעקוב אחר פעולות שליפה מוקדמת פעולות כדי לאפשר מיזוג של שליפות מוקדמות עוקבות או פעולות רגילות לאותו שורת cache. דרך פשוטה להשיג זאת היא להתייחס לשליפות מוקדמות כאל פעולות קריאה וכתיבה רגילות בהיררכיית cache ללא נעילה. מעקב אחר שליפות מוקדמות קל יותר ממעקב אחר פעולות קריאה או כתיבה, כיוון שאין צורך לעקוב אחרי רגיסטר יעד או נתוני כתיבה. בנוסף, מכיוון שניתן לבטל שליפות מוקדמות, המעבד לא חייב לעצור אם חסרים משאבים למעקב אחריהן.

## סדר פעולות בשליפות מוקדמות לא מחייבות ביוזמת הצרכן

אין צורך לעכב שליפות קריאה מוקדמות בגלל פעולות קודמות, וגם אין צורך לעכב פעולות עתידיות כדי להמתין להשלמת השליפה המוקדמת. באופן דומה, אין מגבלות סדר עבור שליפות מוקדמות בלעדיות לקריאה אם מערכת הזיכרון מתמודדת עם השליפות הללו באמצעות תשובות בלעדיות מושהות.

בחלק מהעיצובים ניתן להשתמש ב-buffer נפרד לשליפות מוקדמות, מה שעשוי לגרום לשינויים בסדר שלהן ביחס לפעולות אחרות. עם זאת, אילוצי סדר הופכים חשובים יותר אם מערכת הזיכרון משתמשת בתשובות בלעדיות נלהבות (eager), כיוון שתשובה נלהבת עשויה לאפשר לכתיבה שלאחר מכן לשורת הזיכרון להיות מטופלת על ידי ה-cache, אף על פי שעדיין יש צורך ב-invalidation acknowledgement. פתרון אפשרי לכך הוא להגדיל את ספירת הכתיבות הפעולות בשליפה מוקדמת בלעדית לקריאה, כך שפעולות עוקבות שמחכות להשלמת כתיבה ימתינו גם להשלמת השליפה המוקדמת.

### התייחסות לשליפות מוקדמות במערכת הזיכרון

במערכת הזיכרון, ניתן להתייחס לשליפות מוקדמות ביוזמת הצרכן כמו לפעולות קריאה או פעולות בלעדיות לקריאה, מה שמבטיח שהנתונים שנשלפו מראש יישמרו קוהרנטיים. למשל, אילוצי סדר חלים גם על שליפות מוקדמות. באופן דומה, יש לטפל בתשובות לשליפות מוקדמות כמו בתשובות רגילות במערכות שמיישמות ביטול תוקף מוקדם או אישורי עדכון. לכן, רוב המערכות אינן מבדילות בין שליפות מוקדמות לפעולות רגילות ברגע שהן יוצאות מתחום המעבד.

### שליפה מוקדמת בתוכנה

יש 2 אספקטים שיש ל-prefetching בתוכנה:

- רמזים שאנחנו רוצים לתת לקומפיילר ולמעבד לביצועים טובים יותר
- התמודדות וסנכרון עם prefetching שמתרחש בחומרה.

### Prefetch hints

תוכניות יכולות לבצע שליפה מוקדמת באופן ישיר למשל ב-c באמצעות ההוראה _mm_prefetch עבור כל מצביע בתוכנית.

ב-x86 הוראות השליפה המוקדמות (PREFETCH) מספקות לתוכנה את האפשרות לרמוז למעבד כי הנתונים המסוימים עשויים להידרש בקרוב. בתגובה, המעבד יכול לטעון מראש את שורת ה-cache שבה נמצאים הנתונים, מתוך ציפייה לשימוש עתידי בהם. ההוראה PREFETCH מרמזת לכך שיש לקרוא את הנתונים, בעוד שההוראה PREFETCHW מציינת כי הנתונים עומדים להיכתב. אם השורה נטענת מראש באמצעות PREFETCHW, המעבד עשוי לסמן אותה כ-modified (בפרוטוקול MESI).

ארכיטקטורת ARM מספקת רמזים למערכת הזיכרון, כולל PRFM, RPRFM, LDNP ו-STNP, אשר יכולים לשמש את התוכנה כדי להעביר לחומרה את הציפיות לגבי השימוש במיקומי זיכרון. המערכת יכולה להגיב על ידי נקיטת פעולות שמיועדות להאיץ את הגישה לזיכרון, אם היא מתרחשת, כגון טעינה מראש של הכתובת שצוינה לתוך cache אחד או יותר, בהתאם ליעד רמת ה-cache הפנימי וזרם הרמז.

ב-ARM הוראות טעינה מראש הן רמזים בלבד, ולכן התוכנה יכולה להתייחס אליהן כאל NOPs מבלי להשפיע על ההתנהגות הפונקציונלית של המכשיר. הוראות אלו אינן יוצרות exception של synchronous Data Abort, אך פעולות מערכת הזיכרון שנעשות כתוצאה מהן עשויות, במקרים חריגים, ליצור asynchronous External abort, המתבצע באמצעות SError interrupt exception.

הוראות ה-prefetch hint מגדירות את סוגי הרמזים לשליפה מראש. ההוראה מאותת למערכת הזיכרון שסביר להניח שגישה לזיכרון מסוג רמז לכתובת שצוינה או ממנה תתרחש בעתיד הקרוב. מערכת הזיכרון עשויה לנקוט פעולה כדי להאיץ את הגישה לזיכרון, כגון טעינה מראש של הכתובת שצוינה לתוך cache אחד או יותר.

עוד דוגמה לפעולה דומה לרמז ל-prefetch היא ההוראה DC ZVA לאיפוס שורת ה-cache ב-ARM בכך שהיא מספק רמז למעבד על סבירות השימוש העתידי בכתובות מסוימות. עם זאת, פעולת האיפוס עשויה להיות מהירה יותר משמעותית, מכיוון שהיא לא דורשת המתנה להשלמת גישה לזיכרון חיצוני. במקום לקרוא נתונים מהזיכרון לתוך ה-cache, השורות ב-cache מאופסות באופן ישיר. פעולה זו מאפשרת למעבד לדעת שהקוד ישנה לחלוטין את תוכן שורת ה-cache, ובכך מבטלת את הצורך בקריאה ראשונית של הנתונים.

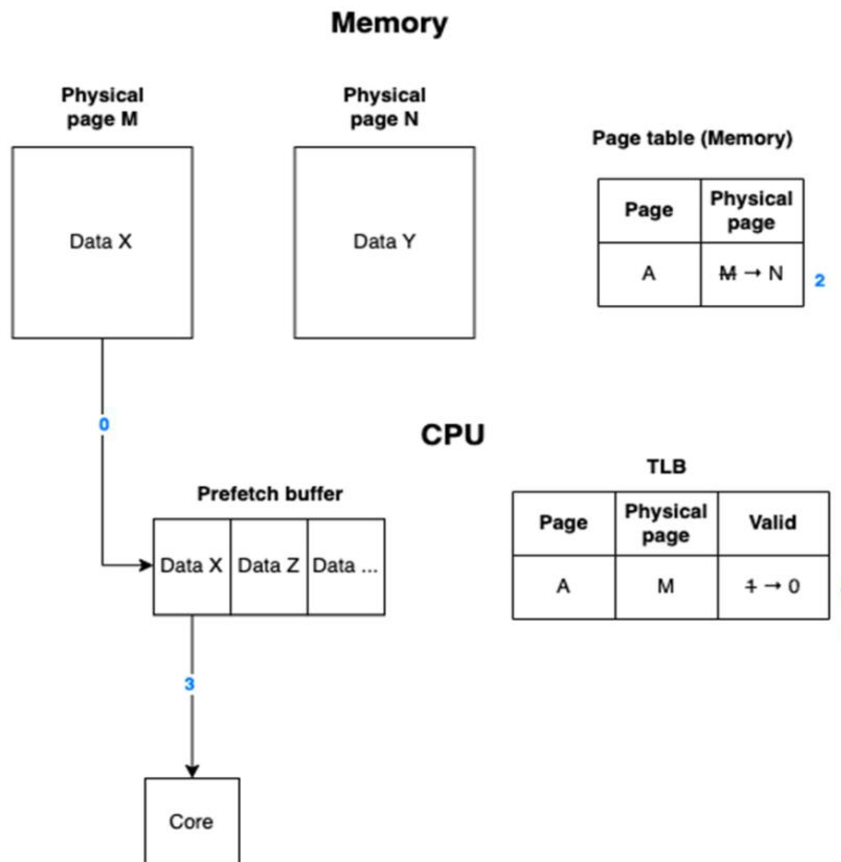
### התמודדות עם Prefetch בחומרה

במקרים מסוימים, שליפת הוראות מראש עלולה ליצור מצבים שבהם טיפול בפרוטוקול קוהרנטיות הזיכרון הופך לבלתי אפשרי. במצבים כאלה, יש צורך שהתוכנה תשתמש בהוראות להסדרת קוהרנטיות ו/או הוראות לניקוי ה-cache כדי להבטיח שקבלת הגישה לנתונים לאחר מכן תהיה קוהרנטית.

דוגמה למצב כזה היא עדכון pages table, ואחריו גישה לדפים הפיזיים שאליהם מתייחסים הטבלאות המעודכנות. להלן רצף האירועים האפשרי כאשר התוכנה משנה את התרגום של דף וירטואלי A מעמוד פיזי M לדף פיזי N:

1. התוכנה מבטלת את ערך ב-TLB. הטבלאות המתרגמות את הדף הווירטואלי A לדף הפיזי M נשמרות כעת רק בזיכרון הראשי ואינן נמצאות ב-cache ה-TLB.
2. התוכנה משנה את ערך טבלת הדפים בזיכרון הראשי עבור הדף הווירטואלי A, כך שהוא מצביע על הדף הפיזי N במקום הדף הפיזי M.
3. התוכנה ניגשת לנתונים בדף הווירטואלי A.

ניתן לראת זאת בתרשים הבא:




בשלב 3, התוכנה מצפה מהמעבד לגשת לנתונים מהדף הפיזי N. עם זאת, המעבד עלול להביא מראש את הנתונים מהדף הפיזי M לפני שנעשה עדכון טבלת הדפים בשלב 2. זאת כיוון שההפניות לזיכרון הפיזי של טבלאות הדפים שונות מההפניות לזיכרון הפיזי של הנתונים, ולכן המעבד אינו מזהה את הצורך לבדוק קוהרנטיות ומניח שהבאת הנתונים מהדף הפיזי M בטוחה. התנהגות דומה יכולה להתרחש גם כאשר הוראות נשלפות מראש מעבר לעדכון טבלת הדפים.

כדי למנוע בעיה זו, התוכנה צריכה להשתמש בהוראת INVLPG או ב-MOV CR3 מיד לאחר עדכון טבלת העמודים ב-x86 וגם ברוב ארכיטקטורות האחרות צריך לטפל במקרה הזה בדרך דומה לרוב. כך מבטיחים ששליפת ההוראות והגישה לנתונים לאחר מכן ישתמשו בתרגום הנכון של דף וירטואלי לדף פיזי. אין צורך לבצע פעולת ניקוי TLB לפני עדכון הטבלה.

עוד מקרה נפוץ שצריך לטפל ב-prefetch בחומרה הוא כשמשתמשים ב-self-modifying-code. הבעיה נגרמת כשכותבים ערכים חדשים להוראות לביצוע והוראות מיושנות עשויות להימצא ב-Instruction Queue וב-Instruction cache, ולכן התוכנית צריכה להסיר אותן לפני שתמשיך.

פעולות כאלו עלולות לדרוש ניקוי או ביטול cache-ים. קוד המעורב בהעתקת זיכרון משתמש בהוראות טעינה ואחסון, הפעולות בצד הנתונים של המעבד. אם cache הנתונים משתמש במדיניות write-back באזור שבו נכתב הקוד, יש לנקות את הנתונים מה-cache לפני הפעלת הקוד החדש. זאת כדי להבטיח שהנתונים שנכתבו יוצאים מה-cache ומועברים לזיכרון הראשי, ואז יהיו זמינים ללוגיקת שליפת ההוראות. בנוסף, אם האזור שאליו נכתב הקוד שימש קודם לכן לתוכנה אחרת, cache ההוראות עלול להכיל קוד ישן שלא הוסר מהזיכרון הראשי. לכן, ייתכן שיהיה צורך גם לנקות את cache ההוראות לפני הסתעפות לקוד החדש שהועתק.

ב-ARM הוראת isb מיועדת למטרה זו. היא מטהרת את כל ההוראות בתור ההוראות וממתינה להשלמה של כל הוראות הביצוע הנוכחיות (הוראות שכבר עברו את שלב התור). לאחר מכן, שליפת ההוראות תתחיל מחדש.

הוראת isb מבטיחה שכל הוראות הביצוע הושלמו, אך מדובר בהשלמה רק של ה-pipeline של ההוראות. כלומר, פעולות הקשורות ל-load/store (כגון גישה לזיכרון, cache, TLBs ויחידות אחרות שקוראות זיכרון) עשויות לא להיות מושלמות.

עוד הוראה נפוצה ב-ARM לפתרון של הבעיות האלו היא ההוראה IC IVAU,Xn שמבטלת את הערכים השמורים ב-cache ההוראות עבור הכתובת שניתנת, בכל הליבות. זה מאלץ כל fetch עתידי של הוראות מאותה כתובת לקבל miss ב-cache ההוראות ולקרוא את הערכים החדשים ישירות מהיררכיית זיכרון הנתונים. בנוסף, ההוראה משפיעה גם על ה-fetch queue, כך שהיא נוגעת במכונות הקשורות לשליפת הוראות.

