Generational GCx with Load-Balancing

מנוחה בנישתי, ת.ז 313311920

רותם אליאס, ת.ז 316538768

בפרוייקט זה נעסוק בשיפורים לאלגוריתמי GC ולאלגוריתמי ניהול זיכרון עבור זיכרונות פלאש (SSD). את השיפורים ביצענו ביחס לעבודה שנעשתה בנושא ע״י אייל דותן ודור סורה ועל גבי הסימולטור שהם פיתחו במהלך עבודתם ואשר בו מומשו כלל האלגוריתמים והשיפורים שהציעו. להלן קישור לדף ה-Github של הפרוייקט הקודם: https://github.com/Eyallotan/GC_Simulator. הקוד הרלוונטי להרצת הסימולטור כולל השיפורים שנציג בעבודה זו נמצאים תחת הקישור הבא: https://github.com/MenuchaBen/Generational-GC-with-Load-Balancing. בנוסף ניתן למצוא בקישור זה את כל ההוראות להורדה והפעלה של הסימולטור המותאם לעבודתנו.

מבוא - Flash Memories

זיכרון מבוסס פלאש מוצג כפתרון בעל ביצועים גבוהים וחסכוני. עם זאת, SSD סובלים מסיבולת מוגבלת עקב שחיקה ולכן עלולים לגרום לאובדן נתונים. לפיכך, מזעור מספר הכתיבות לזיכרון מאט את קצב השחיקה שלו וכתוצאה מכך מגדיל את תוחלת החיים שלו.

ה-SSD מורכב מדפים ובלוקים. דף הוא יחידת כתיבה של מס׳ כלשהו של בתים, ובלוק מורכב ממס׳ כלשהו של דפים. גדלים אלו משתנים ונקבעים ע״י היצרן של הזיכרון.

הזיכרון מאפשר מחיקה וכתיבה מחדש של מידע. אך לרוע המזל, במקום להחליף את הנתונים ישירות in place, SSD צריכים לבצע תחילה פעולת מחיקה, לפני שתתרחש פעולת כתיבה. כלומר מחק ואז כתוב.

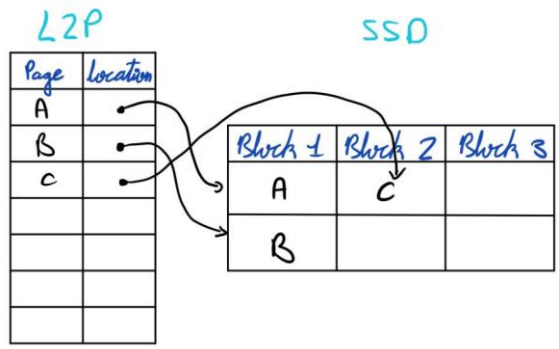
ניתן לבצע שלוש פעולות על הSSD. מחיקה בהיקף של בלוק, כתיבה וקריאה בהיקף של דף.

מכיוון שניתן לבצע מחיקה אך ורק ברזולוציה של בלוק, כאשר נרצה לעדכן דף שקיים כבר בזיכרון, נכתוב את הדף המעודכן בשנית למקום פנוי בזיכרון מבלי למחוק את הדף הישן. נסמן את הדפים המעודכנים כ-valid ואת אלו שאינם כ-invalid. לשם כך יש לתחזק טבלת תרגום logical-to-physical (L2P) הממפה כתובות לוגיות לפיזיות. כאשר דף נכתב בשנית, הL2P מתעדכן כך שיצביע למיקום הפיזי החדש של הדף.

