Cache Coherence

מאת מיכאל שליטין

הקדמה

בעקבות התפתחות של המחשוב בעשורים האחרונים ארכיטקטורות מעבדים מנסות לגרום לשיפור ביצועים במעבדים בכל מיני דרכים שונות כמו הוספת רכיבים למעבדים, הוספת cache-ים וריבוי מעבדים.

לרוב אנחנו פשוט מתעלמים מכל אלה ולא חושבים על זה בכלל ואם כן יוצא לנו אז לרוב זה גם רק בצורה מאוד high-level-ית בלי באמת להבין מה קורה ואנחנו משתמשים בעטיפות שאחרים יצרו בשבילנו כמו ספריות ומנגוני סנכרון קיימים. אבל אם יוצא לכם לעבוד בצורה מספיק low-level-ית או שנכנסתם קצת לקרנל אז כבר זה לא תמיד מספיק לחשוב על זה בצורה high-level-ית ויש צורך אמיתי להבין מה קורה ואיזה בעיות יכולות לקרות ואיך מתמודדים איתם.

בסדרת המאמרים הזאת אני הולך להציג מבוא קצר אל העולם הזה ונתחיל את הסדרה בהסבר על אופטימיזציות מסויימות שקיימות היום כמו ריבוי שיכולות לגרום לקוד שלנו לרוץ out-of-order ואחרי זה נראה דרכים שבהם אנחנו יכולים להתמודד עם הבעיות האלו כמו מחסומי זיכרון ומודלי זיכרון.

כמו שהוזכר קודם יש כל מיני אופטימיזציות היום שנעשות לשיפור ביצועים במעבדים וביניהם cache-ים ותמיכה בפעולות בו-זמנית על ידי ריבוי ליבות וריבוי thread-ים. מכאן עולה צורך חשוב של חיבור בין הטכניקות האלו, אבל איך עושים את זה? איך גורמים לכל האופטימיזציות והרכיבים השונים לעבוד ביחד?

המאמר הקרוב ינסה לתת מבוא לעולם הזה ונעשה את זה על ידי לימוד של מספר נושאים: נתחיל מסקירה כללית על cache לאחר מכן נעבור לקוהרנטיות cache ונבין מה זה אומר לשמור על קוהרנטיות במחשבים מודרנים, משם נגלוש ונראה דרכים שבהם הארכיטקטורות נותנות לנו גישה לשלוט בצורה מסויימת ב-cache, לאחר כל אלה נראה דוגמה לפרוטוקול קוהרנטיות ובצורה הזאת נבין את הרעיונות בצורה בהירה יותר ונסיים עם Multi-copy atomicity שדרכו נוכל לראות כל מיני מקרים מוזרים שיכולים לקרות בסוגים שונים של קוהרנטיות.

גישה זו תספק לקורא הבנה מעמיקה של האתגרים של בקוהרנטיות cache, ותאפשר להעריך כיצד פתרונות טכנולוגיים ופרוטוקולים שונים מסייעים לייעל את תהליכי העיבוד במערכות מרובות ליבות, תוך שמירה על דיוק ואמינות המידע.

הערה: כל החומר הזה למדתי באינטרנט ללא כל הכוונה ואין לי רקע אקדמי אז יכול להיות טעויות במאמר עצמו ואני מניח שיש כאן קוראים במגזין שמבינים יותר ממני אז אם אתם רואים טעיות אני אשמח שתתקנו אותי.

Cache

מבוא

זיכרון RAM מהיר כמו ליבות המעבד המודרניות יקר בהרבה לעומת זיכרון DRAM הקונבנציונלי. העלויות הגבוהות של זיכרון מהיר נוטות להיות מנוטרלות על ידי התקורות הכרוכות בניהול המשאבים. לכן, במקום להפוך את ה-SRAM למשאב שנשלט על ידי מערכת ההפעלה או המשתמש, הוא מנוהל ישירות על ידי המעבד, והשימוש בו שקוף למערכת.

במקרה כזה, ה-SRAM משמש כזיכרון cache, כלומר, כמאגר זמני עבור נתונים מהזיכרון הראשי, אשר סביר להניח שיעשה בהם שימוש בקרוב על ידי המעבד. הדבר מתאפשר בזכות העובדה שהקוד והנתונים של תוכנה נוטים להפגין מקומיות זמנית ומרחבית:

- **מקומיות מרחבית:** יש סבירות גבוהה שגישה למיקום מסוים תוביל גם לגישה למיקומים סמוכים. לדוגמה, קריאת הוראות רציפות או גישה למבני נתונים מקודדים באותו אזור זיכרון, כמו בגישה ל-stack, היא לרוב מוגבלת לאזורים קטנים בזיכרון.

- **מקומיות זמנית:** גישה לאזור זיכרון מסוים נוטה לחזור על עצמה בתוך פרק זמן קצר. לדוגמה, לולאות קוד שמבצעות גישות חוזרות ונשנות לאותו אזור זיכרון. בנוסף, פונקציות יכולות להיקרא מספר פעמים, מה שמוביל לשימוש חוזר בכתובות מסוימות בזיכרון.

לכן, גישה לזיכרון RAM על ידי הליבה אינה אקראית, אלא מציגה דפוסים של מקומיות מרחבית וזמנית. מאפיינים אלו מאפשרים ל-cache לפעול בצורה יעילה מאוד ולשפר את ביצועי המערכת.

כאשר מעבד משנה נתונים בזיכרון, אך השינוי משפיע רק על חלק מ-cache line (רצף של מספר מילים בזיכרון המאוחסנות יחד ב-cache), הוא חייב קודם לטעון את ה-cache line המלאה לזיכרון שלו. כדי להכניס נתונים חדשים ל-cache, בדרך כלל נדרש לפנות מקום.

המעבדים חופשיים לנהל את ה-cache-ים שלהם לפי שיקול דעתם, כל עוד הם נשארים בהתאם למודל הזיכרון שהוגדר עבור ארכיטקטורת המעבד. לדוגמה, זה מקובל שמעבד ינצל תקופות בהן יש פעילות מועטה או ללא פעילות על ה-bus של הזיכרון, כדי לכתוב באופן יזום cache lines מלוכלכות (כאלו שעברו שינויים ועדיין לא נכתבו לזיכרון הראשי) בחזרה לזיכרון הראשי.

סקירה כללית

כאשר המעבד מנסה לקרוא או לכתוב למיקום בזיכרון הראשי, הוא קודם כל בודק אם הנתונים מאותו מיקום כבר מאוחסנים ב-cache. אם הנתונים אכן נמצאים ב-cache, המעבד מבצע את פעולת הקריאה או הכתיבה ישירות מה-cache, שהוא מהיר בהרבה מהזיכרון הראשי. תהליך זה מאפשר גישה מהירה יותר לנתונים ושיפור משמעותי בביצועים.

רשומות (Cache entries) cache

נתונים מועברים בין הזיכרון הראשי ל-cache בצורת בלוקים בגודל קבוע הנקראים cache lines או cache blocks. כאשר cache line מועברת מהזיכרון הראשי אל ה-cache, נוצרת cache entry. ה-entry כוללת את הנתונים שהועתקו מהזיכרון, לצד מיקום הזיכרון שממנו הם נלקחו.

כאשר המעבד מבצע פעולה של קריאה או כתיבה למיקום בזיכרון, הוא תחילה בודק אם מיקום הזיכרון הזה נמצא כבר ב-cache. ה-cache בודק את כל ה-cache lines שעשויות להכיל את הכתובת המבוקשת. אם הכתובת קיימת ב-cache, כלומר אם הנתונים נמצאים שם, נוצר מצב שנקרא cache hit. במצב כזה, המעבד ניגש ישירות לנתונים שב-cache ומבצע את הקריאה או הכתיבה במהירות רבה.

לעומת זאת, אם מיקום הזיכרון המבוקש לא נמצא ב-cache, מתרחש מצב שלcache miss. במקרה כזה, ה-cache מקצה cache entry חדשה ומעתיק את הנתונים הדרושים מהזיכרון הראשי. לאחר מכן, הבקשה של המעבד מתמלאת מתוך ה-cache עצמו, כך שהגישה לנתונים לאחר מכן תהיה מהירה יותר, במקרה שיהיה צורך לגשת אליהם שוב.

Cache Miss

ב-cache miss מתרחש ניסיון שנכשל לקרוא או לכתוב פיסת נתונים מה-cache, מה שמוביל לגישה לזיכרון הראשי, שמאופיין ב-latency גבוה בהרבה. ישנם שלושה סוגים עיקריים של cache miss:

1. **פספוס קריאת הוראות:** כאשר מדובר בהחמצת קריאת הוראות מה-cache, מדובר בדרך כלל בעיכוב משמעותי. הסיבה לכך היא שהמעבד, או ה-thread המבצע, צריך להמתין (stall) עד שההוראה תיטען מהזיכרון הראשי. זה יכול להוביל לעיכובים משמעותיים בתפקוד המעבד, שכן כל הוראה שאינה זמינה ב-cache צריכה להביא את עצמה מהזיכרון הראשי.

2. **פספוס קריאת נתונים:** כאשר מתרחש cache miss בקריאת נתונים מה-cache, העיכוב הוא לרוב קטן יותר. במקרה זה, ניתן להוציא הוראות שאינן תלויות בנתונים החסרים ולהמשיך בביצוע עד שהנתונים יגיעו מהזיכרון הראשי. כך, ההוראות התלויות בנתונים החסרים יכולות להמתין לטעינת הנתונים ולהתחדש עם הגעתם.

3. **פספוס כתיבת נתונים:** פספוסי כתיבת נתונים ב-cache גורמים לעיכוב הקצר ביותר מבין השלושה. הסיבה לכך, היא שניתן להכניס את הכתיבה ל-store buffer, ואין בדרך כלל מגבלות רבות על ביצוע ההוראות הבאות. המעבד יכול להמשיך לעבוד עד שהתור יתמלא, ומאוחר יותר תתבצע הכתיבה לנתונים בזיכרון הראשי.

ביצועי Cache

ביצועי ה-cache הם גורם חשוב מאוד, משום שהפער בין מהירות הזיכרון למהירות המעבד גדל באופן מעריכי. שני גורמים מרכזיים המשפיעים על ביצועי ה-cache הם שיעור ה-cache hits ושיעור ה- cache misses. הפחתת ה-cache misses היא קריטית לשיפור הביצועים הכוללים, שכן כל miss עלול להאריך את זמן הגישה לנתונים ולהשפיע על ביצועי התוכנית.

CPU Stalls

כאשר נדרש זמן ארוך להביא cache line אחת מהזיכרון, המעבד עשוי להיתקל במצב שבו הוא מחכה לזמן ארוך כדי לקבל את הנתונים. מצב זה נקרא stall. כאשר מעבדים הופכים למהירים יותר, הפער בין מהירות המעבד לזיכרון הראשי גדל, מה שמוביל ליותר זמן המתנה במהלך cache miss. מעבדים מודרניים יכולים לבצע מאות הוראות בזמן שלוקח להביא cache line אחת מהזיכרון הראשי.

כדי להתמודד עם בעיות אלו ולהפחית את הזמן האבוד בזמן עיכוב, נעשו פיתוחים שונים. אחד הפתרונות הוא ביצוע out of order.

Multi-Level Caches

ביצועי ה-cache הם פשרה בין זמן השהיית ה-cache וקצב ה-hit. ל-cache גדול יותר יש נטייה לספק שיעורי hits טובים יותר, אך דורש יותר זמן גישה, בעוד ש-cache קטן יותר הוא מהיר יותר, אך פחות יעיל בשיעור hits. כדי לנהל את הפשרה הזו, מחשבים רבים עושים שימוש במבנה של מספר רמות של cache.

בדרך כלל, ה-cache-ים מחולקים למספר רמות כשהרמה הראשונה (L1) הקטנה והמהירה ביותר, שנמצאת הכי קרוב למעבד. אחריה יש רמות אחרות גדולות יותר ואיטיות יותר. הרבה פעמים הרמה האחרונה של ה-cache משותפת למספר ליבות, ומוכרת לרוב בשם LLC.

ריבוי סוגי cache-ים

במחשבים מודרניים יש לפחות שני סוגים עצמאיים של cache-ים:

- **Data Cache:** משמש להאצה של פעולות קריאה וכתיבה של נתונים. בניגוד ל-instruction-cache, הממוקד בהוראות, ה-data-cache מיועד לנתונים שהמעבד זקוק להם לצורך ביצוע חישובים. לרוב, data-cache בנוי כהיררכיה מרובת רמות.

- **Instruction Cache:** המעבד לא מאחסן רק נתונים ב-cache, אלא גם את ההוראות שאותן הוא מבצע. למרות זאת, ה-instruction cache נחשב לפחות בעייתי בהשוואה ל-data cache (זיכרון ה-cache לנתונים).

הסיבה לכך היא שזרימת ההוראות בתוכנית צפויה יותר מדפוסי הגישה לנתונים. המעבדים המודרניים מצוידים ביכולות מתקדמות לזיהוי דפוסים, מה שמאפשר להם לבצע prefetching בצורה יעילה יותר. בנוסף, לקוד יש מקומיות מרחבית וזמנית חזקה יחסית. כלומר, סביר להניח שההוראות הנמצאות בסמוך זו לזו בזיכרון יבוצעו ברצף, ושתהיה חזרה על אותן הוראות בתוך פרקי זמן קצרים.

עיכוב בטעינת ההוראה הבאה מהזיכרון פוגע משמעותית במהירות התהליך ה-pipeline של ביצוע ההוראות.

במעבדי CISC, במיוחד, שלב הפענוח יכול לקחת זמן רב יותר עקב מורכבות ההוראות.

בשנים האחרונות, מעבדי x86 ו-x86-64 מתקדמים, כמו אלה של אינטל, אינם מאחסנים ב-L1i את רצף הבייטים הגולמי של ההוראות. במקום זאת, הם שומרים את ההוראות לאחר שהן כבר מפוענחות (ב-intel ההוראות המפוענחות נשמרות במקום הנקרא µop cache או Decoded Stream Buffer). זה מאפשר להאיץ את ביצוע ההוראות ולמנוע עיכובים הנובעים מצורך לפענח אותן בכל פעם מחדש.

החל מ-L2 ומעלה, זיכרון ה-cache במעבדים עשוי להיות מאוחד, כלומר, הוא מכיל גם קוד וגם נתונים. עם זאת, בניגוד ל-L1i, כאן ההוראות מאוחסנות בתצורתן הגולמית כרצפי בתים שטרם פוענחו.

ה-Instruction Cache אינו בהכרח קוהרנטי עם זיכרון הנתונים: למשל, בארכיטקטורת Arm מערכת מחשבים אינה מחייבת את החומרה לשמור על קוהרנטיות בין cache-ים של ההוראות לבין זיכרון הנתונים. במקום זאת, על המתכנתים להשתמש בהוראות תחזוקה מפורשות של ה-cache כדי להבטיח סנכרון והתעדכנות נכונה.

במקרים רבים, פרוטוקול הקוהרנטיות של ה-Instruction Cache פשוט הרבה יותר בהשוואה לפרוטוקולי קוהרנטיות כמו MOESI. ה-Instruction Cache, משתמש בביט פשוט של valid/invalid כדי לנהל את הבלוקים או ה-lines שבו. ההבדל בין פרוטוקולים אלו נובע מהמידע שכל אחד מהם שומר והכמות של

השינויים שהמידע עובר. בדרך כלל, כמעט ואין צורך לשנות את ההוראות שב-cache בזמן הריצה, כאשר חריגות נראות לעיתים רחוקות, כמו במצבים של JIT או ב-context switch.

Translation Lookaside Buffer (TLB)

ה-TLB שנועד להאיץ את תהליך תרגום הכתובות הווירטואליות לכתובות פיזיות, המשמשות את המעבד לגישה להוראות ולנתונים. ה-TLB יכול להיות משותף עבור שני סוגי הגישות (הוראות ונתונים), או, להפריד בין TLB להוראות ו-TLB לנתונים. למרות שהוא חלק חשוב בהאצת גישה לזיכרון, ה-TLB שייך ליחידת ניהול הזיכרון (MMU) ואינו נחשב כחלק ישיר ממערכת ה-cache של המעבד עצמו.

מדיניות

מדיניות החלפה (Replacement)

כאשר מתרחש cache miss ויש צורך להכניס ערך חדש ל-cache, לעיתים יש צורך לפנות מקום על ידי הסרת אחד מהערכים הקיימים. התהליך שבו ה-cache בוחר איזה ערך להסיר נקרא מדיניות ההחלפה. האתגר המרכזי במדיניות זו הוא לחזות איזה ערך שנמצא כרגע ב-cache יש סיכוי נמוך יותר לשימוש בעתיד. כיוון שתחזיות אלו מורכבות ולעיתים בלתי אפשריות, אין שיטה מושלמת לבחירת הערך שיש להחליף. עם זאת, ישנן מספר שיטות הנמצאות בשימוש נרחב, כאשר אחת פופולריות היא מדיניות ה-LRU. מדיניות זו מסירה את הערך שלא נעשה בו שימוש לאחרונה מתוך הנחה שסביר להניח שהוא יהיה הפחות נחוץ בקרוב.

במקרים מסוימים, ניתן לשפר ביצועים על ידי סימון טווחי זיכרון מסוימים כ-non-cacheable. גישה זו מונעת את הכנסת נתונים אל ה-cache כאשר לא סביר שיהיה בהם שימוש חוזר בעתיד הקרוב, וכך נמנעת התקורה של אחסון מידע מיותר ב-cache.

בנוסף, ניתן גם לנקוט בגישות של ניהול חכם יותר של ה-cache, כמו השבתה או נעילה של ערכים מסוימים בהתאם להקשר התוכנית, כדי לשלוט על הנתונים המאוחסנים ולייעל את הגישה אליהם.

כאשר נתונים נכתבים ל-cache, בסופו של דבר יש צורך לכתוב אותם גם לזיכרון הראשי. התזמון שבו הכתיבה מתבצעת נקרא מדיניות הכתיבה.

מדיניות כתיבת Cache

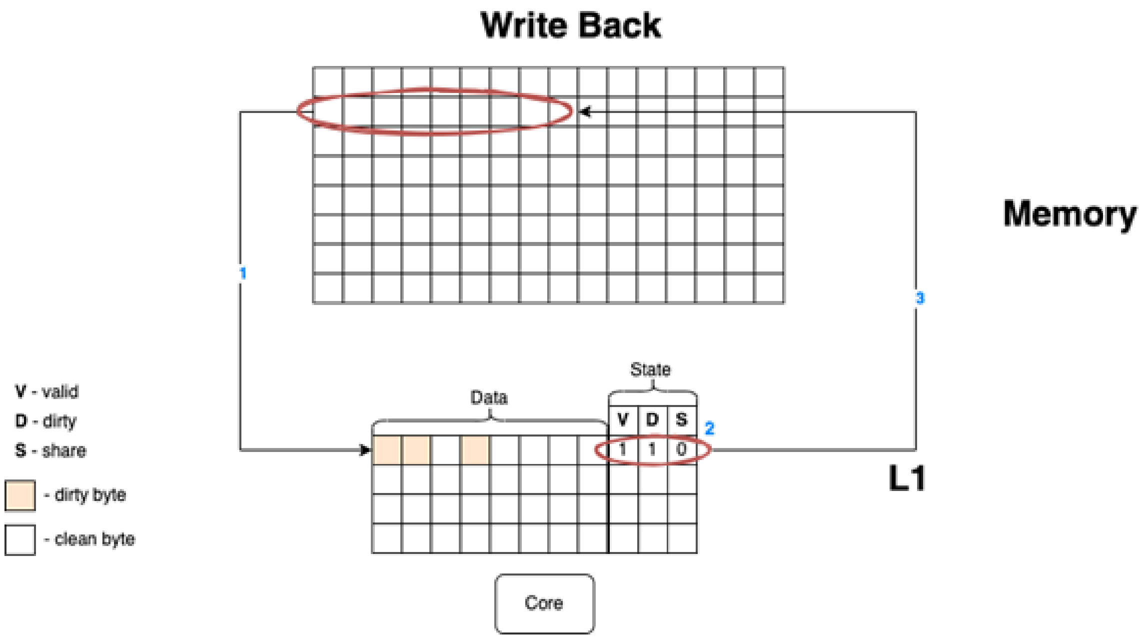
Write-Through

מדיניות ה-write-through היא השיטה הפשוטה ביותר להשגת קוהרנטיות של ה-cache. כאשר מעבד כותב ל-cache line, הנתונים נכתבים באופן מידי גם לזיכרון הראשי. כך מובטח שכל שינוי שמתבצע בזיכרון ה-cache משתקף מיד בזיכרון הראשי, מה שמבטיח שהמידע בשני המקומות נשאר מסונכרן.

יתרה מזאת, כאשר יש צורך להחליף cache line בנתונים חדשים, ניתן פשוט להשליך את התוכן הישן מה-cache מבלי לחשוש לאובדן נתונים, שכן הם כבר נכתבו לזיכרון הראשי. אמנם מדיניות זו היא קלה ליישום ומספקת קוהרנטיות טובה, אך היא אינה מהירה במיוחד, מכיוון שכל כתיבה ל-cache גוררת כתיבה מיידית לזיכרון הראשי, מה שעלול להאט את ביצועי המערכת.

Write-Back

במדיניות ה-write-back, המעבד אינו כותב את ה-cache line ששונתה בחזרה לזיכרון הראשי באופן מידי. במקום זאת, ה-cache line מסומנת כ-Dirty, מה שמעיד שהיא עברה שינוי, אך הנתונים עדיין לא נכתבו לזיכרון הראשי. רק כאשר ה-cache line נדחקת החוצה מה-cache בנקודת זמן כלשהי בעתיד, הביט Dirty מורה למעבד לכתוב את הנתונים המעודכנים לזיכרון הראשי. שיטה זו מאפשרת להימנע מכתיבה תכופה לזיכרון הראשי, מה שמסייע לשיפור ביצועי המערכת, במיוחד במצבים שבהם נעשות הרבה פעולות כתיבה ל-cache:



עם זאת, קיימת בעיה משמעותית ביישום של write-back במערכות מרובות מעבדים (או ליבות, או hyper-threading). כאשר מספר מעבדים ניגשים לאותו אזור זיכרון, יש להבטיח שכל המעבדים רואים את אותם נתונים בכל זמן נתון. אם ה-cache line מלוכלכת באחד המעבדים ולא נכתבה עדיין לזיכרון הראשי, ומעבד אחר מנסה לקרוא את אותו מיקום בזיכרון, הקריאה לא יכולה להתבצע מהזיכרון הראשי, שכן הוא אינו מכיל את הנתונים המעודכנים. במקרה זה, נדרש שהמעבד השני יקבל את תוכן ה- cache line ישירות מהמעבד הראשון, כדי להבטיח קוהרנטיות של הנתונים.

Write-Combining

מדיניות ה-write-combining (שילוב כתיבה) היא סוג של אופטימיזציה ב-cache, שנמצאת בשימוש בעיקר עבור זיכרון RAM של מכשירים חיצוניים, כמו כרטיסים גרפיים. מאחר שעלות העברת נתונים לזיכרון של מכשירים חיצוניים גבוהה בהרבה בהשוואה לגישה לזיכרון RAM המקומי, יש חשיבות גבוהה להימנע מהעברות רבות ומיותרות. לדוגמה, העברת cache line שלמה רק בגלל שינוי של מילה אחת בזיכרון היא פעולה בזבזנית, במיוחד אם הפעולה הבאה תחליף מילה אחרת באותו cache line.

מדיניות write-combining נועדה לפתור בעיה זו על ידי שילוב של כמה פעולות כתיבה לפני כתיבת כל ה-cache line לזיכרון החיצוני. בתרחיש האידיאלי, כל ה-cache line משתנה בהדרגה, מילה אחרי מילה, ורק כאשר כל המילים עברו שינוי, כל ה-cache line נכתבת לזיכרון המכשיר החיצוני. זה מאפשר להפחית את מספר ההעברות ולהגביר את היעילות.

Uncacheable

כאשר אזור זיכרון מוגדר כ-uncacheable, המשמעות היא שבדרך כלל אין לו תמיכה מגובה ב-RAM, כלומר, אין אפשרות לאחסן אותו ב-cache. במקרים מסוימים, זה יכול להיות אזור זיכרון מיוחד שמוקצה לכתובת קשיחה, המשמש למטרות מסוימות מחוץ לפעילות הרגילה של המעבד, כמו גישה למכשירים חיצוניים או פקודות ספציפיות למערכת.

מדיניות ה-write-combining וה-uncacheable מיועדות לשימוש באזורים מיוחדים במרחב הכתובות של המערכת, שאינם מגובים בזיכרון RAM אמיתי. אזורים אלה משמשים למטרות ייחודיות הקשורות לגישה למכשירים או פונקציות חומרה ייחודיות, ולכן אינם מנוהלים באותו אופן כמו זיכרון רגיל.

תמיכה בריבוי מעבדים

במערכות מרובות מעבדים, אין זה מעשי לאפשר גישה ישירה של מעבד אחד ל-cache של מעבד אחר. אחת הסיבות המרכזיות לכך היא שהחיבור בין המעבדים אינו מהיר מספיק כדי לאפשר גישה כזו בזמן אמת. במקום זאת, הפתרון היעיל יותר הוא להעביר את תוכן ה-cache למעבד השני בעת הצורך. זה נכון גם במקרים שבהם ה-cache-ים שייכים לאותו מעבד אך אינם משותפים.

העברת cache line מתבצעת כאשר מעבד אחד זקוק ל-cache line מלוכלכת שנמצאת ב-cache של מעבד אחר, לצורך קריאה או כתיבה. עם זאת, מרבית הגישות לזיכרון הן קריאות, וה-cache lines המתקבלות הן נקיות, כלומר לא שונו ב-cache.

לכן, אין זה מעשי לשדר את המידע על כל שינוי ב-cache lines אחרי כל גישת כתיבה. כדי להתמודד עם הבעיה הזו, פותחו פרוטוקולים לניהול קוהרנטיות ה-cache, כגון MOESI ו-AMBA CHI (שנדון בו בהרחבה

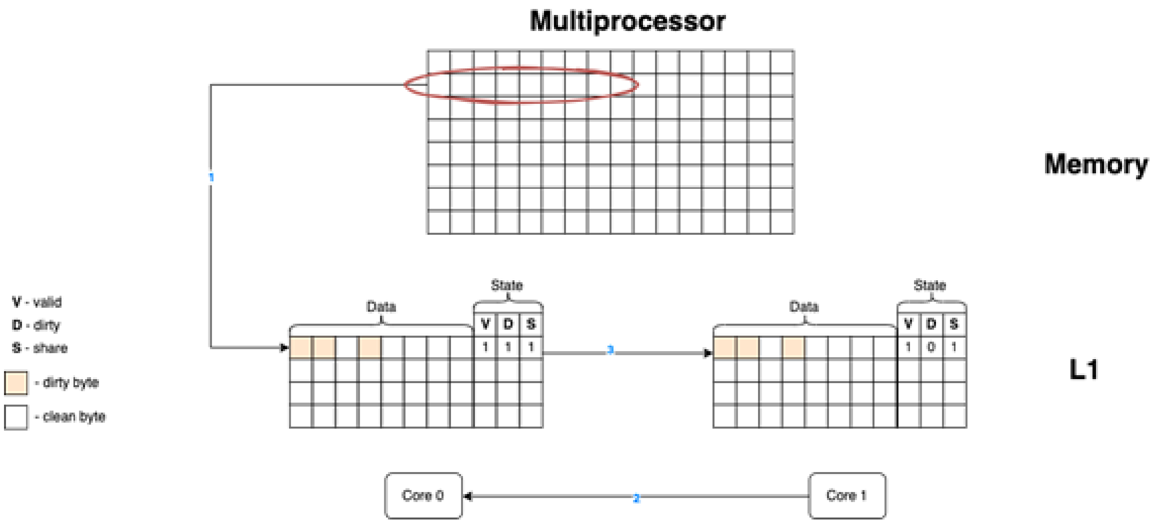
בהמשך), לצד פרוטוקולים נוספים. פרוטוקולים אלה נועדו להבטיח שכל המעבדים במערכת יראו את אותו מצב של הזיכרון, גם כאשר מספר מעבדים ניגשים לאותם נתונים בזיכרון.

במערכות מרובות מעבדים סימטריות (SMP), לא ניתן לאפשר ל-cache-ים של המעבדים לפעול באופן עצמאי לחלוטין. כל המעבדים במערכת חייבים לשמור על תמונה אחידה של תוכן הזיכרון, כך שכולם יראו את אותו המידע בכל רגע נתון (הדבר נכון רק כאשר מדובר בפלטפורמות multicopy-atomic , נדון ברעיון זה בהמשך), שמירה על אחידות זו נקראת קוהרנטיות cache.

אם מעבד היה בודק רק את ה-cache שלו ואת הזיכרון הראשי שלו, הוא לא היה מודע ל-cache lines מלוכלכות ששונו על ידי מעבדים אחרים. מתן גישה ישירה של מעבד אחד ל-cache של מעבד אחר היה גורם לעלויות גבוהות מאוד וליצירת צוואר בקבוק משמעותי בביצועים. במקום זאת, המעבדים מתוכננים לזהות מתי מעבד אחר מבצע קריאה או כתיבה ל-cache line מסוימת, וכך לשמור על הסנכרון והקוהרנטיות של הזיכרון בין כל המעבדים.

כאשר מעבד מזהה ניסיון כתיבה לכתובת מסוימת ויש לו עותק נקי של ה-cache line המתאימה ב-cache שלו, העותק הזה מסומן כ-Invalid. משמעות הדבר היא שגישות עתידיות לאותה cache line ידרשו לטעון אותה מחדש מהזיכרון הראשי. עם זאת, חשוב לציין שקריאה ממעבד אחר אינה מחייבת Invalidate, וניתן לשמור מספר עותקים נקיים של אותה cache line בכמה מעבדים בו זמנית.

במקרים מתקדמים יותר של ניהול cache, ייתכן מצב נוסף. אם ה-cache line שהמעבד השני רוצה לקרוא או לכתוב אליה מסומנת כמלוכלכת ב-cache של המעבד הראשון (כלומר, היא שונתה אך השינויים לא נכתבו לזיכרון הראשי), יש צורך בגישה שונה. במצב כזה, מכיוון שהזיכרון הראשי אינו מעודכן, המעבד המבקש חייב לקבל את תוכן ה-cache line ישירות מהמעבד הראשון (המצב המתואר הוא SD בפרוטוקול AMBA CHI).