עוד דגש קטן שצריך לזכור ב-ARM כשהוראת DSB מתבצעת (ההוראה היא סוג של מחסום זיכרון והנושא יוסבר במאמר עתידי), כל הוראה המופיעה בסדר התוכנית לאחר ה-DSB לא יכולה לשנות מצב של המערכת או לבצע פונקציות כלשהן עד שה-DSB יושלם. החריגים הם פעולות פנימיות של המעבד, כמו שליפה מהזיכרון ופענוח.

שליפה מוקדמת מהזיכרון יכולה להתבצע עד למרחק מסוים מהנקודה הנוכחית של הביצוע. שליפה מוקדמת כזו עשויה לכלול מספר קבוע או משתנה של הוראות, ויכולה לעקוב אחרי כל נתיב ביצוע עתידי אפשרי. לגבי כל סוגי הזיכרון:

- ה-PE (Processing Element) יכול לשלוף הוראות מהזיכרון בכל עת מאז אירוע סינכרון ההקשר האחרון באותו PE.
- הוראות שנשלפו בדרך זו עשויות להתבצע מספר פעמים אם נדרש על פי ביצוע התוכנית, מבלי להחזיר אותן מהזיכרון. בהעדר אירוע סינכרון הקשר, אין הגבלה על מספר הפעמים שבהן הוראה עשויה להתבצע מבלי לבצע fetch מהזיכרון.

## עקביות במודל הזיכרון ב-ARM

ארכיטקטורת ARM מתירה שליפת הוראות מראש, כולל אפשרות של שליפה ספקולטיבית:

"כמה רחוק מנקודת הביצוע הנוכחית נשלפים הוראות יישום מוגדר. שליפה מוקדמת כזו יכולה להיות מספר קבוע או משתנה דינמית של הוראות, ויכול לעקוב אחר כל נתיב ביצוע עתידי אפשרי או אחר. עבור כל סוגי הזיכרון, ייתכן שה-PE היה מביא את ההוראות מהזיכרון בכל עת מאז אירוע סנכרון ה-context האחרון באותו PE."

עם זאת, שליפת פקודות אחת עשויה להיות לא עקבית עם שליפות קודמות לפי סדר התוכנית, מה שעלול להוביל לנתונים ולזרמי הוראות שאינם מסונכרנים זה עם זה. ארכיטקטורת ARM אינה מבטיחה עקביות בין שליפות מאותו מיקום זיכרון: שליפת פקודה אינה מבטיחה ששליפה מאוחרת יותר מאותו מיקום לא תחשוף הוראה ישנה יותר.

## שליפה Prefetching לא מחייבת

שליפה מוקדמת לא מחייבת לבלוק B היא בקשה למערכת הזיכרון הקוהרנטי לשנות את מצב הקוהרנטיות של B באחד או יותר מה-cache-ים. מימושים של שליפות מוקדמות לא מחייבות יכולים להתבצע מבלי להשפיע על מודל עקביות הזיכרון, מה שהופך אותן לשימושיות עבור שליפת cache פנימית מראש (למשל, דרך stream buffers) וגם עבור ליבות עם ביצועים אגרסיביים יותר.

כאשר מבצעים פעולות ספקולטיביות של load או store שמבוטלות בהמשך (נמעכות), הן עשויות להיראות כמו שליפות מוקדמות לא מחייבות, מה שמאפשר לספקולציה זו להתקיים מבלי להשפיע על מודל עקביות זיכרון חזק כמו SC. אם ה-load נמעך, הליבה מוחקת את עדכון הרגיסטר ומסירה כל השפעה פונקציונלית של הפעולה, כאילו היא לא התבצעה כלל.

במקרה של שליפות מוקדמות לא מחייבות, אין צורך לבטל אותן כאשר מדובר ב-cache, מכיוון ששליפה מראש של הבלוק עשויה לשפר את הביצועים אם ה-load תבוצע מחדש. עבור store, הליבה עשויה להוציא בקשה לקבלת הבלוק במצב modified לא מחייבת מוקדמת, אך היא לא תשלים את הפעולה עד שה-store תהיה מובטחת להתחייב.

במערכת בה הליבה מתוזמנת באופן דינמי, ייתכן ש-loads ו-stores יתבצעו בסדר שאינו עוקב אחר סדר התוכנית (program order). מודל ה-SC אינו מושפע מסדר השליפות המוקדמות הלא מחייבות, כיוון ש-SC דורש רק שה-loads וה-stores של ליבה מסוימת ייגשו ל-cache שלה לפי סדר התוכנית. קוהרנטיות הזיכרון מחייבת שבלוקי ה-cache יהיו במצבים המתאימים כדי לקבל פעולות load ו-store. חשוב לציין, ש-SC (או כל מודל עקביות זיכרון אחר):

- מכתיב את הסדר שבו פעולות load ו-store מיושמות על זיכרון קוהרנטי.
- אינו מכתיב את סדר פעילות הקוהרנטיות.

## ספקולציות

אחרי שראינו שהמעבדים עושים prefetch לשיפור ביצועים נבדוק טכניקה נוספת דומה ברעיון שלה ל-prefetching, נדבר על ספקולציות שמעבדים עושים.

### מבוא

ביצוע ספקולטיבי הוא שיטת אופטימיזציה במערכות מחשב, שבה מבוצעות משימות מראש, לפני שנודע בוודאות אם הן נדרשות. המטרה היא לחסוך זמן ולמנוע עיכובים בביצוע העבודה לאחר שמתברר שהיא נדרשת. אם בסופו של דבר מתברר שהמשימה לא הייתה הכרחית, המערכת מבטלת את רוב השינויים שבוצעו ומתעלמת מתוצאות הביצוע.

מטרת הביצוע הספקולטיבי היא להגדיל את ניצול המשאבים במערכת ולהפחית עיכובים. השיטה משמשת בתחומים רבים, כמו חיזוי נתיבי branch במעבדים עם pipeline, חיזוי ערכי נתונים לניצול מקומיות ערכית, ושליפת נתונים מראש מהזיכרון.

מעבדים מודרניים, בעלי pipeline-ים, משתמשים בביצוע ספקולטיבי כדי להפחית את העיכוב שנגרם מהוראות branch מותנות. הם עושים זאת באמצעות חיזוי נתיב ההוראות הנכון על בסיס ההיסטוריה של ביצועי branch קודמים. הוראות מתוזמנות מראש, לפני שמובהר אם הן נחוצות, כדי לשפר את הביצועים ולהשיג ניצול מרבי של המשאבים.

כאשר כתיבה (store) מחויבת (committed) ל-cache, היא הופכת לגלויה ל-thread-ים הפועלים על ליבות אחרות, בהתאם לפרוטוקול הקוהרנטיות של ה-cache. בשלב זה, קשה או בלתי אפשרי להחזיר את הכתיבה לאחור, שכן ייתכן שליבה אחרת כבר קיבלה עותק מהערך שנכתב. לכן, כדי למנוע פגיעות ולוודא שה-store לא תגרום לבעיות, יש להמתין ולוודא שאין ספקולציות שגויות קודמות (כמו חיזוי branch שגוי) לפני שניתן לבצע את ההוראה של ה-store ל-L1d cache.

ללא סידור מחדש של store-load, כל load חייבת להמתין עד שכל ה-stores הקודמים לה יפרשו (הסתיימו לחלוטין ויתקבלו על ה-cache לפני שהיא יכולה לקרוא ערך מה-cache לשימוש בהוראות מאוחרות יותר. רגע ההעתקה של ערך מה-cache לרגיסטר הוא הזמן הקריטי שבו הוא מתרחש עם סדר הקוהרנטיות של הקריאות והכתיבות למיקום זיכרון מסוים.

במעבדים מודרניים, ה-stores עוברות מה-store buffer ל-L1d לאחר שההוראת ה-store המתאימה פרשה מה-ROB. כמות מסוימת של stores בוגרות יכולה להיות מאוחסנת ב-store buffer, מה שמונע את הבזבוז בזמן כתוצאה מ-cache misses. ניתן גם לשים לב שההתחייבות ל-L1d יכולה להתרחש כחלק מהפרישה, אם כי זה עלול לוותר על שיפור הביצועים הקיים. ביצוע store יתבצע על ידי כתיבת הכתובת והנתונים לתוך ה-

store buffer, מה שמאפשר ביצוע ספקולטיבי עד שהנתונים יהיו מוכנים, כאשר ה-store buffer שומר על הספקולציות פרטיות עד שיתחייבו.

## גרסאות של ביצוע ספקולטיבי

### ביצוע להוט (Eager Execution)

ביצוע להוט הוא גרסה של ביצוע ספקולטיבי, שבה המעבד מבצע את כל נתיבי ה-branch המותנה. לאחר מכן, התוצאה הנכונה מחויבת (committed) רק אם ההחלטה הייתה נכונה. בתיאוריה, אם היו משאבים בלתי מוגבלים, ביצוע להוט היה מספק את אותם ביצועים כמו חיזוי branch מושלם. אך בפועל, מכיוון שהמשאבים מוגבלים, יש להפעיל ביצוע להוט בזהירות, שכן כל רמת branch נוספת מגדילה את צריכת המשאבים באופן אקספוננציאלי.

### ביצוע חזוי (Predictive Execution)

ביצוע חזוי הוא סוג נוסף של ביצוע ספקולטיבי שבו המעבד צופה תוצאה מסוימת וממשיך לבצע את ההוראות בהתאם לנתיב החזוי, עד שהתוצאה הסופית מאומתת. אם התחזית הייתה נכונה, הביצוע ממשיך ומחויב; אם לא, יש לבטל את התוצאות ולהתחיל מחדש. טכניקות חיזוי נפוצות כוללות חיזוי branch-ים וחיזוי תלות בזיכרון, ובגרסה כללית יותר, חיזוי ערכים.

### Runahead Execution

טכניקת ה-runahead מאפשרת למעבד לעבד מראש הוראות במהלך מחזורי זמן שבהם מתרחשות cache misses. באמצעות גישה זו, המעבד יוצר קריאות מקדימות של הוראות ונתונים על ידי ביצוע הוראות שעלולות להוביל להחמצות cache, לפני שהן מתרחשות בפועל. טכניקה זו מסתירה את זמני ההשהיה שנגרמים עקב החמצות בזיכרון. ב-runahead, המעבד משתמש במשאבים שאינם פעילים כדי לחשב כתובות ולבצע פעולות שאינן תלויות בהחמצות cache. לאחר שהמעבד פותר את ההחמצה הראשונית, הוא מוחק את כל התוצאות שהושגו במהלך ה-runahead וחוזר לביצוע רגיל. מקרה השימוש העיקרי לטכניקה זו הוא התמודדות עם בעיית ה-memory wall, והיא עשויה לשמש גם כדי לחשב מראש תוצאות branch-ים להשגת חיזוי מדויק יותר.

## קריאות ספקולטיביות

הרעיון המרכזי של קריאות ספקולטיביות הוא להקדים ביצוע של פעולות קריאה לזיכרון כך שהן יבוצעו במקביל לפעולות קודמות באותו מעבד. טכניקה זו מאפשרת לקריאות להסתיים מוקדם יותר, ובכך ייתכן שהן יסתיימו שלא בסדר התוכנית המקורי. מערכת פשוטה עוקבת אחר כך כדי לזהות אם ביצוע מקבילי זה עלול לגרום להפרות בעקביות הזיכרון. אם מתגלה הפרה כזו, הקריאה הספקולטיבית וכל החישובים התלויים בה מבוטלים מבוצעים מחדש. טכניקה זו מותאמת במיוחד למעבדים דינמיים בעלי יכולת חיזוי branch-ים, מכיוון שמנגנון ההחזרה לאחור במקרה של טעות דומה למנגנון ההחזרה שמופעל כשחיזוי branch מתגלה כשגוי.

הדרישות לשמירה על עקביות רציפה עלולות לשלול שימוש בטכניקות שמאיצות את פעולת המעבדים. במקרים מסוימים, המחיר של שמירה על עקביות כזו עשוי להאט את המעבד באופן שלא מצדיק את המאמץ.

### תיאור טכניקת הביצוע הספקולטיבית

הרעיון מאחורי הביצוע הספקולטיבי הוא די פשוט: נניח שיש לנו שתי פעולות בתוכנית, u ו-v, כאשר u היא פעולה שדורשת זמן עיבוד ארוך ו-v היא פעולת קריאה. ונדרש שפעולת הקריאה v תתבצע רק לאחר השלמת הפעולה u. בטכניקת הביצוע הספקולטיבי, המעבד מניח או משיג ערך עבור פעולת הקריאה v עוד לפני ש-u מסתיימת וממשיך בביצוע יתר הפעולות. אם בסיום הפעולה u מתגלה שהערך של פעולת הקריאה v תואם לערך הנוכחי, הספקולציה מוצלחת והחישוב נחשב לנכון. המשמעות היא שגם אם v הייתה מתבצעת לאחר u, התוצאה הייתה זהה. לעומת זאת, אם הערך הנוכחי של v שונה מהערך הספקולטיבי, יש לבטל את החישובים שהיו תלויים בערך השגוי ולבצע אותם מחדש.

כדי ליישם סכימה כזו, נדרש מנגנון שמספק ערך ספקולטיבי עבור פעולת הקריאה, מנגנון שמזהה האם הספקולציה הצליחה, ומנגנון תיקון שמאפשר לבטל ולבצע מחדש חישובים במקרה של כישלון בספקולציה.

