פרויקט סימולטור מטמון

אור פרל 200732444

תומר גארח 305007668

**תיעוד**

פרויקט זה נועד ליצור סימולטור מטמון בעל שתי דרגות אשר יאפשר לנתח את ביצועי הזיכרון על תוכניות כרצוננו, בתחום רחב של גדלי בלוקים ורמות שונות. כפי שמצוין בדרישות הפרויקט המטמונים שבסימולציה הם מסוג Write Back ו-Write Allocate, L1 הוא Direct Mapped ו-L2 הוא Two Way Set Associative.

אופן ההרצה:

הקוד ממומש בשפת פייתון ונבדק בגרסה 3.5. יש לוודא שמותקנת במחשב גרסה זו.

על מנת להריץ את הסימולטור באמצעות CMD ב-Windows יש לכתוב את הפקודות הבאות:

cd *path\_to\_files*

python sim.py *levels b1 b2* .\Test\_Case\_X\trace.txt .\Test\_Case\_X\memin.txt memout.txt l1.txt l2way0.txt l2way1.txt stats.txt

מבנה הקוד:

הקוד מחולק ל-6 קבצים. באופן כללי יש לנו את קובץ הסימולציה הראשי ומחלקת זיכרון אבסטרקטית (MemoryInterface) המגדירה רכיב זיכרון גנרי בהיררכיה. כל אחת מהמחלקות L1Cache, L2Cache ו-MainMemory יורשת מהמחלקה האבסטרקטית ומממשת את המפרט הספציפי של אותה רמה. קובץ הסימולציה הראשי בונה את היררכיית הזיכרון ע"י יצירת אובייקטים מתאימים עבור כל רמה, וכל אובייקט מצביע לרמה הבאה (ה-MM מצביע ל-None). כך הקובץ הראשי מפעיל את הרמה הראשונה שמפעילה את הרמות הבאות, בהתאם להוראות הזיכרון בקלט trace.txt.

פירוט המחלקות:

* sim.py – קובץ הריצה הראשי אשר מריץ את הסימולציה. כאמור הקובץ בונה את היררכיית הזיכרון ע"י אובייקטים מסוג L1Cache, L2Cache (בקונפיגורציה של שתי דרגות)

ו-MainMemory. האתחול של אובייקט ה-MainMemory קורא את הקלט memin.txt וממלא את הזיכרון בהתאם. כמו כן, קובץ הסימולציה (sim.py) מבצע parsing להוראות הזיכרון בקלט trace.txt, מפעיל את ההיררכיה בעזרת הממשק mem\_ifc (המחלקה האבסטרקטית) ומתכלל את הסטטיסטיקות שחוזרות מהפעולות השונות. לבסוף הסימולציה מייצאת את תוכן היררכיית הזיכרון לקבצים השונים – l1.txt, l2way0.txt, l2way1.txt (קבצים אלה ריקים בקונפיגורציה של דרגה אחת) ו-memout.txt. הסימולציה שומרת גם את הסטטיסטיקות (ומבצעת חישובי miss rate ו-AMAT) לקובץ הפלט stats.txt.

* sim\_constants.py – קובץ קבועים עבור הסימולציה (קבועים כלליים ולא ספציפיים לכל רמת זיכרון).
* mem\_ifc.py – המחלקה האבסטרקטית של רכיב זיכרון המשמשת כממשק לקובץ הסימולציה. מחלקה זו מגדירה את הפעולות הגנריות של load ו-store הזהות לכל הדרגות ומשתמשות בפעולות read ו-write שהן ספציפיות לכל דרגה. read ו-write עושות את הקריאה והכתיבה בפועל בעוד load ו-store מרכזות את המעטפת של כל פעולה, הכוללת: בדיקה אם הכתובת קיימת, פנייה לרמה הבאה במידת הצורך, חישוב מחזורי ההשהיה, Write Back ועוד. בנוסף, המחלקה האבסטרקטית מגדירה גם פונקציות כמו is\_address\_present, get\_block\_size, flush\_if\_needed, dump\_memory (dump של תוכן הזיכרון) ו-dump\_output\_file (כתיבה של ה-dump לקובץ).
* l1cache.py, l2cache.py, main\_memory.py – מחלקות הזיכרון של כל רמה המממשות את הפונקציות שהוגדרו ב-mem\_ifc.py בהתאם למפרט של הרמה. קבצים אלה מגדירים את גדלי המטמונים וגדלי ה-Buses, ומכילים את מערך הזיכרון בפועל (ב-L1 וב-MM זהו מערך חד מימדי של בתים וב-L2 זוהי טבלה של בתים מחולקת לשני ה-ways). הם מבצעים parsing לכתובת בהתאם לגדלים שהוגדרו ומחלצים מתוכה את סיביות ה-tag, ה-index