תהליך זה מתבצע באמצעות טכניקה שנקראת snooping, שבה המעבד שמבקש לבצע כתיבה או קריאה שולח בקשה לנתונים מהמעבד הראשון, כך שהמידע עובר ישירות ביניהם, מבלי לעבור דרך הזיכרון הראשי. לעיתים, בקר הזיכרון מזהה את ההעברה הישירה הזו ומעדכן את הזיכרון הראשי עם התוכן המעודכן של ה-cache line.

אם הגישה מיועדת לכתיבה, המעבד הראשון יסמן את ה-cache line המקומית שלו כ-invalid כדי להבטיח שאין עותקים ישנים ולא מעודכנים בזיכרון ה-cache שלו. למרות זאת, עותקים נקיים של אותה cache line יכולים להישמר באופן שרירותי במספר caches של מעבדים שונים במערכת.

Cache Controller

ה-cache controller הוא רכיב חומרה שתפקידו לנהל את זיכרון ה-cache במערכת, מבלי שהניהול יהיה גלוי לתוכנית. ה-cache controller אחראי לכתיבה אוטומטית של הוראות ונתונים מהזיכרון הראשי לתוך ה-cache. הוא מקבל בקשות קריאה וכתיבה מהליבה ומבצע את הפעולות הנדרשות על זיכרון ה-cache או על הזיכרון החיצוני.

כאשר ה-cache controller מקבל בקשה מהליבה, עליו לבדוק אם הכתובת המבוקשת כבר קיימת ב-cache. שלב זה נקרא חיפוש ב-cache.

אם הליבה מבקשת הוראות או נתונים מכתובת מסוימת, אך ה-cache controller לא מוצא את התוכן המבוקש ב-cache, מדובר במצב של cache miss. במקרה כזה, הבקשה מועברת לרמה הבאה בהיררכיית הזיכרון, אשר יכולה להיות cache ברמה גבוהה יותר (כגון L2) או זיכרון חיצוני. התוצאה של מצב זה יכולה להיות מילוי line ב-cache, שבו תוכן מהזיכרון הראשי מועתק ל-cache. במקביל, הנתונים או ההוראות המבוקשים נמסרים לליבה. תהליך זה מתרחש בצורה שקופה, כלומר, אינו גלוי ישירות למשתמש. הליבה אינה צריכה להמתין עד שהמילוי של ה-cache line יושלם לפני השימוש בנתונים.

במקרים רבים, ה-cache controller ייגש קודם כל למילה הקריטית בתוך ה-cache line. לדוגמה, אם הוראת load גורמת ל-cache miss ומפעילה מילוי של cache line, הקריאה הראשונה לזיכרון החיצוני תהיה עבור הכתובת המדויקת שניתנה על ידי הוראת ה-load. הנתונים הקריטיים הללו יסופקו מיד ל-pipeline של המעבד, בעוד שחומרת ה-cache וממשק ה-bus החיצוני ימשיכו לקרוא את שאר ה- cache line ברקע.

בחלק מהמעבדים המודרניים קיימת רכיב הנקרא Line Fill Buffer, אשר נועדה לטפל בתהליך של מילוי ה-cache line.

ניקוי וביטול תוקף של זיכרון Cache

ניקוי וביטול של זיכרון ה-cache נדרשים כאשר תוכן הזיכרון החיצוני משתנה, ויש צורך להסיר נתונים מיושנים מה-cache. פעולה זו עשויה להיות דרושה גם לאחר שינויים במדיניות ה-cache או בהרשאות גישה שמבוצעים על ידי ה-MPU.

המונח "שטיפה" (flush) משמש לעיתים קרובות כדי לתאר את פעולות הניקוי והביטול תוקף, אך בסטנדרטים של ARM ו-x86, השמות הנפוצים הם "clean" ו-"invalidate".

ביצוע של clean על cache או על cache line מעורב בכתיבה של התוכן של cache lines מלוכלכות אל הזיכרון הראשי. פעולה זו מבטיחה שהתוכן שב-cache line והזיכרון הראשי יהיו קוהרנטיים זה עם זה.

ביצוע של invalidation על cache או על cache line גורם להסרת הנתונים מה-cache, על ידי ביטול תוקף של ה-cache line אחת או יותר. בתהליך זה, הביט valid של ה-cache line נוקה, מה שמוביל לכך שהנתונים שב-cache אינם מוגדרים עוד. אם ה-cache מכיל נתונים מלוכלכים, לא נכון לבטל אותו מבלי לנקות את הנתונים. פעולה זו תוביל לאובדן כל הנתונים המעודכנים ב-cache ובאזורי האחסון שלו.

במכשירי ARM, פעולות ה-invalidate וה-clean של ה-cache יכולות להתבצע על פי set/way או על ידי ציון כתובת מסוימת.

חסרונות ה-Cache

למרות היתרונות הברורים של זיכרון ה-cache, שמאיץ את ביצוע התוכניות, הוא גם מוסיף מספר אתגרים ייחודיים. אחד החסרונות המרכזיים הוא האפשרות לכך שזמן הביצוע של התוכנית יהפוך ללא דטרמיניסטי, כלומר, לא ניתן יהיה לחזות באופן מדויק את משך הזמן שיידרש לביצועה בכל פעם.

בנוסף, ייתכן שתוכן ה-cache והזיכרון הראשי לא יהיו מסונכרנים באופן מושלם. במצבים מסוימים, המעבד עשוי לעדכן את ה-cache, אך השינויים טרם ייכתבו חזרה לזיכרון הראשי. מצד שני, רכיב חיצוני כמו בקר DMA יכול לעדכן את הזיכרון הראשי לאחר שהליבה כבר טענה עותק משלה. בעיות אלו נוגעות לניהול קוהרנטיות הזיכרון, והן עשויות להתעורר במיוחד במערכות עם מספר ליבות או רכיבים חיצוניים שניגשים לזיכרון, כמו בקרים חיצוניים.

בעיות ידועות עם Cache

False Sharing

המושג False sharing מתאר בעיה ביצועים במערכות עם ריבוי cache-ים, כאשר מספר מעבדים משתמשים באותו משתנה (הכוונה היא לאותו מיקום זיכרון). הבעיה מתעוררת כאשר משתנים המאוחסנים ב-cache נמצאים באותה cache line. אם אחד המעבדים משנה רק חלק מהנתונים ב- cache

line, אך החלקים האחרים אינם משתנים, כל ה-cache line צריכה לצאת מה-cache ולהגיע לזיכרון הראשי ולאחר מכן להיטען מחדש, משום שמבנה ה-cache עובד לפי שורות שלמות ולא לפי חלקים בתוכן. בעיה זו גורמת לכך שכל עדכון של חלק מסוים ב-cache line יוביל לעדכון מחדש של כל ה- cache line, גם אם רק חלק קטן מהמידע השתנה. זה מוביל לבעיות ביצועים משום שהמעבד צריך להפעיל מנגנוני סנכרון כדי לנהל את העדכונים.

כאשר ה-cache lines מחולקות בין משתנים גלובליים, אם אחד המשתנים משתנה על ידי אחד מהמעבדים, ה-cache line המתאימה מסומנת כמלוכלכת. ב-cache line הנותרת של המעבד, זה עלול לגרום ל-entry מעופשת (stale) שדורשת שטיפה (flush) והחזרה מהזיכרון, דבר שיכול להוביל ל-miss של cache line ולדרוש יותר מחזורי CPU. זה עשוי להפחית את ביצועי המערכת. הרבה מבני הנתונים בקרנל של מערכת ההפעלה משתמשים באסטרטגיות כדי להימנע מ-miss של ה-cache line.

פתרון אפשרי לבעיית ה-false sharing זו, הוא להפריד בין נתונים שצריכים להתעדכן ביחד ונתונים שלא דורשים עדכון משותף. ניתן לעשות זאת על ידי יצירת משתנים אשר נמצאים באותה שורת cache (משתנים שמשתנים או קבועים יחד), או על ידי שימוש ב-padding לצורך יישור נכון של הנתונים. במקרים של structs חשובים וקריטיים, במיוחד בשימוש נרחב, יש לקחת בחשבון בעיות אלו ולתכנן את ה-struct כך שיהיה מיושר (aligned) לגודל ה-cache line.

בעיית false sharing עלולה לגרום לירידה בביצועים, ואם מטפלים בבעיה כראוי יכול להיות שיפור ביצועים משמעותי. הבעיה יכולה להתרחש גם בפרוטוקולי cache אוטומטיים וגם בסביבות נוספות כמו מערכות קבצים מבוזרות או מסדי נתונים, אך היא שכיחה בעיקר ב-cache RAM.

ייעול גישה ל-Cache:

- כאשר יוצרים נתונים גדולים שלא נעשה בהם שימוש מידי, יש לשקול דרכים למנוע את הכנסתם המוקדמת ל-cache. הכנסת נתונים שאינם בשימוש גורמת לעומס מיותר ולתקורה, כיוון שמידע עובר דרך ה-cache ללא צורך, מה שעלול לדחוק מידע חשוב יותר מרמות נמוכות כמו L1 ל-L2 או אף החוצה מה-cache. מצב כזה יוצר חוסר יעילות וזמן עיבוד מבוזבז.

- כאשר יש נתונים רציפים שניתן לאחד ביניהם בעת כתיבה, מומלץ לאחד אותם לשורת cache אחת. איחוד שכזה מפחית את מספר הקריאות והכתיבות ל-cache ומשפר את ביצועי התוכנית על ידי הפחתת התקשורת המיותרת עם הזיכרון הראשי.

- שדות הנגישים לעיתים קרובות ב-struct ממוקמים בתחילתו, הדבר נעשה כדי להגדיל את הסיכוי שהשדות החשובים ימוקמו כולם על אותו cache line, כך שהמעבד יוכל לגשת אליהם בבת אחת.

- הפרדה בין שדות לא קשורים ב-struct כך שאין צורך לעבד אותם יחד, יש להקפיד להפריד אותם בזיכרון כך שהם יהיו על cache lines נפרדים. מטרת ההפרדה היא למנוע תופעה של false sharing.

- כאשר מדובר במבני נתונים קטנים יחסית, יש סיכוי טוב שניתן להכניס את כל המבנה ל-cache line אחת. כדי לממש זאת, חשוב לוודא שאין שדות מיותרים במבנה ושכל שדה מוקצה בגודל מינימלי, ללא בזבוז מקום או padding מיותר בין השדות. פעולה זו מצמצמת את גודל המבנה ומאפשרת לו להיכנס כולו ל-cache line אחד, ובכך מייעלת את הגישה אליו.

- כאשר סדר הגישה למבנה אינו מוגדר על ידי לוגיקה חיצונית, עדיף לגשת לאלמנטים בסדר שבו הם מופיעים במבנה הנתונים. עבור מבנים קטנים, חשוב שהמתכנת יחשוב מראש על סדר האלמנטים ויגדיר אותם בצורה כזו שתתאים לסדר הגישה הסביר ביותר בתוכנית. בנוסף, יש להיות גמישים ולאפשר שינויים כדי להתאים לאופטימיזציות אחרות, כגון מניעת רווחים מיותרים בזיכרון. במבנים גדולים יותר, כל בלוק של מידע צריך להיות מאורגן כך שיתאים לגודל של cache line ולהבטיח גישה יעילה בהתאם לחוקים הללו.

למשל בשפת C עם GCC:

```
struct strtype {
    ...members...
} __attribute((aligned(64)));
```

זה יגרום להקצאת אובייקטים עם יישור מתאים (יישור ל-64 בתים), וזה חיוני לביצועים מיטביים של cache ולמניעת בעיות גישה. הקומפיילר אחראי להקצות אובייקטים עם היישור הנדרש, כולל מערכים. עם זאת, כאשר מדובר בהקצאה דינמית של אובייקטים, על המתכנת להקפיד לבקש את היישור המתאים בעצמו.

אתגרי יישור ב-Stack

כאשר משתנה אוטומטי מוקצה על המחסנית עם דרישת יישור מסוימת, על הקומפיילר להבטיח שהדרישה מתקיימת. עם זאת, לא תמיד מדובר במשימה פשוטה. הסיבה לכך היא שהקומפיילר אינו שולט בכל הקריאות הפנימיות המבוצעות (כגון דחיפות למחסנית) או על האופן שבו הקריאות הללו מטפלות במחסנית. כדי להתמודד עם האתגר הזה, ישנן שתי דרכים עיקריות:

1. הקומפיילר יכול לייצר קוד שמוודא שהמחסנית מיושרת בצורה נכונה בכל שלב. פעולה זו כוללת הכנסת padding במידת הצורך כדי ליישר את הנתונים, וכן קוד נוסף שנדרש גם לבטל את היישור כאשר אין בו עוד צורך. שיטה זו מחייבת בדיקות חוזרות על היישור במהלך זמן הריצה, מה שעלול להוסיף מעט תקורה לביצועים.

2. שימוש במחסנית מיושרת אצל כל ה-caller-ים, כדי לבצע את זה יש לוודא שכל הקריאות שמבוצעות תומכות במחסנית מיושרת מראש. כלומר, כל פונקציה דואגת לכך שהמחסנית תישאר מיושרת כאשר היא קוראת לפונקציה אחרת. גישה זו היא הסטנדרט בכל ה-ABI הנפוצים, והיא נפוצה כי היא דורשת פחות קוד מורכב לזמן ריצה.

ניצול מיטבי של Cache

במערכות שבהן ה-working set גדול, חשוב לנצל את ה-cache באופן מיטבי. לשם כך, ייתכן שיהיה צורך לארגן מחדש את מבני הנתונים ולא לסדר אותם לפי השייכות הרעיונית בלבד. לעיתים קרובות, המתכנת יעדיף לרכז את כל הנתונים הקשורים רעיונית לאותו מבנה נתונים, אך גישה זו אינה בהכרח המיטבית לביצועים. ארגון שגוי יכול להוביל לכך שהגישה לנתונים לא תהיה מיושרת באופן יעיל ל-cache.

השפעות של גישה לא מיושרת

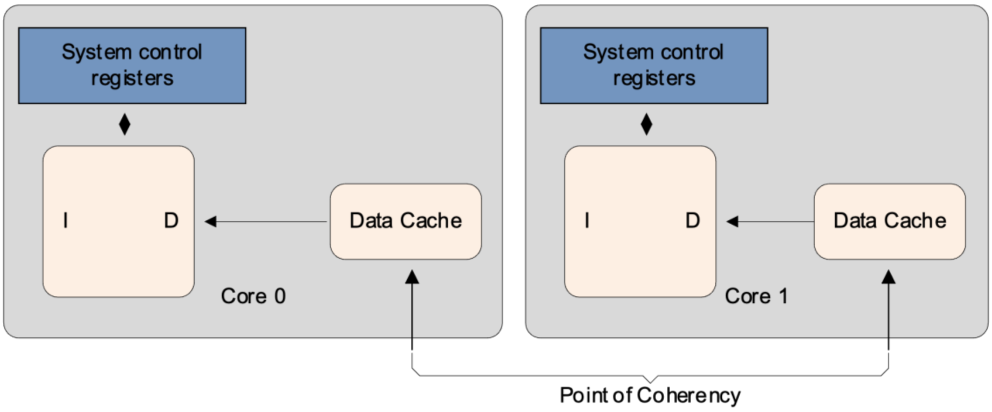
גישה לא מיושרת לנתונים מגדילה את הסיכון ל-miss בזיכרון ה-cache הנגרם מקונפליקטים, מכיוון שכל גישה לנתונים עלולה לדרוש יותר משורת cache אחת. כתוצאה מכך, חלקים שונים מהנתונים עשויים להימצא במיקומים שונים בזיכרון ה-cache, מה שמוביל לכך שקריאות עוקבות לנתונים אלו יגרמו לדחיקות ולבזבוז משאבים ב-cache.

נקודת קוהרנטיות ונקודת איחוד ב-ARM

במערכת לניהול cache, פעולות ניקוי וביטול יכולות להתבצע ברמות שונות של cache. פעולות אלו מוגדרות בהתאם לסוג הקונפיגורציה של cache - לפי set/way. כאשר מדובר בפעולות המשתמשות בכתובת וירטואלית, הארכיטקטורה מגדירה שתי נקודות עיקריות.

נקודת קוהרנטיות (Point of Coherency, PoC):

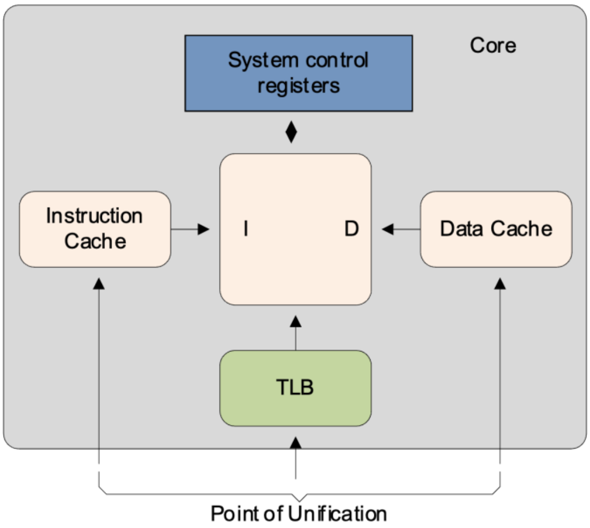
עבור כתובת מסוימת, נקודת הקוהרנטיות היא המקום שבו מובטח לכל המעבדים במערכת (ליבות המעבד, DMA, DSPs וכו') - שיגיעו לאותו עותק של מיקום זיכרון. לרוב, נקודת הקוהרנטיות היא זיכרון המערכת החיצוני הראשי. (בהמשך יש הסבר מורחב יותר על נקודת הקוהרנטיות):



[Figure 11 11 Point of Coherency from Point of coherency and unification (developer.arm.com)]

נקודת איחוד (Point of Unification, PoU):

נקודת האיחוד עבור ליבה היא המקום שבו מובטח שה-cache-ים של ההוראות, הנתונים וה- translation table walks יראו את אותם העותקים. לדוגמה, אם cache רמה 2 פועל כ-cache מאוחד במערכת שבה יש cache רמה 1 ו-TLB לשמירה ב-cache של ערכי טבלת תרגום, אז ה-cache רמה 2 משמש כנקודת האיחוד. במקרה שאין cache חיצוני, אז הזיכרון הראשי משמש כנקודת האיחוד:



[Figure 11 12 Point of Unification from Point of coherency and unification (developer.arm.com)]

Cache Coherence

כעת, שיש לנו ידע בסיסי על cache אפשר להתחיל להתקדם ולראות דברים קצת יותר מעניינים כמו קוהרנטיות cache.

מבוא לקוהרנטיות

בעיית קוהרנטיות יכולה להתרחש כאשר מספר גורמים (כמו מספר ליבות) מנסים לגשת לנתון מסוים שיש לו מספר עותקים (למשל, עותקים שונים ב-cache-ים) ולפחות אחת הגישות היא כתיבה. כדי למנוע גישה לנתונים מיושנים (שגורמת לחוסר קוהרנטיות) נעשה שימוש בפרוטוקול קוהרנטיות.

חוסר קוהרנטיות נגרם בעיקר בשל נוכחותם של מספר מעבדים עם גישה ל-cache-ים ולזיכרון. במערכות מודרניות, מעבדים אלו כוללים ליבות מעבד, מנועי DMA והתקנים חיצוניים, אשר יכולים לקרוא ולכתוב לזיכרון ולפחות לחלקם יש cache.

קוהרנטיות cache שואפת להפוך את ה-cache-ים במערכת זיכרון משותף לבלתי נראים מבחינה פונקציונלית, כאילו כל ה-cache-ים שייכים למערכת בעלת ליבה אחת בלבד. היא משיגה זאת על ידי הפצת כתיבה ממעבד אחד ל-cache-ים של מעבדים אחרים במערכת.

חשוב לציין, שבניגוד לעקביות זיכרון, שהיא מפרט ארכיטקטוני המגדיר את נכונות הזיכרון המשותף, קוהרנטיות אינה עוסקת בהגדרת המודל של עקביות עצמו אלא משמשת כאמצעי לתמוך במודל עקביות הזיכרון. בפרט, פרוטוקולי קוהרנטיות ממלאים תפקיד חשוב בהבטחת עקביות הזיכרון במערכת.

אפשר להשיג קוהרנטיות על ידי השבתה מוחלטת של ה-cache למיקומי זיכרון משותפים, אך שיטה זו נוטה להקטין את הביצועים באופן משמעותי.

מתי קוהרנטיות רלוונטית?

קוהרנטיות נוגעת לכל מבני האחסון שמחזיקים בלוקים מתוך מרחב הכתובות המשותף. מבנים אלה כוללים את ה-cache הנתונים ברמות השונות, וגם את הזיכרון הראשי. בנוסף, בחלק מהארכיטקטורות זה כולל את ה-instructions-cache ברמה L1 ו-TLBs (לפעמים ה-TLB יכול להחזיק מיפויים שאינם עותקים בלבד של בלוקים מתוך הזיכרון המשותף).

קוהרנטיות אינה נראית ישירות למתכנת. במקום זאת, ה-pipeline של המעבד ופרוטוקול הקוהרנטיות עובדים יחד כדי לאכוף את מודל העקביות, ורק מודל העקביות הוא החשוף למתכנת.

הדרישות המרכזיות לקוהרנטיות Cache:

- **התפשטות כתיבה**: כאשר משתנים נתונים המאוחסנים ב-cache, יש לוודא שהשינויים מתעדכנים גם בעותקים האחרים של אותו בלוק בזיכרונות ה-cache של שאר המעבדים במערכת.

- **סידור טרנזקציות**: כל פעולות הקריאה והכתיבה שמתבצעות לאותו מיקום בזיכרון צריכות להיראות לכל המעבדים באותו סדר. כך מובטחת עקביות בסדר הגישה לנתונים המשותפים בין כל המרכיבים במערכת.

בעוד שתיאורטית ניתן להבטיח קוהרנטיות ברמת כל פעולת קריאה וכתיבה, בפועל קוהרנטיות מיושמת לרוב ברמת cache line, משום שזה יעיל יותר מבחינת ביצועים.

קוהרנטיות cache לבדה אינה מספיקה כדי להבטיח התנהגות זיכרון תקינה במערכות שבהן נתונים משותפים נשמרים ב-cache. היא מהווה רק חלק מצומצם מהאילוצים שמגדיר מודל הזיכרון של המערכת. לדוגמה, קוהרנטיות cache אינה מתייחסת לסדר הפעולות הנובע מיחסי סדר בין פעולות זיכרון שונות במיקומים שונים. יתרה מכך, חלק מהאילוצים שמגדיר מודל הזיכרון חלים גם במערכות שאינן תומכות ב-cache-ים כלל. בנוסף, במערכות רבות, קשה להפריד בין פרוטוקול קוהרנטיות ה-cache לפרוטוקול הכללי שנועד לשמירה על עקביות הזיכרון, אך לרוב, פרוטוקול הקוהרנטיות מהווה חלק חיוני בתוכנית הכוללת לתמיכה במודל הזיכרון של המערכת.

קוהרנטיות cache מתארת עקרון לפיו במערכת מרובת מעבדים, כל המעבדים חייבים לקבל תצוגה עקבית של תוכן הזיכרון המשותף. כלומר, כאשר מעבדים שונים מבצעים פעולות כתיבה לאותו מיקום בזיכרון, חייב להיות סדר גלובלי מוסכם של פעולות הכתיבה הללו (הנקרא סדר הקוהרנטיות), כך שכל מעבד יראה את השינויים באותו סדר. סדר זה חייב להיות עקבי עם סדר הפעולות שהוגדרו בתוכנית עבור אותו מיקום בזיכרון.

חשוב להבדיל בין קוהרנטיות לעקביות, בספרות המקצועית ההבדל ביניהם הוא שקוהרנטיות מתייחסת לכל מיקום זיכרון בנפרד (מיקום זיכרון יחיד), בעוד שעקביות מתייחסת למערכת כולה ולכל מיקומי הזיכרון באופן כללי במערכת.

קוהרנטיות מנוהלת תוכנה

קוהרנטיות מנוהלת תוכנה היא גישה לניהול שיתוף נתונים. ביישום זה, הנתונים מאוחסנים ב-cache, אך ניהול הקוהרנטיות שמתבצע באמצעות תוכנה (בעיקר ספריות low level ודרייברים מבצעים פעולות כאלו) אחראי לנקות נתונים שהם במצב dirty או לבטל עותקים ישנים מה-cache. גישה זו עלולה להוסיף זמן עיבוד, להקשות על פיתוח התוכנה ולהפחית ביצועים כאשר שיעור השיתוף בין הליבות גבוה.

קוהרנטיות מנוהלת חומרה

בגישה מנוהלת חומרה, הקוהרנטיות בין ה-data-cache-ים ברמה 1 בתוך cluster נשמרת על ידי החומרה. כאשר ליבה מופעלת, היא משתתפת אוטומטית במערכת הקוהרנטיות, כל עוד ה-data-cache וה-MMU של הליבה פעילים והכתובת נחשבת לקוהרנטית.

תמיכה בקוהרנטיות מנוהלת חומרה מוסיפה מורכבות לחומרה הקשורה ל-interconnect ול-cluster, אך היא מפשטת מאוד את התוכנה ומאפשרת פיתוח יישומים שלא היו אפשריים רק באמצעות קוהרנטיות.

במערכות עם קוהרנטיות חומרה הקוהרנטיות נשמרת לרוב בפירוט (granularity) שמתאים למספר מילים בזיכרון, אשר בדרך כלל תואם לגודל ה-cache line. פירוט זה מכונה coherence granularity. גודל גדול יותר של coherence granularity מסייע בהפחתת התקורה הקשורה לניהול ביטים של מצבי תג ב-cache. בדרך כלל, עותקים ב-cache נשלפים לפי גודל של cache line שלמה, וכאשר מתבצע ביטול, cache line שלמה מבוטלת גם אם הכתיבה שהביאה לכך נוגעת רק למילה בודדת.

כשה-coherence granularity גדול זה מציב פשרה מבחינת ביצועים: כאשר יש מקומיות מרחבית טובה, ניתן להפיק תועלת מהעברת נתונים בגודל cache line שלמה, אך כאשר יש מקומיות מרחבית גרועה, הדבר עלול להוביל ליותר cache misses בשל תופעת false sharing.

קוהרנטיות בין רכיבים שונים

במקרים מסוימים, שינוי הנתונים יכול להתרחש בצורה שהפרוטוקול קוהרנטיות לא יכול לטפל בו עקב השפעות של מנגנוני ייעול של המעבד, למשל כמו prefetching של הוראות וקיום רכיבים שלא משתתפים בפרוטוקול קוהרנטיות. במצבים כאלה, התוכנה חייבת להשתמש בהוראות המיועדות להסדרת ה-cache או הוראות invalidation כדי להבטיח שהגישה לנתונים תהיה קוהרנטית לאחר מכן.

קוהרנטיות הוראות

בארכיטקטורות כמו ARM וארכיטקטורות אחרות, שליפת פקודות אינן בהכרח קוהרנטיות.

קוהרנטיות הוראות להוראה

fetch instruction→fetch instruction - שליפה אחת של פקודות עשויה להיות לא עקבית עם שליפה קודמת לפי סדר התוכנית.

קוהרנטיות הוראה לנתונים

fetch instruction→fetch data - כאשר מתבצע fetch ואז קריאה מאותו מיקום בזיכרון, הקריאה אינה מחויבת לראות את הכתיבה לאותו מיקום באופן קוהרנטי.

קוהרנטיות נתונים להוראה

fetch instruction→fetch data - כאשר מתבצעת קריאה ואחריה fetch מאותו מיקום בזיכרון, הקריאה אינה מחויבת לראות את הכתיבה לאותו מיקום באופן קוהרנטי. במילים אחרות, ייתכן שה-fetch של הוראה לא יראה את השינויים שנעשו בכתיבה הקודמת.

גישה קוהרנטית עם DMA

הטמעה של DMA בצורה קוהרנטית יכולה להתבצע פשוט על ידי הוספת cache קוהרנטי לבקר ה-DMA, מה שמאפשר לו להשתתף בפרוטוקול הקוהרנטיות של המערכת. במודל כזה, בקר ה-DMA יתנהג כמו ליבה ייעודית מבחינת הקוהרנטיות, מה שמבטיח שקריאות DMA תמיד יגיבו לגרסה העדכנית ביותר של בלוק הנתונים, ושהכתיבות של DMA יבצעו ביטול של כל העותקים הישנים של הבלוק.

עם זאת, הוספת cache קוהרנטי לבקר DMA אינה תמיד אופטימלית מסיבות שונות:

1. דפוסי מקומיות של בקרי DMA שונים מאוד מאלו של ליבות קונבנציונליות, לעיתים רחוקות בקרי DMA משתמשים באותם נתונים שוב בטווח זמן קצר והגישות זורמות דרך הזיכרון. לכן, ה-cache של DMA בדרך כלל לא מנצל היטב את התועלת שניתן להפיק מגודלו.

2. כאשר בקר DMA מבצע כתיבה לבלוק, הוא נוטה לכתוב את כל הבלוק בבת אחת, מה שהופך את שליפת הבלוק באמצעות פעולה כמו ReadUnique (בפרוטוקולי AMBA) לבזבזנית, מכיוון שכל הנתונים מוחלפים. פרוטוקולי קוהרנטיות שונים מטפלים במצבים כאלה על ידי שימוש בפעולות קוהרנטיות מיוחדות, כמו הוספת בקשה חדשה מסוג MakeUnique שתבקש הרשאה לכתיבה רק אם מדובר בכתיבה ולא תדרוש שליחה של נתונים חדשים. אפשרות נוספת, היא שימוש בהודעת WriteUniqueFull, שמעדכנת את הבלוק בזיכרון ומבטלת את כל העותקים האחרים, כולל העותקים במצבים של UD ו-SD.

אפשרות נוספת היא להימנע מהצורך בקוהרנטיות של cache חומרה על ידי דרישה ממערכת ההפעלה לשטוף את ה-cache בצורה סלקטיבית. לדוגמה, לפני התחלת DMA לקריאה או כתיבה ב-page מסוים, מערכת ההפעלה יכולה לאלץ את כל ה-cache-ים לשטוף את ה-page הנוגע באמצעות פרוטוקול דמוי TLB Shootdown או באמצעות תמיכה חומרתית אחרת לניקוי page-ים. גישה זו נחשבת לפחות יעילה ומיועדת בדרך כלל למערכות embedded, מכיוון שמערכת ההפעלה צריכה לשטוף את כל ה-page באופן שמרני, גם אם רק חלק קטן ממנו נמצא בקבצים הנדרשים.