הגישה ההגיונית ביותר היא לבצע את פעולת הקריאה ולהשתמש בערך שהוחזר. במצב שבו פעולת הקריאה מתבצעת ב-cache, הערך יתקבל במהירות. במצב של פספוס ב-cache, למרות שהערך לא יתקבל מיד, הקריאה תבוצע במקביל לפעולות אחרות, בדומה למנגנון של prefetch. באופן כללי, ספקולציה על ערך מסוים אינה מועילה אלא אם ידוע שהערך מוגבל לטווח קטן, כמו למשל במקרים של פעולות נעילה.

באשר למנגנון הזיהוי, דרך בסיסית לזהות אם ערך ספקולטיבי שגוי היא לבצע את פעולת הקריאה מחדש בתנאים לא ספקולטיביים, ולבדוק את הערך המתקבל מול הערך הספקולטיבי. עם זאת, אם מנגנון הספקולציה מאחסן את הערך ב-cache, ניתן לעקוב אחרי טרנזקציות קוהרנטיות באותו מיקום כדי לאמת אם הערך הספקולטיבי נכון. כך, טכניקת הביצוע הספקולטיבית מאפשרת גישה אחת ל-cache עבור כל פעולת קריאה, בניגוד לשתי הגישות הנדרשות בטכניקת prefetch.

ניקח שוב פעם לדוגמה את הפעולות u ו-v. ויש דרישה שהשלמת הקריאה v תעוכב עד להשלמת u. הטכניקה הספקולטיבית מאפשרת להנפיק את הקריאה v מוקדם, והמעבד ממשיך עם הערך הספקולטיבי. אם מגיעה בקשה לעדכון או ביטול עבור המיקום של v לפני ש-u הסתיימה, זה מצביע על כך שהערך הספקולטיבי עשוי להיות שגוי. מנגד, היעדר הודעות invalidate או עדכון מרמז שהערך הספקולטיבי כנראה נכון. יש להתייחס גם להחלפות cache באופן נכון, משום שאם מיקום v יוחלף מה-cache לפני ש-u תסתיים, ייתכן שלא יתקבלו עוד בקשות invalidate ועדכון, והערך הספקולטיבי ייחשב לעתיק. במצב כזה, נדרש לחזור על הקריאה לאחר השלמת u כדי לאמת את הערך.

אם הערך הספקולטיבי מתברר כשגוי, מנגנון התיקון מבטל את כל החישובים שהתבססו עליו ומבצע אותם מחדש. מנגנון זה דומה למנגנון המשמש במעבדים עם branch prediction, בהם במצב של חיזוי שגוי מבטלים את ההוראות שבוצעו לאחר ה-branch ומבצעים את ההוראות הנכונות. באופן דומה, אם הערך הספקולטיבי שגוי, מבטלים את פעולת הקריאה ואת החישובים שבאו בעקבותיה ומבצעים אותם מחדש כדי להשיג תוצאה נכונה.

הטכניקה הספקולטיבית מתמודדת עם החסרונות של טכניקת השליפה המוקדמת בכך שהיא מאפשרת גישה לערכים ספקולטיבים שהם מחוץ לסדר. באמצעות ביצוע ספקולטיבי, ניתן להנפיק פעולות קריאה ברגע שהכתובת עבורן ידועה, ללא קשר למודל העקביות שבו המערכת תומכת.

**שילוב קריאה ספקולטיבית עם prefetch חומרה לכתיבה**

שילוב טכניקות הקריאה הספקולטיבית עם prefetch חומרה לכתיבה מציע הזדמנות לשיפור ביצועים משמעותי על ידי חפיפת פעולות זיכרון, ללא תלות במודל הזיכרון הנתמך. התוצאה היא שיפור בביצועים עבור כל סוגי מודלי העקביות, כולל המודלים היותר חלשים. שיפור זה חשוב במיוחד מכיוון שמודלים רגועים נוטים להיות בעלי מודל תכנות מסובך יותר.

העיקרון המרכזי מאחורי טכניקות אלו הוא לשרת את הפעולות בהקדם האפשרי, מבלי להתחשב במגבלות של מודל הזיכרון. עם זאת, מכיוון שצריך לשמור על תקינות, שירות מוקדם (או ספקולטיבי) לא תמיד מועיל.

טכניקות קריאה ספקולטיבית ותחזיות prefetch חומרה לכתיבה מספקות יתרונות ביצועים גבוהים יותר כאשר ישנו תדירות גבוהה של פעולות שדורשות זמן עיכוב ארוך (כמו החמצות cache). כאשר פעולות עם זמן עיכוב ארוך אינן תכופות (כלומר, מספר רב של הוראות מפרידות בין כל החמצת cache), מגבלות על משאבי buffer מונעות לרוב מהמעבד לחפוף מספר פעולות עיכוב ארוך בשל יכולת ההסתכלות המוגבלת. עם זאת, יישומים כאלה בדרך כלל פועלים ביעילות והפחתת זמן העיכוב לתקשורת פחות חשובה.

המשמעות המרכזית של הטכניקות המוצעות היא שהביצועים של מודלי עקביות שונים מתקרבים זה לזה לאחר יישום טכניקות אלו. לכן, הבחירה במודל עקביות לתמיכה בחומרה נעשית פחות קריטית אם נבצע את הטכניקות הללו. העלות היא כמובן המורכבות הנוספת של החומרה הנדרשת ליישום. בעוד שטכניקת השליפה המוקדמת היא פשוטה יחסית לשילוב במעבדים מרובי cache קוהרנטיים, הטכניקה של ביצוע ספקולטיבי דורשת תמיכה בחומרה מתקדמת יותר.

**ביצוע ספקולטיבי של פעולות loads**

כאשר מעבד מבצע פעולת load באופן ספקולטיבי ממיקום שנמצא ב-cache, הדבר עלול להוביל לטעינת שורת cache חדשה. פעולה זו עשויה גם לגרום לפינוי של שורת cache קיימת, מה שיכול להשפיע על הביצועים הכלליים של המערכת.

**Out Of Thin Air (OOTA)**

המונח OOTA מתאר מצב תיאורטי שבו קבוצת thread-ים מבצעת פעולות load מ-stores, כאשר כל store תלויה בערך המוחזר מה-load המתאימה. התופעה יוצרת מחזור OOTA שבו עשוי להיווצר ערך לא תקני או "יש מאין", אך זו לא התוצאה היחידה האפשרית.

בכדי למנוע את תופעת ה-OOTA, אשר עלולה להוביל לתוצאות שגויות או בלתי צפויות, יש להקפיד על מספר דברים:

1. **איסור על טעינת ערכים מכתיבות ספקולטיביות**: אסור להניח ערכים מכתיבות ספקולטיביות לקריאות שאינן ספקולטיביות. חומרה המבצעת אופטימיזציות ספקולטיביות עשויה ליצור מחזורי OOTA במהלך הביצוע הספקולטיבי, אך נדרשת לכך מנגנונים שמונעים ממחזורי OOTA ספקולטיביים להיכנס למצב מחויב.
2. **מניעת חשיפת ערכים ספקולטיביים**: ערכים המאוחסנים באופן ספקולטיבי אינם מורשים להיות נגישים לקריאות לא ספקולטיביות מ-thread-ים אחרים. כאשר מדובר ב-thread-ים מרובים שחולקים את אותה הליבה, ספקולציות הכוללות מספר thread-ים חייבות להיות מנוהלות באופן שיבטל את כל ההשפעות של ספקולציות אלה על כל ה-thread-ים המעורבים.
3. **הקפדה על סידור נכון של stores**: המעבדים בעולם האמיתי חייבים לסדר את ה-stores לאחר ה-loads המתאימים להם כדי למנוע תוצאות OOTA. כלומר, הכתיבות נדרשות להתבצע לאחר שהקריאות המתאימות הושלמו ומסופקות.
4. **אישור ספקולציות**: לפני ש-store מחויבת, יש לוודא שכל הספקולציות שה-store תלויה בהן אושרו. זאת אומרת, ה-store אינה נחשפת ל-thread-ים אחרים עד שכל הספקולציות שקשורות אליה אושרו.

**ליבות ספקולטיביות**

ליבה שמבצעת הוראות לפי סדר התוכנית עשויה גם לבצע חיזוי branch-ים, דבר המאפשר להתחיל בביצוע הוראות עוקבות, כולל פעולות loads ו-stores, גם כאשר ייתכן שיהיה צורך לבטל את הפעולות הללו במקרה של חיזוי branch שגוי.

כאשר קריאות או כתיבות אלו מבוטלות, ניתן להתייחס אליהן כאל שליפות מוקדמות (prefetches) לא מחייבות, מכיוון שהן לא משפיעות על זיכרון עקבי. לדוגמה, אם קריאה מבוטלת, הליבה פשוט מסלקת את עדכון הרגיסטר, ומבטלת את כל ההשפעות התפקודיות של אותה קריאה, כאילו לא התרחשה בכלל.

ב-cache אין צורך לבטל את השליפות המוקדמות הלא מחייבות, מכיוון שהשליפה המוקדמת של הבלוק עשויה לשפר את הביצועים אם הקריאה תתבצע מחדש. כאשר מדובר בכתיבה, הליבה עשויה ליזום פעולת ReadUnique (בפרוטוקול AMBA CHI) לא מחייבת (שתתחיל את הבאת הבלוק), אך היא לא תתחייב לכתוב את הערך ל-cache עד שתהיה וודאות כי הפעולה אכן תתחייב.

### ליבות מתוזמנות דינמית

ליבות מודרניות רבות משתמשות בתזמון דינמי כדי לבצע הוראות שלא בסדר התוכנית המקורי, במטרה לשפר את הביצועים, בניגוד לליבות מתוזמנות סטטית, שחייבות לבצע הוראות בסדר תכנותי קפדני. מעבדים בעלי ליבה אחת המשתמשים בתזמון דינמי או בביצוע מחוץ לסדר צריכים להבטיח רק את תלות הנתונים האמיתית בתוך התוכנית. אבל, במעבדים מרובי ליבות, התזמון הדינמי מציב אתגר חדש - ספקולציות על עקביות זיכרון.

לדוגמה, כאשר ליבה מנסה לסדר מחדש באופן דינמי את הביצוע של שתי פעולות loads מהזיכרון, L1 ו-L2, ייתכן שהיא תבצע את L2 לפני L1 באופן ספקולטיבי, מתוך הנחה שהסידור מחדש לא ייחשף לליבות אחרות. סידור כזה עשוי לפגוע בעקביות זיכרון.

## ספקולציות לאחר פרישה

שימוש ב-store buffer בטוח כאשר מדובר בליבה אחת, בתנאי שהפעולות הקריאה יבדקו ב-store buffer האם ישנן פעולות כתיבה בהמתנה לאותה כתובת. אבל, במערכות מרובות ליבות, כללי הסדר של עקביות זיכרון מגבילים שימוש פשוט ב-store buffer.

ליבות עם תזמון דינמי יכולות להסתיר חלק מהשהיית הפספוסים של פעולות ה-store, אך לא את כולן. כדי להסתיר יותר מהשהיית הפספוסים, הוצעו טכניקות אגרסיביות יותר שמבצעות ספקולציות מעבר לחלון ההוראות. הרעיון המרכזי הוא לפרוש באופן ספקולטיבי פעולות קריאה וכתיבה שמחכות לפספוסים, תוך שמירה על המצב הספקולטיבי של ההוראות הללו בנפרד.

אחת השיטות הנפוצות היא עיכוב בקשות קוהרנטיות. בשיטה זו, כאשר פעולת זיכרון צעירה שפרשה ויש פעולת זיכרון ישנה יותר שעדיין ממתינה, בקשות קוהרנטיות למיקום הזיכרון של הפעולה הצעירה מתעכבות עד שהפעולה הישנה יותר תפרוש. שיטה זו עלולה להוביל ל-deadlock, ולכן יש צורך במנגנונים קפדניים להימנע ממנו.

### תמיכה פשוטה יותר בספקולציות ליבה

במערכות שבהן יש מודלים חזקים לעקביות זיכרון, ליבה עשויה לבצע ספקולציות על פעולות קריאה לפני סדר התוכנית המקורי, גם אם הן עדיין אינן מוכנות להתחייב.

**Prefetch - Speculation**

היכולת של מעבד לבצע הוראות באופן שאינו לפי סדרן המקורי מאפשרת לו להעביר הוראות שהתנגשויות ביניהן אינן מהוות בעיה. עם זאת, אף על פי שהמעבד אינו יכול להשתמש בערך שנמצא עדיין, הוא יכול להתחיל את העבודה על ההוראות בהתאם.

היתרון המרכזי של loads ספקולטיביות הוא בכך שהן מספקות את הערכים הנדרשים מוקדם יותר, ובכך מסייעות להסתיר את השהיות הנובעות מהמתנה לטעינה של נתונים מהזיכרון.

כאשר מדובר ב-load ספקולטיבי, המנגנון מבצע את פעולת הקריאה כך שהשהייה הקשורה לטעינה מוסתרת מהמעבד. טעינות ספקולטיביות מספקות יתרון משמעותי בכך שהן טוענות את הערכים ישירות אל הרגיסטר, ולא לתוך שורת ה-cache. היתרון הזה חשוב במיוחד כאשר ה-thread לא מתוזמן באופן תדיר או עלול להיתקל בעיכובים אחרים, שכן הערך לא יתפוס מקום בשורת ה-cache ויתפנה בעתיד.