וה-offset. כאמור הם מממשים את הפונקציות השונות, שהעיקריות שבהן הן read, write

ו-flush\_if\_needed, ומחשבות כמה מחזורים לקחה כל פעולה בפועל.

הערה חשובה: הסבר של פעולת כל פונקציה, והקלטים והפלטים שלה, מפורטים היטב בהערות בתוך הקבצים.

מקרה בדיקה 1: בדיקת sanity

עבור המקרה מטה בחרנו שורה של צעדים שיגרמו לקומבינציות שונות של hit / miss בכל אחד מרמות הcache. וידאנו שבעת פגיעה מתעדכן המקום המתאים בcache ע"פ הלוגיקה שלו (Direct mapped לL1, 2-way associative לL2). כמו כן וידאנו את המצבים בהם כל אחת מרמות הcache מבצעת flush לרמה הבאה או אל הזכרון הראשי.

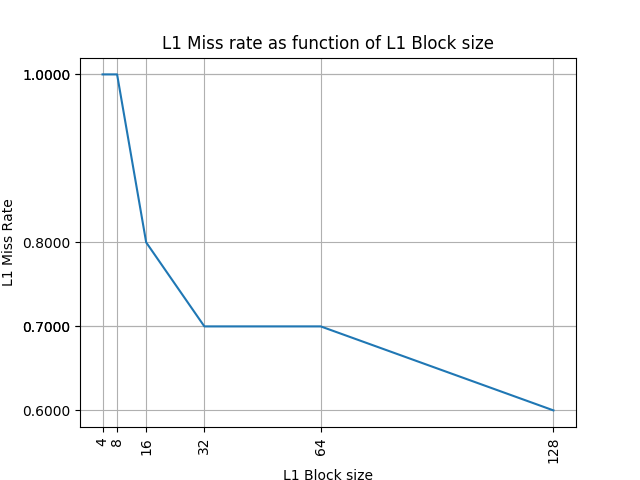
בעמודת הHit / Miss מטה ניתן לראות מה מצופה בדיוק מכל שלב בתוכנית לבדוק.

השתמשנו בגדלי בלוקים של 128 עבור L1 ו512 עבור L2 כיוון שהם גדולים יותר מרוחב הbus ולכן נדרשים מחזורים נוספים בכדי להעביר את המידע בין הזכרון ושתי רמות הזכרון. באופן זה ניתן לבדוק את המנגנון אשר מונה כמה מחזורים לקחה התוכנית.

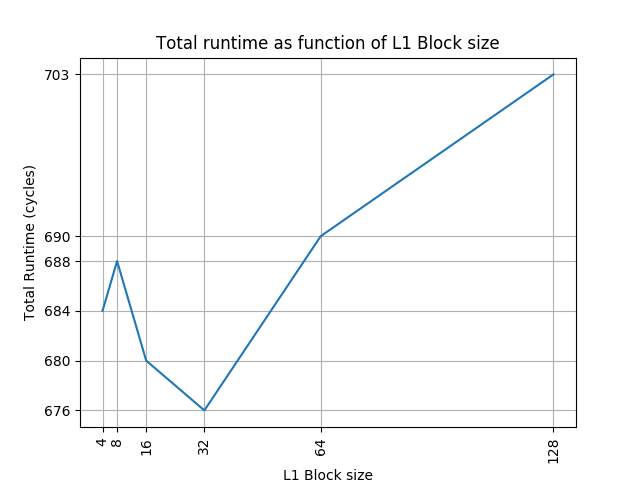
השתמשנו גם בoffsetים שונים בפקודות בתוכנית. על אף שהם לא משפיעים על הtag שיתאחסן שבכל אחד מהתאים בcache, הם מאפשרים לנו לבצע מעקב ולוודא שאכן הdata נכתב בצורה נכונה לקבצי הפלט.