פרוטוקול קוהרנטיות

מימושים התומכים ב-caching חייבים להיעזר בפרוטוקול קוהרנטיות cache על מנת לשמור על קוהרנטיות בין הזיכרון הראשי לבין ה-cache ושמירה על קוהרנטיות בין ה-cache-ים עצמם.

פרוטוקול קוהרנטיות, הוא מערכת של כללים המיושמת על ידי קבוצה של מעבדים מבוזרים בתוך המערכת. קיימים מגוון רחב של פרוטוקולי קוהרנטיות, אך העקרון המרכזי הוא שכל הווריאציות מבטיחות שהכתיבה של מעבד אחד תהיה גלויה לשאר המעבדים על ידי הפצת הכתיבה לכל ה-cache-ים.

פרוטוקול קוהרנטיות שומר על קוהרנטיות על ידי אכיפת האינווריאנטים הבאים:

1. **כותב אחד וריבוי קוראים (Single-Writer, Multiple-Read (SWMR))**: עבור כל מיקום זיכרון A, בכל זמן נתון (לוגי), ישנה רק ליבה אחת בלבד שיכולה לכתוב ל-A (ויכולה גם לקרוא אותה), או מספר ליבות שיכולות לקרוא את A בלבד מבלי לכתוב אליו.

2. **ערך ללא שינוי (Data-Value Invariant)**: הערך של מיקום הזיכרון בתחילת עידן חייב להיות זהה לערך של אותו מיקום זיכרון בסוף העידן הקודם שבו בוצעה קריאה או כתיבה.

כדי ליישם את האינווריאנטים הללו, מקשרים לכל מבנה אחסון, כל cache וזיכרון מכונת מצב סופי שנקראת בקר קוהרנטיות (coherence controller). האוסף של בקרי הקוהרנטיות הללו יוצר מערכת מבוזרת שבה הבקרים מתקשרים זה עם זה באמצעות הודעות, על מנת להבטיח שכל בלוק זיכרון שומר על האינווריאנטים SWMR ו-Data-Value לאורך זמן.

האינטראקציות בין מכונות המצב הסופי הללו מוגדרות על ידי פרוטוקול הקוהרנטיות. בקרי הקוהרנטיות אחראים למספר תחומים: בקר הקוהרנטיות ב-cache, לדוגמה, נקרא לעיתים בקר cache.

בקר ה-cache אחראי לטפל בבקשות משני מקורות עיקריים:

- **בצד הליבה**: בקר ה-cache מתקשר עם ליבת המעבד. הוא מקבל בקשות לקריאות ולכתיבות מהליבה ומחזיר את ערכי הקריאות לליבה. כאשר מתרחש cache miss, הבקר יוזם טרנזקציה קוהרנטית על ידי שליחת בקשת קוהרנטיות, כמו בקשה להרשאת קריאה בלבד, עבור הבלוק המכיל את המיקום שניגשת אליו הליבה. בקשת הקוהרנטיות נשלחת דרך רשת interconnection לבקרי קוהרנטיות אחרים במערכת. טרנזקציה זו כוללת את הבקשה וכן את שאר ההודעות הנדרשות להשלמתה, כגון הודעות תגובה עם נתונים שנשלחות מבקר קוהרנטיות אחר למבקש. סוגי הטרנזקציות וההודעות שנשלחות במסגרת כל טרנזקציה משתנים בהתאם לפרוטוקול הקוהרנטיות הספציפי במערכת.

- **בצד הרשת Interconnection**: בקר ה-cache מתקשר עם שאר המערכת דרך רשת interconnection. הוא מקבל בקשות קוהרנטיות ותגובות קוהרנטיות שצריך לעבד. עיבוד ההודעות הקוהרנטיות הנכנסות תלוי גם הוא בפרוטוקול הקוהרנטיות הספציפי המנוהל במערכת.

בקר זיכרון דומה לבקר cache, אך בדרך כלל יש לו רק צד רשת. כלומר, הוא לא מוציא בקשות קוהרנטיות עבור פעולות קריאה או כתיבה, ולא מקבל תגובות קוהרנטיות.

ה-agent-ים אחרים במערכת, כמו התקני io, עשויים להתנהג כמו בקרי cache, בקרי זיכרון או לשלב את התכנים של שניהם, בהתאם לצרכים הספציפיים שלהם. כל בקר קוהרנטיות פועל על פי קבוצת מכונות מצב סופי - למעשה, מכונת מצב סופי אחת עבור כל בלוק באופן לוגי - ומעבד אירועים (כגון הודעות קוהרנטיות נכנסות) בהתאם למצב הנוכחי של הבלוק. כאשר אירוע מסוים, E (למשל, בקשת store מהליבה), מתרחש עבור בלוק B, בקר הקוהרנטיות מבצע פעולות (כגון הוצאת בקשת קוהרנטיות להרשאת קריאה-כתיבה) התלויות באירוע E ובמצב הנוכחי של B. לאחר ביצוע הפעולות הללו, הבקר עשוי לעדכן את מצב הבלוק B בהתאם.

ההבדלים בין פרוטוקולי קוהרנטיות נובעים מהשונות במפרטי הבקר השונים. הבדלים אלה כוללים קבוצות שונות של מצבי בלוק, טרנזקציות, אירועים ומעברים בין מצבים.

מאפייני פרוטוקול קוהרנטיות

קיימים כל מיני מאפיינים לפרוטוקול קוהרנטיות ובעזרתם אפשר לחלק את הפרוטוקולים לקבוצות ולראות תכונות בולטות בין הפרוטוקולים.

מאפייני סנכרון

הפרוטוקולים שונים במועדי ובאופן הסנכרון שיתקיים, קיימים שני סוגים עיקריים של פרוטוקולי קוהרנטיות:

1. בגישה הראשונה, פרוטוקול הקוהרנטיות מבצע הפצת כתיבה ל-cache-ים באופן סינכרוני. כלומר, הכתיבה מועברת לכולם במקביל.
2. בגישה השנייה, פרוטוקול הקוהרנטיות מבצע את ההפצה באופן אסינכרוני. כלומר, הכתיבה מופצת בהדרגה, תוך שמירה על מודל העקביות של המערכת.

במעבדים מודרניים, כתיבה לרוב מופצת באופן אסינכרוני. מה שאומר שהכתיבה עשויה להתבצע לפני שהיא הופכת לגלויה לכל המעבדים במערכת. תהליך זה יכול להוביל לצפייה בערכים מיושנים בזמן אמת. עם זאת, כדי לשמור על עקביות נכונה, פרוטוקולי קוהרנטיות צריכים להבטיח שהסדר שבו הכתיבה הופכת לגלויה יהיה בהתאם לכללי הסדר שנקבעו על ידי מודל העקביות של המערכת.

קוהרנטיות בתוך המעבד ובין המעבדים

בדרך כלל, ניתן להבחין בין שני מרכיבים עיקריים בקוהרנטיות cache:

- פרוטוקול הקוהרנטיות שמטפל בניהול הקוהרנטיות בתוך היררכיית ה-cache של כל מעבד בנפרד.
- פרוטוקול הקוהרנטיות שמנהל את הקוהרנטיות בין cache-ים של מעבדים שונים במערכת.

ההבחנה הזו מאפשרת לטפל בנפרד בקוהרנטיות בתוך המעבד ובין המעבדים, מה שמייעל את הניהול של קוהרנטיות במערכות מבוזרות.

פרוטוקולי Invalidate לעומת Update

כדי להבטיח שהעותקים ב-cache-ים השונים יישארו מעודכנים, כאשר מתבצעת כתיבה לזיכרון, יש לבטל או לעדכן את העותקים הישנים, המכונים נתונים מעופשים (stale), בעותקים האחרים. ניתן לבצע זאת באמצעות שתי גישות: ביטול (invalidate) של הנתונים המעופשים או עדכון הערך המאוחסן ב-cache לערך החדש שנכתב. תהליך זה של ביטול או עדכון מלווה לעיתים קרובות בהודעת אישור שמעידה על השלמת הפעולה. רוב העיצובים של פרוטוקולי קוהרנטיות משתמשים או בגישה של ביטול או בגישה של עדכון, אך ישנם עיצובים היברידיים שבהם המערכת יכולה לבחור באופן סטטי או דינמי בין שתי השיטות, בהתאם למיקום הזיכרון או לתנאים אחרים.

בפרוטוקול קוהרנטיות, אחת ההחלטות העיקריות בתכנון היא כיצד לנהוג כאשר ליבה כותבת לבלוק זיכרון. קיימות שתי אפשרויות עיקריות:

פרוטוקולי Invalidate

כאשר ליבה רוצה לכתוב לבלוק, היא יוזמת טרנזקציה קוהרנטית שמביאה לביטול של העותקים של הבלוק בכל ה-cache-ים האחרים. ברגע שהעותקים בוטלו והפכו ללא חוקיים, הליבה המבקשת יכולה לבצע את הכתיבה ללא חשש שהליבות האחרות יקראו את הערך הישן של הבלוק. אם ליבה אחרת רוצה לקרוא את הבלוק לאחר שהעותק שלו בוטל, עליה ליזום טרנזקציה קוהרנטית חדשה כדי לקבל את הבלוק, ובכך תקבל עותק מעודכן מהליבה שביצעה את הכתיבה. תהליך זה מבטיח שמירה על הקוהרנטיות.

פרוטוקולי Update

כאשר ליבה רוצה לכתוב לבלוק, היא יוזמת טרנזקציה קוהרנטית שמביאה לעדכון כל העותקים של הבלוק בכל ה-cache-ים האחרים כדי לשקף את הערך החדש שנכתב.

פרוטוקולי Update מפחיתים את זמן ההשהיה לקריאה של בלוק חדש שנכתב, משום שהליבה אינה נדרשת ליזום ולחכות להשלמת טרנזקציה חדשה. עם זאת, פרוטוקולי Update בדרך כלל צורכים יותר רוחב פס מפרוטוקולי Invalidate, מכיוון שהודעות ה-Update כוללות מידע נוסף (כתובת וערך חדש) לעומת הודעות Invalidate (שכוללות רק כתובת).

פרוטוקולי Update גם מסבכים את היישום של מודלים שונים של עקביות זיכרון. לדוגמה, שמירה על multi-copy atomicity נעשית מורכבת יותר כאשר יש צורך לעדכן מספר עותקים של בלוק בזיכרון. בשל המורכבות הרבה של פרוטוקולי Update, הם מיושמים רק לעיתים נדירות.

תנאי סיום בפרוטוקול קוהרנטיות

תנאי הסיום של פרוטוקול קוהרנטיות דורשים שכל פעולת כתיבה שהונפקה על ידי מעבד תסתיים לבסוף באופן שהכתיבה תיראה על ידי כל המעבדים באותו סדר עקבי. כלומר, סדר ההשלמה של פעולות הכתיבה חייב להיות אחיד ונראה לכל המעבדים במערכת. דרישה זו חיונית במיוחד במערכות שבהן נתונים משוכפלים ב-cache-ים בין כמה מעבדים, שכן היא מבטיחה שכל שינוי בנתונים יהיה גלוי ומסונכרן בין כל הרכיבים.

בנוסף, הקוהרנטיות מחייבת שכל פעולת כתיבה לאותה כתובת בזיכרון תופיע בכל המעבדים באותו הסדר. כלומר, אם מספר מעבדים כותבים לאותה כתובת בזיכרון, כל המעבדים האחרים צריכים לראות את סדר הכתיבות באופן זהה. (דרישה ל-multi-copy atomicity)

Point of Coherence

כבר הוסבר בקצרה הרעיון של נקודת הקוהרנטיות אבל עכשיו שאנחנו מבינים יותר טוב את המשמעות של קוהרנטיות אפשר להסביר בצורה קצת יותר מעמיקה.

נקודת הקוהרנטיות היא מיקום רעיוני בתוך המערכת שאליו מגיעות כל הכתיבות והמחסומים במהלך פעולתן. ניתן לדמות את נקודת הקוהרנטיות כמקום הסופי שבו מתבצעת הכתיבה, לאחר שהיא עוברת דרך כל ה-cache-ים וה-buffer-ים הרלוונטיים ומגיעה לזיכרון הראשי לשם אחסון סופי של הנתונים.

העיקרון המרכזי של נקודת הקוהרנטיות הוא שכתיבות לאותה כתובת זיכרון חייבות להגיע לנקודה זו בסדר קוהרנטיות אחיד. סדר זה מגדיר את הסדר שבו כל מעבד במערכת יראה את הכתיבות. ייתכן שמערכת הזיכרון עצמה תקבע את סדר הכתיבות מראש ותשלח אותן לנקודת הקוהרנטיות לפי סדר מוגדר, או שהיא תעביר את הכתיבות לפי סדר הגעתן לנקודה זו, ומשם תקבע את הסדר. מה שחשוב הוא, שברגע שכתיבה מסוימת מגיעה לנקודת הקוהרנטיות, מיקומה בסדר הקוהרנטיות נקבע ולא ניתן לשנותו. כלומר, כתיבות עתידיות לא יוכלו לעקוף כתיבות קודמות בסדר זה.

פרט לדרישות שמטילים מחסומי זיכרון (יוסבר בהרחבה במאמר נפרד), אין הכרח שסדר ההתפשטות של הכתיבות בין המעבדים או הדרך שבה הן מגיעות לנקודת הקוהרנטיות יהיה מוגדר מראש. כתיבות יכולות להגיע לנקודת הקוהרנטיות בסדר שונה מזה שבו נשלחו במקור. לדוגמה, ייתכן ששני מעבדים יכתבו לאותה כתובת זיכרון בזמנים שונים, אך הכתיבה השנייה תגיע לנקודת הקוהרנטיות ותהפוך לגלויה לפני הכתיבה הראשונה.

הדרישה המרכזית היא שקוהרנטיות אחת בלבד תהיה תקפה עבור משתנה נתון בזמן נתון. דרישה זו מתממשת באמצעות ה-cache line המכיל את המשתנה, שיכול לעבור בין מעבדים לפי הצורך. נוסף לכך, המערכת יכולה להפיק תועלת מיכולת scalability של נקודת הקוהרנטיות עבור כל cache line, כך שהיא תוכל לתמוך גם במערכות גדולות ומורכבות יותר. יישום חומרה פשוט שיכול להתרחב בצורה יעילה

במערכות גדולות מסתמך על שימוש ב-store buffer-ים, שמסייעים בשמירה על קוהרנטיות וסדר הכתיבות.

עוד דרך לראות את ה-point of coherence הוא המקום בהיררכיית הזיכרון שבו מאוחסן הערך האחרון של מיקום זיכרון, ושם מובטחות גישה קוהרנטית למיקום זה. הוא מייצג את המיקום הספציפי שבו ניתן למצוא את הערך הקוהרנטי העדכני ביותר של בלוק זיכרון, לאחר השלמת כל הפעולות הממתינות.

Serialization Point

דרישת הסדר הכולל בטיפול בבקשות קוהרנטיות יוצרת השפעות משמעותיות על עיצוב רשת interconnection. במיוחד בפרוטוקולים מסורתיים מבוססי snooping (יוסבר בהמשך המאמר). מאחר שבקרי קוהרנטיות שונים יכולים להנפיק בקשות במקביל, יש צורך ברשת שתארגן את הבקשות לפי סדר כולל שייקבע בצורה עקבית. מנגנון זה נקרא נקודת הסדרה (serialization point), ותפקידו לקבוע את הסדר שבו הבקשות יישלחו לבקרים ויטופלו.

כאשר בקר קוהרנטיות שולח בקשת קוהרנטיות, רשת ה-interconnection מתאמת את הבקשה בהתאם לנקודת ההסדרה ושולחת אותה לכל הבקרים. הבקר ששולח את הבקשה יכול לגלות באיזה סדר היא טופלה על ידי ביצוע snooping לבקשות המגיעות מהבקרים האחרים, כך שיוכל לדעת אילו בקשות קודמות או עוקבות בקשר לבקשה שלו.

בואו ניקח דוגמה פשוטה למערכת המשתמשת ב-bus לשידור בקשות קוהרנטיות. במערכת כזו, בקרי הקוהרנטיות חייבים לפעול עם לוגיקת סלקציה שמבטיחה שבכל רגע רק בקשה אחת תעבור על ה-bus. לוגיקת הסלקציה פועלת כנקודת ההסדרה בכך שהיא קובעת את הסדר שבו הבקשות יישלחו על ה-bus. ברגע שהלוגיקה הזו מארגנת את הבקשה, הבקשה יכולה להתבצע, אך הבקר המנפיק יידע את הסדר הכולל שבו הבקשה שלו שולבה רק על ידי ביצוע snooping ב-bus. במהלך זה הוא יגלה אילו בקשות אחרות קיימות לפני או אחרי הבקשה שלו.

כתוצאה מכך, בקרי הקוהרנטיות יכולים להבין את הסדר הכולל של הבקשות רק מספר מחזורי שעון לאחר שנקודת ההסדרה קבעה את הסדר בפועל.

ה-serialization point מתייחסת לרגע או למקום בזמן שבו פעולה בזיכרון משותף מסודרת באופן גלובלי ביחס לפעולות אחרות. זוהי הנקודה שבה כל המעבדים מסכימים על סדר פעולות הזיכרון. סדר זה חיוני לעקביות, מכיוון שהוא מבטיח שכל המעבדים יצפו בפעולות זיכרון באותו רצף. לדוגמה, במערכת עם מספר cache-ים, כאשר מתרחשת כתיבה למשתנה משותף, ה-serialization point הוא הזמן או המנגנון שקובעים מתי כתיבה זו תהיה גלויה לכל המעבדים.

Serialization Point vs Point of Coherence

ה-serialization point מתארת את סדר הפעולות ומתי הן מזוהות גלובלית על ידי כל המעבדים, בעוד ש-point of coherence הוא על המיקום שבו קיים ערך הנתונים העדכני והנכון ביותר.

נקודת סריאליזציה היא מושג זמני הקשור להבטחת עקביות זיכרון, בעוד שנקודת קוהרנטיות היא מושג מרחבי הקשור למציאת הערך הקוהרנטי של cache line.

Cache maintenance

אחרי שהבנו מה זה קוהרנטיות בואו נראה איך אנחנו יכולים להשפיע בצורה מסוימת על קוהרנטיות.

תחזוקת cache בתוכנה

לעיתים יש צורך בתוכנה לבצע פעולות ניקוי או ביטול של ה-cache. פעולות אלו נדרשות כאשר תוכן הזיכרון החיצוני השתנה, ויש צורך להבטיח שה-cache אינו מכיל נתונים מיושנים. פעולות אלו יכולות להיות נדרשות גם לאחר שינויים הקשורים ל-MMU כמו שינוי הרשאות גישה, מדיניות cache, מיפוי כתובות וירטואליות לכתובת פיזית, או כאשר ה-I-caches ו-D-caches חייבים להיות מסונכרנים לקוד שנוצר באופן דינמי, כגון ב-JIT-compilers וטועני ספריות דינמיות.

- **ביטול (Invalidation) של cache או cache line:** פעולה זו כוללת ניקוי של ה-cache מנתונים ללא כתיבה לרמה הבאה או לזיכרון הראשי על ידי ביטול של cache line אחת או יותר. משמעות הדבר היא שה-cache מסומן כ-invalid, ולכן תוכן השורות אינו מוגדר. אפשר לראות זאת כדרך להסיר שינויים בתחום הזיכרון מה-cache, כך שהנתונים המחודשים מהזיכרון החיצוני יכנסו ל-cache בצורה נכונה.

- **ניקוי של cache line או cache:** פעולה זו כוללת כתיבה של התוכן של שורות ה-cache המסומנות כמלוכלכות לרמה הבאה של ה-cache או ישירות לזיכרון הראשי, ולאחר מכן ניקוי ביט ה-dirty ב-cache line. פעולה זו מבטיחה שהתוכן של ה-cache line יהיה תואם עם הרמה הבאה של ה-cache או עם מערכת הזיכרון.

פעולות תחזוקת cache והשפעתן

בנסיבות רגילות, פעולות של ניקוי או ביטול של כל ה-cache מתבצעות בעיקר על ידי ה-firmware כחלק מתהליך ההדלקה או הכיבוי של הליבה. תהליך זה עשוי לקחת זמן משמעותי, במיוחד כאשר מדובר ב-L2 cache, שבו מספר השורות יכול להיות גדול מאוד. במקרה כזה, יש לעבור על כל השורות בלולאה אחת אחת, מה שיכול להאריך את זמן הביצוע.

x86

Cache Line Writeback and Flush

הוראת CLFLUSH מקבלת כתובת ומבצעת בדיקה האם הכתובת נמצאת ב-cache. אם הכתובת קיימת ב-cache, כל שורת ה-cache הכוללת את הכתובת נעשית לא חוקית. במקרה שחלק כלשהו מהשורה במצב מלוכלך (כלומר, modified או owned ב-MOESI), כל השורה נכתבת חזרה לזיכרון הראשי לפני שהיא מבוטלת. CLFLUSH משפיעה על כל ה-cache-ים שבהיררכיית הזיכרון, כולל ה-cache-ים הפנימיים וחיצוניים למעבד, וכן על כל הליבות במעבד. לעומת זאת, הוראת CLWB פועלת בצורה דומה, אך היא לא מבטלת את שורת ה-cache. התהליך של הבדיקה והביטול נמשך עד שהכתובת עודכנה בזיכרון, ובמקרה של CLFLUSH, עד שכל ה-cache-ים הקשורים בכתובת זו בוטלו.

באופן כללי, סוג הזיכרון הבסיסי שהוקצה לכתובת לא משפיע על אופן פעולת ההוראה הזו. עם זאת, כאשר סוג הזיכרון הבסיסי עבור הכתובת מוגדר כ-Uncacheable או Write Combining, המעבד לא ימשיך לבדוק את כל ה-cache-ים כדי לוודא אם הכתובת נמצאת בהם. במצבים אלו, הכתובת אינה ניתנת לשמירה והביטול של התוקף הוא מיותר. כמו כן, אם הכתובת הפיזית המתאימה נופלת בטווח הכתובת הפעילה של Write-Combining Buffer, הנתונים ב-buffer יכתבו חזרה לזיכרון.

Invalidate & Cache Writeback

בניגוד להוראות CLFLUSH ו-CLWB, אשר פועלות על cache line בודדת, ההוראות WBINVD ו-WBNOINVD משפיעות על כל ה-cache במערכת. כאשר מופעלת הוראת WBINVD, היא מתחילה בכתיבת write-back של כל שורות ה-cache המלוכלכות (כלומר, שורות במצב modified או owned ב-MOESI) אל הזיכרון הראשי. הוראת WBINVD מבטלת גם את כל שורות ה-cache. תהליך זה נמשך עד שכל ה-cache-ים הפנימיים בנתיב של הליבה המבצעת לזיכרון המערכת מתבטלים. במצבים מסוימים, תהליך זה עשוי לכלול גם cache-ים ברמות אחרות בהיררכיית ה-cache של המערכת. לכל אחת מההוראות, מתבצע מחזור bus מיוחד שנשלח ל-cache-ים חיצוניים ברמות גבוהות יותר, כדי להנחות אותם לבצע פעולות כתיבה וביטול תוקף.

Cache Invalidate

הוראת INVD מיועדת לבטל את כל שורות ה-cache. בניגוד להוראת WBINVD, הוראת INVD לא מבצעת כתיבה write-back של שורות ה-cache המלוכלכות אל הזיכרון הראשי. התהליך נמשך עד שכל ה-cache-ים הפנימיים בוטלו. בנוסף, מתבצע מחזור bus מיוחד המועבר ל-cache-ים חיצוניים ברמות גבוהות יותר, כדי להנחות אותם לבצע ביטול תוקף. יש להשתמש בהוראת INVD רק במצבים שבהם קוהרנטיות הזיכרון איננה נדרשת.

ARM

ב-arm יש תבנית דומה לפעולות תחזוקת cache ל-data cache ול-instruction cache, וכל פעולה כזאת מקבלת פרמטרים שבעזרתם ניתן לבצע את הפעולה לדיוק ולמיקום הנכון.

סוג ה-cache

הדבר הראשון שצריך לבחור זה כמובן את סוג ה-cache שעליו רוצים להפעיל את הפעולה:

- פעולות על ה-data cache מתבצעות על ידי ההוראה DC
- פעולות על ה-instruction cache מתבצעות על ידי ההוראה IC

סוג הפעולה

כמו ב-x86 גם ב-ARM ניתן לבחור בין 2 פעולות על ה-cache-ים:

- ניקוי של cache line או cache (מסומן על ידי C)
- ביטול של cache line או cache (מסומן על ידי I)

ערכים מושפעים

בעת ביצוע פעולות תחזוקת cache, ניתן לבחור על אילו ערכים הפעולה צריכה לחול:

- לכל הערכים (All): פעולה זו משפיעה על כל ה-cache במערכת. הבחירה ב-All אינה זמינה עבור data cache.

- לכתובת הוירטואלית (VA): פעולה זו מתבצעת על שורת ה-cache המכילה תוכן של כתובת וירטואלית ספציפית. כאן מתייחסים ל-cache על פי כתובת וירטואלית.

- ה-Set ו-Way: פעולה זו מתבצעת על cache line ספציפית שנבחרת לפי מיקומה בתוך מבנה ה-cache, כלומר לפי ה-Set וה-Way שלה.

הגדרת נקודת ביצוע והיקף תחזוקת cache

ה-scope ביצוע הפעולה יכול להיות PoC או PoU.

Point of Unification

עשויות להתקיים מספר נקודות איחוד (PoU) במערכת:

1. **נקודת איחוד לכל ליבה:** כל ליבה במערכת עשויה להכיל נקודת איחוד משלה. בנקודה זו, כל הוראות הזיכרון ונתוני ה-cache הקשורים לליבה זו מאוחדים, כלומר, כל שינויים בזיכרון הנתונים או הוראות cache בליבה זו מתואמים ונראים כאחידים בליבה עצמה.
2. **נקודת איחוד למערכת כולה או לדומיין שיתוף:** בנוסף, עשויה להתקיים נקודת איחוד ברמה גבוהה יותר, המתייחסת למערכת כולה או לדומיין (תחום) השיתוף הכולל. בנקודה זו, כל ה-cache-ים במערכת או בדומיין השיתוף מאוחדים, כך שכל השינויים המתרחשים ב-cache הנתונים או הוראות ה-cache נראים כאחידים בכל המעבדים המשתפים את אותו דומיין.

כאשר מתבצע fetch מכתיבה, זה מרמז שהשינוי הגיע לנקודת האיחוד הקרובה ביותר (PoU). כלומר, כל השינויים נשמרים ומעודכנים עד לנקודה זו. עם זאת, זה לא מרמז שהשינוי הגיע לכל נקודת איחוד אחרת במערכת, גם אם הכתיבה בוצעה על ידי ליבה מרוחקת או מעבד אחר במערכת.

הכרת ה-PoU מאפשרת self-modifying code כדי להבטיח ש-fetch של הוראות עתידיות יבוצעו בצורה נכונה מהגרסה של הקוד שעבר את השינוי.

Point of Coherency

נקודות הקוהרנטיות (PoC) - היא הנקודה שבה עדכוני ה-cache נדחפים לזיכרון הפיזי. לכן כל הליבות, DSPs, התקני I/O, מנועי DMA וכו', רואים את אותו עותק של זיכרון.

סוג השיתוף

לפעולות שניתן לשדר, ניתן לבחור את רמת השיתוף: סוג השיתוף "Inner Shareable" מסומן על ידי IS.

ה-cache ברמת המערכת

מבחינה קונספטואלית, ניתן להבחין ב-2 קטגוריות עיקריות של cache-י מערכת ב-arm:

1. **ה-cache-י מערכת לפני נקודת הקוהרנטיות הניתנים לניהול חלקי:** cache-י מערכת אלה נמצאים לפני נקודת הקוהרנטיות וניתנים לניהול באמצעות הוראות תחזוקה של cache על פי כתובת וירטואלית החלות על נקודת הקוהרנטיות. עם זאת, הם אינם ניתנים לניהול על ידי הוראות תחזוקה

לפי set/way. תחזוקה של cache-ים כאלה, כמו במקרה של ניהול צריכת חשמל, חייבת להתבצע באמצעות מנגנונים שאינם ארכיטקטוניים.

2 **ה-cache-י מערכת מעבר לנקודת הקוהרנטיות:** cache-ים אלה נמצאים מעבר לנקודת הקוהרנטיות ולכן אינם נגישים לתוכנה. ניהול cache-ים כאלה אינו נכלל בתחום הארכיטקטורה ונתון לניהול מחוץ למערכת הארכיטקטונית.

במערכת מרובת מעבדים, כאשר משתמשים בהוראת IC IVAU עבור מיקום non-cacheable, ההוראה משודרת לכל המעבדים בתוך התחום Inner Shareable של המעבד שמבצע את ההוראה. זאת למרות שמיקומי זיכרון רגילים שאינם נתמכים על ידי cache, מטופלים כאל Outer Shared במצבים אחרים של הארכיטקטורה.

ייתכן שיידרשו צעדים נוספים בתוכנה כדי לסנכרן את פעולות המעבדים עם המעבדים האחרים. זה עשוי לכלול ביצוע הוראת ISB לאחר השלמת ביטול התוקף, על מנת למנוע בעיות הקשורות לשינויים וביצוע במקביל של רצפי הוראות.

בלוקים גדולים יותר של הוראות ניתן לשנות באמצעות הוראת IC IALLU במערכת חד-מעבד, או באמצעות IC IALLUIS במערכת מרובת מעבדים.

איפוס (Zero) cache

יכולת טעינת ה-cache מראש עם ערכי אפס באמצעות הוראת DC ZVA, מאפשרת למעבדים לפעול בצורה הרבה יותר מהירה לעומת גישה למערכות זיכרון חיצוניות, שלעיתים יכולה לקחת הרבה זמן.