**Branch Predictor**

ה-Branch Predictor הוא רכיב המיועד לנחש את הכיוון שבו יתבצע ה-branch לפני שהמיקום הסופי שלו יוכר באופן חד משמעי. תהליך זה נועד לשפר את הביצועים של המעבד על ידי צמצום הזמן שבו המעבד מחכה להחלטות לגבי קפיצות מותנות.

במידה ואין שימוש ב-branch prediction, המעבד נאלץ להמתין עד שהוראת הקפיצה המותנית תושלם בשלב הביצוע לפני שתוכל להתחיל לעבד את ההוראה הבאה ב-pipeline. ה-branch prediction פועל במטרה למנוע בזבוז זמן זה על ידי ניחוש אם הקפיצה המותנית צפויה להתבצע או לא. על פי התחזיות, ההוראות הנכונות מבוצעות באופן ספקולטיבי, כלומר, לפני שהכיוון הסופי של הקפיצה נקבע.

במידה ומתגלה מאוחר יותר שהתחזיות היו שגויות, ההוראות שבוצעו או התבצעו חלקית על פי התחזיות יבוטלו, וה-pipeline יתאפס ויתחיל מחדש עם ה-branch הנכון. תהליך זה כרוך בעיכוב, אך מאפשר לנצל בצורה יעילה יותר את משאבי המעבד ולשפר את ביצועיו הכוללים.

כאשר המעבד נתקל בהוראה מותנית, כמו הוראת branch, הוא עשוי להתחיל לבצע הוראות באופן ספקולטיבי, עוד לפני שידוע אם ההוראה הספציפית הזו באמת צריכה להתבצע. ביצוע מוקדם זה יכול לשפר את הביצועים, מכיוון שהתוצאה תהיה זמינה מיד ברגע שהתנאים יאשרו שהספקולציה הייתה נכונה.

**טעינת נתונים וזיכרון במעבדים**

כאשר מעבד מבצע הוראת טעינה (L), הוא קודם כל בודק אם יש הוראות stores שנעשו קודם לכן, לאותו מיקום זיכרון, לפני הוראת ה-load בסדר התוכנית. אם יש הוראת store מתאימה, המעבד משתמש בערך של

ה-store האחרון שנעשה (העונה על התנאים) כערך שהטעינה (L) מקבלת. אנו אומרים שהערך של ה-store הועבר (forwarded) ל-L.

אם אין הוראת store מתאימה לפני L, המעבד פונה לתת-מערכת הזיכרון כדי לטעון את הערך מהזיכרון. במצב זה, אנו אומרים ש-L מסופקת (satisfied) מהזיכרון. תת-מערכת הזיכרון מחזירה את הערך של ה-store האחרון למיקום זה, אם הוא כבר התפשט למעבד הנוכחי.

בפועל, התמונה עשויה להיות מורכבת יותר. למעבדים יש cache-ים מקומיים, ולכן הפצת ה-store למעבד משמעותה שהיא מתפשטת ל-cache המקומי של המעבד. cache מקומי יכול לקחת זמן לעבד את ה-stores שהוא מקבל, ולכן לא ניתן להשתמש ב-store כדי לספק את אחד מה-loads של המעבד עד שהוא עובר עיבוד. רוב הארכיטקטורות מאחסנות ב-cache-ים המקומיים בסדר FIFO, מה שגורם לעיכובים בעיבוד. עם זאת, בארכיטקטורות מסוימות, כמו [Alpha](), ה-cache-ים המקומיים מעוצבים בצורה שאינה FIFO, מה שיכול להוביל להתנהגות שונה.

חשוב לציין שהוראות טעינה עשויות להתבצע באופן ספקולטיבי ולעיתים ידרשו להופיע מחדש בנסיבות מסוימות. חלק ממודלי הזיכרון (כמו ה-LKMM) מתעלמים מהביצועים המוקדמים הללו, ומניח שפעולת הטעינה מתבצעת לפי הזמן האחרון שהיא הועברה או מסופקת.

**ספקולציות תלות זיכרון**

מעבדים מוקדמים הפועלים בשיטת out-of-order ביצעו פעולות זיכרון בצורה לא ספקולטיבית בלבד. פעולות קריאה לא בוצעו עד שכל הכתובות של פעולות כתיבה קודמות היו ידועות, כך שניתן היה לקבוע בוודאות אם הקריאה תלויה בכתיבה מוקדמת ויש להמתין זמן נוסף. טכניקות של ספקולציה על תלות זיכרון מאפשרות למעבד לנחש אם קריאה תלויה בכתיבה קודמת, עוד לפני שהכתובות של הכתיבה ידועות, ובכך לסכן את עצמם בסידור שגוי שבו קריאה תתבצע לפני כתיבה שהיא תלויה בה.

מנבא תלות פשוט יכול לחזות שלרוב הקריאות אין תלות בכתיבות קודמות, מכיוון שבקוד בפועל רוב הקריאות אינן תלויות בכתיבות קודמות. מנבא מתקדם יותר עשוי להשתמש בהתנהגות העבר של הקריאה כדי לשפר את החיזוי.

במעבדים הממשיכים לבדוק את ה-store queue כדי לאתר תלות, ספקולציה לא תמיד נדרשת. כאשר מתבצעת קריאה, ייתכן שכל כתובות הכתיבה הקודמות כבר ידועות, כך שהמעבד יכול לדעת בוודאות אם יש תלות ללא צורך בספקולציה. ההתנהגות של קריאה שחזויה כתלויה אינה זהה בהכרח לקריאה שידוע כתלויה.

**כיצד סידור מחדש של Store-Load מסייע למעבדים**

מעבדי x86 מודרניים מבצעים loads מוקדמות באופן ספקולטיבי לפני loads אחרות, גם כאשר יש cache miss. תהליך זה יכול להוביל לתוצאות שגויות אם המעבד מזהה שהעותק שלו של שורת ה-cache כבר לא

תקף. כאשר מצב זה מתגלה, המעבד יזרוק את תוכן ה-reorder buffer כדי לחזור למצב עקבי ויתחיל לבצע מחדש את ההוראות מהשלב שבו הם נתקעו. בדרך כלל, תופעה זו מתרחשת כאשר ליבה אחרת משנה את שורת ה-cache, אך היא יכולה לקרות גם אם העיכוב התרחש בגלל טעינה ששגתה וטעינה מחדש לא נדרשה. חשוב לציין שלמעבדי x86 יש יכולת לבצע סידור מחדש של loads לפני stores בצורה חופשית.

אם לא הייתה אפשרות לסדר מחדש של Store-Load, המעבד היה יכול לבצע טעינות באופן ספקולטיבי, אך עדיין היה נדרש לבצע התחייבות לכתיבות מוקדמות יותר לפי התהליך הרגיל. ספקולציות טעינה יכולות להמשיך לעקוב אחרי כתיבות המאוחסנות ב-store buffer עד שהן פורשות.

במצב כזה, ההגבלה תהיה שטעינות לא יוכלו להתפרש עד שכל הכתיבות הקודמות יתממשו. הכתיבות הללו עשויות להישאר ב-store buffer לאחר הפרישה, מה שמונע מהן להיות ספקולטיביות נוספות. עם ROBs ענקיים ו-store buffers גדולים במעבדים מודרניים, המצב אינו נורא מאוד, אך הוא מהווה בעיה גדולה עבור מעבדים שמבצעים את הביצוע בסדר קבוע או עבור מעבדים עם יכולות ביצוע צנועות מחוץ לסדר.

אפילו עם יכולות ניהול עצומות מחוץ לסדר, תהליכי הספקולציה יכולים להיכנס לחלון זמן גדול שבו המעבד עלול להיתקל בעיכוב משמעותי אם יידרש להשליך את ה-pipeline ולזרוק את ה-ROB. בקוד מקביל, הניגש לזיכרון משותף, זה יכול להוביל לבעיות כמו false sharing, שבו משתנים עשויים להימצא באותה שורת cache, מה שמוביל לעונשים משמעותיים ולתופעות שליליות אחרות.

בנוסף, אם יש מעט כתיבות בקוד, אחת מהן שמחמיצה את ה-cache יכולה להימצא ב-store buffer במשך זמן רב, עד שיתקבל הנתון מזיכרון הראשי או עד שיתבצע עדכון עם בעלות בלעדית. במהלך עיכוב זה, עשויות לעבור מאות הוראות לפני שהכתיבה תתממש, מה שעלול לעכב את ההוראות החדשות מהן לא יכולה להיכנס ל-out-of-order back-end ולהיות מתוזמנות ליחידות ביצוע בזמן העיכוב.

למרות זאת, רוב הקוד לא מבצע הרבה כתיבות, ולכן ה-store buffer יתמלא במהירות. בארכיטקטורות עם סדר חלש שמאפשר סידור מחדש של Store-Store, הבעיה עם store-miss אינה משפיעה על כתיבות מאוחרות יותר.

מעבדי x86 אכן מבצעים טעינות מוקדמות באופן ספקולטיבי, ולכן ניתן להחיל את כלל Store-Load היפותטי וגם את כלל סידור Load-Load בפועל של x86.

### התמודדות עם ספקולציות

כדי להתמודד עם ספקולציות ארכיטקטורות מספקות כל מיני כלים והוראות מיוחדות.

### Memory Barriers

אחת הדרכים להתמודד עם ספקולציות של המעבד היא מחסומי זיכרון. ב-ARMv8 יש מספר מחסומים שמיועדים להתמודד עם ספקולציות:

#### מחסום DMB

ההוראה DMB אינה מונעת ביצוע ספקולטיבי של קריאות עתידיות לזיכרון. במידה ובוצעה קריאה בזיכרון באופן ספקולטיבי, הליבה מחויבת למחוק את הנתונים הספקולטיבים מהרגיסטר. לאחר מכן, הליבה מחויבת לבצע מחדש את הקריאה מהזיכרון לאחר שצפתה וסיימה את כל הגישות הקודמות לזיכרון באופן מפורש.

### מחסום ספקולציה (SB)

הוראת SB ‏(Speculation Barrier) משמשת כמחסום המונע ביצוע ספקולטיבי של הוראות עד שהמחסום מושלם. כלומר, כל הוראה שנמצאת אחרי המחסום בסדר התוכנית אינה יכולה להתבצע ספקולטיבית במקרים שבהם הספקולציה עשויה להיחשף דרך ערוצים צדדיים. לדוגמה, אם מתבצעת הקצאה ספקולטיבית במבני ה-cache, הדבר יכול להצביע על ערך נתונים שנמצא בזיכרון או ברגיסטרים, וכתוצאה מכך לחשוף את הנתונים בערוצים צדדיים.

### מחסום צריכת נתונים ספקולטיבי (CSDB)

הוראת CSDB ‏(Consumption of Speculative Data Barrier) נועדה לשלוט בביצוע ספקולטיבי הנובע מחיזוי ערכי נתונים. כאשר הוראת CSDB מופיעה בקוד, אף פקודה (למעט הוראות branch) שמופיעה לאחריה בסדר התוכנית אינה יכולה להתבצע ספקולטיבית באמצעות תוצאות של ספקולציות שמתבצעות עבור הוראות הקודמות לה.

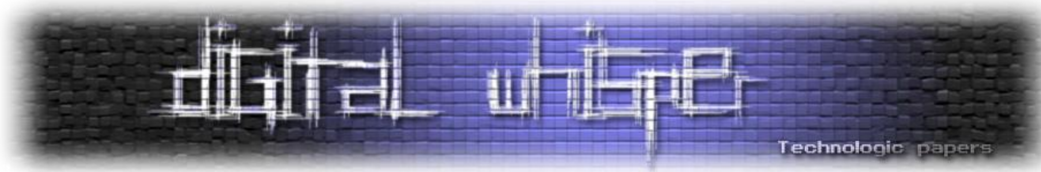
הוראת CSDB מגנה על התוכנית מפני הביצוע ספקולטיבי של הוראות כאשר ספקולציות קודמות עדיין לא נפתרו ארכיטקטונית. לדוגמה:

- ספקולציות על ערכי נתונים מכל סוג של הוראות.
- ספקולציות על הוראות (למעט branch) שקדמו ל-CSDB ולא נפתרו ארכיטקטונית.
- תחזיות מצב חיזוי SVE עבור כל הוראות SVE.

### הוראות אחרות

#### הוראת CLRBHB - ניקוי היסטוריית ה-branch-ים

ב-ARM קיימת ההוראה CLRBHB המשמשת לניקוי היסטוריית ה-branch-ים של ההקשר הנוכחי. הוראה זו מונעת שימוש במידע היסטורי של branch-ים שנוצר לפני ביצוע CLRBHB כדי לשלוט בביצוע של קוד המופיע לאחר מכן.





למרות שלארכיטקטורות שונות יש מנגנונים למניעת קריאות ספקולטיביות, כגון הוראות isb ו-isync ב-ARM ו-PowerPC בהתאמה, הקרנל אינו כולל פונקציה ייחודית למניעת קריאות ספקולטיביות בעקבות branch מותנה. אם יש צורך למנוע קריאות ספקולטיביות במקרה זה, ניתן להשתמש בהוראה smp_rmb כדי להבטיח סנכרון נכון בין ההוראות והקריאות למערכת הזיכרון.