כל אחד משלבי הסימולציה מפורט מטה (באופן בינארי, כדי להקל על המעקב אחר כל אחד מחלקי הכתובת); ביצענו אותם ידנית תחילה ווידאנו שהסימולטור אכן עובד כמצופה (ואכן הנתונים בקבצי הפלט של הסימולטור תואמים את התוצאות שלנו מהחישוב הידני).

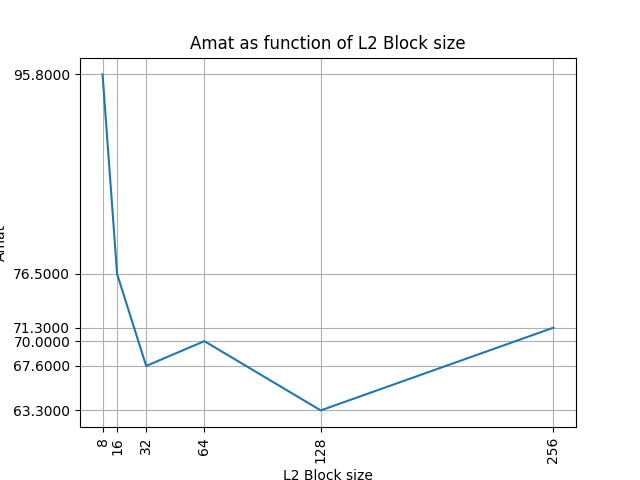
**הגרפים אותם קיבלנו:**



התוכנית פונה לכתובות הנמצאים במרווחים קטנים וגדולים (תכננו אותה כך שתתאים למקרים בהם הבלוק של L1 הוא בגודל 128 בתים), ולכן עבור גדלי בלוק גדולים יותר נקבל ניצול טוב יותר של תכונת הlocality במקום של התכנית, אותה גדלי בלוקים קטנים יותר לא יוכלו לנצל במקרה זה. מכאן שהגיוני שקיבלנו גרף שיורד עם גודל הבלוק.



גודל בלוק L2 הוא 128 בתים. במקרה זה ניתן לראות שגודל בלוק 32 הוא האופטימלי ביותר לL1 מבחינת זמן הריצה של התכנית. ככל שגודל הבלוק עולה, כך מספר הבלוקים בL1 קטן (פחות ביטים בindex שהולכים לטובת הoffset). לכן ניתן להניח שעבור גדלי בלוק גדולים 64, 128 בתים, התכנית תגרום לconflict misses שיגרמו לflush – פעולה שלוקחת זמן רב ומקפיצה את אורך התוכנית ומכאן נקבל את העליה בחלק הימני של הגרף. את החלק השמאלי של הגרף ניתן להסביר בכך שגדלי בלוקים קטנים מדי (4,8) יגרמו למס' רב של L1 write / read compulsory misses עליהם נצטרך לשלם בזמני מחזור נוספים עד שהמידע יגיע מ L2 ומכאן העליה המתונה לעומת גודל בלוק של 32 בתים אשר מוצא איזון נכון בין מס' flushים למס' הmisses.



גודל בלוק של L1 הוא 8 בתים, כלומר קטן ולכן נצפה למס' רב של L1 compulsory misses אותם הL2 ינסה לבלום. ה-average memory access time גבוה כאשר גודל הבלוק של L2 נמוך. במקרה של גודל בלוק 8 יתכן כי הבלוק קטן מדי בכדי לנצל את תכונת הלוקאליות במקום של התכנית. עבור גודל בלוק של 256 נצטרך לשלם במחזורים נוספים על כל L2 Miss שכן רוחב הbus לזכרון הראשי הוא רק 64 בתים ולכן נראה עליה בקצה הימני של הגרף. גם גודל בלוק של 128 דורש מחזורים נוספים על כל L2 miss, אך כאן כיוון שהוא נותן את זמן הגישה הממוצע הטוב ביותר לזכרון ניתן להניח שהוא מצליח להגן על missים נוספים אשר אותם גדלי בלוקים נמוכים יותר לא מצליחים לבלום.