איפוס cache line מתבצע בצורה דומה לשליפה מוקדמת, בכך שמדובר בדרכים לרמוז למעבד על השימוש הצפוי בכתובות מסוימות בעתיד. עם זאת, פעולת האיפוס יכולה להיות מהירה יותר, מכיוון שאין צורך להמתין להשלמת גישה לזיכרון חיצוני. במקום לקרוא את הנתונים מהזיכרון לתוך ה-cache, שורות ה-cache מתמלאות באפסים. זה מאפשר למעבד להבין שהקוד מחליף לחלוטין את תוכן שורת ה-cache, ולכן אין צורך בקריאה ראשונית.

דוגמה לשימוש: אם אתה זקוק ל-buffer אחסון זמני גדול או מאתחל מבנה חדש, תוכל לבחור בין כתיבת קוד שיתחיל להשתמש בזיכרון או קוד שיבצע טעינה מראש של הנתונים לפני השימוש. שני האפשרויות ידרשו זמן ומעבד לשימוש ברוחב פס זיכרון בקריאת התוכן הראשוני ל-cache.

AMBA ACE/CHI

מעבדים ותחזוקת Cache

מעבדים משתמשים בטרנזקציות תחזוקת cache כדי לגשת ולתחזק את ה-cache-ים של מעבדים אחרים במערכת. טרנזקציות תחזוקה אלו מאפשרות למעבדים לראות את ההשפעה של פעולות load ו-store על cache מערכת שלא ניתן לגשת אליו בדרכים אחרות. טרנזקציות תחזוקת cache יכולות להתפשט גם ל-cache-ים ב-downstream, מה שמאפשר תחזוקה של כל ה-cache-ים במערכת.

מעבד היוזם טרנזקציית תחזוקת cache אחראי גם לביצוע הפעולה המתאימה ב-cache המקומי שלו.

טרנזקציות תחזוקת **cache**

- **טרנזקציית CleanShared** מאפשרת למעבד לבצע פעולת ניקוי על ה-cache-ים של מעבדים אחרים במערכת. כאשר cache של מעבד שמחזיק cache line מלוכלכת מקבל טרנזקציית CleanShared, עליו לספק את שורת ה-cache כדי לכתוב אותה לזיכרון הראשי. ה-cache שמבצעים עליו snooping יכול לשמור את העותק המקומי שלו של שורת ה-cache.

- **טרנזקציית CleanInvalid** מאפשרת למעבד לבצע פעולת ניקוי וביטול תוקף על ה-cache-ים של מעבדים אחרים במערכת. כאשר cache שמחזיק cache line נקייה מקבל טרנזקציית CleanInvalid, עליו להסיר את העותק המקומי שלו של שורת ה-cache. אם ה-cache שמבצעים עליו snooping מחזיק cache line מלוכלכת, הוא נדרש לספק את שורת ה-cache כדי לכתוב אותה לזיכרון הראשי ולהסיר את העותק המקומי שלו.

- **טרנזקציית MakeInvalid** מאפשרת למעבד לבצע פעולת ביטול תוקף על ה-cache-ים של מעבדים אחרים במערכת. כאשר ה-cache שמבצעים עליו snooping מקבל טרנזקציית MakeInvalid, עליו להסיר את העותק המקומי שלו של שורת ה-cache, אך אינו נדרש לספק נתונים, גם אם שורת ה-cache מלוכלכת.

Directory & Snoop

עכשיו אחרי שאנחנו משופשפים קצת יותר בקוהרנטיות cache, וגם ראינו שיש לנו יכולת לשלוט במידה מסוימת על ה-cache, ואפילו יש כל מיני דרכים לעשות את זה וכל דרך יעילה לצורך מסוים, אפשר להעמיק ולראות למה הדברים נראים ככה. נתחיל קצת להבין איך המעבדים ממשים את זה ונבדוק את הפרוטוקול שהם עובדים איתו (או לפחות אחד מהם).

פרוטוקול Snooping

מנגנון Snooping הוא מנגנון שבו כל אחד מה-cache-ים במערכת עוקב אחרי שורות כתובת מסוימות כדי לזהות אם שורות זיכרון ששמורות ב-cache שלו נגישות או משתנות על ידי מעבדים אחרים. פרוטוקולי קוהרנטיות כמו פרוטוקולי invalidate ופרוטוקולי update משתמשים במנגנון זה.

פרוטוקול snooping מתבסס על bus משותף שמחבר בין כל ה-cache-ים לבין הזיכרון הראשי. כאשר מעבד מבצע כתיבה ל-cache שלו, הוא משדר את הכתובת של הבלוק שעבר שינוי אל ה-bus. מעבדים אחרים שברשותם עותק של אותו הבלוק ב-cache יכולים לבחור לבטל את העותק או לעדכן אותו, תלוי בגרסת הפרוטוקול שנמצאת בשימוש (פרוטוקול מבוסס invalidate או מבוסס update). היתרון העיקרי של פרוטוקול זה הוא הפשטות והמהירות שלו, אך הוא מגיע עם מספר חסרונות:

- ה-bus עלול להפוך לצוואר בקבוק ככל שמספר המעבדים והגישה ל-cache-ים גדלים.
- הפרוטוקול דורש שכל ה-cache-ים יעקבו כל הזמן אחר תעבורת ה-bus, דבר המוביל לצריכת חשמל ורוחב פס מוגבר.
- פרוטוקול זה אינו מתאים למערכות מבוזרות, שבהן ה-bus מוחלף ברשת תקשורת.

למרות החסרונות, הפרוטוקול עדיין מתאים לשימוש במערכות עם מספר קטן של מעבדים, בהן הפשטות שלו עדיפה על מגבלותיו. כמו כן, הוא שימושי במערכות בזמן אמת או ביישומים שבהם יש צורך בגישה מהירה ועם latency נמוך לנתונים משותפים.

קוהרנטיות בפרוטוקולי snooping

פרוטוקולי snooping מבוססים על עיקרון מרכזי אחד: כל בקרי הקוהרנטיות עוקבים אחר בקשות הקוהרנטיות (snoop) בסדר אחיד, ופועלים בצורה מתואמת לשמירה על קוהרנטיות הנתונים במערכת. על ידי כך שכל הבקשות המתקבלות עבור בלוק נתון יופיעו לפי סדר קבוע, הפרוטוקול מאפשר לבקרי הקוהרנטיות המבוזרים לעדכן את מצבם באופן נכון, תוך התאמה למכונת המצבים המשותפת המתארת את המצב של בלוק ה-cache בכל רגע נתון.

בפרוטוקולי snooping מסורתיים, כל בקשה משודרת לכל בקרי הקוהרנטיות במערכת, כולל הבקר שיזם את הבקשה. בקשות אלה נשלחות לרוב דרך רשת שידור מסודרת, כמו bus, שמבטיחה שכל בקרי הקוהרנטיות יקבלו את הבקשות באותו סדר בדיוק. כלומר, נוצרת מערכת של סדר כולל לבקשות הקוהרנטיות, המבטיחה שכל בקר קוהרנטיות יצפה באותה סדרה של פעולות באותו רצף.

הסדר הכולל הזה מאפשר לכל בקר קוהרנטיות לעדכן בצורה מדויקת את המצב של בלוק cache, מאחר שההחלטות מתבססות על אותה סדרת פעולות לכל הבקרים, ללא קשר למיקומם או לפעולות אחרות שמתרחשות במקביל במערכת.

כיצד Snooping תלוי בסדר כולל של בקשות קוהרנטיות

פרוטוקולי Snooping מסורתיים מבוססים על קיומו של סדר כולל של כל בקשות הקוהרנטיות במערכת. הסדר הזה מאפשר לכל בקר קוהרנטיות לקבוע מתי בקשה מסוימת התקבלה, בהתבסס על סדר snooping לוגי שמסנכרן בין כל הבקרים.

היכולת של פרוטוקול snooping לזהות באופן עקיף את סדר הגעת הבקשות היא מה שמייחד אותו מפרוטוקולי directory, שבהם נדרש מעקב מפורש אחר כל בקשה.

כדי ליישם את הדרישה הזאת לסדר כולל, רשת interconnect בפרוטוקול snooping צריכה לארגן את כל בקשות הקוהרנטיות בסדר קבוע. מאחר שבקרים שונים יכולים להוציא בקשות בו-זמנית, הרשת חייבת לטפל בתיעדוף וסידור הבקשות בנקודת הסדרה (serialization point). כאשר בקר קוהרנטיות שולח בקשת קוהרנטיות, רשת interconnect ממיינת את הבקשה, משדרת אותה לכל הבקרים, והבקר ששלח את הבקשה, לומד מה סדר ההגעה שלה על ידי צפייה בזרם הבקשות שהוא מקבל.

למשל, מערכת המבוססת על bus משתמשת ב-bus עצמו לשידור מסודר של כל בקשות הקוהרנטיות. כל בקשה שנשלחת עוברת לכל הבקרים בסדר אחיד, מה שמקל על שמירת קוהרנטיות.

אולם, הדרישה לרשת שידור מסודרת עבור פרוטוקולי snooping הופכת את המימוש ליקר ולא יעיל בהשוואה לרשתות פשוטות וזולות יותר שיכולות לשרת פרוטוקולי directory. בפרט, במערכות גדולות עם מספר רב של ליבות, snooping יוצר צוואר בקבוק, הן מבחינת רוחב הפס של הרשת הנדרשת לשידור הבקשות, והן מבחינת העומס על בקרי הקוהרנטיות שצריכים לעקוב אחרי כל הבקשות. בשל כך, מערכות בעלות scalability דורשות פרוטוקול קוהרנטיות יעיל יותר, והצורך הזה הוביל לפיתוח פרוטוקולי directory.

עם זאת, snooping עדיין יכול להיות חלק מהפתרון גם במערכות גדולות. אחת הטכניקות היעילות ל-scalability היא "הפרד ומשול". לדוגמה, במערכת הכוללת שבבים מרובי ליבות, ניתן להשתמש בפרוטוקול snooping לשמירה על קוהרנטיות בתוך כל שבב, בעוד שפרוטוקול directory מטפל בקוהרנטיות בין השבבים. גישה זו מאפשרת לשלב בין היתרונות של שתי השיטות.

פרוטוקול מבוסס Directory

בפרוטוקול מבוסס directory, ישנו רכיב מרכזי או מבוזר הנקרא directory, אשר אחראי לניהול המעקב אחר מצבם ומיקומם של כל בלוקי ה-cache במערכת. כאשר מעבד מבקש לקרוא או לכתוב בלוק מסוים ב-cache, הוא שולח בקשה ל-directory. ה-directory בודק את המידע על הבלוק ומחליט אם לאשר או לדחות את הבקשה, ובמידת הצורך, מאמת את התקשורת בין המעבד המבקש למעבדים אחרים שמחזיקים עותקים של הבלוק.

ה-directory פועל כמעין מתווך, וכל מעבד שמעוניין לטעון ערך מהזיכרון הראשי ל-cache שלו חייב לבקש רשות מה-directory.

כאשר ערך בזיכרון משתנה על ידי אחד מהמעבדים, ה-directory מעדכן את שאר ה-cache-ים שמחזיקים עותקים של אותו ערך, או לחלופין מבטל את תוקף העותקים שלהם, בהתאם לפרוטוקול הקוהרנטיות הנדרש. בכך ה-directory מבטיח שכל ה-cache-ים במערכת יישארו מסונכרנים וימנעו ממצבים של חוסר קוהרנטיות בנתונים.

פרוטוקול זה מצליח להתגבר על חלק מהמגבלות של פרוטוקול snooping:

- **הפחתת עומס תעבורה**: התעבורה ב-bus או ברשת מצטמצמת, מכיוון שרק המעבדים הרלוונטיים מעורבים בכל פעולה, ולא כל המעבדים במערכת כמו ב-snooping.
- **שיפור scalability**: ה-directory יכול להיות מבוזר על פני מספר צמתים, מה שמאפשר תמיכה במערכות גדולות יותר עם מספר רב של מעבדים.
- **גמישות**: ניתן ליישם מדיניות שונות עבור אזורי זיכרון שונים, בהתאם לצורכי היישום.

עם זאת, ישנם גם חסרונות לפרוטוקול מבוסס directory:

- **נקודת כשל מרכזית**: במערכות עם directory מרכזי, הרכיב מהווה נקודת תורפה בודדת. אם ה-directory הופך לבלתי זמין, בין אם עקב תקלת חומרה או בעיה אחרת, הדבר עלול לשבש את פעולתה של המערכת כולה.

קוהרנטיות בפרוטוקולי Directory

פרוטוקולי directories נוצרו במטרה להתמודד עם המגבלות של פרוטוקולי snooping מבחינת scalability. בפרוטוקולי snooping מסורתיים, כל הבקשות מועברות דרך רשת חיבורית מסודרת, וכל בקרי הקוהרנטיות חייבים לעקוב ולעבד את כל הבקשות, מה שיוצר עומס כבד במערכות גדולות. לעומת זאת, פרוטוקולי directories מציעים גישה שונה שמבוססת על עקיפה, המאפשרת להימנע מהצורך ברשת שידור מסודרת ומהחובה שכל בקר cache יטפל בכל בקשה.

ה-directory עוקב אחרי ה-cache-ים המחזיקים בכל בלוק, ובאילו מצבים הם מחזיקים אותו (למשל, מצבי MOESI). כאשר בקר cache רוצה לבצע בקשת קוהרנטיות, כמו בקשה לקריאת נתונים, הוא שולח את הבקשה ישירות ל-directory בהודעת unicast.

ה-directory בודק את מצב הבלוק המבוקש כדי לקבוע כיצד להמשיך. לדוגמה, אם ה-directory מזהה שהבלוק נמצא בבעלות של ליבה אחרת, נניח Core 2, הוא ינתב את הבקשה ישירות ל-Core 2, ויבקש מליבת Core 2 לשלוח עותק של הבלוק ל-cache המבקש. לאחר קבלת ההודעה, ה-cache של Core 2 ישיב עם הנתונים ל-cache שביצע את הבקשה.

הבדל מעניין בין פרוטוקולי directories לבין פרוטוקולי snooping הוא באופן הטיפול הבסיסי בבקשות. בעוד ש-snooping מסתמך על כך שכל הבקרים יתעדו את כל הבקשות ברשת מסודרת, פרוטוקולי directories עובדים בצורה ממוקדת יותר, כאשר כל בקשה מטופלת ישירות על ידי ה-directory והרכיבים הרלוונטיים, מה שמאפשר למערכת להיות scalability ויעילה יותר.

פרוטוקול Directory

בפרוטוקול directory, ה-directory אחראית לניהול מצב כל בלוק נתונים במערכת. כל בקשת קוהרנטיות שמגיעות מבקרי cache נשלחות ישירות ל-directory, אשר בוחנת את מצב הבלוק ומביאה החלטות בהתאם. בהתאם למצב, ה-directory עשויה או להגיב לבקשה בעצמה, או להעביר את הבקשה לבקרי קוהרנטיות נוספים, שיגיבו לאחר מכן.

טרנזקציות קוהרנטיות בפרוטוקול directory נוגעות בדרך כלל לשני סוגי שלבים:

1. **שני שלבים**: שלב בקשת unicast ולאחר מכן תגובת unicast.
2. **שלושה שלבים**: שלב בקשת unicast, שלב שבו הבקשה מועברת ל-K בקרי קוהרנטיות (כאשר K הוא מספר השותפים המהווים את ה-cache הנדרש), ולאחר מכן תגובות מ-K הבקרים.

בחלק מהפרוטוקולים קיימת גם אפשרות של שלב רביעי, לדוגמה, אם ה-directory מעורבת בתגובות עקיפות או אם המבקש מודיע ל-directory על סיום הטרנזקציה.

בניגוד לכך, בפרוטוקולי snooping, כל מצב הבלוק מופץ בין כל בקרי הקוהרנטיות במערכת. מכיוון שאין מרכז ניהול של מצב בלוק. כל בקשה קוהרנטית נשלחת לכל הבקרים, מה שדורש שידור של הבקשה לכל בקר, ולאחר מכן קבלת תגובה unicast. כך, טרנזקציות קוהרנטיות ב-snooping כוללות תמיד שני שלבים: שלב בקשת שידור ושלב תגובה unicast.

כמו כן, גם בפרוטוקול directory, יש צורך להגדיר מתי וכיצד טרנזקציות קוהרנטיות מסודרות ביחס לטרנזקציות אחרות. ברוב פרוטוקולי ה-directory, הסדר של טרנזקציה נקבע לפי הסדר שבו הבקשות מסודרות ב-directory. כאשר מספר בקרי קוהרנטיות שולחים בקשות בו-זמנית, ה-directory קובעת את סדר העיבוד של הבקשות.

כאשר ישנן שתי בקשות מתחרות ל-directory, הרשת קובעת איזו בקשה תעובד קודם. ההתנהלות עם הבקשה השנייה תלויה בפרוטוקול הספציפי של ה-directory ובסוג הבקשות המתחרות. האפשרויות כוללות:

- **עיבוד מיד לאחר הבקשה הראשונה**: הבקשה השנייה מעובדת ברצף לאחר הראשונה.
- **שמירה בהמתנה**: הבקשה השנייה נשמרת ב-directory עד להשלמת הבקשה הראשונה.
- **אישור שלילי (NACK)**: ה-directory שולחת הודעת אישור שלילי למבקש, שמאלץ להנפיק מחדש את בקשתו.

השימוש ב-directory כנקודת סידור (ordering point) מדגיש הבדל מרכזי נוסף בין פרוטוקולי directory לפרוטוקולי snooping:

בפרוטוקולי snooping המסורתיים, הסדר הכולל של טרנזקציות נקבע על ידי סידור כל הבקשות ברשת השידור המסודרת לחלוטין. הסדר הכולל הזה אינו רק מבטיח שכל בקשה לבלוק נתון תעובד לפי סדר הגעתה, אלא גם מקל על ביצוע מודל עקביות זיכרון.

כאשר בקשת קוהרנטיות נשלחת, היא מועברת לכולם באותו סדר, כך שכאשר בקר snooping מקבל את בקשת ה-ReadUnique (בפרוטוקול AMBA CHI) שלו, הוא יכול להסיק ש-cache-ים האחרים יבטלו את התוקף של הבלוקים במצב share אצלם. זה מאפשר לו לדעת מתי להתחיל עידן קוהרנטיות חדש.

לעומת זאת, בפרוטוקולי directory, הסדר של טרנזקציות קובע ה-directory, כדי להבטיח שבקשות סותרות יעובדו לפי סדר הגעתן בכל הצמתים. היעדר הסדר הכולל ברשת השידור פירושו שבקר בפרוטוקול directory נדרש ליישם אסטרטגיה נוספת כדי לקבוע מתי הבקשה שלו הוסדרה, ולמעשה, מתי הוא יכול להתחיל בעידן הקוהרנטיות שלו. מכיוון שרוב פרוטוקולי ה-directory אינם משתמשים בשידור מסודר לחלוטין, אין סדר גלובלי שניתן לעקוב אחריו. במקום זאת, כל בקשה צריכה להיות מסודרת באופן אינדיבידואלי ביחס לכל ה-cache-ים שיכולים להחזיק עותק של הבלוק. במקרה של בקשת ReadUnique, כל בקר cache שמחזיק עותק משותף חייב לשלוח הודעת אישור (Ack) מפורשת לאחר שהוא עיבד את הבקשה וסילק את התוקף מהעותקים שלו, על מנת להודיע למבקש שהבקשה שלו הוסדרה.

השוואת פרוטוקולי directory ו-snooping מדגישה את הפשרה הבסיסית ביניהם. פרוטוקול directory מציע scalability גבוהה יותר, כלומר הוא דורש פחות רוחב פס, אך זאת במחיר של רמת עקיפה גבוהה יותר. בפרוטוקולי directory, חלק מהטרנזקציות כוללות שלושה שלבים, בניגוד לשני שלבים בפרוטוקולי snooping, מה שמגביר את ההשהיה של חלק מהטרנזקציות הקוהרנטיות.

בפרוטוקול directory, קליטת הודעה יכולה להניע את בקר הקוהרנטיות לשלוח הודעה נוספת. באופן כללי, אם אירוע A (כגון קליטת הודעות) יכול לגרום לאירוע B (כגון שליחת הודעות), ושני האירועים הללו דורשים הקצאת משאבים (כמו קישורי רשת ו-buffer-ים), יש להיזהר מהתמודדות עם מצב של deadlock, שעלול להתרחש אם נוצרות תלות מעגלית במשאבים, שבה כל רכיב מחכה למשאב שנמצא בבעלות רכיב אחר.

AMBA CHI

עכשיו ראינו מה ההבדלים בין קבוצות פרוטוקולים שונות ואפשר לצלול יותר עמוק כדי להבין ממש מה קורה במעבד. נלמד על הפרוטוקול AMBA CHI של ARM שהוא פרוטוקול מסוג Directory.

הערה: ההסבר הבא הוא קצת ארוך. מי שרוצה להבין את הרעיון של פרוטוקול קוהרנטיות לא חייב לעשות את זה על ידי AMBA CHI. אפשר למשל לקרוא על פרוטוקול MOESI. אבל, כאן אני מסביר על AMBA CHI כי הוא פרוטוקול שלם ומאוד מפורט, ולדעתי זה נותן הבנה הרבה יותר בהירה של הרבה רעיונות שבאים לידי ביטוי בפרוטוקולים השונים.

מהו AMBA?

פרוטוקול AMBA (קיצור של Advanced Microcontroller Bus Architecture) הוא פרוטוקול שפותח על ידי ARM ומיועד לסטנדרטיזציה של התקשורת בין רכיבי חומרה שונים במערכות SoC. משפחת פרוטוקולים זו מכסה מגוון רחב של היבטים בתכנון המערכת, כולל טופולוגית bus, בוררות (arbitration) בין רכיבים המבקשים גישה למשאבים משותפים, ניהול signaling, תכנון interconnect, ניהול צריכת חשמל ואבטחה. הפרוטוקול AMBA כולל מספר מפרטים שונים שמותאמים לצרכים מגוונים של טרנזקציות וסוכנים במערכת, כולל AHB, APB, AXI, ACE ו-CHI. כל אחד מפרוטוקולים אלו מתמקד בהיבט מסוים של התקשורת בין רכיבי המערכת.

מהי מערכת קוהרנטית?

מערכת קוהרנטית היא מערכת שבה מספר מעבדים או agent-ים, יכולים לגשת לאותם מיקומי זיכרון ולבצע בהם שינויים, תוך כדי שמירה על תצוגה עקבית של הנתונים בין כל הרכיבים. כל שינוי במיקום זיכרון מסוים צריך להיות נגיש לכל ה-agent-ים בזמן אמת, כך שכולם יראו את המידע המעודכן בצורה עקבית. זה מאפשר עיבוד יעיל ומהיר יותר של נתונים ומקל על תהליכי התכנות והניפוי. עם זאת, שמירה על קוהרנטיות מביאה עימה גם אתגרים טכניים, כמו ניהול עקביות בין מספר agent-ים, הימנעות מקונפליקטים בין פעולות קריאה וכתיבה, וכן שמירה על latency נמוך ורוחב פס מספק. לפיכך, יש צורך בפרוטוקול מוגדר היטב שיסדיר את אופן התקשורת והסנכרון בין ה-agent-ים כדי להבטיח את הקוהרנטיות.

מהו AMBA5 CHI?

פרוטוקול AMBA5 CHI (קיצור של Coherent Hub Interface), הוא פרוטוקול שנועד להחליף את פרוטוקול AMBA4 ACE, תוך תמיכה בקוהרנטיות cache במערכות בעלות מספר רב של מאסטרים, כגון CPUs ו-GPUs. פרוטוקול זה מאפשר שיתוף נתונים בין מספר רב של מעבדים או agent-ים במסגרת תחום

קוהרנטי, כאשר הם ניגשים לזיכרון דרך רכיב מרכזי שנקרא Hub קוהרנטי. ה-Hub מתפקד כמתווך וכבקר זיכרון עבור התחום הקוהרנטי.

רכיבים ב-AMBA CHI

Interconnect

ה-Interconnect (או בקיצור ICN), הוא מנגנון התקשורת המשמש בפרוטוקול CHI להעברת מידע בין צמתי פרוטוקול. מנגנון זה יכול להיות מיושם במגוון טופולוגיות, כמו רשת של switch-ים המחוברים ב-ring, crossbar, או רשתות אחרות. ה-Interconnect עשוי לכלול גם צמתי פרוטוקול שונים, כגון צומת Home וצמתים אחרים, בהתאם לצרכי התקשורת במערכת.

Requester

ה-Requester הוא רכיב במערכת שתפקידו ליזום טרנזקציה על ידי שליחת הודעת בקשה. מדובר ברכיב שמתחיל את תהליך התקשורת באופן עצמאי, כאשר הוא נדרש למשאב מסוים או לנתונים מסוימים. המונח Requester, מתייחס גם לרכיב interconnect שמוציא הודעות בקשה ב-downstream כחלק מתהליך ניהול התקשורת, בין אם מדובר בבקשה יזומה ישירות, או בתגובה לטרנזקציות אחרות המתרחשות במערכת. כלומר, רכיב ה-Requester לא רק שולח בקשות עצמאיות אלא גם עשוי להגיב לטרנזקציות שהושקו על ידי רכיבים אחרים במערכת.

סוגי Requester-ים:

- **צמתים קוהרנטיים לחלוטין (RN-F):** צמתי Requester אלה מכילים זיכרון cache התומך בקוהרנטיות מלאה. כתוצאה מכך, הם יכולים לקבל ולהגיב ל-snoop-ים, שהם בקשות שמטרתן לשמור על עקביות הנתונים ב-cache-ים במערכת מרובת agent-ים.

- **צמתים I/O קוהרנטיים (RN-I):** צמתי Requester אלו אינם מכילים cache קוהרנטי ולכן אינם מסוגלים לקבל snoops. בדרך כלל, הם משמשים לצורך חיבור התקני I/O למערכת.

הערה: במאמר הזה אני משתמש במונח Requester אבל הכוונה היא ל-RN-F.

Subordinate

ה-Subordinate הוא ה-agent במערכת שאחראי לקבל את הטרנזקציות מה-Requester, ולבצע אותן כהלכה. תפקידו הוא להשלים את הבקשות באופן תקין, בין אם מדובר בגישה לנתונים, משאבים או זיכרון. בדרך כלל, ה-Subordinate הוא ה-agent הנמוך ביותר בהיררכיית התקשורת של המערכת. כלומר, הנקודה הסופית אליה מגיעות הטרנזקציות לביצוע. לעיתים מתייחסים אליו גם כאל "משלים"

(Completer) או "נקודת קצה" (Endpoint), שכן הוא ה-agent שסוגר את המעגל ומבצע את הפעולה שהתבקשה בטרנזקציה, דוגמה ל-Subordinate זה הזיכרון הראשי.

- **צומת SN-F עבור זיכרון רגיל:** צמתים אלו מחוברים להתקני זיכרון המגבים את שטח הזיכרון הקוהרנטי. לדוגמה, בקר זיכרון יתחבר לצומת SN-F מסוג זה.
- **צומת SN-I עבור זיכרון היקפי או רגיל:** צמתים אלו מחוברים לציוד היקפי I/O או לזיכרון לא קוהרנטי, המספק גישה למשאבים שאינם מחויבים לעמוד בקוהרנטיות מלאה.

הערה: במאמר הזה אני משתמש במונח Subordinate אבל הכוונה היא ל-SN-F.

Home

ה-(HN) Home Node הוא יחידה בתוך המערכת, המשמשת כנקודת ריכוז עבור טרנזקציות של פרוטוקול קוהרנטיות המגיעות מצמתי Requester. תפקידו הוא לנהל ולתאם את הגישה לנתונים או למשאבים המשותפים, והוא ממוקם בתוך ה-interconnect, המהווה את החיבור המרכזי בין מרכיבי המערכת השונים.

צמתי Home יכולים להיות קוהרנטיים או לא קוהרנטיים בהתאם לתפקידם.

צמתי Home קוהרנטיים לחלוטין (HN-F): צמתי Home אלו אחראים לניהול וסידור כל הבקשות לזיכרון קוהרנטי במערכת. הם גם מוציאים פעולות snoop לצמתי RN-F כדי לשמור על קוהרנטיות בין כל הצמתי ה-Requester-ים.

במערכת המבוססת על פרוטוקול CHI, כאשר מתבצעת בקשת קריאה, הנתונים הנדרשים יכולים להגיע ממספר מקורות אפשריים:

- ה-cache שנמצא בתוך ה-interconnect.
- צומת Subordinate.
- עמית RN-F: צומת Requester אחר במערכת בעל יכולות קוהרנטיות מלאות, שיכול לספק את הנתונים הנדרשים לבקשת הקריאה.

תרחיש אחד אפשרי, הוא שה-Home Node מבקש מ-RN-F או מצומת ה-Subordinate להחזיר את הנתונים ישירות אליו (אל ה-Home). לאחר מכן, ה-Home Node מבצע את התיאום הנדרש ומעביר עותק של הנתונים שקיבל אל ה-Requester.

כדי לשפר את יעילות המערכת ולצמצם שלבים מיותרים בתהליך זה, ניתן לאפשר לספק הנתונים (למשל, ה-RN-F או ה-Subordinate) להעביר את תגובת הנתונים ישירות ל-Requester שביקש את הנתונים, מבלי שהנתונים יעברו דרך ה-Home Node תחילה. דרך פעולה זו מפחיתה את ההשהיה (latency)

ומשפרת את זרימת הטרנזקציות, שכן היא מבטלת את הצורך במעבר נוסף דרך ה-Hom. היכולת היא בעצם תמיכה ב-2 יכולות: Direct Memory Transfer (DMT) ו-Direct Cache Transfer (DCT).

הערה: במאמר הזה אני משתמש במונח Home אבל הכוונה היא ל-HN-F.

Snoopee & Snoop

ה-Snoop היא הודעת בקשה שמיודעת ל-Requester. יש כל מיני סוגים של פעולות שיכולות להתבצע כתוצאה ממנה ב-Requester היעד, לדוגמה בקשת snoop לקבל cache line מסוימת.

ה-Snoopee הוא Requester שמקבל snoop (כלומר עושים עליו פעולת snoop ועליו לפעול בהתאם).

allocate

allocate היא פעולת הקצאת משאבים שנעשה ב-node מסוים. לדוגמה, זה יכול להיות הקצאת מקום ב-cache של ה-node, כדי לאפשר קבלת מידע מבקשת כתיבה. דוגמא נוספית היא הקצאה של משאבים לסיפוק של בקשה מסוימת, אם היא צריכה להחזיר תגובת ack.

בסיום הטרנזקציה שגרמה לפעולת allocate, ה-node שביצע את ה-allocate צריך לשחרר את המשאבים ולכן הוא יבצע פעולת deallocate.