## ספקולציות בארכיטקטורות

### ספקולציה ותחייבות של קריאות וכתיבות ב-Power PC

במערכת [Power PC](), ניתן לבצע קריאות וכתיבות באופן ספקולטיבי במצבים שונים:

1. **קריאות ספקולטיביות**: ניתן לבצע קריאות באופן ספקולטיבי, כלומר, לקרוא ערכים בזיכרון על סמך תחזיות מוקדמות. קריאה נחשבת מסופקת כאשר היא מקשרת (binds) את הערך שלה, כלומר, הערך שהתקבל מעודכן ומחובר לקריאה. קריאה נחשבת מתחייבת כאשר לא ניתן לשנות את הערך הזה יותר, והוא קבוע ואמין.
2. **כתיבות ספקולטיביות**: כתיבה ספקולטיבית מתבצעת כאשר הכתובת והערך שלה מחושבים, אולי באופן ספקולטיבי, ולכתיבה אסור להתפשט לקריאות מקומיות של thread-ים שונים. כתיבה נחשבת מתחילה כאשר הכתובת והערך שלה נקבעים, והכתיבה יכולה להתפשט ל-thread-ים אחרים כאשר היא מוכנה.

המפרט של מערכת Power מספק הגדרות ברורות לפעולות קריאה וכתיבה ספקולטיביות באמצעות המונחים satisfy read events, initiate write events, ו-commit events. זה מבטיח שכל פעולה ספקולטיבית נשמרת באופן מסודר ומבוקר, ושהערכים נחשבים סופיים כאשר הם לא יכולים להיות להשתנות מחדש.

### ספקולציות ב-AMD64

קריאה ספקולטיבית מותרת ומתרחשת כאשר המעבד מתחיל לקרוא מהזיכרון עוד לפני שברור שההוראה תושלם בהצלחה. לדוגמה, המעבד עשוי לחזות מראש את ביצועו של branch מסוים ולהתחיל לבצע את ההוראות שמגיעות בעקבותיו, מבלי לדעת אם החיזוי נכון. כאשר אחת מההוראות הללו כוללת קריאה לזיכרון, הקריאה היא ספקולטיבית. גם מילוי cache יכול להתבצע באופן ספקולטיבי.

לעומת זאת, כתיבה ספקולטיבית אינה מותרת. בדומה לכתיבה מחוץ לסדר, כתיבות ספקולטיביות אינן יכולות לרשום את תוצאותיהן לזיכרון עד שכל ההוראות הקודמות יושלמו לפי הסדר הנכון. במקרים כאלה, המעבד עשוי לשמור את תוצאות הכתיבה ב-buffer פרטי, שאינו נגיש לתוכנה, עד לרגע שבו ניתן לבצע את הכתיבה בפועל.

כאשר שורת cache נמצאת כבר במצב Modified, כתיבה שביצע hit אינה משנה את מצבה. מעברי מצב ב-cache הנגרמים על ידי קריאות, בדיקות קריאה או בדיקות כתיבה, יכולים להתרחש בעקבות שליפה מוקדמת או ביצוע ספקולטיבי. במשפחת המעבדים h17, מעבר למצב Modified אינו מתבצע באופן ספקולטיבי, אך עדיין אפשרי ש-cache יקבל נתונים כתובים מ-cache אחר באופן ספקולטיבי, מה שמוביל את השורה למצב Dirty. יישומים מסוימים עשויים לתמוך בתת-קבוצה של מצבי MOESDIF.

**ספקולציות ב-ARM**

ספקולציה במעבדים מתארת מצב שבו ליבה מבצעת או מתחילה לבצע הוראה לפני שיש לה ודאות שההוראה הזו אכן צריכה להתבצע. היתרון בכך הוא שהמעבד עשוי לסיים את הביצוע מוקדם יותר אם יתברר שהספקולציה הייתה נכונה. דוגמאות לכך כוללות שימוש בביצוע מותנה ב-ARM, או כאשר הליבה נתקלת ב-branch מותנה. במקרים כאלה, הליבה יכולה להתחיל לבצע באופן ספקולטיבי את ההוראה המותנית או את ההוראות שאחריה. במקרה שהספקולציה מתבררת כלא נכונה, הליבה חייבת לוודא שכל הסימנים לפעולה זו יימחקו.

במקרה של הוראות טעינת זיכרון, ספקולציות עשויות להיות מורכבות אף יותר. למשל, אם מתבצעת באופן ספקולטיבי הוראת load ממיקום בזיכרון הניתן ל-cache, זה עלול לגרום להעתקת אותו מיקום מהזיכרון החיצוני אל ה-cache, פעולה שעלולה לפנות שורת cache קיימת. מעבדים מודרניים מתקדמים אף יותר ועוקבים אחר דפוסי גישה לנתונים על מנת לנבא מראש את הכתובות הבאות שיידרשו, ובכך להכניס אותן ל-cache עוד לפני שההוראות המתאימות נכנסות ל-pipeline של המעבד.

המעבד מסוגל לגשת באופן ספקולטיבי למיקומי זיכרון המסומנים כרגילים. זה אומר שהוא עשוי לקרוא נתונים או הוראות מהזיכרון גם מבלי שהגישה תתבצע באופן מפורש בתוכנית, או לפני שהפניה המפורשת לזיכרון מתבצעת בפועל. גישות ספקולטיביות כאלה עשויות להתרחש כתוצאה מחיזוי של branch-ים, טעינה ספקולטיבית של שורות ב-cache, ביצוע קריאות לא מסודרות לזיכרון (out-of-order loads), או כתוצאה מאופטימיזציות חומרה נוספות.

בנוסף, המעבד עשוי לבצע גישה ספקולטיבית לזיכרון המסומן כרגיל בכל רגע נתון. לכן, בעת הערכה האם נדרשים מחסומים לזיכרון, יש לקחת בחשבון לא רק את הגישות המפורשות המתבצעות באמצעות הוראות load או store, אלא גם את הגישות הספקולטיביות שאינן מופיעות בתוכנית בצורה ישירה.

# אופטימיזציות קומפיילר

## מבוא

אחרי שראינו מספר דרכים שבהם המעבד מבצע אופטימיזציות בואו נעבור צד ונראה איך הקומפיילר מנסה לעשות אופטימיזציות לקוד שלנו.

הקומפיילר יכול לשנות את הסדר של הפעולות של התוכנית, להוסיף ולבטל loads ו-stores, לעשות ספקולציות של ערכים, ועוד אופטימיזציות יצרתיות בתנאי שזה לא משפיע על הפעולה הנראית לעין של התוכנית. לדוגמה זה יכול לבוא לידי ביטוי על ידי רצף של stores שיכול להשתנות, זה לא מפריע לרוב ב-thread יחיד אבל כשיש מספר thread-ים אז זה כבר בעייתי.

## אופטימיזציות קומפיילר

בדומה לאופטימיזציות שנעשות ברמת ארכיטקטורת החומרה, גם אופטימיזציות קומפיילר נפוצות יכולות לשנות את האופן שבו פעולות זיכרון משותף מתבצעות, וכתוצאה מכך, להשפיע על ההתנהגות של תוכנות מקביליות. אופטימיזציות כגון הקצאת רגיסטרים, תנועת קוד (code motion), חיסול תת-ביטויים נפוצים, שינויי מבנה לולאות, וחסימת טרנספורמציות (blocking transformations), כולן משפיעות על סידור פעולות הזיכרון ואף עלולות לגרום לביטולן. ההשפעות של אופטימיזציות כאלה עלולות להיות קריטיות כאשר מקמפלים תוכנות מקביליות שמוגדרות באופן מפורש.

אופטימיזציות קומפיילר נועדו לשפר את ביצועי הקוד על ידי סידור מחדש של הוראות כך שהן מנצלות בצורה אופטימלית את תכונות החומרה ומפחיתות את זמן השהייה.

במערכת עם ליבה אחת, שינויים בסדר ההוראות שביצע הקומפיילר לרוב שקופים למתכנת, מאחר והמעבד הבודד מסוגל לנהל את התלויות בין הוראות ולוודא שהן מכובדות. אבל, במערכות הכוללות ליבות מרובות, שבהן הליבות מתקשרות זו עם זו דרך זיכרון משותף או משתפות נתונים בדרכים אחרות, הופך ניהול סדר הזיכרון לעניין חשוב. במקרים אלו, חשוב להבין את השפעת אופטימיזציות הקומפיילר על סדר הגישות לזיכרון, שכן הן עשויות להשפיע על התנהגות התוכנית בכל ליבה ולהשפיע על התקשורת בין הליבות.

אופטימיזציות קומפיילר עשויות להשפיע באופן משמעותי על הקוד שלך במטרה להסתיר השהיות ב-pipeline ולנצל אופטימיזציות של המיקרו-ארכיטקטורה. קומפיילר יכול להחליט להזיז גישות לזיכרון מוקדם יותר או מאוחר יותר, בהתאם למטרה להעניק יותר זמן להשלמת פעולות מסוימות או לאזן את סדר הגישות בתוכנית. כשהקומפיילר מזיז גישות לזיכרון קדימה, המטרה כאן היא להקדים את ביצוען, כדי להבטיח שהן יושלמו לפני שהערכים שנדרשים מהן יהיו נדרשים על ידי קוד נוסף בהמשך.

באופן כללי, אופטימיזציות קומפיילר עשויות להוביל להפרת סדר התוכנית המקורי על ידי סידור מחדש של פעולות זיכרון באותו מעבד.

קומפיילרים יכולים להשתמש בכלל ה-"כאילו". כלומר, הם רשאים להפיק כל קוד שהם רוצים עבור גישה רגילה, כל עוד התוצאה המתקבלת מביצוע ה-thread נראית כאילו הקומפיילר עמד בכל הכללים הרלוונטיים של השפה. ניתן לראות את זה בצורה ברורה יותר אם מקמפלים קוד עם רמת אופטימיזציה גבוהה ואז מבצעים דיבוג על הבינארי שנוצר (נראה בהמשך דוגמה לזה).

במערכת החומרה, מקביליות ניהול הוראות נעשית בצורה דינמית, כלומר המעבד קובע בזמן הריצה אילו הוראות ניתן לבצע במקביל. לעומת זאת, במערכת התוכנה, מקביליות מנוהלת בצורה סטטית, כלומר הקומפיילר הוא זה שמחליט מראש אילו הוראות יבוצעו במקביל בזמן התרגום.

**הדגמה של השפעת האופטימיזציה**

קומפיילר יכול לזהות רצפים של הוראות שאינן תלויות זו בזו, ואם קיימים מספיק רגיסטרים פנויים, הוא יכול להקצות לכל רצף רגיסטרים שונה במהלך שלב הקצאת הרגיסטרים בתהליך יצירת הקוד.

האיורים הבאים מדגימים כיצד אופטימיזציה כמו הקצאת רגיסטרים, יכולה גם היא לגרום לסידור מחדש של פעולות זיכרון באופן דומה. שתי התוכניות שמוצגות שם נראות תואמות לפני ואחרי הקצאת הרגיסטרים:

**(a) before**

| P1 | | P2 | |
|---|---|---|---|
| *a1:* | B = 0; | *a2:* | A = 0; |
| *b1:* | A = 1; | *b2:* | B = 1; |
| *c1:* | u = B; | *c2:* | v = A; |

**(b) after**

| P1 | | P2 | |
|---|---|---|---|
| *a1:* | r1 = 0; | *a2:* | r2 = 0; |
| *b1:* | A = 1; | *b2:* | B = 1; |
| *c1:* | u = r1; | *c2:* | v = r2; |
| *d1:* | B = r1; | *d2:* | A = r2; |

[Program segments before and after register allocation A B from MEMORY CONSISTENCY MODELS FOR SHARED-MEMORY MULTIPROCESSORS (infolab.stanford.edu)]

ביצוע התוכנית לאחר הקצאת הרגיסטרים מוביל לתוצאה u=0, v=0. בפועל, הקצאת הרגיסטרים גרמה לסידור מחדש של הקריאה ל-B עם הכתיבה ל-A ב-P1, ושל הקריאה ל-A עם הכתיבה ל-B ב-P2.

שימו לב שהטרנספורמציה שנעשתה תיחשב בטוחה אם נבחן את הקוד של כל מעבד כיחידה נפרדת, כלומר, תוכנית חד-מעבד. אבל, במערכות מקביליות, שינויי סדר כאלה עלולים להוביל להתנהגויות בלתי צפויות ולשבור את הסנכרון בין מעבדים שונים.

## ההשפעות של אופטימיזציות קומפיילר על ביטול פעולות זיכרון

האיורים הבאים ממחישים בעיות נוספות הנובעות מאופטימיזציות קומפיילר, כמו הקצאת רגיסטר, שהשפעתן נובעת מביטול פעולות זיכרון מסוימות. לדוגמה, בקטע הקוד בקטע c באיור מופיעה לולאת while, שמסתיימת בכל ביצוע תחת מודל הזיכרון רציף. לעומת זאת, הקוד בקטע d באיור מציג מצב שבו רגיסטר משמש לאחסון הערך של משתנה Flag שהוקצה ל-P2. בכך נמנעת הקריאה החוזרת של Flag בתוך לולאת ה-while.

**(c) before**

```
     P1                  P2
a1:  A = 1;         a2:  while (Flag == 0);
b1:  Flag = 1;      b2:  u = A;
```

**(d) after**

```
     P1                  P2
a1:  A = 1;         a2:  r1 = Flag;
b1:  Flag = 1;      b2:  while (r1 == 0);
                    c2:  u = A;
```

[Program segments before and after register allocation C D from MEMORY CONSISTENCY MODELS FOR SHARED-MEMORY MULTIPROCESSORS (infolab.stanford.edu)]

נניח שהתוכנית עברה טרנספורמציה שבה הקריאה למשתנה Flag ב-P2 מחזירה את הערך 0. במצב כזה, לולאת ה-while ב-P2 לא תסתיים, דבר שמפר את מודל הזיכרון הרציף. הסיבה לכך היא שהקצאת רגיסטר למשתנה Flag גורמת לכך ש-P2 לא יראה את השינויים שבוצעו על ידי P1 בערך של משתנה Flag. זה יוצר מצב שבו נראה כאילו פעולת הכתיבה של Flag ב-P1 לא נצפתה כלל על ידי P2.