**הסימולציה הידנית עבור מקרה הבדיקה:**

|  |  |
| --- | --- |
| Direct mapped  Block transfer time [L2 to L1]: 7 cycles  (Bus size = 256)  Block transfer time [L1 to CPU]: 1 cycle  (Bus size = 32, always 4 bytes) | 2-Way-Set Associative  Block transfer time [MM to L2]: 163 cycles  (Bus size = 64) |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Command** | **Hit / Miss** | **L1 Index** | **L2 Index** | **L1 Tag** | **L2 Tag** | **Dirty?** | **Cycles** | **Type** |
| 1 LD 000000000000 00000 0000100 | L1 R-Miss L2 R-Miss | 00000 | 00000 | 000000000000 | 0000000000 | L1: No L2: No | 1+163+7+1 | Compulsory |
| 2 LD 000000000000 00000 0001000 | L1 R-hit | 00000 | - | 000000000000 | - | L1: No | 2+1 |  |
| 3 ST 000000000001 00010 0001000 A1A2A3A4 | L1 W-miss  L2 R-miss | 00010 | 01000 | 000000000001 | 0000000000 | L1: Yes  L2: No | 3+163+7+1 | Compulsory |
| 4 ST 000000000001 00010 0000100 B1B2B3B4 | L1 W-hit | 00010 | - | 000000000001 | - | L1: Yes | 4+1 |  |
| 5 LD 000000100100 00010 0010000 | L1 R-miss  L2 R-miss  L1 Flush  L2-W hit | 00010 | 00000 | 000000100100 | 0000001001 | L1: No L2: No, Yes | 5+163+7+7+1 | Conflict+  Compulsory |
| 6 ST 000000100100 00010 0000000 E1E2E3E4 | L1 W-hit | 00010 |  | 000000100100 |  | L1: Yes | 6+1 |  |
| 7 LD 000000000001 00010 0000000 | L1 R-miss  L2 R-hit  L1 Flush  L2 W-hit | 00010 | 01000 | 000000000001 | 0000000000 | L1: No  L2: No, Yes | 7+7+7+1 | Conflict |
| 8 ST 000000000000 00000 1111100 F1F2F3F4 | L1 W-hit | 00000 |  | 000000000000 |  | L1: Yes | 8+1 |  |
| 9 ST 100000000000 00000 1000000 C1C2C3C4 | L1 W-miss  L2 R-miss  L1 Flush  L2 W-miss  L2 Flush | 00000 | 00000 | 100000000000 | 1000000000 | L1: Yes  L2: No, Yes | 9+163+7+7+163+1+163 | Conflict |
| 10 ST 000000000101 00011 1010101 D1D2D3D4 | L1 W-miss  L2 R-miss | 00011 | 01000 | 000000000101 | 0000000001 | L1: Yes  L2: No | 10+163+7+1 | Compulsory + Conflict |

**Cache state:**

* Dirty marked in red

Steps 1, 2:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Index** | **L1 Tag** | **L2 – Way 0 Tag** | **L2 – Way 1 Tag** | **L2 – LRU** |
| 00000 | 000000000000 | 0000000000 |  | 1 |
| 00001 |  |  |  | 0 |
| 00010 |  |  |  | 0 |
| 00011 |  |  |  | 0 |
| 00100 |  |  |  | 0 |
| 00101 |  |  |  | 0 |
| 00110 |  |  |  | 0 |
| 00111 |  |  |  | 0 |
| 01000 |  |  |  | 0 |
| … | | | | |

Steps 3, 4:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Index** | **L1 Tag** | **L2 – Way 0 Tag** | **L2 – Way 1 Tag** | **L2 – LRU** |
| 00000 | 000000000000 | 0000000000 |  | 1 |
| 00001 |  |  |  | 0 |
| 00010 | 000000000001 |  |  | 0 |
| 00011 |  |  |  | 0 |
| 00100 |  |  |  | 0 |
| 00101 |  |  |  | 0 |
| 00110 |  |  |  | 0 |
| 00111 |  |  |  | 0 |
| 01000 |  | 0000000000 |  | 1 |
| … | | | | |

Step 5:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Index** | **L1 Tag** | **L2 – Way 0 Tag** | **L2 – Way 1 Tag** | **L2 – LRU** |
| 00000 | 000000000000 | 0000000000 | 0000001001 | 0 |
| 00001 |  |  |  | 0 |
| 00010 | 000000100100 |  |  | 0 |
| 00011 |  |  |  | 0 |
| 00100 |  |  |  | 0 |
| 00101 |  |  |  | 0 |
| 00110 |  |  |  | 0 |
| 00111 |  |  |  | 0 |
| 01000 |  | 0000000000 |  | 1 |
| … | | | | |