סיווג תפקיד רכיבים במערכת

רכיבים שונים במערכת יכולים להיות מסווגים כ-“Requester” או “Completer”, תלוי בתפקידם בטרנזקציה:

- **צומת Requester:** רכיב שמתחיל טרנזקציה על ידי שליחת הודעת בקשה. זהו רכיב שמסוגל ליזום טרנזקציות עצמאיות, או לפעמים, רכיב interconnect שמוציא הודעת בקשה במורד הזרם כחלק מטרנזקציה אחרת. כלומר, ה-Requester לא חייב להיות מקור הבקשה הראשוני, אלא יכול גם לתפקד כחלק משרשרת טרנזקציות במערכת.
- **צומת Completer:** רכיב שמקבל טרנזקציה ומגיב לה. ה-Completer יכול להיות רכיב interconnect, כמו HN, או רכיב מחוץ למערכת ה-interconnect, כגון subordinate, שמבצע את הטרנזקציה בפועל.

מודל זיכרון

Multi-copy atomicity

מודל הזיכרון של AMBA CHI מחייב יישום של Multi-copy atomicity. משמעות הדבר היא, שכל רכיב התואם לפרוטוקול חייב להבטיח שבקשות הכתיבה שלו מקיימות את התנאים של Multi-copy atomicity. כתיבה נחשבת כעומדת בתנאי זה אם מתקיימים שני תנאים עיקריים:

- כל פעולות הכתיבה לאותו מיקום מתבצעות בסדר מוגדר, כך שכל ה-Requester-ים רואים (observed) את פעולות הכתיבה באותו סדר. יש Requester-ים שאולי לא יראו את כל הכתיבות.

- קריאה מאותו מיקום לא תוכל להחזיר את הערך שנכתב, עד שכל ה-Requester-ים יוכלו לראות את הכתיבה (כלומר, אסור ש-Requester אחד יוכל לראות את הערך החדש ו-Requester אחר יראה ערך ישן).

Downstream

ה-Downstream מוגדר מנקודת המבט של ה-Requester. כשה-Requester שולח בקשה, ה-Downstream cache הוא ה-cache שאליו ניגשת הבקשה באמצעות טרנזקציות הבקשה בפרוטוקול CHI. כלומר, ה-cache ממוקם "במורד ה-stream" ביחס ל-Requester.

Upstream

במהלך טרנזקציה, יש אינטראקציה בין ה-Requester לבין רכיבי Subordinate אחד או יותר, כאשר התקשורת יכולה לעבור גם דרך רכיבי ביניים נוספים. במצב כזה, ה-Upstream מייצג את הכיוון מהצומת הנוכחי (כגון רכיב ביניים), חזרה לכיוון צומת ה-Requester המקורי שהוציא את הטרנזקציה. כלומר, כל רכיב ביניים שמבצע טרנזקציה מתייחס ל-Upstream בתור הכיוון שמוביל אל צומת ה-Requester המקור, כולל הצומת עצמו.

יש לציין כי ההגדרות של "במורד הזרם" (Downstream) ו"במעלה הזרם" (Upstream) נקבעות לפי הטרנזקציה בכללותה, ולא על סמך זרימת נתונים ספציפיים בתוך הטרנזקציה.

PoS - Point of Serialization

ה-(PoS) Point of Serialization, היא הנקודה בתוך ה-interconnect שבה נקבע סדר הביצוע של הבקשות המגיעות מ-agent-ים שונים. בנקודה זו, המערכת קובעת את סדר הפעולות על מנת להבטיח עקביות בביצוע הטרנזקציות, מה שמאפשר שמירה על סדר פעולות קריאה וכתיבה מצד ה-agent-ים השונים במערכת.

PoC - Point of Coherence

ה-(PoC) Point of Coherence היא הנקודה במערכת שבה מובטח שכל ה-agent-ים שיכולים לגשת לזיכרון יראו את אותו העותק העדכני של מיקום הזיכרון. כלומר, ב-PoC מתקיים סנכרון בין כל הגישות

לזיכרון, כך שכולם עובדים מול אותו מצב נתונים. במערכת טיפוסית שמבוססת על CHI, ה-PoC נמצא ב-Home הקוהרנטי (HN-F) שב-interconnect.

מאפייני Cache line

ל-AMBA CHI יש מודל קוהרנטיות דומה ל-ACE, והוא מציג מספר המצבי ה-cache line, בניגוד למשפחת פרוטוקולי MESI, הפרוטוקול מציג מצבים שבנויים על ידי מספר מאפיינים, וחלק מהמאפיינים יכול להתקיים במקביל על אותו cache line.

מוצגים כאן המאפיינים של ה-cache line, והם מוצגים בצורה של קבוצות מנוגדות. כלומר, כל קבוצה מכילה מאפיינים שלא יכולים להיות במקביל על אותה cache line, אבל מאפיינים בקבוצות שונות יכולים להתקיים במקביל.

Valid, Invalid

- **מאפיין Valid:** ה-cache line תקפה (Valid), כלומר היא קיימת ב-cache.

- **מאפיין Invalid:** ה-cache line לא תקפה (Invalid), כלומר היא לא קיימת ב-cache.

Unique, Shared

- **מאפיין Unique:** ה-cache line היא העותק היחיד שקיים.

- **מאפיין Shared:** ה-cache line יכולה להיות משותפת (עם מספר עותקים) ב-cache-ים שונים, אבל לא בטוח שיש עוד עותקים.

Clean, Dirty

- **מאפיין Clean:** ה-cache שמחזיק ב-cache line לא אחראי על עדכון הזיכרון הראשי, וה-cache line מכילה ערך שיכול להיות שונה מהזיכרון הראשי או זהה לזיכרון הראשי.

- **מאפיין Dirty:** האחריות של ה-cache שמחזיק ב-cache line היא להבטיח שהערך מתעדכן בזיכרון הראשי (ההבטחה לא בהכרח אומרת שה-cache בעצמו יבצע את העדכון), ה-cache line יכול להכיל ערך שונה מהזיכרון הראשי אבל לא חייב להכיל ערך שונה.

Full, Partial, Empty

- **מאפיין Full:** כל הבתים שה-cache line מכילה חוקיים (במצב Valid).
- **מאפיין Partial:** חלק מהבתים שה-cache line מכילה יכולים להיות חוקיים (במצב Valid), אבל גם יכול להיות שאין בכלל או כל הבתים חוקיים.
- **מאפיין Empty:** ה-cache line לא מכילה בכלל בתים חוקיים (במצב Valid).

מאפיין Empty

כש-cache line היא Empty, היא מוחזקת במצב Unique כדי למנוע עותקים אחרים של אותהcache line.

מצבים בהם cache line ריקה יכולה להיווצר כוללים לדוגמה את המקרים הבאים:

- רכיב requester יכול בכוונה להשיג cache line ריקה לפני תחילת כתיבה, וזאת במטרה להפחית את צריכת רוחב הפס של המערכת. במקרה שבו ה-requester מתכנן לכתוב לשורת ה-cache, הוא יכול לקבל cache line ריקה עם הרשאות אחסון, במקום לקבל שורה מלאה וחוקית של נתונים קיימים.

- רכיב requester שמחזיק כבר בעותק של שורת ה-cache יכול לעבור למצב ריק כאשר מתבקשת הרשאת אחסון. העותק שבידיו יהיה לא חוקי (invalid) עד שה-requester יקבל הרשאת אחסון. כאשר בקשה זו תושלם, ה-requester יוכל להחזיק ב-cache line ריקה עם הרשאות אחסון.

מצבי Cache line

מצב Invalid

ה-cache line לא קיימת ב-cache.

מצב Unique Dirty

ה-cache line קיימת רק ב-cache הזה בלבד, והיא בעלת האחריות לעדכון הזיכרון הראשי. ה-cache שמחזיק את ה-cache line יכול לבצע כתיבה כי הוא היחיד שמחזיק בה, וכבר יש לו הרשאות כתיבה.

בבקשת snoop, ה-cache צריך להעביר את ה-cache line ל-requester המבקש, ולשנות את המצב של ה-cache line בהתאם (ואולי גם לכתוב את הנתונים לזיכרון הראשי אם בקשת ה-snoop דורשת את זה).

מצב Unique Dirty Partial

ה-cache line הזו קיימת ב-cache הזה בלבד, והיא בעלת האחריות לעדכון הזיכרון הראשי. זה יכול להיות כמה בתים חוקיים, כשחלקם יכול להיות אף אחד או את כולם.

לאחר שה-requester מקבל בעלות על cache line ריקה, הוא רשאי, אך אינו מחויב, לאחסן נתונים ב-cache line. אם ה-requester מחליף חלק מנתוני ה-cache line תישאר במצב Unique Dirty Partial. כלומר, ה-cache line תכיל נתונים חלקיים שעברו שינוי.

המצב הזה הוא מצב זמני ופרטי ולא יכול להיות קבוע. לכן, בבקשת snoop, לא ניתן להעביר את ה- cache line ל-requester המבקש, כי חסר לו מידע כדי להעביר cache line מלאה, וצריך לבצע טרנזקציות מתאימות כדי לקבל את שאר המידע הרלוונטי, על מנת לקבל את המידע המלא. דוגמה לתהליך כזה הוא כאשר ה-cache line מפונה, יש למזג נתונים מה-cache או הזיכרון ברמה הבאה, עם ה-cache line שמפונה כדי ליצור את שורת ה-cache התקינה המלאה.

מצב Shared Dirty

ה-cache line בעלת האחריות לעדכון הזיכרון הראשי. מכיוון שה-cache line משותפת (Shared), ייתכן שהיא קיימת ב-cache מקומי אחד או יותר, אבל זה לא מובטח. אם ה-cache line קיימת במספר cache-ים, ה-cache-ים האלו יהיו בעלי שורה זו ב-Shared Clean.

מצב Unique Clean

ה-cache line לא בעלת האחריות לעדכון הזיכרון הראשי, והיא קיימת רק ב-cache מקומי יחיד. ניתן לשנות אותו מבלי ליידע cache-ים אחרים.

מצב Unique Clean Empty

ה-cache line קיימת רק ב-cache הזה, אך אף אחד מהבתים חוקי. ניתן לשנות את ה-cache line מבלי להודיע ל-cache-ים אחרים.

מצב Shared Clean

ה-cache line עשויה להיות מוחזקת ב-cache מקומי אחד או יותר. ייתכן שה-cache line השתנתה ביחס לזיכרון הראשי, אך cache זה אינו אחראי לכתיבת השורה חזרה לזיכרון בעת פינוי.

מיפוי בין MOESI ל-AMBA CHI

יש תמיכה של AMBA CHI ב-MOESI, וכמו שאפשר לראות בטבלה הבאה, התמיכה הגיעה די בחינם, ולא היה צורך לעשות שינויים על מנת לאפשר את התמיכה:

MOESI	AMBA
Modified	UniqueDirty
Owned	SharedDirty
Exclusive	UniqueClean
Shared	SharedClean
Invalid	Invalid

לפי דעתי, הגישה הזאת נוחה יותר על מנת להבין את הפעולות עצמן במערכת. משום שבעזרת AMBA יש קשר הדוק בין השמות של המצבים שה-cache line נמצאת בהם, לבין הטרנקציות שמתרחשות במערכת.

טרנזקציות ב-AMBA CHI

טרנזקציה היא קבוצת הודעות שהמערכת מחליפה בין node-ים כדי להשלים בקשה שמגיעה מ-node כלשהו.

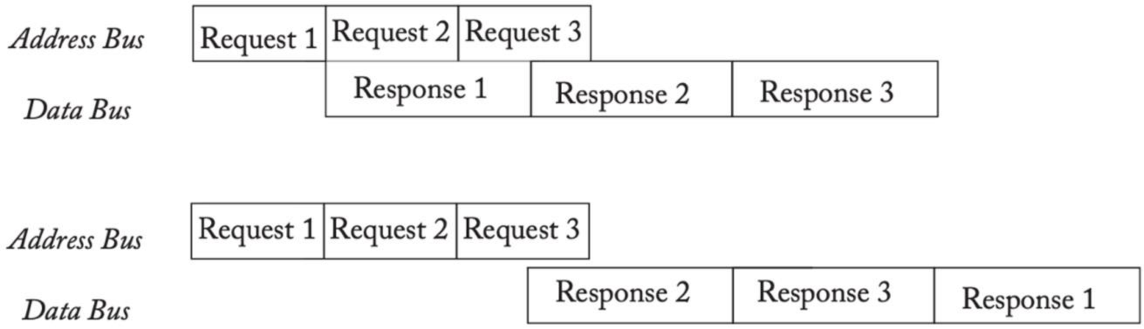
תגובות In order לעומת Out-of order

אחד היתרונות המרכזיים של bus pipeline שאינו פועל באופן אטומי, הוא היכולת לטפל בבקשות באופן רציף, ללא צורך להמתין להשלמת תגובה לבקשה קודמת. מבנה זה מאפשר ל-bus לנצל את המשאבים

המשותפים בצורה טובה יותר ולהשיג רוחב פס גבוה יותר, כל זאת מבלי להגדיל את מספר ה-thread-ים או להוסיף משאבים נוספים. עם זאת, בניית טרנזקציות אטומיות בתנאים כאלה הופכת למורכבת יותר (אך עדיין אפשרית). טרנזקציה אטומית מבטיחה כי כל פעולה על בלוק מסוים תסתיים לפני שתחל פעולה חדשה על אותו הבלוק, אולם אין חובה לקיים את אותה ההבטחה עבור בלוקים שונים.

גישה אחת ליישום bus שאינו אטומי, היא שימוש בטרנזקציות מפוצלות (split-transaction). ב- bus pipeline, התגובות לבקשות נשלחות תמיד באותו הסדר שבו התקבלו הבקשות, אך ב- split-transaction bus, התגובות לבקשות עשויות להגיע בסדר שונה מזה שבו התקבלו הבקשות.

פרוטוקול AMBA CHI עובד עם טרנזקציות מפוצלות (split-transaction):



[Split-Pipelined-transactions from A Primer on Memory Consistency and Cache Coherence Second Edition (pages.cs.wisc.edu)]

יתרונות וחסרונות של bus מפוצל טרנזקציות

היתרון העיקרי של bus מפוצל טרנזקציות לעומת bus pipeline, הוא בכך שהוא מאפשר למערכת לספק תגובות עם השהיות קצרות יותר, מבלי להמתין לתגובות בעלות השהיות ארוכות יותר. לדוגמה, בקשה 1 מתייחסת לבלוק שאינו נמצא ב-cache, ולכן מחייב גישה לזיכרון הראשי. בעוד בקשה 2 מתייחסת לבלוק שכבר נמצא ב-cache. אז pipeline bus היה מאלץ את התגובה לבקשה 2 להמתין עד להשלמת הטיפול בבקשה 1. מצב זה היה יוצר עיכוב בביצועים. לעומת זאת, ב-split-transaction bus, התגובה לבקשה 2 יכולה להישלח מיד, ללא תלות בתגובה לבקשה 1, ובכך נמנע עיכוב מיותר.

אתגרים ביישום bus מפוצל טרנזקציות

בעיה מרכזית שמופיעה כאשר משתמשים ב-bus מפוצל, היא הצורך בהתאמת התגובות לבקשות המתאימות. ב-bus אטומי, התגובה האחרונה תמיד תתאים לבקשה האחרונה שהתקבלה, דבר שמפשט את ניהול התגובות. ב-pipeline bus, המבקש צריך לעקוב אחר מספר הבקשות שממתינות למענה כדי להתאים תגובה לבקשה הנכונה. עם זאת, ב-bus מפוצל, המערכת חייבת לצרף לכל תגובה מידע המציין את זהות הבקשה או המבקש, כדי להבטיח שהתגובה תגיע לבקשה הנכונה, וכדי למנוע בלבול במצבים שבהם התגובות מתקבלות בסדר שונה מזה שבו נשלחו הבקשות.

התמודדות עם מצב Deadlock בפרוטוקול קוהרנטיות

בפרוטוקולי קוהרנטיות במערכות עם cache, יכול להיווצר מצב של deadlock, כאשר יש תלות מעגלית בין משאבים. כלומר, כל משאב ממתין למשאב אחר שיסיים את עבודתו לפני שהוא ממשיך, וכך אף משאב לא מסוגל להתקדם. תרחישים כאלו יכולים להתרחש במספר רמות שונות של מערכת הקוהרנטיות. לדוגמה, במערכת שבה שני מעבדים שולחים בקשות ל-cache אחד של השני, מצב של deadlock יתרחש אם כל אחד מהמעבדים ממתין לתגובה מהשני לפני שהוא יכול לעבד את הבקשה שלו. בנוסף, deadlock יכול להיגרם גם עקב buffering של הודעות שלא מעובדות במלואן בהיררכיית ה-cache או בעיבודן ברשת התקשורת בין הרכיבים.

פתרון נפוץ למניעת מצב זה הוא הפרדה בין נתיבי התקשורת של הודעות הבקשה להודעות התגובה, כך שכל אחד מהם ישתמש בערוץ לוגי נפרד או בנתיבים וירטואליים שונים (כמו בפרוטוקול AMBA CHI) . הפרדה כזו מונעת חסימות שנגרמות כאשר הודעות משני סוגים שונים ממתינות זו לזו באותו ערוץ, ובכך נמנע מצב של deadlock. עם זאת, כאשר יש הפרדה בנתיבי התקשורת, עלול להיווצר חוסר עקביות בסדר הגעת ההודעות מאותה נקודת מקור לנקודת יעד, דבר שמצריך פתרונות נוספים, כמו שימוש במחסומי זיכרון, או בטרנזקציות עם מאפייני סידור, כדי להבטיח סדר אחיד והגיוני של פעולות התקשורת.

אישור השלמה (Completion acknowledgment - CompAck)

ה-CompAck הוא מנגנון המבטיח את סדר השלמת הטרנזקציות במערכת. באמצעות אישור זה, ניתן לשלוט בסדר היחסי של טרנזקציות שהונפקו על ידי ה-Requester, ובין טרנזקציות ה-Snoop, הנגרמות על ידי בקשות שונות. המנגנון מבטיח שטרנזקצית Snoop שמסודרת לאחר טרנזקציה מה-Requester, מובטחת להתקבל לאחר תגובת הטרנזקציה.

הסדר שבו טרנזקציית Read מושלמת ונשלח אישור CompAck הוא כדלקמן:

1. ה-Requester מקבל אחת מההודעות הבאות: Comp, RespSepData, או CompData, או לאחר קבלת שילוב של RespSepData ו- DataSepResp.

2. ה-Requester שולח CompAck.

3. ה-Home חייב להמתין לקבלת CompAck, לפני שליחת בקשת Snoop נוספת לאותה כתובת.

סדר זה מבטיח שה-Requester יקבל השלמת טרנזקציה ובקשת Snoop לאותה כתובת של ה-cache line באותו הסדר שבו הם נשלחו על ידי ה-Home, ומבטיח שהפעולות לכתובת מסוימת ב-cache יתבצעו בסדר הנכון.

כאשר ל-Requester יש טרנזקציה פעילה המשתמשת באישור CompAck, (למעט במקרים של מספר טרנזקציות ספציפיות כמו ReadNoSnp), מובטח שטרנזקציית Snoop לא תישלח לאותה כתובת בין הזמן שבו מתקבל ה-CompAck לבין הזמן שבו ה-CompAck נשלח.

שימוש ב-CompAck עבור טרנזקציה נקבע לפי שדה ExpCompAck שנקבע על ידי ה-Requester בבקשה המקורית.

חשוב לציין שלא כל טרנזקציה במערכת CHI דורשת הודעת Completion Acknowledgment.

עצירת טרנזקציות לשמירה על הסדר

צמתי ה-HN-F יכולים לשלוט בסדר הטרנזקציות במערכת על ידי עצירת טרנזקציות מסוימות. למשל, אם כבר קיימת טרנזקציה בביצוע (outstanding) שפועלת על cache line מסוימת, ו-Requester אחר מנפיק טרנזקציה חדשה שמובילה לפעולת snoop לאותה שורת cache, ה-HN-F יכול להשהות את הטרנזקציה המאוחרת כדי לשמור על הסדר.

לאחר שהטרנזקציה הקוהרנטית המקורית שהונפקה על ידי ה-RN-F מושלמת, ה-RN-F שולח CompAck ל-HN-F. בעקבות זאת, ה-HN-F משחרר את פעולות ה-snoop שחיכו להשלמת אותה טרנזקציה.

מנגנון זה מזכיר את הפונקציונליות של מנגנוני RACK/WACK בפרוטוקול ACE, בהם שומרים על סדר טרנזקציות באמצעות מנגנוני אישור.

תגובה ונתונים נפרדים בטרנזקציית קריאה

בטרנזקציית קריאה, ישנה אפשרות לקבל תגובות נפרדות עבור השלמת הקריאה ועבור הנתונים שנקראו. כדי לתמוך במנגנון זה, נוספו שתי הודעות חדשות:

- הודעת **RespSepData**, אשר מסמנת שהקריאה הגיעה לנקודת הסידור. ה-Home שולח הודעה זו ל-Requester. באמצעות הודעה זו, משך החיים של בקשת הקריאה ב-Home מתקצר, בגלל שה-Requester יכול לשלוח מיד את תגובת ה-CompAck לאחר קבלת קריאה שאינה מסודרת.

- הודעת DataSepResp, המתייחסת לנתונים בלבד, שמטרתה לשלוח את נתוני הקריאה. הודעה זו יכולה להישלח על ידי ה-Home או על ידי Subordinate, בהתאם אם נעשה שימוש במנגנון DMT.

משמעות הסדר של RespSepData ו-DataSepResp

כאשר ה-Requester מקבל את הודעת ה-DataSepResp ראשון (לפני ה-RespSepData), ניתן להתייחס לטרנזקציית הקריאה ככזו שנצפתה גלובלית. הסיבה לכך, היא שאין יותר פעולות שיכולות לשנות את הנתונים שכבר התקבלו במסגרת הקריאה. כלומר, מרגע שהתקבלה תגובת DataSepResp, הנתונים שנקראו הם סופיים ואין עוד סיכון לשינוי במידע שנמסר.

כאשר ה-Requester מקבל תגובת RespSepData מה-Home, הדבר מעיד על כך שהבקשה הרלוונטית לטרנזקציה סודרה ב-Home. מרגע זה, ה-Requester לא יקבל שום בקשת Snoop עבור טרנזקציות שנשלחו לפני תגובת ה-RespSepData. לפני שה-Home שולח את תגובת ה-RespSepData ל-Requester, עליו לוודא שאין בקשות Snoop פעילות לאותו Requester ולאותה כתובת ספציפית. עם זאת, חשוב לציין שקבלת RespSepData לא מבטיח שה-Home השלים את כל פעולות ה-snooping הנדרשות מול node-ים אחרים במערכת, אלא רק לגבי ה-Requester הנוכחי.

סידור טרנזקציות

מעבר לשימוש בתגובת Comp לצורך שמירה על הסדר של בקשות שנשלחות על ידי ה-Requester , ב-AMBA CHI יש מנגנונים נוספים שמבטיחים סידור של בקשות בין ה-Requester ל-Home, כמו שדה ה-Order, המשמש לצורך קבלת אישור שה-Requester התקבל בהצלחה.

הסדר בין ה-Requester ל-node-ים השונים נתמך על ידי שדה Order שמופיע בכל בקשה הנשלחת.

טרנזקציות קריאה

תגובות להשלמת טרנזקציות קריאה

התגובות להשלמת הטרנזקציות CompData ו-DataSepResp כוללות שדה בשם Resp, אשר משמש לציון הפרטים הבאים:

שדה Cache state

השדה מציין את מצב ה-cache line הסופי ב-Requester עבור כל סוגי הקריאות, למעט ReadNoSnp ו-ReadOnce. השדה מצוין בשם בתגובה באמצעות תוספת של שם המצב בקיצור.

לדוגמה, התגובה CompData_UC מתארת שה-Requester שקיבל את התגובה צריך להעביר את המצב של ה-cache line שלו ל-Unique Clean.

שדה Pass Dirty

השדה מציין האם האחריות לעדכן את הזיכרון עוברת אל ה-Requester. אם השדה Pass Dirty מופעל, המשמעות היא שה-Requester מקבל אחריות על עדכון הזיכרון במקרה שהנתונים הנוכחיים מלוכלכים, ומצב זה מצוין בשם התגובה באמצעות הסיומת PD.

לדוגמה, התגובה CompData_UD_PD מתארת שה-Requester שקיבל את התגובה, קיבל גם את האחריות לכתוב את הערך של ה-cache line לזיכרון הראשי (או לדאוג שמישהו אחר יעשה את זה בעתיד). בשילוב עם השדה Cache state, ה-Requester צריך להעביר את המצב של ה-cache line שלו ל-Unique Dirty.

סידור ותגובת ReadReceipt

ההודעה מסוג ReadReceipt מסמנת שהמערכת יכולה להנפיק את הבקשה הבאה לאחר שהבקשה הקודמת סודרה והתקבלה כראוי.

כאשר ה-subordinate שולח תגובת ReadReceipt, הוא מתחייב לשמור על הסדר שבו קיבל את הבקשות השונות. כלומר, הוא מבטיח שהבקשות יטופלו ויבוצעו לפי הסדר שבו הגיעו, כדי לשמור על עקביות והתנהלות נכונה של הטרנזקציות במערכת.

מטרת ההודעה היא להבטיח שהבקשה התקבלה ותטופל לפי סדר הבקשות שהוגדר.

הודעת ReadReceipt נשלחת כאשר ישנה בקשה המחייבת שמירה על סדר בהשוואה לבקשות אחרות מאותו Requester.

ReadNoSnp

ה-ReadNoSnp היא בקשת קריאה שנעשית על ידי Requester עבור אזור כתובת שאינו snoopable, או שבקשה זו יכולה להישלח מ-Home לכל אזור כתובת כדי לקבל עותק של הנתונים הכתובים (addressed data).

בטרנזקציית ReadNoSnp, מדובר בקריאה שאינה מבצעת הקצאה (Non-allocating). הנתונים שמתקבלים כתוצאה מהקריאה לא צפויים להישמר ב-cache, ואם הם כן נשמרים, הם לא נשמרים בצורה קוהרנטית עם שאר המערכת.

בנוסף, אין צורך להודיע ל-Home על השלמת הטרנזקציה. כלומר, אין צורך לשלוח תגובת CompAck ל-Home כדי לסמן שהבקשה הושלמה.

- כאשר משתמשים בטרנזקציית ReadNoSnp מ-Home ל-Subordinate, על Home לקבל הודעת ReadReceipt מה-Subordinate או לחכות לתגובת CompAck מה-Requester, בהתאם למבנה הטרנזקציה.

על ה-Requester להתעלם ממצב ה-cache בתגובת CompData או DataSepResp ל-ReadNoSnp ולהניח באופן מרומז שמצב ה-cache הוא Invalid.

ReadNoSnpSep

ה-ReadNoSnpSep היא בקשת קריאה הנשלחת מ-Home ל-Subordinate, ובקשה זו מיועדת ל-completer לשלוח רק את תגובת הנתונים, ללא צורך בתגובה להשלמת הטרנזקציה.

סוג זה של טרנזקציה משמש כאשר התגובות עבור השלמת הקריאה והנתונים הנקראים נשלחות בנפרד.כלומר, תהליך הקריאה מושלם בשני שלבים נפרדים – שלב אחד עבור ההשלמה ושלב נוסף עבור שליחת הנתונים.

כאשר משתמשים ב-ReadNoSnpSep מ-Home ל-Subordinate, על ה-Home לקבל הודעת ReadReceipt מה-Subordinate.

ReadClean

ה-ReadClean היא בקשת קריאה לאזור כתובת snoopable במטרה לקבל עותק נקי של cache line. בקשה זו מיועדת לשימוש כאשר ה-Requester מקצה את השורה ל-cache שאינו תומך בשורות מלוכלכות, כמו ה-Instruction cache (לרוב הוא שומר רק מידע אם ה-cache line היא valid או invalid). במצב זה, על הנתונים שמתקבלים להיות במצבי UC או SC בלבד.

הטרנזקציה מחייבת את השדה ExpCompAck להיות דלוק (כלומר מחייבת לשלוח CompAck בסוף הטרנזקציה).

ReadShared

ה-ReadShared היא בקשת קריאה לאזור כתובת snoopable, ומשמשת לביצוע פעולת load של cache line. הבקשה מאפשרת ל-Requester לקבל עותק של cache line לצורך קריאה בלבד, במצבי UC, UD, SC או SD.

במקרים שבהם ל-Requester יש אפשרות לקבל נתונים במצב SD, נעשה שימוש ב-ReadShared במקום ב-ReadNotSharedDirty, שמיועדת למקרים בהם אין תמיכה בקבלת נתונים במצב SD.

ReadUnique

ה-ReadUnique היא בקשת קריאה לאזור כתובת snoopable, ומשמשת לביצוע store ל-cache line. השורה שתתקבל תינתן ל-Requester רק במצבים UC או UD.

הטרנזקציה מחייבת את השדה ExpCompAck להיות דלוק (כלומר מחייבת לשלוח CompAck בסוף הטרנזקציה).

MakeReadUnique

ה-MakeReadUnique היא בקשת קריאה לאזור כתובת snoopable, שמטרתה לקבל עותק ייחודי של cache line. בקשה זו אופיינית כאשר ל-Requester יש עותק משותף של cache line, והוא מעוניין לקבל הרשאה לאחסן את השורה כעותק ייחודי.

כאשר ה-Requester מקבל בקשת snoop שגורמת ל-invalidate, אם יש צורך לשמור את הנתונים, מובטח ש-Requester יקבל עותק ייחודי של השורה בלי הצורך להוציא מחדש בקשה.

הטרנזקציה מחייבת את השדה ExpCompAck להיות דלוק (כלומר מחייבת לשלוח CompAck בסוף הטרנזקציה).

טרנזקציות Dataless

צמתי Requester משתמשים בטרנזקציות Dataless כדי לבצע פעולות קוהרנטיות מבלי להעביר נתונים ל-Requester או ממנו.

תגובות להשלמת טרנזקציות Dataless

תגובת Comp כוללת שדה בשם Resp, שמציין את הפרטים הבאים:

Cache state

השדה Resp מתאר את המצב הסופי שבו יכולה להימצא cache line אצל ה-Requester לאחר ביצוע הטרנזקציה, למעט עבור טרנזקציות מסוג תחזוקת cache. עבור טרנזקציות תחזוקת cache, צריך להתעלם מהשדה והמצב של cache line נשאר ללא שינוי.