חשוב לציין שאופטימיזציה זו תיחשב בטוחה בתוכנית במעבד יחיד, שכן שם אין צורך לדאוג לסנכרון זיכרון thread-ים שונים.

המקרים המתוארים מדגימים כיצד אופטימיזציות קומפיילר נפוצות, שמיועדות לתוכניות למעבד יחיד, עלולות להוביל להתנהגות לא נכונה כאשר הן מיושמות על תוכניות מקבילות במפורש. אופטימיזציות אלו עשויות לשנות את האופן שבו זיכרון משותף מנוהל, דבר שעלול להוביל למירוצי נתונים ולבעיות סנכרון במערכות מרובות מעבדים.

**השפעת אופטימיזציות על מודל הזיכרון הרציף**

איור הבא מציג דוגמה להשפעתה של אופטימיזציה קומפיילרית נפוצה. באיור בקטע a מופיעה התוכנית המקורית, שנכתבה במפורש תחת מודל הזיכרון הרציף. בתוכנית זו, P1 כותב ערך למשתנה A, מגדיר Flag, ולאחר מכן קורא שוב את הערך של A. בינתיים, P2 ממתין עד ש-Flag יוגדר לפני שהוא קורא את A. תחת מודל רציף, לולאת ה-while של P2 תמיד תסתיים, והתוצאה האפשרית היחידה היא u=1.

```
     P1                    P2

a1:  A = 1;           a2:  while (Flag == 0);
b1:  Flag = 1;        b2:  u = A;
c1:  ... = A;
```

**(a) before**

```
     P1                    P2

a1:  r1 = 1;          a2:  r2 = Flag;
b1:  Flag = 1;        b2:  while (r2 == 0);
c1:  ... = r1;        c2:  u = A;
d1:  A = r1;
```

**(b) after**

[Effect of register allocation from MEMORY CONSISTENCY MODELS FOR SHARED-MEMORY MULTIPROCESSORS (infolab.stanford.edu)]

עם זאת, קטע b באיור מציג את ההשפעה של אופטימיזציה מסוג הקצאת רגיסטר למשתנים A ו-Flag. אופטימיזציה זו נחשבת תקינה לחלוטין אם בוחנים את הקוד של כל מעבד בנפרד, כאילו הוא תוכנית במעבד יחיד. אך כאשר מדובר בתוכנית מקבילית, האופטימיזציה עלולה להפר את הסמנטיקה של מודל SC.

לדוגמה, הקצאת רגיסטר למשתנה A ב-P1 עשויה לאפשר ל-P2 לקרוא את הערך 0 עבור A, מכיוון שהכתיבה ל-A ב-P1 מתבצעת רק לאחר הכתיבה ל-Flag. זהו למעשה מקרה של סידור מחדש של פעולות כתוצאה מהאופטימיזציה. בנוסף, אם Flag מוקצה גם הוא לרגיסטר ב-P2, לולאת ה-while עלולה שלא להסתיים לעולם, מכיוון ש-P2 יקרא את Flag פעם אחת בלבד ולא ימשיך לעדכן את הערך שלו מתוך הזיכרון. כלומר, אם הקריאה הראשונית ל-Flag מחזירה את הערך 0, P2 עשוי לא לקרוא את Flag שוב ולעולם לא לסיים את לולאת ה-while.

שתי ההתנהגויות שתוארו אינן מותרות בתוכנית המקורית תחת מודל הזיכרון הרציף.

תוכנית מקורית מניחה לרוב מודל זיכרון בסיסי מסוים. עם זאת, ייתכן שארכיטקטורת היעד או החומרה שבהן התוכנית תפעל תומכות במודל זיכרון שונה מהמודל המקורי.

הנ"ל יוצר שתי דרישות מהקומפיילר:

1. **הגבלת אופטימיזציות הקומפיילר:** הקומפיילר חייב להגביל את האופטימיזציות שהוא מבצע כדי לוודא שהן תואמות לסמנטיקה של מודל הזיכרון שהתוכנית המקורית מסתמכת עליו.
2. **מיפוי נכון בין המודלים:** הקומפיילר צריך למפות את הסמנטיקה של מודל הזיכרון המקורי כך שתהיה תואמת למודל הזיכרון של החומרה בארכיטקטורת היעד, במיוחד כאשר המודלים שונים זה מזה.

יתכן שגם במצבים שבהם נדרשות אופטימיזציות בטוחות, הקומפיילר צריך להיות מודע למודל הזיכרון של ארכיטקטורת היעד כדי להחליט אילו אופטימיזציות ניתן ליישם מבלי לפגוע בתפקוד התקין של התוכנית.

כדי להבטיח אופטימיזציות בטוחות, יש לספק לקומפיילר מידע מפורט, כגון שימוש ב-label-ים או מחסומי זיכרון, שיסייעו לו להקפיד על הסדר והגישה הנכונים בזיכרון.

**תנאי תלות בערך, התחלה, ותלות חד-מעבד**

כדי להבטיח את תקינות התלות בנתונים בתוכניות חד-מעבד, הקומפיילר צריך לוודא שאם קריאה מחזירה את הערך שנכתב על ידי אותו מעבד, אז הכתיבה היא הפעולה האחרונה לפני הקריאה, לפי סדר התוכנית המקורי. זה פשוט יחסית כאשר אין אופטימיזציות כמו הקצאת רגיסטרים, מכיוון שהקומפיילר יכול פשוט להימנע מסידור מחדש של פעולות כתיבה וקריאה לאותה כתובת זיכרון.

עם זאת, כאשר מתבצעת אופטימיזציה כמו הקצאת רגיסטרים, התקינות נשמרת אם במהלך הקצאת הרגיסטרים הקומפיילר מבטיח שסדר הפעולות בתוך הרגיסטרים תואם את סדר הפעולות בזיכרון בתוכנית המקורית. בנוסף, תנאי התלות החד-מעבד מחייבים שמירה על סדר הפעולות בין כל הפעולות שמתנגשות בזיכרון.

תנאים אלו חלים רק על פעולות המכוונות לאותה כתובת זיכרון. מכיוון שאופטימיזציות קומפיילר מתבצעות בזמן הקומפילציה, הקומפיילר אינו יכול לדעת בזמן זה את הכתובות המדויקות שייחשפו בזמן הריצה. לכן, לעיתים הקומפיילר יצטרך להניח הנחות שמרניות לגבי התנגשויות אפשריות בין פעולות בזיכרון. מצד שני, במקרים מסוימים, הקומפיילר יכול לבצע אופטימיזציות מדויקות יותר מהחומרה בזמן ריצה על ידי שימוש בידע סטטי על התוכנית ומבני הנתונים שלה כדי לקבוע שאין התנגשויות בין פעולות מסוימות.

באיור הבא, מופיע תבנית SB (הכוונה בתבנית של מבחני לקמוס שיוסברו במאמר עתידי), נבדוק כיצד אופטימיזציות שונות עשויות להפר את תנאי הסיום ותנאי התקינות של תוכניות חד-מעבד.

| P1 | P2 |
|---|---|
| *a1:* A = 1; | *a2:* B = 1; |
| *b1:* while (B == 0); | *b2:* while (A == 0); |

[Example to illustrate the termination condition from MEMORY CONSISTENCY MODELS FOR SHARED-MEMORY MULTIPROCESSORS, מקור]

נניח שמודל זיכרון דורש שכתיבת ערך ל-A על ידי P1 וכתיבת ערך ל-B על ידי P2 יעמדו בתנאים מסוימים של סיום. תנאים חד-מעבדיים, כגון נכונות, ערך ותנאי סיום, מחייבים שכל לולאת while בתוכנית תסיים את ביצועיה בכל ריצה אפשרית של הקוד. עם זאת, אופטימיזציות מסוימות עלולות להפר את התנאים הללו:

1. **העברת לולאת while מעל הכתיבה:** אם לולאת while בכל מעבד מועברת מעל הכתיבה המבוצעת באותו מעבד, אופטימיזציה זו משנה את הסדר שבו מבוצעות הפעולות. מצב זה אינו מתואם עם תנאי התקינות החד-מעבדיים, שכן הוא עשוי לשנות את הסדר שבו מבוצעות פעולות הזיכרון ולמנוע סיום תקין של הלולאות.
2. **כתיבה לרגסיטר בלבד:** אם הכתיבה ל-A ול-B מתבצעת ישירות לרגיסטר ולא נכתבת חזרה לזיכרון, הערכים החדשים לא יירשמו בזיכרון. אופטימיזציה זו מפרה את תנאי הסיום, מכיוון שהערכים החדשים אינם מתעדכנים בזיכרון, דבר שמונע סיום נכון של הלולאות לפי תנאים שהוגדרו.
3. **הקצאת מיקומים ב-P1 ו-P2:** אם מיקום B מוקצה ל-P1 או מיקום A מוקצה ל-P2, הדבר מפר את ההגדרה של הוראות ביצוע שמגבילות את מספר תתי-הפעולות שיכולות להקדים כל פעולה מסוימת בסדר הביצוע. אופטימיזציה זו עשויה לשנות את סדר פעולות הזיכרון ולפגוע בתנאי הסיום והתקינות של התוכנית.

באופן כללי, אופטימיזציות אלו יכולות לשנות את הסדר שבו מתבצעות פעולות הזיכרון בתוכנית, דבר שעשוי להפר את תנאי הסיום ואת תנאי התקינות החד-מעבדיים הנדרשים במודל הזיכרון.

## מחסומי קומפיילר

מחסומי זיכרון קומפיילר, המכונים גם מחסומי קומפיילר, הם הוראות או הנחיות שמיועדות לכפות סידור וסנכרון מדויקים של גישות לזיכרון בתוך הקוד שהקומפיילר יוצר. השימוש במחסומים אלו נועד למנוע מהקומפיילר לבצע סידור מחדש או אופטימיזציה של פעולות זיכרון באופן שעלול להפר את ההתנהגות הרצויה של התוכנית.

בתכנות מקבילי, מחסומי זיכרון קומפיילר חיוניים כדי להבטיח התנהגות צפויה ונכונה בעת עבודה עם זיכרון משותף ותיאום בין thread-ים. הם מבטיחים שפעולות זיכרון יתבצעו בסדר המתוכנן על ידי המתכנת, גם אם הקומפיילר יכול היה לייעל את הקוד או לשנות את סדר ההוראות.

שימוש נכון במחסומי זיכרון קומפיילר מאפשר למתכנתים להבטיח שהקוד שנוצר יכבד את אילוצי הסידור והסנכרון שנדרשים, ובכך להבטיח פעולה נכונה ואמינה של תוכנות מקביליות או מבוססות thread-ים.

ב-GCC רוב קריאות הפונקציות פועלות כמחסומי קומפיילר גם אם הפונקציה עצמה אינה מכילה מחסום כזה. החריגים לכך הם פונקציות inline, פונקציות שמסומנות כ-pure, ומקרים בהם נעשה שימוש בהפקת קוד בזמן תהליך ה-linking. במקרים שבהם הפונקציה חיצונית, היא משמשת כמחסום קומפיילר חזק יותר, שכן הקומפיילר אינו יכול לדעת את תופעות הלוואי של הפונקציה החיצונית. לכן, הקומפיילר חייב להימנע מהנחות לגבי הגישה לזיכרון שעשויות להיות גלויות לפונקציה זו. הקומפיילר משנה את סדר האינטראקציות בזיכרון

מסיבות דומות לאלה של המעבד - אופטימיזציה של ביצועים. שינויים אלה הם תוצאה ישירה מהמורכבות של מעבדים מודרניים, שמצדיקים אופטימיזציות לשיפור הביצועים.

## מחסומי קומפיילר בקרנל - המאקרו barrier

מחסום קומפיילר הוא סוג של מחסום המתייחס רק לתהליך הקומפילציה ולא ישירות לביצוע ההוראות על ידי המעבד. בקוד המקור של הקרנל של לינוקס, המאקרו barrier משמש כמחסום קומפיילר. הוא לא מייצר הוראות מכונה ישירות בקוד האובייקט, אלא משפיעה על הדרך בה הקומפיילר יוצר את שאר קוד האובייקט.

המאקרו barrier שמוגדר ב-include/linux/compiler.h מונע מהקומפיילר להזיז פעולות גישה לזיכרון משני צדי המחסום. זהו מחסום כללי שאינו מחלק את פעולות הזיכרון לקריאות או כתיבות נפרדות (כמו סוגי מחסומים אחרים). בהינתן קוד מקור המכיל מחסום קומפיילר כזה:

```
... some memory accesses ...
barrier();
... some other memory accesses ...
```

הפונקציה barrier מבטיחה שכל הוראות המכונה שקשורות לקבוצת גישות אחת יושלמו במלואן לפני שהמעבד יתפנה לבצע הוראות המכונה שקשורות לקבוצת גישות אחרת. זאת אומרת, כל גישה במסגרת הקבוצה הראשונה תושלם לפני כל גישה מהקבוצה השנייה, גם אם חלק מהגישות הן גישות פשוטות (plain). יש לציין שהמעבד עשוי להוציא את ההוראות בסדר שאינו תואם את סדר התוכנית, אך ניתן להתמודד עם בעיות כאלה בדרכים אחרות. ללא השימוש ב-barrier, לא תהיה הבטחה לכך שסדר ההוראות נשמר, וייתכן ששתי קבוצות הגישות יתערבבו או אפילו יחליפו את מקומן בקוד האובייקט.