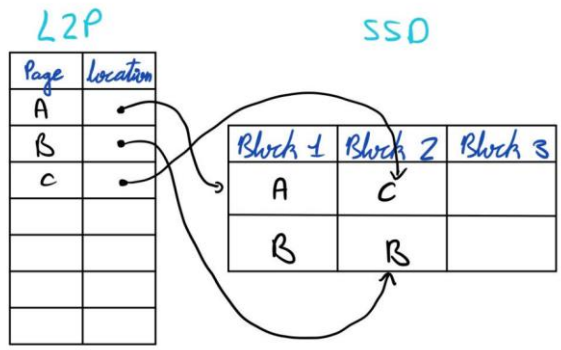
לשם המחשה נסתכל על זיכרון SSD המורכב מ-3 בלוקים כך שבכול בלוק 2 דפים. נניח כי בתחילה הזיכרון פנוי כולו, כלומר באופן סכמתי זהו מצבו ההתחלתי:

ונניח כי המשתמש רוצה לבצע את סדרת הכתיבות הבאה של דפים לוגיים: A, B, C, B

שלושת הדפים הראשונים נכתבים לזיכרון למקום פנוי ומוסף המיפוי המתאים לטבלת ה-L2P:



בכתיבה הרביעית למעשה הדף B מתעדכן ולכן מצב הזיכרון וטבלת ה-L2P תהיה:



מכיוון שכמות הזיכרון בה משתמשים גדול מכמות הזיכרון בה המשתמש ביקש להשתמש, מגדירים את המושגים הבאים עבור הזיכרון:

Number of Physical Blocks – מס' הבלוקים שבפועל יש בזיכרון, מסומן ב-T.

Number of Logical Blocks – מס' הבלוקים אותם המשתמש יכול למלא במידע האמיתי, מסומן ב-U. כמובן שבלוקים אלו אינם רציפים בזיכרון ולמעשה מגדירים את מס' הדפים שהמשתמש יכול לשמור בזיכרון בו-זמנית.

Over Provisioning(OP) – ערך שמוגדר באופן הבא: $OP = \frac{T-U}{U}$ ומהווה מדד לכמות הזיכרון הפיזית שאינה פנויה לשימוש המשתמש אלא מנוצלת לצורך ניהול הזיכרון. באופן אופטימלי ערך זה יהיה 0.

כמובן שמתישהו יהיה צריך לפנות את הדפים שהם invalid מהזיכרון על מנת שיהיה מקום לכתוב מידע חדש. מכיוון שאין הבטחה כי כאשר הזיכרון יתמלא יהיה בלוק שמכיל רק דפים שהם invalid, ייתכן כי לא תהיה ברירה אלא למחוק בלוק שיש בו דפים שהם valid ולכן יהיה צורך לכתוב אותם מחדש לזיכרון לפני המחיקה על מנת לא לאבד מידע שהוא valid. כלומר כמות הכתיבות בפועל לזיכרון גדול מכמות הכתיבות שהמשתמש מבקש לכתוב. נתייחס אל הכתיבות של מידע חדש שמגיע מהמשתמש כאל כתיבות לוגיות לעומת כל סוגי הכתיבות גם הלוגיות וגם כתיבות של דפים שמועתקים מבלוק אחד למשנהו בתוך הזיכרון (כתיבות להן המשתמש לא מודע כלל) אליהן נתייחס ככתיבות פיזיות. למעשה תמיד התקיים כי: $logica\_writes \leq physical\_writes$.

מכיוון שה-SSD סובל משחיקה ככל שנעשות אליו יותר כתיבות, נרצה להקטין כמה שאפשר את מס' הכתיבות הפיזיות כאשר המצב האופטימלי כמובן יהיה שמס' הכתיבות הפיזיות יהיה שווה למס' הכתיבות הלוגיות.

הורדת כמות הכתיבות הפיזיות יכול להיעשות ע"י אלגוריתם הכתיבה לזיכרון (הכתיבות הלוגיות) אשר קובע לאיזה בלוק יכתבו הדפים הלוגיים שמגיעים מהמשתמש, וע"י אלגוריתם הפינוי, ה- Garbage Collector(GC), אשר מפנה מהזיכרון את הדפים שהם invalid.

המושג Write-Amplification(WA) המוגדר באופן הבא:

$$WA = \frac{physical\ writes}{logical\ writes}$$

הינו מדד ליעילות האלגוריתמים הללו. ככל שה-WA יותר קרוב ל-1 כך יעילותם גדולה יותר.