CleanUnique

טרנזקציה זו נועדה לאפשר ל-Requester לשנות את מצב ה-cache שלו למצב Unique עבור כתובת נתונה. זה מאפשר ל-Requester לבצע כתיבה ל-cache line המדוברת. התרחיש הנפוץ הוא כאשר ל-Requester יש עותק משותף של ה-cache line, והוא מבקש הרשאה להפוך את העותק הזה ל-Unique כדי שיוכל לכתוב עליו. כל עותק מלוכלך של אותו cache line ב-cache-ים שביצעו snoop, ייכתב בחזרה לזיכרון כדי לשמור על הקוהרנטיות.

הטרנזקציה מחייבת את השדה ExpCompAck להיות דלוק (כלומר מחייבת לשלוח CompAck בסוף הטרנזקציה).

MakeUnique

טרנזקציה זו מאפשרת ל-Requester לקבל בעלות על שורת ה-cache מבלי לדרוש תגובת נתונים. כלומר, ללא העברת תוכן ה-cache line. הטרנזקציה מבוצעת כאשר ה-Requester מתכוון לכתוב את כל תוכן ה-cache line, לכן אין צורך להעביר נתונים קיימים. כל עותק מלוכלך של השורה ב-cache-ים אחרים יהפוך ל-invalid מבלי להעביר את הנתונים (כלומר נתונים מלוכלכים ימחקו).

הטרנזקציה מחייבת את השדה ExpCompAck להיות דלוק (כלומר מחייבת לשלוח CompAck בסוף הטרנזקציה).

טרנזקציות Dataless לתחזוקת cache

פעולות תחזוקת cache (המכונות גם CMO - Cache Maintenance Operations) משמשות לניהול המבנה והמצב של ה-cache-ים בעזרת תוכנה, על מנת לשמור על עקביות הזיכרון במערכת.

CleanShared

תגובה לטרנזקציית CleanShared מבטיחה שכל עותקי ה-cache של ה-cache line הרלוונטית ישתנו למצב clean. אם יש עותקים מלוכלכים ב-cache-ים, הם ייכתבו בחזרה לזיכרון כדי לשמור על עקביות. כך מבטיחים שמצב ה-cache-ים מסונכרן עם הזיכרון.

CleanInvalid

תגובה לטרנזקציית CleanInvalid מבטיחה שכל העותקים המאוחסנים של ה-cache line המסוים ייכנסו למצב Invalid, כלומר לא יהיה ניתן להשתמש בהם עוד. כמו כן, עותקים מלוכלכים יאולצו להיכתב בחזרה לזיכרון כדי למנוע אובדן נתונים.

MakeInvalid

תגובה לטרנזקציית MakeInvalid מבטיחה שכל עותקי ה-cache של השורה המדוברת יהפכו ל-invalid. פעולה זו מתירה למחוק את העותקים המלוכלכים מה-cache מבלי לשמור אותם בזיכרון.

טרנזקציות כתיבה

טרנזקציות כתיבה מעבירות נתונים מצומת Requester לצומת Completer, אשר יכול להיות cache ברמה הבאה, זיכרון או רכיב היקפי כלשהו. סוג הנתונים המועבר, בין אם הוא קוהרנטי או לא קוהרנטי, תלוי בסוג הטרנזקציה שבוצעה.

תגובת WriteData

שדה Cache state

שדה זה מציין את מצב cache line לפני שנשלחת תגובת WriteData. מצב זה יכול להשתנות לעומת המצב שבו ה-cache line הייתה בעת שליחת בקשת הטרנזקציה המקורית. שינוי זה עשוי להתרחש אם בקשת snoop שמופנית לאותה כתובת התקבלה אצל ה-Requester לאחר שליחת בקשת הטרנזקציה המקורית, אך לפני שה-Requester שולח את תגובת WriteData.

לדוגמה, כש-Requester מסוים מבצע כתיבה ל-cache line במצב UD, לפני שהוא שולח את המידע, ולאחר שהוא קיבל תגובה עם אישור על הקצאה, הוא משנה את ה-cache line ל-I ושולח את התגובה CBWrData_UD_PD.

שדה Pass Dirty

שדה זה מציין אם האחריות לעדכן את הזיכרון עוברת מה-Requester. במידה והאחריות להעברת הנתונים המלוכלכים לזיכרון עוברת ל-Requester, הדבר מסומן באמצעות הסיומת _PD בשם התגובה.

סידור בטרנזקציות כתיבה

ב-CHI, טרנזקציות מסודרות בהתאם ל-Endpoint Order ו-Request Order, והם מוגדרים כך:

- **Endpoint Order:** שומר על הסדר של טרנזקציות שמגיעות מ-Requester יחיד, המיועדות לטווח כתובות יחיד של Subordinate. לדוגמה, ב-Endpoint Order, מספר פעולות גישה למכשיר מונפקות ל-register bank של Subordinate.

- **Request Order:** שומר על הסדר של טרנזקציות שמגיעות מ-Requester יחיד, לאותה כתובת ספציפית. סדר זה נדרש, למשל, כאשר ישנן מספר בקשות שמונפקות לכתובת חופפת שהיא non-cacheable, כמו Device-GRE.

הערה: כאשר מוגדר סדר Request Order, הסדר של Endpoint Order משתמע אוטומטית.

רק סוגי בקשות מסוימים יכולים לנצל את סדרי ה-Request Order וה-Endpoint Order. סוגי הבקשות האלה כוללים:

- בקשות מסוג ReadNoSnp ו-ReadOnce:
  - ה-Requester מנפיק בקשה מסוג ReadNoSnp או ReadOnce שדורשת שמירה על סדר.
  - ה-Subordinate מקבל את הבקשה ומחזיר תשובה בהודעת ReadReceipt. הודעה זו מאשרת שניתן להנפיק את הבקשה הבאה ששומרת על הסדר.
  - באמצעות הנפקת הודעת ReadReceipt, ה-Subordinate מבטיח כי הבקשות יישמרו ויטופלו לפי סדר קבלתן.

- בקשות מסוג WriteNoSnp ו-WriteUnique מטופלות כך:
  - ה-Requester מגיש בקשה מסוג WriteNoSnp או WriteUnique, הדורשת שמירה על סדר (Ordered).
  - ה-Subordinate מגיב בהודעת DBIDResp, המאותתת שהוא מוכן לקבל את ההודעה. הודעת DBIDResp מסמנת שיש slot פנוי ב-Data Buffer לקליטת נתוני הכתיבה, ומאפשרת ל-Requester להמשיך ולהנפיק את הבקשה המסודרת הבאה.
  - הנפקת הודעת DBIDResp מצד ה-Subordinate מבטיחה שהבקשות ישמרו ויטופלו לפי סדר קבלתן.

סדר האירועים במצב כזה מתנהל כך:

1. ה-Requester יוזם בקשת קריאה מספר 1 ל-Subordinate, כאשר מוגדר ReqOrder.
2. ה-Requester מגיש גם בקשת קריאה מספר 2 לאותו Subordinate, עם ReqOrder מוגדר, אך נחסם מלשלוח אותה כי בקשה מספר 1 עדיין ממתינה להשלמה.
3. ה-Subordinate מגיב להודעה של בקשת קריאה מספר 1 עם ReadReceipt, המאשר שהבקשה התקבלה.

4. לאחר מכן 2 האירועים הבאים יכולים לקרות בכל סדר:
   1. ה-Requester שולח את בקשת קריאה מספר 2 ל-Subordinate.
   2. ה-Subordinate משיב ל-Requester עם נתוני הקריאה עבור בקשת קריאה מספר 1.

שדה הקצאת מידע DBID

ה-DBID (קיצור של Data Buffer ID) מופיע רק בהודעות תשובות ומידע. מזהה זה משמש את צומת היעד כדי להודיע על זמינות לקבלת נתוני כתיבה, וכדי להקצות טרנזקציות הדורשות Completion Acknowledgement.

- כאשר מדובר בכתיבות, ה-Requester אינו רשאי לשלוח נתוני כתיבה (Write Data), עד לקבלת ערך DBID בתגובה מה-Completer.

- בחלק מטרנזקציות הקריאה, מתבצע Completion Acknowledgement, שבו ה-Requester מציין שהוא קיבל את נתוני הקריאה. במהלך החזרת נתוני הקריאה ל-Requester, מועבר גם ערך DBID, שה-Requester ישתמש בו כדי לשלוח הודעת השלמה.

תגובת DBIDResp

- נשלחת כדי להודיע ל-Requester שישנם משאבים זמינים לקבלת נתוני הכתיבה (WriteData).
- מציינת שה-completer מספק ערבויות מסוימות לגבי סדר הטרנזקציות.

טרנזקציות כתיבה מיידיות

טרנזקציות כתיבה מיידיות, הידועות גם כ-Non-CopyBack Write, הן סוג מיוחד של טרנזקציות כתיבה, בהן הנתונים מועברים מצומת Requester ישירות לצומת Home, מבלי לקבל תחילה בעלות קוהרנטית על הנתונים. טרנזקציות אלו משמשות להעברת נתונים ישירות. לדוגמה, מהצומת Home לצומת Subordinate, מבלי לעבור תהליך של קבלת בעלות קוהרנטית לפני כן. טרנזקציות אלו עשויות לדרוש ביצוע Snoop מול agent-ים אחרים במערכת, כדי לשמור על עקביות הנתונים.

WriteNoSnpFull

טרנזקציה זו מתבצעת כאשר יש צורך לכתוב שורת cache מלאה של נתונים מצומת Requester לאזור כתובת שלא ניתן לבצע בו snooping, או כאשר יש צורך לכתוב שורת cache מלאה של נתונים מצומת Home לצומת Subordinate. נתונים מועברים בשלמותם, ואין צורך ב-snooping במהלך הפעולה.

WriteNoSnpPtl

כתיבה חלקית (Ptl - partial) לשורת cache line, מתבצעת מצומת הבקשה לאזור כתובת שאינו ניתן ל-Snoop, או מצומת Home לצומת Subordinate. הכתיבה נעשית עבור בתים מסוימים ב-cache line.

WriteUniqueFull

פעולה זו מתבצעת כאשר יש צורך לכתוב שורת cache מלאה לאזור כתובת ניתן ל-Snoop. במקרה זה, שורת ה-cache נכתבת לזיכרון או ל-cache ברמה הבאה, כאשר שורת ה-cache אצל ה-Requester היא במצב Invalid.

WriteUniquePtl

פעולה זו דומה ל-WriteUniqueFull, אך כאן הכתיבה מתבצעת עבור חלק מהבתים בלבד של שורת ה-cache.

WriteUniqueZero

כתיבה לאזור כתובת הניתן ל-Snoop, כאשר כל הבתים בשורת ה-cache נכתבים עם ערך אפס. פעולה זו אינה מעבירה נתונים אמיתיים אלא מציינת שהשורה מכילה אפסים.

טרנזקציות CopyBack

טרנזקציות CopyBack הן סוג מיוחד של טרנזקציות כתיבה, שבהן מועברים נתונים קוהרנטיים מ-cache אחד ל-cache אחר או לזיכרון ברמה הבאה. טרנזקציות אלו מתבצעות כאשר יש צורך להחזיר נתונים מלוכלכים מ-cache לזיכרון, תוך שמירה על הקוהרנטיות של המערכת. חשוב לציין, כי טרנזקציות CopyBack לא דורשות ביצוע snooping ל-agent-ים אחרים במערכת, מכיוון שהן מתמקדות בהעברת נתונים שנמצאים במצב מלוכלך.

WriteBackFull

פעולה זו מתבצעת כאשר יש צורך להחזיר שורת cache מלאה של נתונים מלוכלכים לזיכרון, או ל-cache ברמה הבאה. אם הנתונים מועברים באמצעות CopyBackWriteData_I, אז ייתכן ויש צורך לדלג על חלק מהביטים ב-cache line.

WriteBackPtl

פעולה זו מיועדת להחזיר באופן חלקי שורת cache של נתונים מלוכלכים לזיכרון או ל-cache ברמה הבאה.

WriteCleanFull

בפעולה זו מוחזרים נתונים מלוכלכים לזיכרון או ל-cache ברמה הבאה, וגם צריך לשמור עותק במצב clean של שורת ה-cache המקורית. אם הטרנזקציה מתבצעת עם CopyBackWriteData_I, אז ייתכן ויש צורך לדלג על חלק מהביטים ב-cache line.

בקשות Snoop

ה-Interconnect יוצר בקשה מסוג Snoop, או בתגובה לבקשה מה-Requester, או בעקבות טריגר פנימי, כגון פעולה של cache או פעולת תחזוקה של מסנן snoop. כל טרנזקציית Snoop, למעט טרנזקציית SnpDVMOp, מתמקדת בנתונים המאוחסנים ב-cache של ה-Requester.

הערה: המונח Snoopee מתייחס ל-node שנשאל או נחקר במהלך פעולת snoop, זה ה-node שמבצעים עליו את ה-snooping והוא היעד של הודעת ה-snoop (שולחים לו את ההודעה).

תגובת Snoop

שדה Cache state

שדה זה מייצג את המצב הסופי של ה-cache line בצומת שמבצעים עליה את פעולת ה-snoop. כלומר, לאחר ביצוע פעולת ה-snoop, השדה הזה מגדיר מה יהיה המצב של ה-cache line בצומת הנבדקת.

לדוגמה, Requester מקבל בקשת snoop של SnpMakeInvalid (כלומר צריך להפוך את ה-cache line למצב invalid). ה-cache line שיש לו לפני שהוא מקבל את הבקשה נמצאת במצב SC, וכשהוא מקבל את הבקשה, הוא הופך את ה-cache line ל-I, ומגיב עם הודעת SnpResp_I.

שדה Pass Dirty

שדה זה מציין שהאחריות לעדכון הזיכרון עבור הנתונים מועברת ל-Requester (שיזם את הטרנזקציה המקורית) או ל-interconnect. השדה הזה קיים רק כאשר תגובת ה-snoop כוללת נתונים. כלומר, כאשר הצומת ששולחת את התגובה שולחת גם את המידע הנדרש. כש-Pass Dirty דולק, זה מוצג על ידי _PD בשם התגובה.

טרנזקציות Snoop

SnpClean & SnpCleanFwd

בקשה מסוג Snoop לקבלת עותק של ה-cache line במצב Clean, תוך שמירה על עותק במצב משותף. אין להשאיר את ה-cache line במצב Unique.

SnpNotSharedDirty& SnpNotSharedDirtyFwd

בקשה מסוג Snoop לקבלת עותק של ה-cache line במצב SharedClean, תוך שמירה על עותק במצב Shared. אין להשאיר את ה-cache line במצב Unique.

הערה: התגובה הזאת מעניינת, כי היא נוספה ב-CHI-B כדי לאפשר תמיכה בפרוטוקול MESI שבו אסור מצב SD (זהה ב-MOESI למצב Owned שלא קיים ב-MESI). בנוסף, כדי לאפשר תמיכה נוספה גם הטרנזקציה ReadNotSharedDirty.

SnpShared & SnpSharedFwd

בקשה מסוג Snoop לקבלת עותק של ה-cache line במצב Shared, תוך שמירה על עותק במצב Shared. אין להשאיר את ה-cache line במצב Unique.

SnpUnique & SnpUniqueFwd

בקשה מסוג Snoop לקבלת עותק של ה-cache line במצב Unique, תוך ביטול עותקים שמאוחסנים ב-cache. חייב להתרחש שינוי במצב ה-cache line ל-invalid. אם יש ל-Snoopee עותק במצב מלוכלך, הוא מחויב להעביר גם את המידע ל-Requester המבקש.

SnpCleanShared

בקשה מסוג snoop המיועדת להסיר כל עותק מלוכלך של ה-cache line בצומת ה-Snoopee. אין להשאיר את ה-cache line במצב מלוכלך.

SnpCleanInvalid

בקשה מסוג snoop המיועדת לבטל את תוקף ה-cache line בצומת ה-Snoopee, ולהשיג כל עותק מלוכלך. בקשה זו עשויה להיווצר גם על ידי ה-Interconnect ללא בקשה מתאימה מה-Requester. חייבת להתרחש פעולה של שינוי במצב ה-cache line ל-invalid.

SnpMakeInvalid

בקשה מסוג snoop המיועדת לבטל את תוקף ה-cache line בצומת ה-Snoopee, ובכך לבטל כל עותק מלוכלך:

- אין החזרת נתונים עם תגובת Snoop; נתונים מלוכלכים נמחקים.
- חייבת להתרחש פעולה של שינוי במצב ה-cache line ל-invalid.

SnpQuery

בקשה מסוג Snoop שמטרתה לבדוק את מצב ה-cache line ב-Requester:

- ה-Home יכול לשלוח בקשת snoop מסוג SnpQuery ללא כל בקשה מתאימה מה-Requester.
- תגובת Snoop חייבת לכלול את המצב המדויק של ה-cache line ב-snoopee המיועד.
- אין להחזיר נתונים עם תגובת snoop.
- טרנזקציית SnpQuery אינה אמורה לשנות את מצב ה-cache line ב-Snoopee.

מעברי cache שקטים

ה-cache יכול לשנות את מצבו בעקבות אירועים פנימיים, מבלי לשלוח הודעה לשאר רכיבי המערכת. במקרים מסוימים, קיימת אפשרות (אך אין חובה) להוציא טרנזקציה שמודיעה על שינוי המצב. כאשר טרנזקציה כזו מונפקת, המעבר של מצב ה-cache הופך לגלוי ל-Interconnect, ולכן אינו נחשב למעבר שקט. לדוגמה, יכול להתרחש מעבר מצב cache שקט בתוך ה-RN-F, בפעולת store: ממצב UC ל-UD.

סוגי טרנזקציות ומנגנונים אחרים

יש עוד הרבה סוגים של טרנזקציות, אופטימיזציות ומנגנונים שקיימים ב-AMBA CHI, אבל אני לא אסביר על כולם במאמר הזה, כן אפרט בקצרה על עוד מספר מנגנונים:

Direct Cache Transfer (DCT)

מנגנון Direct Cache Transfer נועד להפחית את זמן ההשהיה שנגרם עקב פעולות snoop. מנגנון זה דומה ל-(Direct Memory Transfer (DMT, אך מתמקד ב-snoop-ים. ה-DCT מאפשר לנתוני ה-snoop לעקוף את צומת ה-Home, ולעבור ישירות מצומת Requester מסוים, אל ה-Requester המקורי שהוציא את הבקשה. מה שמאיץ את ביצועי המערכת במקרים שבהם הנתונים צריכים לחזור מ-Requester-ים שונים. מקרי שימוש שבהם מנגנון DCT מועיל, כוללים מקרים של מנעולים וניהול עומסי עבודה במודל יצרן-צרכן, בהם מהירות הגישה לנתונים היא קריטית.

לדוגמה, ניתן להשוות בין המסלול שהנתונים עוברים בבקשת קריאה רגילה לעומת מסלול הנתונים בעת שימוש ב-DCT:

- ללא DCT:
  1. מעבד A שולח בקשת קריאה ל-Home.
  2. הבקשה מובילה ל-cache miss ב-Home.
  3. ה-Home שולח בקשת snoop למעבד B, שמחזיק את ה-cache line הרלוונטית.
  4. מעבד B מחזיר את הנתונים לצומת ה-Home.
  5. ה-Home מחזיר את הנתונים למעבד A, ה-Requester המקורי.
- עם DCT:
  1. מעבד A שולח בקשת קריאה ל-Home.
  2. הבקשה מובילה ל-cache miss ב-Home.
  3. ה-Home שולח בקשת snoop למעבד B, שמחזיק את שורת ה-cache הרלוונטית.
  4. מעבד B עוקף את ה-Home ושולח את הנתונים ישירות למעבד A, ה-Requester המקורי.

באמצעות DCT, ניתן לקצר את זמן ההשהיה במקרה של snoop hit, ולשפר את יעילות הגישה לנתונים.

העברת (forward) בקשות Snoop

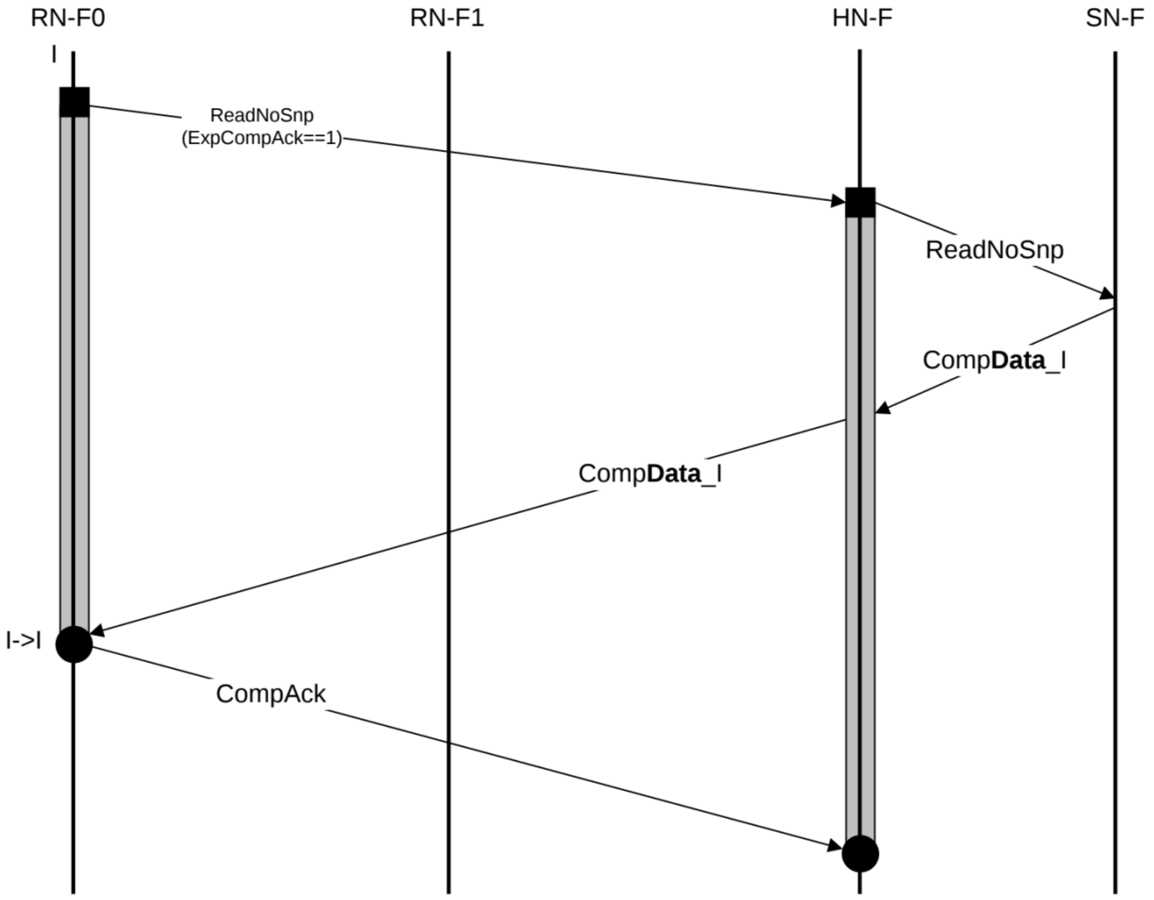
בכדי לאפשר את ה-DCT, הוסיפה CHI-B את מנגנון בקשת Snoop Forwarding. מנגנון זה מאפשר ל-Requester, שהוא המבצע של ה-snoop, לשלוח את נתוני ה-snoop ישירות ל-Requester המקורי במקום דרך ה-Home.

כמעט כל פעולות קריאה הניתנות ל-Snoop, למעט טרנזקציות אטומיות וקריאות בלעדיות, יכולות לנצל את היתרון של ה-DCT לצורך קיצור זמן ההשהיה ושיפור הביצועים.

הדגמת טרנזקציות

טרנזקציית קריאה ReadNoSnp ללא DMT או DCT

באיור B5.5 מוצגת דוגמה ל-flow של טרנזקציית ReadNoSnp שאינה עושה שימוש במנגנוני DMT או DCT:



[Figure B5 5 ReadNoSnp transaction flow from AMBA CHI Architecture Specification (developer.arm.com)]

במקרה זה, לטרנזקציית ReadNoSnp מוגדר השדה ExpCompAck בבקשה המקורית, לכן זה מחייב ש-Requester 0 ישלח בסוף התהליך הודעת CompAck. הטרנזקציה אינה גורמת ל-snooping, מכיוון שהנתונים מתקבלים ישירות מהזיכרון דרך ה-Home, זאת משום שהבקשה מוגדרת כ-NoSnp.

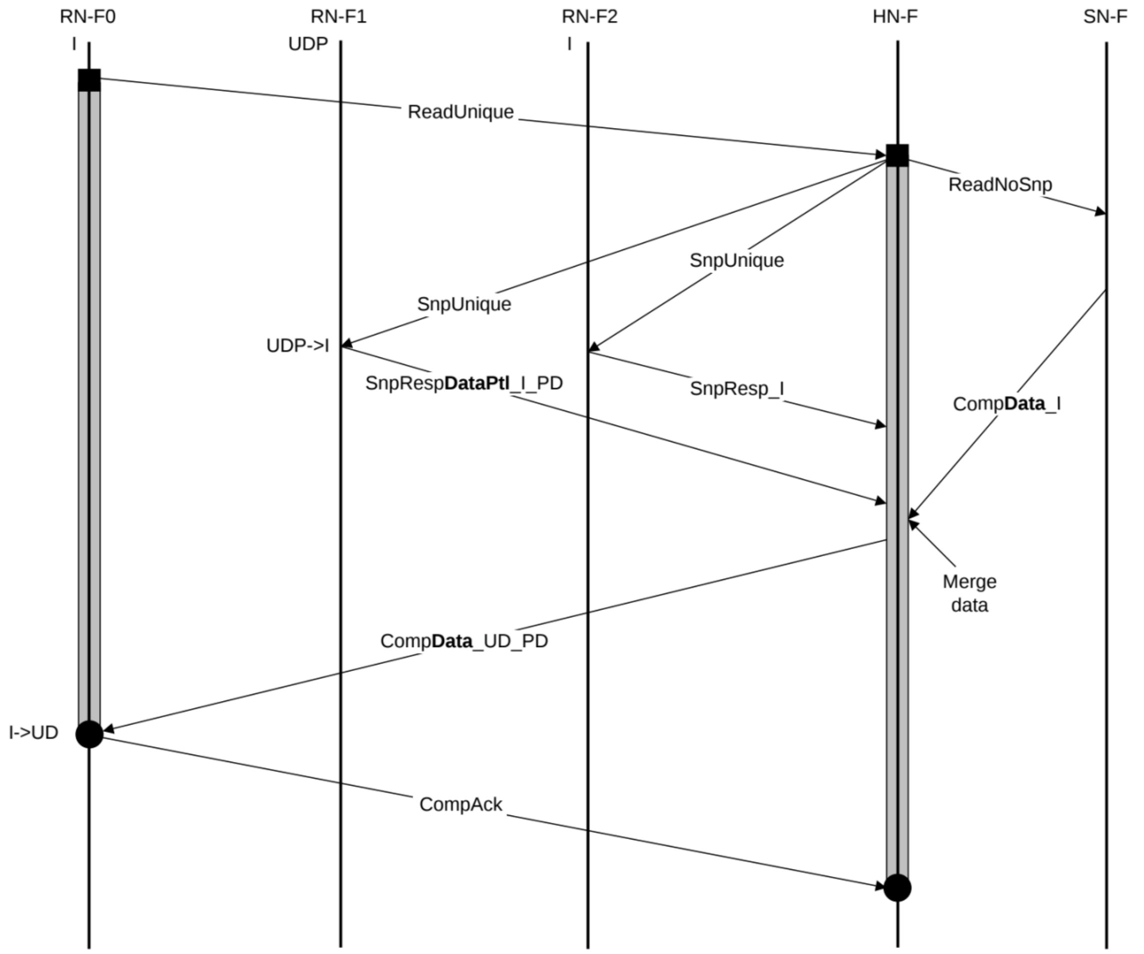
תהליך העבודה של טרנזקציית ReadNoSnp מתבצע בשלבים הבאים:

1. ה-Requester 0 מנפיק בקשת ReadNoSnp, כאשר השדה ExpCompAck מוגדר ל-1. כלומר, ה-Home צריך לצפות לתגובת השלמה בסוף התהליך.
2. ה-Home מקבל את הבקשה, מקצה את הבקשה ושולח את בקשת ReadNoSnp ל-Subordinate.
3. ה-Subordinate מחזיר תגובת CompData_I אל ה-Home.
4. ה-Home מחזיר את הנתונים שהתקבלו ל-Requester 0. (אם השדה ExpCompAck לא היה מוגדר בבקשה המקורית של ReadNoSnp, ה-Home היה יכול לבטל את הקצאת הבקשה בשלב זה).

5. ה-Requester 0 מסיים את התהליך על ידי ביטול הקצאת הבקשה, ושולח את הודעת CompAck ל-Home. בתשובה שמגיעה בהודעה CompData_I מה-Home, אז Requester 0 לא יכול לשמור את הנתונים בצורה קוהרנטית (בגלל שלא נעשה snoop בתהליך הטרנזקציה). לכן, גם באיור, המידע שמועבר לא צריך להישמר ב-Requester 0 בסוף כל הטרנזקציה, כתוצאה מכך הוא "עובר" ממצב invalid ל-invalid.
6. ה-Home מקבל את תגובת CompAck מ-Requester 0 ומבטל את הקצאת הבקשה.