Step 6:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Index** | **L1 Tag** | **L2 – Way 0 Tag** | **L2 – Way 1 Tag** | **L2 – LRU** |
| 00000 | 000000000000 | 0000000000 | 0000001001 | 0 |
| 00001 |  |  |  | 0 |
| 00010 | 000000100100 |  |  | 0 |
| 00011 |  |  |  | 0 |
| 00100 |  |  |  | 0 |
| 00101 |  |  |  | 0 |
| 00110 |  |  |  | 0 |
| 00111 |  |  |  | 0 |
| 01000 |  | 0000000000 |  | 1 |
| … | | | | |

Step 7:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Index** | **L1 Tag** | **L2 – Way 0 Tag** | **L2 – Way 1 Tag** | **L2 – LRU** |
| 00000 | 000000000000 | 0000000000 | 0000001001 | 0 |
| 00001 |  |  |  | 0 |
| 00010 | 000000000001 |  |  | 0 |
| 00011 |  |  |  | 0 |
| 00100 |  |  |  | 0 |
| 00101 |  |  |  | 0 |
| 00110 |  |  |  | 0 |
| 00111 |  |  |  | 0 |
| 01000 |  | 0000000000 |  | 1 |
| … | | | | |

Step 8:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Index** | **L1 Tag** | **L2 – Way 0 Tag** | **L2 – Way 1 Tag** | **L2 – LRU** |
| 00000 | 000000000000 | 0000000000 | 0000001001 | 0 |
| 00001 |  |  |  | 0 |
| 00010 | 000000000001 |  |  | 0 |
| 00011 |  |  |  | 0 |
| 00100 |  |  |  | 0 |
| 00101 |  |  |  | 0 |
| 00110 |  |  |  | 0 |
| 00111 |  |  |  | 0 |
| 01000 |  | 0000000000 |  | 1 |
| … | | | | |

Step 9:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Index** | **L1 Tag** | **L2 – Way 0 Tag** | **L2 – Way 1 Tag** | **L2 – LRU** |
| 00000 | 100000000000 | 1000000000 | 0000000000 | 0 |
| 00001 |  |  |  | 0 |
| 00010 | 000000000001 |  |  | 0 |
| 00011 |  |  |  | 0 |
| 00100 |  |  |  | 0 |
| 00101 |  |  |  | 0 |
| 00110 |  |  |  | 0 |
| 00111 |  |  |  | 0 |
| 01000 |  | 0000000000 |  | 1 |
| … | | | | |

Step 10 [Final state of the cache]:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Index** | **L1 Tag** | **L2 – Way 0 Tag** | **L2 – Way 1 Tag** | **L2 – LRU** |
| 00000 | 100000000000 | 1000000000 | 0000000000 | 1 |
| 00001 |  |  |  | 0 |
| 00010 | 000000000001 |  |  | 0 |
| 00011 | 000000000101 |  |  | 0 |
| 00100 |  |  |  | 0 |
| 00101 |  |  |  | 0 |
| 00110 |  |  |  | 0 |
| 00111 |  |  |  | 0 |
| 01000 |  | 0000000000 | 0000000001 | 1 |
| … | | | | |

**Statistics:**

Total cycles: 1269;

Mem access cycles: 1214

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Read hits: | Read misses: | Write hits: | Write misses: | Miss Rate: |
| L1: | 1 | 3 | 3 | 3 |  |
| L2: | 1 | 5 | 2 | 1 |  |

Global miss rate = L1 miss-rate \* L2 miss rate =

AMAT:

מקרה בדיקה 2:

במקרה זה מטרתנו הייתה לגרום לתגובת שרשרת של misses ו-writes back ארוכה כמה שיותר. עשינו זאת ע"י כך שגרמנו לבלוק להיות dirty ב-L1 ולא להימצא כלל ב-L2, ובמקומו שני בלוקים dirties. אחר כך דאגנו להביא בלוק חדש נוסף שייפול על אותם אינדקסים ב-L2 וב-L1 יחד.

זרימת התוכנית היא כדלקמן:

כותבים לבלוק שנסמן אותו ב-B אשר נמצא באינדקס כלשהו ב-L1 וב-L2.

אחר כך כותבים לבלוקים C ו-D הנמצאים באותו אינדקס כמו של B ב-L2 אך באינדקסים שונים ב-L1.