ה-barrier מונע מהקומפיילר לבצע סידור מחדש או אופטימיזציה של פעולות זיכרון מעבר למחסום, בשני הכיוונים. בתוך לולאות, המחסום מאלץ את הקומפיילר לטעון מחדש את המשתנים בהם נעשה שימוש בכל מחזור של הלולאה, ומונע אופטימיזציות שעלולות להוביל להתנהגות לא צפויה.

## שימוש ב-READ_ONCE ו-WRITE_ONCE

פונקציות המאקרו READ_ONCE ו-WRITE_ONCE נועדו לעזור למנוע מהקומפיילר לבצע אופטימיזציות מוטעות שעלולות לשנות את אופן פעולת התוכנית. לדוגמה, לעיתים הקומפיילר עלול להניח שהמעבד הנוכחי הוא היחיד שמבצע עדכונים למשתנה מסוים, ולכן הוא מרשה לעצמו לבצע שינויים בקוד על בסיס הנחות אלו. השימוש ב-READ_ONCE ו-WRITE_ONCE מונע הנחות כאלו בכך שהוא מכריח את הקומפיילר לא לשמור בזיכרונו מידע מוטעה אודות תוכן משתנים מסוימים.

בעת השימוש ב-READ_ONCE ו-WRITE_ONCE, הקומפיילר נדרש לשכוח את הערכים של מיקומי זיכרון מסוימים, אבל בניגוד ל-barrier, הוא אינו צריך למחוק את כל התוכן שהוא שומר ברגיסטרים. (זה נשמע קצת מוזר בהתחלה בגלל שלקומפיילר אין רגיסטרים, אבל הכוונה כאן היא שאסור לו להניח הנחות קודמות ולכן

הוא חייב לעשות את זה). בנוסף, הקומפיילר מחויב לכבד את סדר הפעולות שבו מתבצעים ה-READ_ONCE וה-WRITE_ONCE, אך המעבד עצמו אינו חייב לעשות זאת, מה שעלול לגרום לשינויים נוספים ברמת החומרה.

משתנים פרטיים, שאינם משותפים בין מעבדים שונים, ניתן לגשת אליהם בדרך כלל מבלי להשתמש ב-READ_ONCE או WRITE_ONCE. למעשה, משתנים פרטיים אינם חייבים להיות מאוחסנים בזיכרון רגיל, ויכולים להיות מאוחסנים ישירות ברגיסטר של המעבד. כאשר משתנה מסומן כ-volatile, אין צורך להשתמש ב-READ_ONCE וב-WRITE_ONCE, מכיוון שהשניים מיושמים כהטלות volatile. משתנים מסומנים כ-volatile מונעים אופטימיזציות מיותרות מצד הקומפיילר. ההגנות שמספקות הפונקציות READ_ONCE, WRITE_ONCE ודומיהן אינן מושלמות, ובמצבים מסוימים הקומפיילר עשוי לשנות את התנהגות הזיכרון בדרכים שאינן תואמות למודל הזיכרון המיועד. פרימיטיבים של ONCE מונעים תופעות של קריעת נתונים (כמו כשהקומפיילר מפרק גישה יחידה למשתנה גדול למספר גישות קטנות יותר) והיתוך גישות (כאשר הקומפיילר מאחד מספר גישות לאותו מיקום בגישה אחת בלבד). בעת גישה למשתנה בודד משותף בין מעבדים מרובים השימוש בהם מונע קריעת זיכרון (Memory Tearing).

### מחסומי קומפיילר והשפעתם על המעבד

חשוב להבין כי המחסומים הקומפיילרים, כמו READ_ONCE, WRITE_ONCE ו-barrier, אינם משפיעים באופן ישיר על אופן הפעולה של המעבד. לאחר יצירת הקוד, המעבד עדיין חופשי לסדר מחדש פעולות בהתאם לאופטימיזציות פנימיות שלו. המחסומים משפיעים רק על אופן יצירת הקוד על ידי הקומפיילר, ולא על אופן הביצוע הפנימי של המעבד.

### יצירת פעולות טעינה נוספות על ידי הקומפיילר

לעיתים, הקומפיילר עלול להוסיף פעולות loads נוספות שמעולם לא היו בקוד המקורי. פעולות loads אלו לא תמיד גורמות לנזק, אך הן יכולות לגרום להקפצת שורות cache (cache bouncing), תופעה שבה כמה מעבדים ניגשים לאותם מיקומי זיכרון ובכך גורמים להקטנת ביצועים ופגיעה ב-scalability של המערכת. כדי למנוע פעולות loads שהומצאו על ידי הקומפיילר, צריך להשתמש ב-READ_ONCE.

### שימוש בפרימיטיבים חזקים באינטראקציה בין מעבדים

כאשר יש צורך באינטראקציה בין מעבדים שונים, יש להשתמש בפרימיטיבים חזקים יותר, כמו smp_load_acquire ו-smp_store_release, כדי להבטיח שהסדר הנכון של פעולות הזיכרון נשמר, ושיש שקיפות מלאה של השינויים בין המעבדים. במקרים בהם יש תלות בקרה, כלומר כאשר סדר הביצוע תלוי בבדיקת תנאי כלשהו, יש להשתמש ב-WRITE_ONCE עבור פעולת ה-store כדי להבטיח שהסדר ישמר לפי תלות הבקרה.

**הערה:** יהיה הסבר מורחב במאמר עתידי על כל המחסומי זיכרון שצויינו כאן.

**השפעת Volatile**

רוב הקומפיילרים אינם מותאמים במיוחד לעבודה עם תוכניות מרובות מעבדים. כך נוצר שאופטימיזציות שמיועדות לתוכניות למעבד יחיד עלולות לשבור את הסמנטיקה במודלי זיכרון חלשים במערכות מרובות מעבדים. אחת מהשיטות הנפוצות להתמודדות עם בעיה זו היא השימוש במילת המפתח volatile, המוכרת בשפות כמו C, לזיהוי משתנים משותפים בין thread-ים. הפונקציונליות של volatile נועדה במקור לטפל בפעולות קלט/פלט בסביבת חד-מעבד, אך הקומפיילר נמנע מהקצאת משתנים אלו לרגיסטרים ושומר על הסדר המקורי של הקוד לגביהם.

השימוש במילת המפתח volatile מבטיח שהקומפיילר ייצור קוד שבו גישה למשתנה מתבצעת תמיד. המשמעות היא שהקומפיילר אינו מורשה לבצע אופטימיזציות שעלולות להסתמך על ערכים שנשמרו או הונחו בעבר. השימוש ב-volatile נועד לציין בפני הקומפיילר שהמשתנה עשוי להשתנות בדרכים חיצוניות שאינן נראות ישירות בקוד, ולכן יש לגשת אליו בכל פעם ישירות מהזיכרון הראשי. עם זאת, מילת המפתח volatile מתאימה בעיקר למודלים מחמירים של זיכרון כמו SC, אך אינה גמישה מספיק לתמיכה במודלים רגועים יותר. אחת הבעיות המרכזיות היא שפעולות למשתנים שאינם מוגדרים כ-volatile אינן מסודרות ביחס לפעולות למשתנים המוגדרים כ-volatile.

בכל המעבדים המודרנים שמבצעים ביצוע מחוץ לסדר הגדרת כל פעולות הסנכרון כ-volatile אינה מספיקה כדי להבטיח את תקינות התוכנית, מכיוון שהקומפיילר עדיין עלול להזיז או לבצע אופטימיזציה של פעולות הקצאת נתונים מעבר לפעולות הסנכרון. הכרזה על אובייקטים כ-volatile משמשת את המתכנת לציין שהחומרה צריכה לגשת לאובייקטים אלה בדיוק כפי שמצוין בקוד המקור. זאת עשויה להיות דרישה מיוחדת עבור רגיסטרי התקנים ממופי זיכרון או DMA buffers ולא עבור מיקומי זיכרון רגילים. הציפייה היא שתהליך תרגום הקוד של קומפיילרים לאובייקטים volatile לקוד מכונה יהיה מדויק ככל האפשר.

כשעוסקים בגישה לאובייקטים volatile חלקית, הדרישות משתנות בהתאם:

- **שמירה על סדר**: הדרישה לשמירה על סדר מתמעטת. היא חלה רק על זוגות של גישות לאותו אובייקט volatile. כלומר, בעוד שסדר הגישות לאובייקט volatile אחד חייב להישמר, גישות לאובייקטים שונים יכולות להיות מסודרות מחדש זה ביחס לזה.
- **מיזוג והשמטה**: הקומפיילר מורשה למזג או להשמיט גישות לאובייקטים volatile חלקית.
- **שמירה על יחס rf**: המיפוי חייב לשמר את יחס כתיבה-לקריאה (rf). זה אומר, אם גישת קריאה R מבצעת קריאה מהכתיבה W אז יכולים להיות שינויים אבל חייב להישמר היחס ש-R קורא את הערך ש-W כתב (W → rf R).

**הערה:** הרעיון של יחס rf יוסבר בהרחבה במאמר עתידי.

המשמעות המדויקת של הגבלת הגישה ל-volatile אינה תמיד ברורה. טיפול ב-volatile, כפי שהובן על ידי מפתחי קומפיילרים, נתפס לעיתים יותר כעניין של מוסכמה חברתית (folklore) מאשר כללים ברורים. כדי להנחות את משתמשי ומיישמי C ו-C++, התקן מספק הערות שהן רמזים עבור מימושים:

- **הסמנטיקה של גישה לאובייקט volatile גלובלי**: הסמנטיקה של גישה לאובייקט volatile גלובלי מוגדרת לפי המימוש. כלומר, המימוש עצמו אחראי להגדיר את ההתנהגות של גישות כאלה.
- **מניעת אופטימיזציה**: ה-volatile מהווה רמז למימוש להימנע מאופטימיזציות אגרסיביות הכוללות את האובייקט, משום שערכו עשוי להשתנות באמצעים בלתי ניתנים לזיהוי על ידי המימוש.

יש לציין ש-volatile אינה מספקת מחסום זיכרון לצורך אכיפת עקביות cache, ולכן אינה מספיקה לצורך תקשורת בין thread-ים במערכות מרובות ליבות. תקני C ו-C++ שקדמו ל-C11 ו-C++11 אינם מתייחסים לתמיכה בעבודה עם מספר thread-ים או מעבדים, ולכן השימוש ב-volatile תלוי בקומפיילר ובחומרה הספציפיים.

### Data Race

בפשטות מירוץ נתונים מתרחש כאשר מספר thread-ים ניגשים במקביל למשתנה מסוים, כאשר לפחות אחת מהגישות היא גישה רגילה ולפחות אחת מהן היא פעולת כתיבה (store). מצב זה עלול לגרום להתנהגות לא צפויה של התוכנית, מכיוון שאין סנכרון נכון בין הגישות למשתנה.

מירוצי נתונים (Data Races) הם הפרות של תקני השפה, (לדוגמה, התקן C11) והם נגרמים מכך ש-thread-ים ניגשים במקביל למשתנה ללא סנכרון נכון. לעומתם, Race Conditions הם באגים לוגים הנובעים מנעילה שגויה או משימוש לא נכון בסמנטיקה של פעולות release/acquire, מה שיכול להוביל להתנהגות לא צפויה בתוכנית גם אם אין הפרות של תקני השפה.

מנקודת מבט טכנית, הקומפיילר יכול להניח שבזמן שהקוד רץ, לא יתרחשו מירוצי נתונים. מירוץ נתונים מתרחש כאשר מתבצעות שתי גישות לזיכרון בתנאים הבאים:

1. הן ניגשות לאותו מיקום בזיכרון.
2. לפחות אחת מהן היא גישה של store.
3. לפחות אחת מהן היא גישה פשוטה (plain).
4. הגישות מתבצעות במעבדים שונים או ב-thread-ים שונים באותו מעבד.
5. הגישות מתבצעות במקביל אחת לשנייה.

בספרות המקצועית, נאמר ששתי גישות מתנגשות אם הן עומדות בתנאים 1 ו-2. אם נוסיף לתנאים אלה גם את התנאים 3 ו-4, נאמר ששתי גישות הן מועמדות למירוץ. אבל יש מקרים שבהם הקומפיילר יכול לעשות טוב יותר מחומרת זמן ריצה על ידי שימוש בידע שלו על התוכנית ומבני הנתונים כדי לקבוע ששתי פעולות לא יכולות להתנגש.

הקומפיילר אינו רשאי לבצע טרנספורמציות מתקדמות על גישות המסומנות כ-volatile, ולכן כל גישה כזו בקוד המקור מתורגמת כמעט ישירות להוראת מכונה בקוד האובייקט. לעומת זאת, גישות רגילות (plain) יכולות לעבור שינויים משמעותיים: הקומפיילר יכול לשלב אותן, לפצל אותן, לשכפל אותן, להסיר אותן, להמציא חדשות, וכדומה. לכן, לראות גישה רגילה בקוד המקור לא מספק מידע ברור לגבי הוראות המכונה שיופיעו בקוד האובייקט. עם זאת, הקומפיילר אינו חופשי לגמרי לפעול כרצונו. הוא כפוף למגבלות מסוימות. אסור לו להוסיף מירוצי נתונים לקוד האובייקט אם הקוד המקור אינו מכיל כבר מירוצי נתונים, שכן הדבר היה מבטל את תועלתם של מודלי זיכרון ולא היה מאפשר כתיבת קוד מרובה-thread-ים בטוח. בנוסף, הקומפיילר לא יכול להעביר גישה רגילה מעבר למחסום קומפיילר.