טרנזקציית קריאה עם תגובת snoop ונתונים חלקיים ללא עדכון זיכרון

דוגמה לסוג טרנזקציה כזו היא ה-ReadUnique. איור B5.6 מציג את זרימת הטרנזקציה, כאשר הנתונים המועברים מסומנים בהדגשה:

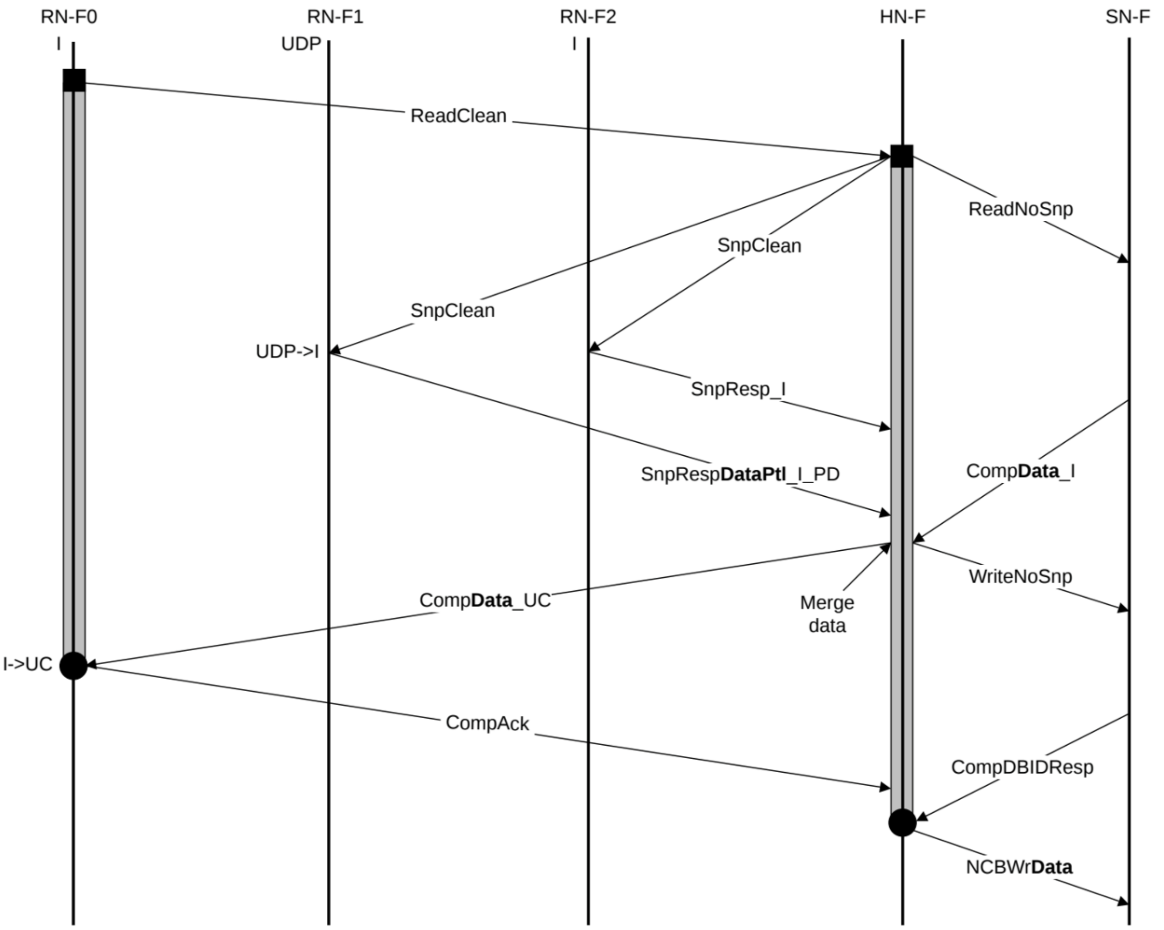


[Figure B5 6 ReadUnique with partial data snoop response from AMBA CHI Architecture Specification (developer.arm.com)]

1. שלב הבקשה: Requester 0 שולח בקשת ReadUnique אל ה-Home.
2. הפצת הבקשה: ה-Home שולח בקשות ReadNoSnp אל ה-Subordinate ובמקביל שולח בקשות SnpUnique אל Requester 1 ו-Requester 2, בגלל שיכול להיות שיש להם עותקים של הנתונים ב-cache שלהם.
3. תגובה מ-Requester 1: ה-Requester 1, שיש לו את הנתונים במצב UDP, משנה את מצב ה- cache line שלו למצב I ושולח ל-Home תגובת SnpRespDataPtl_I_PD. כלומר, תגובת snoop עם נתונים חלקיים ומצב של Invalid.
4. תגובה מ-Requester 2: ה-Requester 2, שאין לו את הנתונים, מחזיר תגובת SnpResp_I. כלומר, תגובת snoop פשוטה המציינת שהנתונים אינם נמצאים ב-cache שלו (או שיש לו נתונים במצב Invalid אבל זה כאילו אין לו נתונים).
5. תגובה מ-Subordinate: במקביל, ה-Subordinate מחזיר תגובת CompData_I לבקשת ReadNoSnp משלב 2.
6. מיזוג הנתונים: ה-Home ממזג את הנתונים שהתקבלו מ-Requester 1 (הנתונים החלקיים), יחד עם הנתונים שהתקבלו מה-Subordinate.
7. שליחת הנתונים ל-Requester 0: לאחר המיזוג, ה-Home שולח את הנתונים המלאים. כעת, במצב CompData_UD_PD , ל-Requester 0, שמשנה את מצב ה-cache line שלו מ-I ל-UD.
8. השלמת הטרנזקציה: Requester 0 שולח ל-Home תגובת CompAck, המאשרת את קבלת הנתונים וסיום הטרנזקציה בהצלחה. ובשלב זה, ה-Home יכול לבצע deallocate לבקשה. (Requester 0 מחויב לשלוח תגובת CompAck, בגלל שלבקשת ReadUnique חייב השדה ExpCompAck להיות דלוק).

טרנזקציית קריאה עם תגובת snoop עם נתונים חלקיים ועדכון זיכרון

דוגמה ל-flow של טרנזקציה מסוג זה היא טרנזקציית ReadClean. באיור B5.7 ניתן לראות את ה-flow המלא של הטרנזקציה:



[Figure B5 7 ReadClean with partial data snoop response from AMBA CHI Architecture Specification (developer.arm.com)]

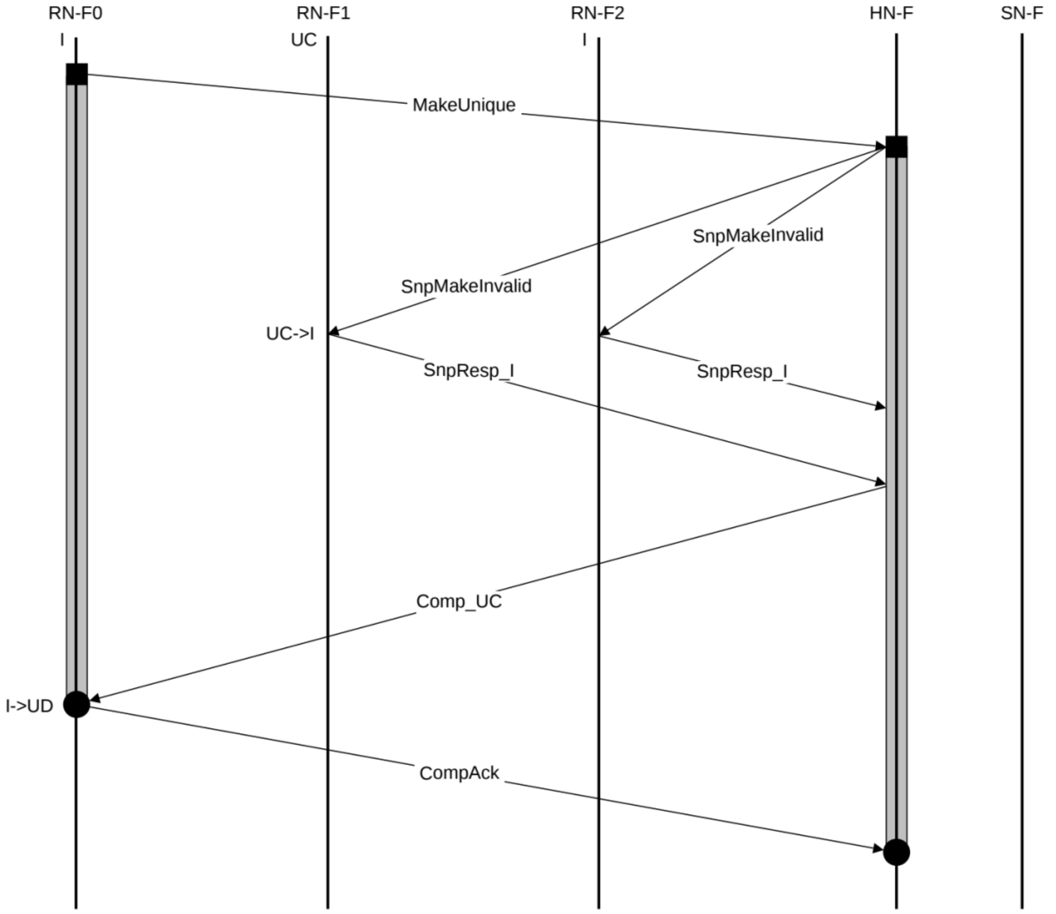
השלבים בטרנזקציית ReadClean עם תגובת Snoop עם נתונים במצב Ptl (חלקי):

1. שליחת בקשה: Requester 0 שולח בקשת ReadClean אל ה-Home.
2. הפצת בקשות: ה-Home מפיץ בקשות ReadNoSnp אל ה-Subordinate, וגם מפיץ בקשות SnpClean אל Requester 1 ו-Requester 2.
3. תגובת Requester 1: ה-Requester 1 מקבל את בקשת ה-Snoop, מעביר את ה-cache line ממצב UDP למצב I, ולאחר מכן מחזיר ל-Home תגובת SnpRespDataPtl_I_PD, הכוללת נתונים חלקיים.
4. תגובת Requester 2: צומת Requester 2 מקבל את הבקשה ומחזיר ל-Home תגובת SnpResp_I, שמשמעותה שאין לו נתונים להחזיר (מצב Invalid).
5. תגובת ה-Subordinate: במקביל, ה-Subordinate מחזיר ל-Home את התגובה CompData_I, הכוללת נתונים מלאים במצב Invalid.

6. מיזוג נתונים: ה-Home מקבל את התגובות מה-Subordinate ומ-Requester 1, וממזג את הנתונים שהתקבלו (הנתונים המלאים מה-Subordinate אבל לא כל המידע בהם עדכני והנתונים החלקיים מ-Requester 1).
7. שליחת הנתונים: ה-Home שולח CompData_UC ל-Requester 0, ומנפיק בקשת WriteNoSnp אל ה-Subordinate כדי לעדכן את הזיכרון.
8. כש-Requester 0 מקבל את הנתונים מה-Home, הוא מעביר את ה-cache line שלו ממצב I למצב UC. לאחר מכן, הוא משדר תגובת CompAck ל-Home כדי לציין שהטרנזקציה הושלמה בהצלחה. (Requester 0 מחוייב לשלוח תגובת CompAck, בגלל שלבקשת ReadClean חייב השדה ExpCompAck להיות דלוק).
9. ה-Subordinate מנפיק CompDBIDResp ל-Home כתגובה לבקשה בשלב 7.
10. שליחת נתונים סופיים: ה-Home שולח נתונים ל-Subordinate באמצעות NCBWrData. לאחר מכן, הוא מסיים את הטרנזקציה ומבצע deallocate לבקשה.

טרנזקציית Dataless ללא עדכון זיכרון

דוגמה ל-flow של טרנזקציה מסוג זה היא טרנזקציית MakeUnique, שבה לא מתבצע עדכון בזיכרון, אלא רק תחזוקת מצב ה-cache.



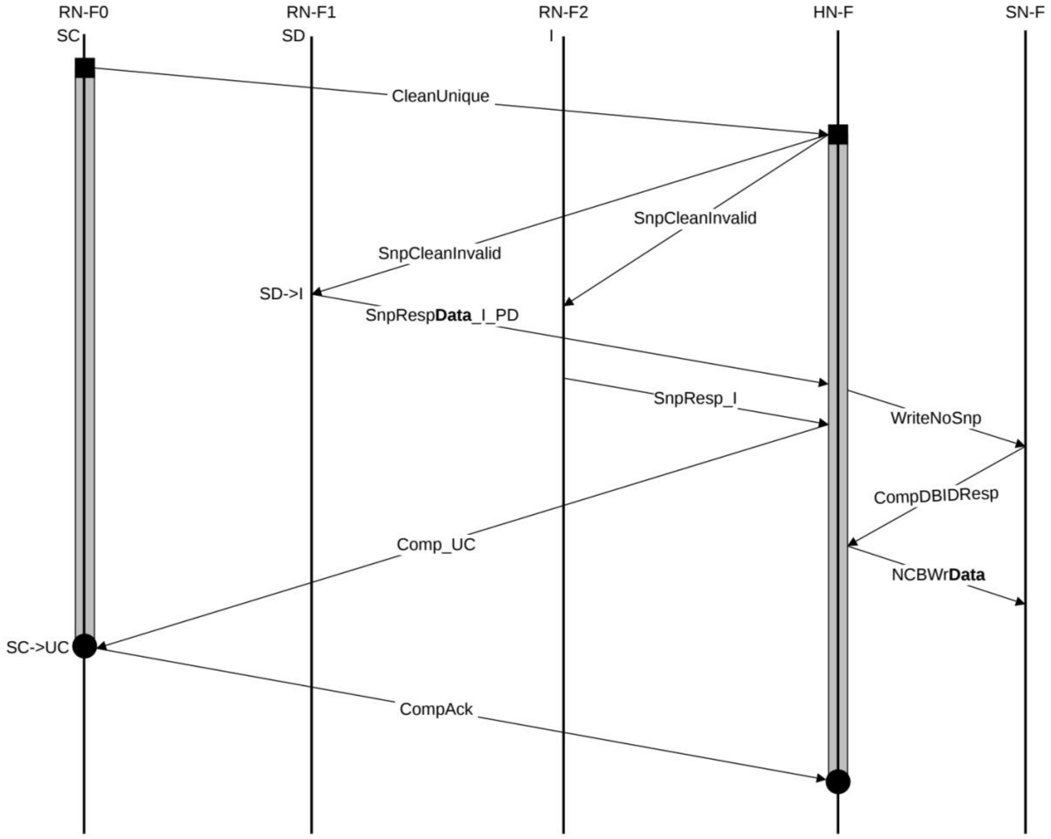
[Figure B5 11 MakeUnique without memory update from AMBA CHI Architecture Specification (developer.arm.com)]

שליחת בקשה: Requester 0 שולח בקשה מסוג MakeUnique אל ה-Home.

1. הפצת בקשות SnpMakeInvalid: ה-Home משדר בקשות SnpMakeInvalid ל-Requester 1 ול-Requester 2, שמטרתן לבטל את תוקף ה-cache line ב-node-ים האלה.
2. כש-Requester 1 ו-Requester 2 מקבלים את הבקשה SnpMakeInvalid (כל אחד בנפרד), הם משנים את המצב ה-cache line מ-UC ל-I. ולאחר מכן כל אחד מהם מחזיר ל-Home תגובות SnpResp_I, שמודיע שה-cache line שלהם במצב Invalid.
3. לאחר קבלת 2 התגובות של Requester 1 ו-Requester 2 בשלב 3, ה-Home שולח ל-Requester 0 תגובת Comp_UC שמציינת שהבקשה הושלמה בהצלחה.
4. כש-Requester 0 מקבל את ה-Comp_UC מה-Home הוא צריך לעדכן את מצב ה-cache line שיש לו מ-I ל-UD, לאחר מכן הוא מאשר את קבלת ההודעה על ידי שליחת תגובת CompAck ל-Home, ובכך מציין שהטרנזקציה הסתיימה בהצלחה. (Requester 0 מחוייב לשלוח תגובת CompAck בגלל שלבקשת ReadClean חייב השדה ExpCompAck להיות דלוק).
5. כשה-Home מקבל את ה-CompAck, הוא יכול לסיים את הטרנזקציה והוא מבצע deallocate לבקשה.

טרנזקציית Dataless עם עדכון זיכרון

דוגמה לטרנזקציה מסוג זה היא CleanUnique, שבה מתבצע עדכון זיכרון אך אין העברת נתונים בטרנזקציה עצמה.



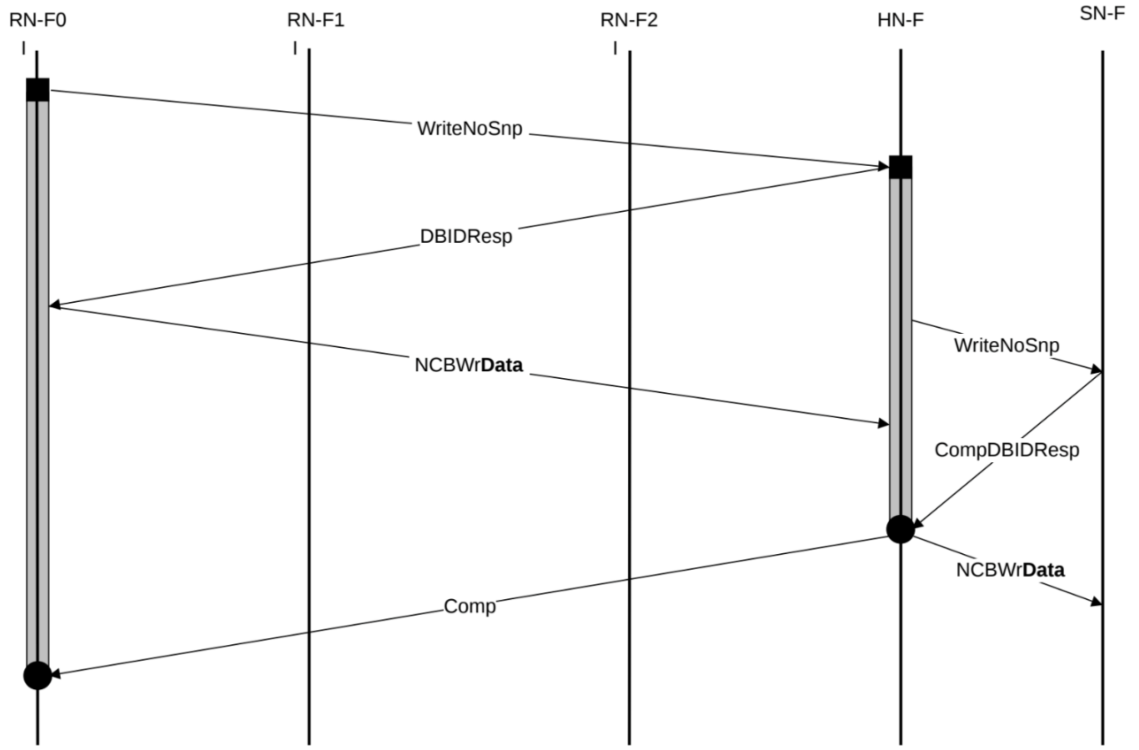
[Figure B5 12 CleanUnique with memory update from AMBA CHI Architecture Specification (developer.arm.com)]

השלבים בזרימת הטרנזקציה CleanUnique עם עדכון זיכרון כפי שמוצגים באיור B5.12:

1. שליחת בקשה: Requester 0 שולח בקשת CleanUnique אל ה-Home.
2. שליחת בקשות SnpCleanInvalid: ה-Home משדר בקשות SnpCleanInvalid ל-Requester 1 ו-Requester 2, שמטרתן לנקות ולבטל את תוקף ה-cache line בצמתים אלו.
3. לאחר שהבקשה מגיעה ל-Requester 1, הוא משנה את המצב של ה-cache line ממצב SD ל-I. לאחר מכן הוא מחזיר ל-Home תגובת SnpRespData_I_PD, המכילה נתונים חלקיים במצב Invalid.
4. כשהתגובה SnpRespData_I_PD מגיעה מ-Requester 1 אל ה-Home, ה-Home משדר בקשת WriteNoSnp ל-Subordinate כדי לעדכן את הזיכרון.
5. תגובת Requester 2: ה-Requester 2 מחזיר ל-Home תגובת SnpResp_I לבקשה משלב 2.
6. כשהתגובה מ-Requester 2 מגיעה אל ה-Home (וכבר בשלב זה הגיעה התגובה מ-Requester 1), כעת כל התגובות שהוא חיכה להן הגיעו, וה-Home משדר ל-Requester 0 תגובת Comp_UC, שמשמעותה שהבקשה הושלמה בהצלחה.
7. בינתיים, ה-Subordinate קיבל את ה-WriteNoSnp משלב 4 , ומשיב ל-Home תגובת CompDBIDResp
8. כשה-Home מקבל את ה-CompDBIDResp מה-Subordinate, הוא שולח ל-Subordinate את הנתונים הסופיים באמצעות NCBWrData.
9. כש-Requester 0 מקבל את התגובה מ-Home, הוא משנה את המצב של ה-cache line מ-SC ל-UC, לאחר מכן, הוא שולח ל-Home תגובת CompAck, המאשרת שהטרנזקציה הסתיימה בהצלחה. (Requester 0 מחוייב לשלוח תגובת CompAck, בגלל שלבקשת ReadClean חייב השדה ExpCompAck להיות דלוק).
10. כשה-Home מקבל את ה-CompAck, הוא יכול לסיים את הטרנזקציה והוא מבצע deallocate לבקשה.

טרנזקציית כתיבה ללא snoop ותגובות נפרדות

האיור B5.15 מציג את ה-flow של טרנזקציית WriteNoSnp, שבה נתונים נכתבים ללא צורך בביצוע snoop.



[Figure B5 15 WriteNoSnp with separate responses from Home Node to Request Node from AMBA CHI Architecture Specification (developer.arm.com)]

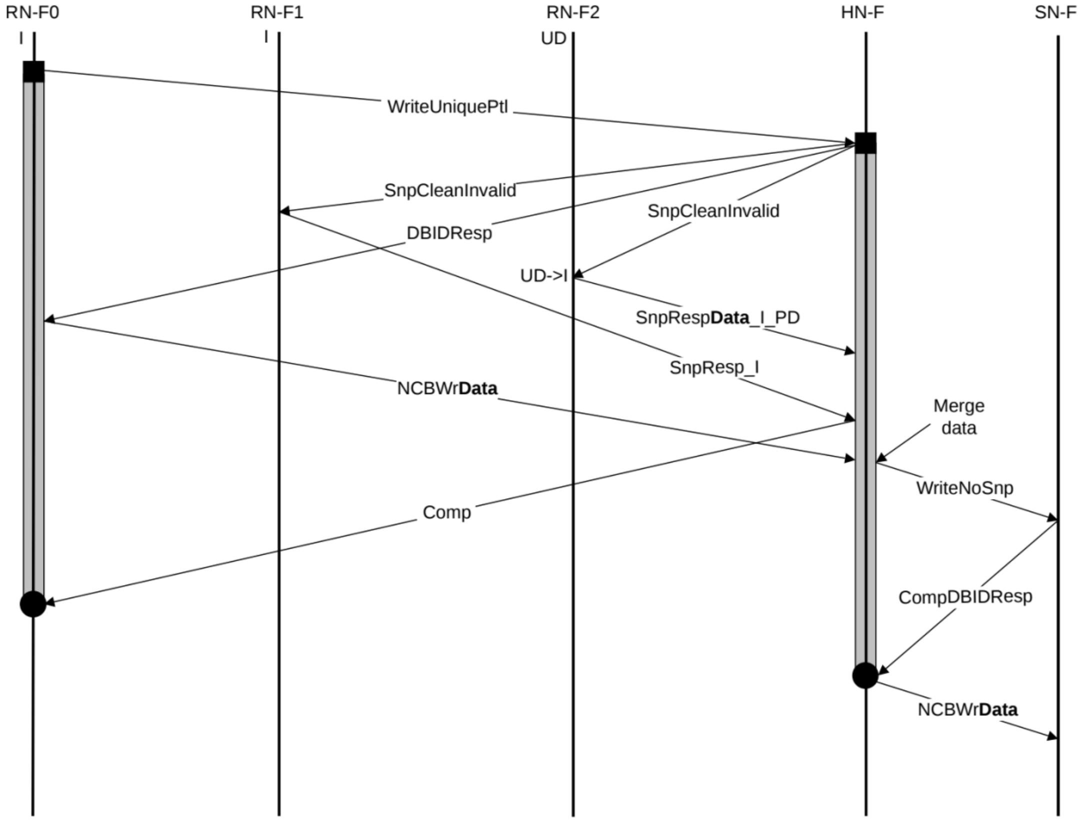
שלבים בזרימת הטרנזקציה WriteNoSnp כפי שמוצגים באיור B5.15:

1. הנפקת הבקשה: Requester 0 שולח בקשה לכתיבה מסוג WriteNoSnp אל ה-Home.
2. לאחר שה-Home מקבל את הבקשה מ-Requester 0, הוא מקצה משאבים לבקשה וגם מחזיר ל-Requester 0 תגובת DBIDResp, ללא Comp.
3. במקביל, ה-Home משדר בקשת WriteNoSnp אל ה-Subordinate.
4. העברת הנתונים: Requester 0 משדר ל-Home את הנתונים הדרושים לכתיבה באמצעות NCWrData.
5. לאחר שה-Subordinate מקבל את בקשת הכתיבה, הוא מחזיר ל-Home את התגובה CompDBIDResp.
6. ה-Home מקבל תגובת CompDBIDResp מה-Subordinate, ולאחר מכן ה-Home משדר את הנתונים הסופיים (NCBWrData) ל-Subordinate כדי לעדכן את הזיכרון.
7. ה-Home שולח ל-Requester 0 את תגובת ה-Comp, המודיעה על השלמת הטרנזקציה. למרות שבדוגמה זו ה-Comp נשלח לאחר קבלת תגובת CompDBIDResp מה-Subordinate, ה-Home רשאי לשלוח את ה-Comp בכל שלב לאחר שקיבל את בקשת ה-WriteNoSnp מ-Requester 0.

8. כש-Requester 0 מקבל מה-Home את הודעת ה-Comp שהוא חיכה לה, המסמנת את סיום התהליך, הוא מבצע deallocate לבקשה.

טרנזקציית כתיבה עם snoop ותגובות נפרדות

דוגמה ל-flow מסוג זה היא טרנזקציית WriteUniquePtl, שבה כתיבה מתבצעת עם הפעלת snoop.



[Figure B5 16 WriteUniquePtl with snoop 1 from AMBA CHI Architecture Specification (developer.arm.com)]

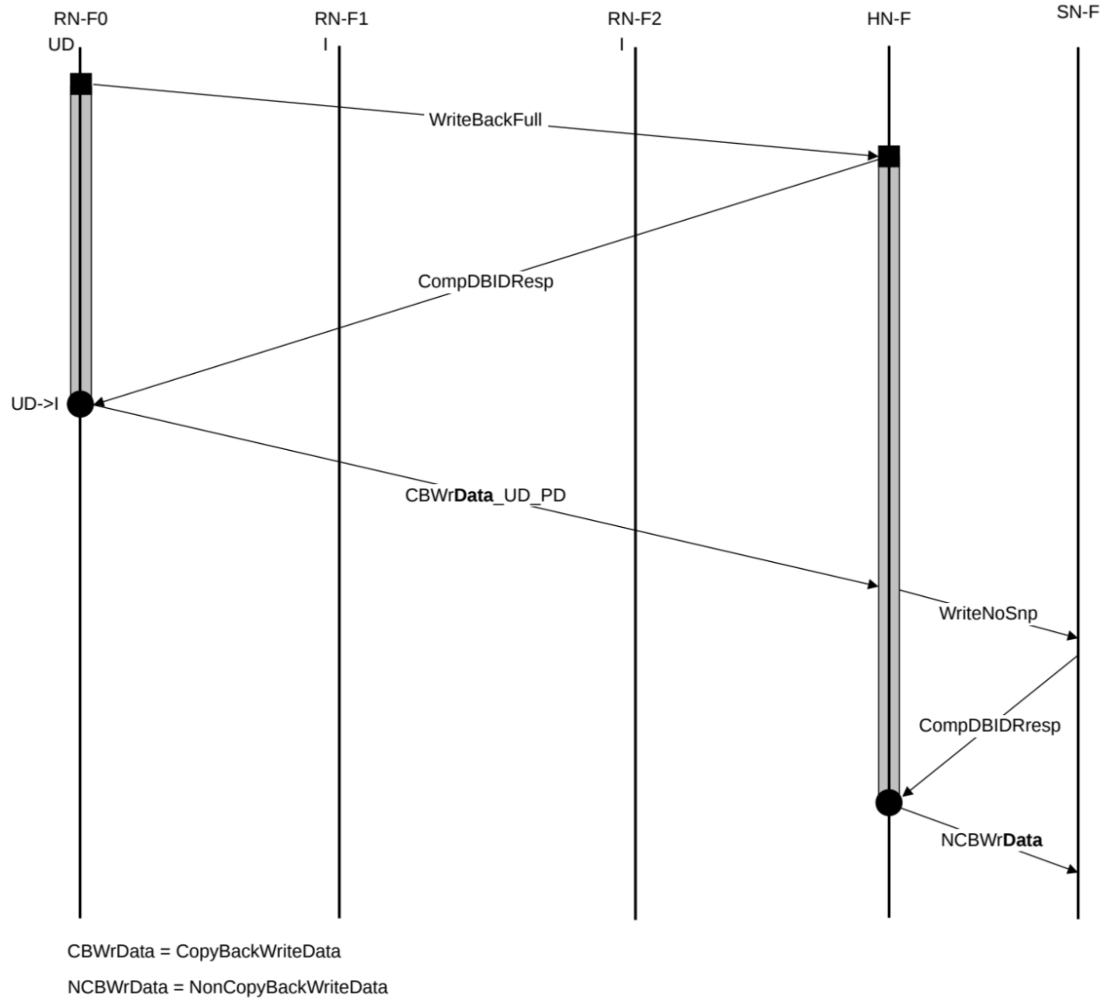
שלבי זרימת הטרנזקציה WriteUniquePtl עם Snoop (איור B5.16):

1. הנפקת הבקשה: Requester 0 שולח בקשת WriteUniquePtl אל ה-Home.
2. שליחת בקשות Snoop ותגובת DBIDResp: ה-Home משדר בקשות SnpCleanInvalid ל- Requester 1 ול-Requester 2, שמטרתן לנקות ולבטל את תוקף ה-cache line ב-node-ים האלו.
3. במקביל, ה-Home שולח ל-Requester 0 תגובת DBIDResp, שמעידה על קבלת הבקשה.
4. כש-Requester 1 מקבל את ה-SnpCleanInvalid, הוא משיב ל-Home עם תגובת SnpResp_I, המציינת שה-cache line נמצא במצב Invalid.
5. כש-Requester 2 מקבל את ה-SnpCleanInvalid מ-Home, הוא משנה את מצב ה-cache line מ-UD ל-I. לאחר מכן, הוא שולח ל-Home תגובת SnpRespData_I_PD, הכוללת את הנתונים המלוכלכים במצב Invalid.
6. כש-Requester 0 מקבל את ה-DBIDResp מ-Home, הוא משדר ל-Home את הנתונים המיועדים לכתיבה באמצעות NCBWrData.