לאחר מכן כותבים לבלוק E הנמצא באותו אינדקס כמו של C ב-L1 אך באינדקס שונה ב-L2.

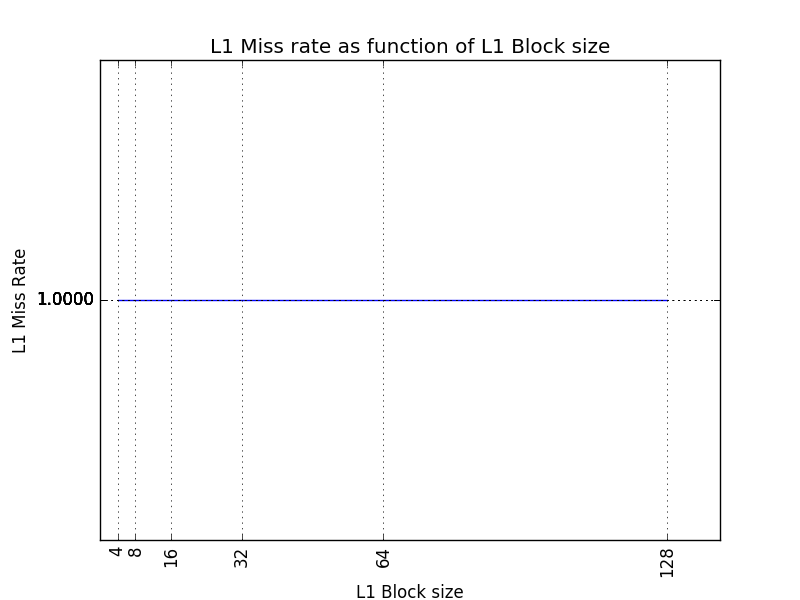
כותבים לבלוק F הנמצא באותו אינדקס כמו של D ב-L1 אך באינדקס שונה ב-L2.

שתי הכתיבות האחרונות נעשות על מנת לגרום לבלוקים C ו-D ב-L2 להיות dirties.

לבסוף כותבים לבלוק A הנמצא באותו אינדקס של C ו-D ב-L2 ובאותו אינדקס כמו של B ב-L1.

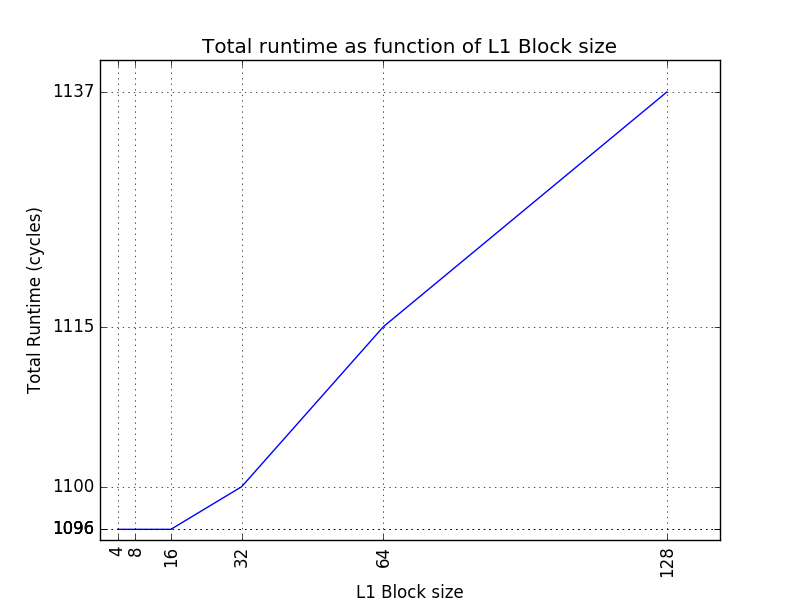
השתמשנו בגדלי בלוקים של 16B עבור L1 ו-64B עבור L2. בעמודות ה-hits/misses בטבלה מטה (סימולציה ידנית) ניתן לראות מה מצופה מכל שלב בתוכנית.