בקרנל שימוש ב-READ_ONCE מונע מהקומפיילר לסדר מחדש את פעולת ה-READ_ONCE עם גישות אחרות שמסומנות באופן מיוחד, כמו פעולות גישה לזיכרון באמצעות גישות מסומנת ולא גישות זיכרון רגילות בשפת C. זה מבטיח שהקומפיילר לא ישנה את סדר פעולות הגישה, מה שעשוי להוביל למירוצי נתונים.

### שפת C

שפת C מכילה היבטים שבהם הסדר אינו מוגדר. לדוגמה, בביטוי כמו:

```
foo(x) + bar(y)
```

אין קביעה חד-משמעית של הסדר שבו ייקראו הפונקציות foo ו-bar. הקומפיילר חופשי לבחור את הסדר, ואף רשאי לשלב את החישובים בתהליך הקומפילציה. בנוסף, ישנן התנהגויות מסוימות בתוכנית שמוגדרות כהתנהגות לא מוגדרת(undefined behavior) לפי תקן שפת C. כאשר התנהגויות כאלה מתרחשות, אין כל הבטחה לתוצאה של הפעולה. דוגמאות לכך כוללות גישה למשתנה לא מאותחל או גישה אל מחוץ לטווח של מערך, בין אם לפני תחילתו או מעבר לסופו.

הקרנל של לינוקס מסתמך אך ורק על שימוש באסמבלי inline לצורך יישום ההוראות האטומיות שלה, ולא עושה שימוש בהוראות האטומיות של תקן C11 או בהוראות המובנות של הקומפיילר (compiler built-ins). מודל זיכרון של הקרנל של לינוקס קפדני יותר מזה של C11.

## יעול הקוד בעזרת הקומפיילר

### ביצוע מותנה

כאשר משתמשים בביצוע מותנה שבו הביטוי נוטה להניב תוצאה אחת הרבה יותר מאשר את השנייה, יש סיכון לחיזוי branch סטטי שגוי, מה שעלול להוביל לבועות ב-pipeline.

כדי למנוע בעיה זו, ניתן להנחות את הקומפיילר להעביר את הקוד שמבוצע בתדירות נמוכה מהנתיב של הקוד הראשי. במקרה זה, ה-branch המותנה שנוצר עבור הצהרת if יקפוץ מחוץ לזרם הביצוע הרגיל והביצוע

הנפוץ יהיה רציף. במצבים כאלה, משתמשים בהוראה __builtin_expect כדי להנחות את הקומפיילר לגבי הסבירות של הערך בביטוי התנאי (הפרמטר הראשון) בהשוואה לערך הצפוי (הפרמטר השני).

הוראה זו מאפשרת לקומפיילר לבצע אופטימיזציות שמביאות את הקוד המיועד לביצוע בתדירות נמוכה יותר, ובכך לצמצם את הסיכון לבעיות ביצועים. לרוב, משתמשים בהוראה זו דרך פקודות מאקרו נפוצות:

```
unlikely(expr) __builtin_expect(!!(expr), 0)
likely(expr) __builtin_expect(!!(expr), 1)
```

**בקרנל של לינוקס**

בקובץ /proc/kallsyms, הפונקציות המסומנות ב-cold. הן פונקציות שהוגדרו על ידי תכונת הקומפיילר המצביעה על כך שסביר להניח שהפונקציה לא תתבצע בתדירות גבוהה. פונקציות אלה ממוקמות בדרך כלל בקטע קישור נפרד, במטרה לשפר את מיקום הקוד של הפונקציות הנדרשות לביצוע מהיר, שאינן מסומנות כ-cold. זהו חלק מהאופטימיזציה שמטרתה לייעל את ביצועי הקוד על ידי ארגון נכון של הפונקציות השונות בקוד.

**דוגמה לסידור מחדש של הקומפיילר**

ב-gcc בגרסה 11.4.0 ב-ubuntu 22.04 ב-x86-64 הקוד הבא לא בטוח בריצה של מספר thread-ים, בגלל שאין כאן מחסומים בכלל, ואפילו אם המעבד לא מבצע סידור מחדש הקומפיילר יכול לעשות סידור מחדש (ובאמת עושה):

```
// Shared values
volatile int lock;
int data;

// Some expensive computation
int compute(int a);

void my_function(int v, int a) {
    // Wait for the lock
    while(lock != v);

    // Critical section
    data = compute(a);

    // Release lock
    lock = v + 1;
}
```

על ידי קימפול בעזרת gcc:

```
gcc -O1 -S test.c -o test.s
```

נקבל את הקטע הבא (רק הקטע הרלוונטי מוצג):

```
movl    my_lock(%rip), %eax
cmpl    %ebx, %eax
jne .L2
```

```
call     compute@PLT
movl     %eax, data(%rip)
addl     $1, %ebx
movl     %ebx, my_lock(%rip)
```

אפשר לראות שהוא נמצא בסדר שהקוד מקור נמצא, סדר הפעולות הוא:

1. קריאה מהמנעול my_lock
2. כתיבה ל-data
3. כתיבה ל-my_lock

וכשמנסים להגביר את האופטימיזציות:

```
gcc -O2 -S test.c -o test.s
```

נקבל את הקטע הבא (רק הקטע הרלוונטי מוצג):

```
movl     my_lock(%rip), %eax
cmpl     %ebx, %eax
jne .L2
call     compute@PLT
addl     $1, %ebx
movl     %ebx, my_lock(%rip)
popq     %rbx
.cfi_def_cfa_offset 8
movl     %eax, data(%rip)
```

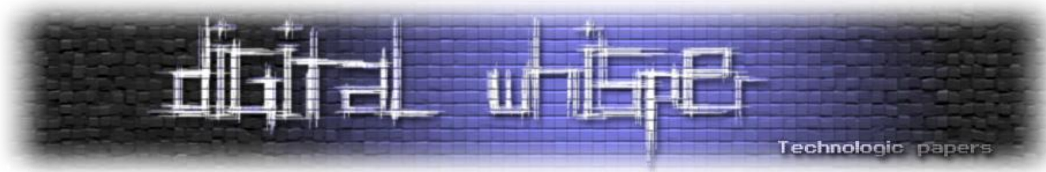
1. קריאה מהמנעול my_lock
2. כתיבה ל-my_lock
3. כתיבה ל-data

כדי לדכא את הסידור מחדש הזה של הקומפיילר צריך להוסיף מחסום זיכרון קומפיילר בין הכתיבה ל-data לכתיבה ל-my_lock.

המחסום קומפיילר נכתב בצורה הבאה:

```
asm volatile("" : : : "memory");
```

ה-memory היא סוג של clobber שאומר לקומפיילר שהקוד אסמבלי מבצע קריאת זיכרון או כתיבה לפריטים אחרים מאלה הרשומים באופרנדים של הקלט והפלט (לדוגמה, גישה לזיכרון שעליו מצביע אחד מפרמטרי הקלט). כדי להבטיח שהזיכרון מכיל ערכים נכונים, ייתכן ש-GCC יצטרך לשטוף (flush) ערכי רגיסטרים ספציפיים לזיכרון לפני ביצוע קטע ה-asm. יתר על כן, הקומפיילר אינו מניח שערכים כלשהם הנקראים מהזיכרון לפני ה- asm נשארים ללא שינוי לאחר ה-asm זה; זה טוען אותם מחדש לפי הצורך. השימוש בקלובר memory יוצר למעשה מחסום זיכרון קריאה/כתיבה עבור הקומפיילר.



כעת נוסיף את המחסום לקוד שלנו:

```
// Shared values
 volatile int lock;
 int data;

 // Some expensive computation
 int compute(int a);

 void my_function(int v, int a) {
    // Wait for the lock
    while(lock != v);

    // Critical section
    data = compute(a);

    // Compiler memory barrier
    asm volatile("" : : : "memory");

    // Release lock
    lock = v + 1;
}
```

ועכשיו כשנריץ את gcc עם O2- על הקוד החדש נקבל:

```
movl    my_lock(%rip), %eax
cmpl    %ebx, %eax
jne .L2
call    compute@PLT
movl    %eax, data(%rip)
addl    $1, %ebx
movl    %ebx, my_lock(%rip)
popq    %rbx
```

וכאן אפשר לראות שהכתיבה ל-data מסודרת לפני הכתיבה ל-my_lock.

## סיכום

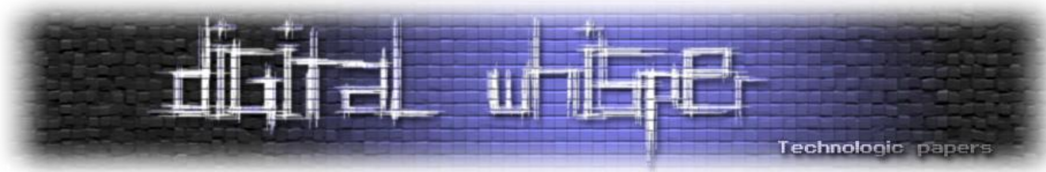
אם הגעתם עד לכאן אז שוב כל הכבוד, התחלנו בסקירה כללית של Store Buffer, בהמשך ראינו איך מעבדים מבצעים שליפת מידע ושליפות מוקדמות. הסברנו איך מעבדים מבצעים קוד בצורה ספקולטיבית, ובסוף בדקנו חלק מהאופטימיזציות שבהן קומפיילרים משתמשים.

אני מיכאל, חייל בן 22 שמתעניין בטכנולוגיה ומחשבים, מאוד אוהב מערכות הפעלה בדגש על Linux Kernel וארכיטקטורות מעבדים.

את החומרים של המאמר הזה ומאמרים עתידיים תוכלו למצוא בבלוג שלי או לפנות אלי ב-linkedin.

## מקורות


- https://en.wikipedia.org/wiki/write_buffer
- https://developer.arm.com/documentation/den0042/a/Caches
- http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.07.23a.pdf
- https://en.wikipedia.org/wiki/MESI_protocol
- https://pages.cs.wisc.edu/~markhill/papers/primer2020_2nd_edition.pdf
- https://stackoverflow.com/questions/54876208/size-of-store-buffers-on-intel-hardware-what-exactly-is-a-store-buffer
- https://stackoverflow.com/questions/64141366/can-a-speculatively-executed-cpu-branch-contain-opcodes-that-access-ram/64148401#64148401
- https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf
- https://arxiv.org/pdf/1810.04610
- https://github.com/paulmckrcu/perfbook
- https://lwn.net/Articles/252125
- http://infolab.stanford.edu/pub/cstr/reports/csl/tr/95/685/CSL-TR-95-685.pdf
- https://lwn.net/Articles/255364/
- https://dl.acm.org/doi/pdf/10.1145/3524059.3532396
- https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)
- https://developer.arm.com/documentation/den0024/a/Caches/Cache-maintenance
- https://developer.arm.com/documentation/ddi0487/latest/
- https://inria.hal.science/hal-02509910/document
- https://developer.arm.com/documentation/102336/0100/Data-Synchronization-Barrier
- https://mariokartwii.com/armv8/ch30.html
- https://lwn.net/Articles/252125/
- https://en.wikipedia.org/wiki/Speculative_execution
- https://github.com/paulmckrcu/oota
- https://en.wikipedia.org/wiki/Branch_predictor
- https://developer.arm.com/documentation/den0024/a/Memory-Ordering
- https://www.kernel.org/doc/Documentation/memory-barriers.txt
- https://github.com/torvalds/linux/blob/master/tools/memory-model/Documentation/explanation.txt
- https://github.com/ARM-software/speculation-barrier/





- https://developer.arm.com/documentation/100076/0100/A32-T32-Instruction-Set-Reference/A32-and-T32-Instructions/CSDB
- https://developer.arm.com/documentation/ddi0602/2023-06/Base-Instructions/SB--Speculation-Barrier-
- https://developer.arm.com/documentation/102336/0100/Memory-barriers
- https://mirrors.edge.kernel.org/pub/linux/kernel/people/paulmck/LWNLinuxMM/StrongModel.html
- https://acg.cis.upenn.edu/papers/cav12_axiomatic_power.pdf
- https://developer.arm.com/documentation/100941/0101/Memory-attributes
- https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/memory-access-ordering-part-2---barriers-and-the-linux-kernel
- https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/memory-access-ordering---an-introduction
- https://blog.stuffedcow.net/2014/01/x86-memory-disambiguation/
- https://stackoverflow.com/questions/37725497/how-does-memory-reordering-help-processors-and-compilers
- https://preshing.com/20120625/memory-ordering-at-compile-time/
- https://diy.inria.fr/linux/long.pdf
- https://github.com/torvalds/linux/blob/master/tools/memory-model/Documentation/recipes.txt
- https://github.com/torvalds/linux/blob/master/tools/memory-model/Documentation/ordering.txt
- https://github.com/torvalds/linux/blob/master/tools/memory-model/Documentation/control-dependencies.txt
- https://en.wikipedia.org/wiki/Instruction-level_parallelism
- https://en.wikipedia.org/wiki/Register_renaming
- https://en.wikipedia.org/wiki/Memory_barrier
- https://lwn.net/Articles/718628/
- https://coffeebeforearch.github.io/2020/11/21/compiler-memory-ordering.html
- https://gcc.gnu.org/onlinedocs/gcc/Constraints.html
- https://0xax.gitbooks.io/linux-insides/content/Theory/linux-theory-3.html
- https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html
- https://gcc.gnu.org/onlinedocs/gcc/Simple-Constraints.html