7. ה-Home מקבל SnpRespData_I_PD מ-Requester 2.
8. ה-Home מקבל SnpResp_I מ-Requester 1, בשלב זה הגיעו כל התגובות מבקשות ה-snoop שהוא שלח. לכן, הוא לא צריך יותר לחכות, והוא שולח ל-Requester 0 את תגובת ה-Comp.
9. בינתיים, ה-Home מקבל את ההודעה מ-Requester 0 עם הנתונים לכתיבה, הוא ממזג את נתוני הכתיבה עם ה-cache line המלוכלכת שהתקבלה מ-Requester 2, ושולח בקשת WriteNoSnp אל ה-Subordinate, כדי לעדכן את השורה בזיכרון.
10. ה-Subordinate מחזיר ל-Home את תגובת CompDBIDResp.
11. כש-Requester 0 מקבל מה-Home את הודעת ה-Comp שהוא חיכה לה, המסמנת את סיום התהליך, הוא מבצע deallocate לבקשה.
12. ה-Home שולח ל-Subordinate את הנתונים הסופיים באמצעות NCBWrData, ובכך מסיים את תהליך הכתיבה והוא מבצע deallocate לבקשה.

טרנזקציית כתיבה CopyBack לזיכרון

טרנזקציית WriteBackFull מתארת תהליך שבו נתונים נשמרים מה-cache של ה-Requester לזיכרון או ל-cache ברמה הבאה.



[Figure B5 17 WriteBackFull transaction flow from AMBA CHI Architecture Specification (developer.arm.com)]

שלבים בזרימת טרנזקציית WriteBackFull (איור B5.17):

1. הנפקת הבקשה: Requester 0 שולח בקשת WriteBackFull אל ה-Home.
2. כשה-Home מקבל את הבקשה, הוא מחזיר ל-Requester 0 תגובת CompDBIDResp.
3. לאחר ש-Requester 0 מקבל את התגובה מ-Home, הוא משנה את מצב ה-cache line מ-UD ל-I. לאחר מכן, הוא משדר ל-Home את הנתונים שיש לכתוב באמצעות CBWrData_UD_PD, שכוללים את הנתונים המלוכלכים מה-cache. לאחר מכן, הוא מסיים את הביצוע של התהליך מבחינתו, לכן הוא מבצע deallocate לבקשה.
4. כשה-Home מקבל את הנתונים מ-Requester 0, ה-Home שולח בקשת WriteNoSnp ל-Subordinate.
5. ה-Subordinate שולח ל-Home את תגובת CompDBIDResp.
6. ה-Home שולח את הנתונים הסופיים אל ה-Subordinate באמצעות NCBWrData, ובכך משלים את הכתיבה לזיכרון, ומסיים את הטרנזקציה. לכן, הוא מבצע deallocate לבקשה.

Multi-copy atomicity

דרישת אטומיות

שמירה ב-cache של נתונים משותפים מהווה אופטימיזציה חשובה שמסייעת להפחית את זמן השהייה של גישות לזיכרון במערכות זיכרון משותף. אולם, שמירה ב-cache יוצרת מספר עותקים של נתונים בצמתים שונים ברשת, והצורך לשמור על עדכניות העותקים הללו בכל כתיבה יכול להוות אתגר. קיימת האפשרות לבטל את העותקים הישנים או לעדכן אותם לערך החדש. אך הפצת העותקים והשוני במסלולי הרשת מקשים על ביטול או עדכון של העותקים בצורה אטומית.

לרוב, יש פתרונות אפקטיביים לדרישת אטומיות עבור מערכות המשתמשות ב-invalidating. פתרונות למערכות המשתמשות ב-updating נוטים להיות מסובכים ובלתי יעילים לעומת הפתרונות ל-invalidating.

שיפורים ארכיטקטוניים שמטרתם להתמודד עם זמני השהייה ארוכים בזיכרון, עשויים להשפיע על התנהגות הזיכרון, בכך שהם מאפשרים לפעולות להתרחש מחוץ לסדר המתוכנן, או לגרום להן להיראות כאילו אינן אטומיות.

במערכות מודרניות, שבהן קיימים מספר מעבדים וכל אחד מהם מחזיק עותק של כל זיכרון, אנו נדרשים להשתמש במודלים של זיכרון, עם עותקים בכל מעבד, על מנת לתפוס את ההשפעות הלא אטומיות שנובעות מהימצאות מספר עותקים של מיקום זיכרון יחיד.

מכיוון שהזיכרון אינו מתפקד עוד כעותק לוגי יחיד, יש צורך להרחיב את המושג של פעולות קריאה וכתיבה, על מנת להתמודד עם נוכחותם של מספר עותקים. כתיבה במצבים כאלה עשויה לא להיראות אטומית.

מודל Multi-Copy Atomicity

הרעיון של multi-copy atomicity, דורש שכל שאר הליבות יראו את הכתיבה של ליבה מסוימת באופן לוגי בו-זמנית.

במודל שהוא multi-copy atomicity, יש נקודת זמן אטומית שבה הכתיבה נכנסת לתוקף בזיכרון. לפני נקודת הזמן הזו, אף ליבה אחרת לא יכולה לראות את הערך החדש שנכתב. לאחר נקודה זו, כל הליבות האחרות חייבות לראות את הערך החדש או ערך מכתיבה מאוחרת יותר, אך לא ערך מלפני הכתיבה.

כדי לתמוך ב-multi-copy atomicity, יש צורך לטפל נכון בתרחיש הקרוי Independent Read Independent Write (IRIW) (שנתקל בו עוד במאמרים עתידיים), שבו קריאות וכתיבות למיקומים שונים בזיכרון נעשות בצורה בלתי תלויה. עם זאת, טיפול נכון ב-IRIW אינו מבטיח אוטומטית שמערכת תעמוד בדרישות של multi-copy atomicity.

המשמעות של עיקרון multi-copy atomicity, הוא שניתן להתמקד בסנכרון ברמה המקומית של כל thread, מבלי לנתח את האינטראקציה הכוללת בין כל ה-thread-ים, בתוכנית מרובת-thread-ים. באופן זה, אפשר להבטיח סנכרון נכון של דפוסים כמו MP ו-WRC, באמצעות הנמקות סנכרון מקומיות לכל thread, במקום להחיל שיקולי סנכרון על התוכנית כולה. זה שונה מארכיטקטורות אחרות, כמו IBM Power, בהן המבחן WRC דורש מנגנון נוסף שנקרא cumulative (שיקבל הסבר במאמר אחר בהמשך), כדי להבטיח סנכרון נכון.

מבחינת יישום החומרה, מודל multi-copy atomicity מצריך מנגנונים פחות מורכבים כדי למנוע התנהגויות חריגות כמו אלו שנצפות ב-WRC. אף על פי שהדרישה הזו מטילה עומס נוסף על החומרה, היא מציעה יתרון בכך שהיא יוצרת מודל עקביות פשוט יותר עבור המפתחים.

הקשר ל-Single-Copy Atomicity

במקור, השם של ה-multi-copy atomicity הגיע מ-single-copy atomicity, בגלל שהם דומים מבחינה רעיונית. ה-single-copy atomicity, לא מגדיר את הרגע המדויק שבו הנתונים מתעדכנים. החשיבות היא לוודא שאף מעבד לא יוכל לראות מצב של נתונים מעודכנים חלקית.

ב-multi-copy atomicity אותו הרעיון קיים, פשוט לא לערך בודד, שאסור שיכיל ערבוב של מספר כתיבות. אלא למיקום זיכרון שצריך להיראות זהה לכל המעבדים.

סיבתיות ו-Multi-Copy Atomicity

סיבתיות, למרות שהיא מספקת הבטחה של סדר לוגי של אירועים, אינה בהכרח מצביעה על multi-copy atomicity. דוגמה לכך, היא מצב שבו מספר ליבות חולקות store buffer: הליבות שחולקות את ה- store

buffer יראו את הכתיבה, בעוד שליבות שאינן חולקות אותו לא יראו את הכתיבה. מצב זה מפר את עקרון האטומיות של הכתיבה, שכן הכתיבה אינה גלויה באופן אחיד לכל הליבות במערכת.

כאשר כתיבה עונה על התנאים של multi-copy atomicity, זה מרמז על כך שהכתיבה משמרת סיבתיות, כלומר כל הליבות במערכת יקבלו את המידע החדש לפי סדר לוגי של התרחשות האירועים. לכן, מודל ששומר על multi-copy atomicity, מרמז על כך שהסיבתיות נשמרת.

מודל Non-Multi-Copy Atomic

מודל שהוא Non-Multi-Copy Atomic, מתאר אופן פעולה. נוח יותר להבין על ידי דוגמה של מודל תיאורטי:

תת-מערכת האחסון במעבד מורכבת מהיררכיה בצורת עץ של תורים, הנמצאים מעל מערכת זיכרון פשוטה. בתת-מערכת זו, כל תור מתייחס לפעולות כתיבה, בקשות קריאה ומחסומי זיכרון הקשורים ל-thread מסוים. כאשר thread מבצע פעולה כלשהי, כמו גישה לזיכרון או הפעלת מחסום, הפעולה נכנסת לראש התור המשויך לאותו thread.

לאורך הזמן, האירועים הנמצאים בתחתית התור יכולים "לזרום" לתורים הבאים בהיררכיה, עד שהם מגיעים לזיכרון הראשי. במקרה של כתיבה, כאשר אירוע כתיבה מגיע לזיכרון, הוא מעדכן את מפת הזיכרון, שהיא מפה של מיקומי בתים המייצגים את כתיבות הזיכרון האחרונות שבוצעו למיקומים אלה. עבור קריאה, כאשר האירוע מגיע לתחתית התור, נשלחת תגובת קריאה עם הערך מהזיכרון חזרה ל-thread המבצע את הקריאה. מחסומים בתור מוסרים כאשר הם מגיעים לשלב הסופי שלהם.

במקרה של קריאה, ייתכן שלא יהיה צורך להמתין להשלמת הכתיבה בזיכרון הראשי. אם בתור יש אירוע כתיבה שנמצא ישירות מתחת לאירוע הקריאה, והוא מתייחס לאותו מיקום בזיכרון, הקריאה יכולה להיות מסופקת מיד מהתור, ולא לחכות לכתיבה בזיכרון הראשי.

התורים במערכת non-multi-copy atomicity מאפשרים גם גמישות מסוימת בסידור מחדש של האירועים. אירועים בתורים סמוכים, אפילו אם הם קשורים ל-thread-ים שונים, עשויים להחליף את הסדר ביניהם, בתנאי שזה לא מפר את המגבלות של סידור האירועים במערכת.

פלטפורמה שאינה תומכת באטומיות מרובת עותקים, עשויה לגרום לכך ש-stores יגיעו ל-thread-ים שונים בזמנים שונים.

דוגמה למבחן IRIW

כדי להבין את זרימת האירועים במודל non-multi-copy atomicity, ניתן להסתכל על מבחן לקמוס שנקרא IRIW+addrs1:

הערה: הקוד הבא הוא דמוי מבחן לקמוס וכאן המשתנים x ו-y הם משתנים גלובאלים משותפים בין כל ה-thread-ים, והמשתנים המקומיים r0 ו-r1 הם פרטיים לכל thread. הפעולות WRITE ו-READ הן פעולות store ו-load בהתאמה, וכל הפונקציות עם הקידומת thread רצות במקביל:

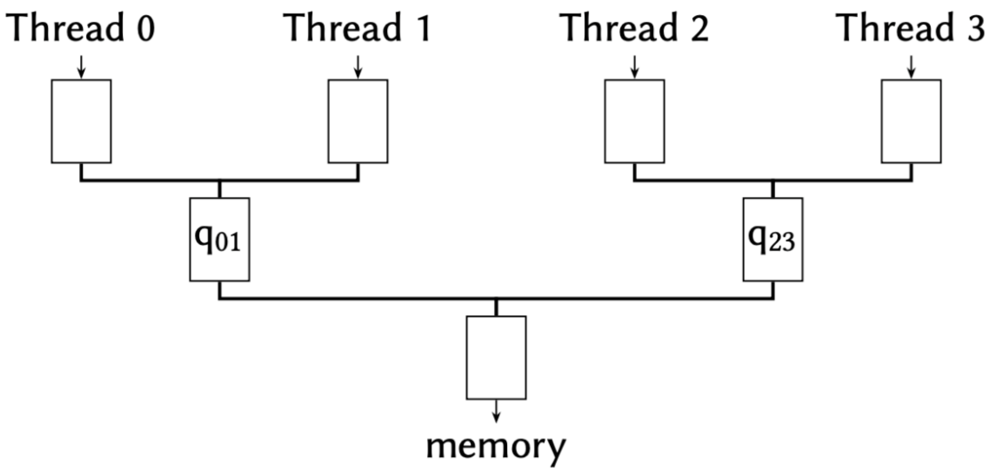
```
thread0()
 {
    WRITE(*x, 1);
 }

 thread1()
 {
    int r0;
    int r1;

    r0 = READ(*x);
    r1 = READ(*y + (r0 ^ r0));
}

thread2()
{
    WRITE(*y, 1);
}

thread3()
{
    int r0;
    int r1;

    r0 = READ(*y);
    r1 = READ(*x + (r0 ^ r0));
}
```

הערה: במאמרים בהמשך אנחנו נלמד על מבחני לקמוס, סידור מחדש, ועל מודלי זיכרון פורמליים. אז נוכל להבין יותר טוב את מה שקורה כאן.

מבחן זה ממחיש מקרים שבהם תוצאה לא עקבית יכולה להתרחש עקב סידור מחדש של אירועים בתורים, אשר מתאפשרת על ידי המודל non-multi-copy atomicity.

בתרחיש המתואר, ה-thread-ים 0 ו-2 מבצעים כתיבות למיקומי זיכרון שונים (x ו-y, בהתאמה), ולאחר מכן ה-thread-ים 1 ו-3 קוראים מהמיקומים הללו, אך בסדר הפוך. כלומר, thread 1 קורא תחילה את x ואחר כך את y, בעוד thread 3 קורא תחילה את y ואחר כך את x. תוצאה מעניינת מתרחשת כאשר כל אחד מה-thread-ים הקוראים רואה את הערך החדש במיקום הזיכרון הראשון שהוא ניגש אליו, אך את הערך ההתחלתי במיקום הזיכרון השני.

תיאור טופולוגיית הזרימה



[figure02 Flowing topology Simplifying ARM Concurrency Multicopy-Atomic Axiomatic and Operational Models for ARMv8 from Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8 (www.cl.cam.ac.uk)]

כאשר ה-thread-ים 0 ו-2 מבצעים את הכתיבות שלהם (x=1 ו-y=1), הכתיבה של thread 0 (x=1) יכולה לעבור לתור q01, שהוא התור המשותף ל-thread-ים 0 ו-1. במקביל, הכתיבה של thread 2 (y=1) עוברת לתור q23 המשותף ל-thread-ים 2 ו-3.

כעת, thread 1 יכול לבצע את הקריאה הראשונה שלו למיקום x, והקריאה "זורמת" בתור q01, עד שהיא פוגשת את הכתיבה x=1, כך שהוא יקרא את הערך המעודכן. לאחר מכן, הוא מבצע את הקריאה השנייה שלו למיקום y, והקריאה הזו זורמת כל הדרך אל הזיכרון הראשי, שם היא מסודרת מחדש עם הכתיבה x=1 והקריאה של y מחזירה את הערך ההתחלתי y=0.

באופן דומה, thread 3 מבצע את הקריאה הראשונה שלו למיקום y, והיא זורמת בתור q23 עד שהיא פוגשת את הכתיבה y=1, כך שהוא יקרא את הערך המעודכן. לאחר מכן, הוא מבצע את הקריאה השנייה שלו למיקום x, והיא זורמת אל הזיכרון, מסודרת מחדש עם הכתיבה y=1, ומחזירה את הערך ההתחלתי x=0.

ההתנהגות הזו מראה כיצד כתיבות יכולות להיות נראות על ידי חלק מה-thread-ים לפני שהן נראות על ידי כולם, מה שמוביל לתוצאות שונות בין ה-thread-ים הקוראים.

באופן כללי, המודל non-multi-copy atomicity מאפשר שתי אופטימיזציות חומרה עיקריות:

1. קיום של buffer משותף לפני ה-cache: זהו buffer שנמצא לפני ה-cache, המאפשר העברת נתונים מוקדמת בין תת-קבוצה של thread-ים במערכת. בכך, הנתונים יכולים לזרום בצורה מהירה יותר בין thread-ים מסוימים לפני שהם מגיעים לזיכרון הראשי או ל-cache-ים האחרים.

2. קיום של snoop invalidation מהיר: מנגנון שמאפשר שליחת בקשת invalidation ל-cache-ים האחרים שמשתתפים בפרוטוקול הקוהרנטיות, מבלי להמתין לאישור מלא מהם. כך מתאפשרת פעולה מהירה יותר במערכת הקוהרנטיות.

מודל Other-multi-copy atomic

בפלטפורמה התומכת באטומיות מרובת עותקים מלאה, כל ה-thread-ים הפעילים בה מבוטחים להסכים על סדר ה-stores, גם כאשר מדובר במיקומי זיכרון שונים. כלומר, כל ה-stores שנעשו על ידי כל ה-thread-ים נצפים באותו סדר על ידי כל ה-thread-ים במערכת.

לעומת זאת, רוב ספקי המעבדים המספקים אטומיות מרובת עותקים למעשה מספקים גרסה חלשה יותר המכונה other-multi-copy atomicity. בגרסה זו, אין דרישה שכל המעבדים יסכימו על סדר כל ה-stores. במקום זאת, רק תת-קבוצה של המעבדים המספקים את ה-stores תסכים על סדר זה.

השם "אחר" (other) ב-other-multi-copy atomicity מתאר את העובדה שלמעבד עצמו מותר לראות את ה-store שלו מוקדם יותר, מה שמאפשר ל-loads המאוחרים שלו לקבל את הערך החדש שהוא כתב ובכך לשפר את הביצועים.

במודל Other-multi-copy atomic, הכתיבה שנעשית על ידי מעבד חייבת להיות גלויה לכל המעבדים האחרים שניגשים למיקום זה באופן קוהרנטי, אם היא נצפתה על ידי מעבד אחר. כלומר, כל מעבד במערכת צריך לראות את הכתיבה אם היא נראית על ידי מעבד אחר. אך יש לציין שמעבד יכול לראות את הכתיבות שלו לפני שהן נחשפות לצופים אחרים במערכת.

בפועל, מודל זיכרון המתואר כ-Other-multi-copy atomic, מאפשר למעבדים להפעיל store buffers מקומיים שאינם קוהרנטיים עם המעבדים האחרים במערכת. ה-store buffers האלה נבדקים רק במובן המקומי עבור תלות בין פעולות, ולא בהכרח בשקיפות מלאה מול יתר המעבדים במערכת, ומעבדים שמבצעים כתיבה יכולים לקרוא ישירות מה-store buffer, ולקבל את הערך החדש שהם כתבו (ידוע בתור forward).

המודל Other-multi-copy-atomicity מוגדר בתיעוד של Arm באופן הבא: במערכת שבה כל כתיבה שנצפתה על ידי צופה אחד (כמו מעבד או thread), צריכה להיות נראית לכל שאר הצופים שניגשים לאותו מיקום בזיכרון באופן קוהרנטי. עם זאת, מותר לכל thread לראות את הכתיבות שלו עצמו לפני שהן הופכות לגלויות לשאר ה-thread-ים במערכת.

אטומיות מרובת עותקים ב-Arm

במודל המקורי של ארכיטקטורת ARM, המערכת לא הייתה אטומית מרובת עותקים. עם זאת, התנהגות זו לא יושמה בייצור בפועל. הסיבה לכך היא שהאופטימיזציות הפוטנציאליות שביצועים כאלה מציעות לא

סיפקו יתרונות משמעותיים עבור ARM, והן גרמו לסיבוכים מורכבים בשילוב עם תכונות אחרות של ARMv8.

כתוצאה מכך, ארכיטקטורת ARMv8 עברה שינוי, והמודל הנוכחי שלה הוא Other-multi-copy-atomicity. המשמעות היא, שכאשר כתיבה הופכת גלויה ל-thread אחד, היא תהיה גלויה לכל ה-thread-ים האחרים במערכת. השינוי במודל זה פישט גם היבטים נוספים בארכיטקטורה, כולל הגדרות ברורות יותר של תלות בין פעולות, והארכיטקטורה כעת כוללת מודל רשמי לניהול מקביליות.

בגרסאות של ARMv7 ו-ARMv8 ישנות, במדריכים הרשמיים נאמר כי כתיבה יכולה להפוך גלויה לחלק מה-thread-ים לפני שתהיה גלויה לכולם, תכונה שהייתה דומה לארכיטקטורת IBM POWER. אך בהקשר של ARM, היתרונות הביצועיים של מודל non-multi-copy atomicity לא הצדיקו את המורכבות הנוספת שהגיעו איתם, במיוחד באימות הלוגיקה והיישום.

האפשרות לאפשר התנהגות non-multi-copy atomicity גרמה למורכבות גדולה במודל הזיכרון, בעיקר כשמנסים לשלב זאת עם הרצון להעניק חופש ביישום ארכיטקטוני ועם ההוראות החדשות שנוספו ב-ARMv8, כמו store-release ו-load-acquire (שגם הם יקבלו הסבר מופרט במאמר אחר). כתוצאה מכך, ARM עדכנו את המפרט של ARMv8.

בנוסף, המפרט המעודכן של ARMv8, כולל כעת מודל זיכרון רשמי שמפרט בבירור אילו סוגי התנהגויות מותרות ואילו אסורות, והמודל הפורמלי מלווה בגרסה תיאורית כתובה. השינויים הללו כוללים גם הבהרות בנוגע לתלות בין הוראות, מה שמוסיף עוד מידה של פשטות ושקיפות לארכיטקטורה.

למרות שהאופטימיזציות הללו יכולות להיות יעילות במצבים מסוימים, בארכיטקטורת ARM הגיעו למסקנה פנימית שהן לא מספקות יתרון משמעותי. אחת הסיבות לכך היא שארכיטקטורת ה-buses של ARM, מתבססת על פרוטוקול AMBA, שתומך ב-multi-copy atomicity. כלומר, בארכיטקטורה זו יש גישה אחידה וברורה יותר לניהול כתיבות זיכרון, ואין צורך באותן אופטימיזציות מורכבות שמספקות non-multi-copy atomicity. לכן, המורכבות שהתווספה על ידי non-multi-copy atomicity בארכיטקטורת ARMv8 לא הצדיקה את הגמישות הנוספת שהוא סיפק.

בעיה נוספת ב-non-multi-copy atomicity, היא העובדה שהוא מאפשר התפשטות של כתיבה ל-thread-ים אחרים לפני שכתיבה קודמת באותו מיקום בוצעה במלואה. כלומר, ייתכן מצב שבו כתיבה מאוחרת תבוצע לפני שכתיבה קודמת לאותו מיקום הושלמה, מה שיוצר חוסר עקביות.

במודלים שהם multi-copy atomicity, בעיות כאלה אינן מתקיימות. לדוגמה, מבחנים כמו IRIW+addrs ו-WRC+addrs לא יכולים להתקיים, מכיוון שמודלים multi-copy atomicity מבטיחים סדר יותר קפדני של כתיבות וקריאות, כך שאין צורך להצטברות מחסומים כדי להבטיח קוהרנטיות במערכת.

AMBA CHI

בגרסאות של AMBA4 ACE ו-CHI Issue A, מעבדים יכולים להעביר מחסומי זיכרון לתוך המערכת. פעולה זו מאפשרת להבטיח שהסדר והצפייה בטרנזקציות נשמרים כפי שנדרש במערכת, כלומר, היא מספקת ערבויות לגבי אופן ביצוע הטרנזקציות והסדר שלהן כפי שנצפה על ידי כל הרכיבים המשתתפים.

כאשר מפיצים מחסומי זיכרון במערכות בעלות תכונה של Multi-Copy Atomicity, עלולה להתרחש ירידה בביצועי המערכת. הדבר נובע מכך שמחסומים אלו יוצרים hazards מיותרים בין הטרנזקציות, מה שמאט את פעולת המערכת.

בפרוטוקולי קוהרנטיות חדשים, כמו AMBA CHI 5, הוסרה או צומצמה התמיכה במחסומי זיכרון. במערכות Armv8, שהן Multi-Copy Atomic, לא נדרשת עוד הפצה של מחסומים לצורך שמירה על הסדר. במערכת כזו, המחסומים אינם משפיעים על אופן סידור הטרנזקציות, והערבויות על הסדר נשמרות באופן טבעי, ללא תלות בשאלה אם המחסומים מופצים או לא.

מערכת מוגדרת כ-Multi-Copy Atomicity אם מתקיימים שני התנאים הבאים:

1. כתיבה לאותו מיקום בזיכרון נצפית על ידי כל הרכיבים (agents) באותו סדר, ללא הבדלים בסדר הצפייה.
2. אם agent אחד רואה כתיבה למיקום מסוים בזיכרון, כל ה-agents האחרים במערכת יראו גם הם את הכתיבה לאותו מיקום.

במערכות כאלה, אין צורך בהפצה של מחסומים לצורך שמירה על עקביות וסדר בין הכתיבות, מכיוון שהן שומרות על סדר אטומי מעצם פעולתן.

פרוטוקולי AXI/ACE

בפרוטוקולי AXI/ACE, כאשר רכיב ביניים מספק תגובת כתיבה מוקדמת (early write response), המשמעות היא שהתוצאה של הכתיבה חייבת להיות גלויה לכל המעבדים במערכת. אם מערכת אינה נותנת תגובות כתיבה מוקדמות, ייתכן שהיא פועלת במודל של Multi-Copy Atomicity, כלומר שכל הכתיבות נעשות זמינות לכל המעבדים בו-זמנית.

גישה למיקומים Inner Shareable ו-Outer Shareable

במערכת עם מיקומים Inner Shareable ו-Outer Shareable, נדרשת אטומיות מרובת עותקים (Multi-copy atomicity). כלומר, תגובת snoop לכתיבה תתקבל רק כאשר כל המעבדים בתחום השיתוף הנדרש ראו את הכתיבה. בנוסף, בעת קבלת תגובת snoop, המעבד המשויך כבר צריך להיות חשוף לכתיבה. נקודת הצפייה מוגדרת כלחיצת היד בערוץ התגובה של snoop. חשוב לציין, שערוץ data snoop אינו משמש להגדרת נקודת הצפייה, כך שאין דרישה למחסומים בערוץ הכתובת של snoop.

סיכום

אם הגעתם עד לכאן אז כל הכבוד, התחלנו בסקירה כללית של cache במחשבים מודרניים, בהמשך המסע, נחשפנו למושגים הבסיסיים, שמירה על קוהרנטיות - נושא קריטי במערכות מרובות ליבות. הסברנו כיצד ארכיטקטורות שונות מאפשרות שליטה מסוימת בתהליך ניהול ה-cache, ונגענו בפרוטוקולי קוהרנטיות שמיישמים עקרונות אלו בפועל. כל זה היה צעד ראשון כדי להכנס לעולם המקביליות. אני מקווה שתישארו איתי להמשך כי זאת רק ההתחלה ויש לנו עוד הרבה מה ללמוד.

על המחבר

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.

מוזמנים ליצור איתי קשר דרך חשבון ה-linkedin שלי.

מקורות מידע

- https://en.wikipedia.org/wiki/Cache_placement_policies
- https://lwn.net/Articles/255364/
- https://en.wikipedia.org/wiki/CPU_cache
- https://lwn.net/Articles/252125/
- https://developer.arm.com/documentation/den0042/a/Caches
- https://stackoverflow.com/questions/25947962/cacheline-aligned-in-smp-for-structure-in-the-linux-kernel
- https://developer.arm.com/documentation/107565/0101/Memory-system/Caches-and-memory-hierarchy/Introduction-to-caches?lang=en
- https://documentation-service.arm.com/static/65fdad3c1bc22b03bca90781?token=
- https://inria.hal.science/hal-02509910/document
- https://mariokartwii.com/armv8
- https://en.wikipedia.org/wiki/False_sharing

- https://developer.arm.com/documentation/den0024/a/Caches/Point-of-coherency-and-unification
- https://en.wikipedia.org/wiki/Cache_coherence
- https://pages.cs.wisc.edu/~markhill/papers/primer2020_2nd_edition.pdf
- https://developer.arm.com/documentation/den0024/a/Multi-core-processors/Multi-core-cache-coherency-within-a-cluster?lang=en
- https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf
- http://infolab.stanford.edu/pub/cstr/reports/csl/tr/95/685/CSL-TR-95-685.pdf
- https://mirrors.edge.kernel.org/pub/linux/kernel/people/paulmck/LWNLinuxMM/StrongModel.html
- https://developer.arm.com/documentation/den0024/a/Caches/Cache-maintenance
- https://developer.arm.com/documentation/ihi0050/latest/
- https://developer.arm.com/documentation/107565/0101/Memory-system/Caches-and-memory-hierarchy/Implications-of-caches-for-programmers?lang=en
- https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/memory-access-ordering---an-introduction
- https://www.linkedin.com/advice/3/what-trade-offs-between-snooping-directory-based
- https://developer.arm.com/documentation/102407/0100/Introduction-to-CHI
- https://developer.arm.com/documentation/ihi0022/e/
- https://developer.arm.com/documentation/100236/0100/functional-description/cache-behavior-and-cache-protection/coherency-between-data-caches-with-the-moesi-protocol
- https://www.linkedin.com/advice/1/what-key-features-differences-amba5-chi-amba4
- https://developer.arm.com/documentation/102407/0100/Transaction-flows
- https://developer.arm.com/documentation/102407/0100/Other-protocol-changes-and-extensions
- https://www.cl.cam.ac.uk/~pes20/armv8-mca/armv8-mca-draft.pdf
- https://developer.arm.com/documentation/102336/0100/Memory-barriers
- http://www0.cs.ucl.ac.uk/staff/j.alglave/papers/toplas21.pdf
- https://github.com/paulmckrcu/perfbook
- https://developer.arm.com/documentation/ka002179/latest/