**הגרפים אותם קיבלנו:**

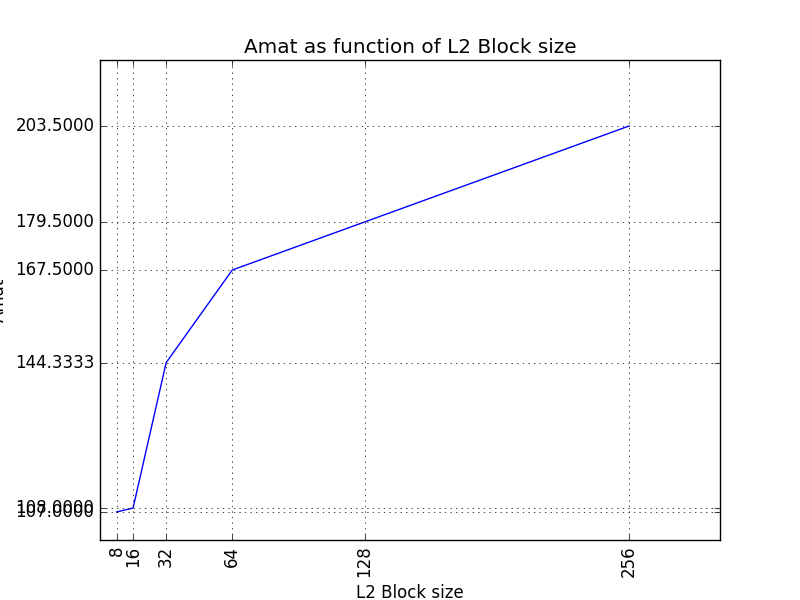


גרף ה-Miss Rate של L1 פחות מעניין במקרה זה שכן קבענו את התוכנית שלנו לגרום ל-misses

ב-L1 בכל הוראה.



גודל בלוק L2 הוא 128 בתים. במקרה זה ניתן לראות שגדלי בלוקים 4, 8 ו-16 הם האופטימליים ביותר לL1 מבחינת זמן הריצה של התכנית. בתוכנית זו אנחנו משלמים בעיקר על ה-flushes ולכן ככל שגודל הבלוק עולה כך זמן הריצה גדל. בנוסף, בתוכנית זו גדלי בלוקים קטנים לא יגרמו ליותר compulsory misses כיוון שאין הרבה פקודות מלכתחילה.



כאמור מספר ה-compulsory misses כמעט לא משתנה ולכן גודל בלוק קטן ב-L2 לא יעלה את

ה-average memory access time. כאמור בתוכנית זו אנחנו משלמים בעיקר על ה-flushes ולכן בדומה לגרף הקודם, הגדלה של גודל הבלוק ב-L2 מגדילה את זמן הריצה ולכן גם את ה-AMAT.

**הסימולציה הידנית עבור מקרה הבדיקה (במקרה זה לא פירטנו את כל צעדי הריצה):**

|  |  |
| --- | --- |
| Direct mapped  Block transfer time [L2 to L1]: 4 cycles  (Bus size = 256)  Block transfer time [L1 to CPU]: 1 cycle  (Bus size = 32, always 4 bytes) | 2-Way-Set Associative  Block transfer time [MM to L2]: 107 cycles  (Bus size = 64) |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| # | address | L1 tag | L2 tag | L1 index | L2 index | L1 read hit | L1 write hit | L1 read miss | L1 write miss | L2 read hit | L2 write hit | L2 read miss | L2 write miss |
| 1 | 000000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 2 | 004010 | 4 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 3 | 008020 | 8 | 2 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 4 | 001010 | 1 | 0 | 1 | 40 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 5 | 002020 | 2 | 0 | 2 | 80 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 6 | 00C000 | C | 3 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
|  |  |  |  |  |  | 0 | 0 | 0 | 6 | 0 | 2 | 6 | 1 |

|  |  |  |  |
| --- | --- | --- | --- |
| # | other opers cycles | transaction cycles | total cycles |
| 1 | 2 | 100 + 7 + 4 + 1 | 114 |
| 2 | 4 | 100 + 7 + 4 + 1 | 116 |
| 3 | 6 | 100 + 7 + 4 + 1 | 118 |
| 4 | 1 | 100 + 7 + 4 + 4 + 1 | 117 |
| 5 | 4 | 100 + 7 + 4 + 4 + 1 | 120 |
| 6 | 2 | 100 + 7 + 100 + 7 + 100 + 7 + 100 + 7 + 4 + 4 + 1 | 439 |
|  |  |  | 1024 |