Garbage Collection Algorithms for Flash Memories -סקירה:

להלן רקע קצר על חלקים מהעבודה הקודמת בנושא עם רלוונטיות לפרוייקט זה.

הנחות עבודה:

1. אלגוריתם הכתיבה מקבל רצף של N דפים, Writing-Sequence, שעליו לכתוב לזיכרון בזה אחר זה לפי הסדר בו הם נתונים.
2. התפלגות הדפים ב-Writing-Sequence הינה התפלגות יוניפורמית.
3. בטרם כתיבת ה- Writing-Sequence לזיכרון, הזיכרון נמצא במצב Steady-State – הזיכרון מלא בדפים אשר נכתבו בהתפלגות יוניפורמית.
4. קיים Temporary-Buffer אליו מועתקים באופן זמני דפים שהם valid ופונו מהזיכרון ע"י ה-GC וממנו מועתקים חזרה לזיכרון לבלוק פנוי ע"י ה-GC.
5. ה-metadat של הזיכרון ומבני הנתונים ששייכים לניהול הזיכרון (לדוגמא הטבלה: L2P), אינם שמורים בבלוקים של ה-SSD אלא ב-RAM של המחשב שמנהל את הזיכרון.

הגדרות:

1. Age – לכל דף פיזי הוגדר גיל לפי תוחלת החיים של הדף בזיכרון כדף שהוא Valid.
תוחלת החיים של הכתיבה ה-i ברצף הכתיבות (ws) סומנה ב-age(i) והוגדרה באופן הבא:

$$next(i) = \begin{cases} j, & \exists j\ s.t\ ws[i] = ws[j]\ and\ i < j \\ N, & else \end{cases}$$

$$age(i) = next(i) - i$$

כלומר, גילו שך דף פיזי שווה למספר הכתיבות שיעברו מהכתיבה שלו לזיכרון עד שיהפוך ל-invalid. במקרה בו הדף הלוגי לא ייכתב בשנית, תוחלת החיים שלה תוגדר להיות המרחק מהכתיבה האחרונה (N).

אלגוריתם הפינוי Greedy Lookahead:

האלגוריתם Greedy Lookahead הינו אופטימיזציה לאלגוריתם Greedy GC. האלגוריתם משתמש בידע הקיים לו ע"י רצף הכתיבות, תחת ההנחה כי התפלגות הכתיבות יוניפורמית, על מנת לבחור בלוק לפינוי שיגרום למס' כתיבות פיזיות קטן יחסית ובכך מקטין את ה-WA.
האלגוריתם מסתכל על כל הבלוקים. בשלב הראשון האלגוריתם לוקח רק את הבלוקים עם מספר הדפים הvalidים המינימלי ביותר, בדומה לאלגוריתם Greedy GC. לאחר מכן מתוך קבוצה זו הוא מחשב לכל בלוק ניקוד שמבטא את "תוחלת החיים" של הדפים בו ולוקח את הבלוק עם תוחלת החיים המקסימלית.

נקבל כי לפי אלגוריתם זה, בלוק אשר הדפים בו נשארים validים ליותר זמן יקבל ניקוד גבוה יותר וכך ייבחר קודם למחיקה.

באלגוריתם נבחר בלוק מתוך הבלוקים המכילים מעט ביותר דפים validים ובכך קטן מספר הכתיבות הפיזיות. בנוסף מתוך הבלוקים המכילים את המספר המינימלי של דפים validים נבחר בלוק שמכיל דפים שיש להם תוחלת חיים ארוכה יותר ובכך נמנע מצב שבו נבחר בלוק לפינוי שמכיל דפים שבעתיד הקרוב יהפכו לinvalid ובעת מחיקת הבלוק מתבזבזים משאבי הזיכרון על העתקה שלהם.

נשים לב כי באלגוריתם זה הדפים שהם valid בבלוק שנבחר להימחק ע"י ה-GC, מועתקים קודם המחיקה ל-Temporary-Buffer, הבלוק נמחק ולאחר מכן הדפים הללו מועתקים חזרה לאותו הבלוק מה-Temporary-Buffer.

הכתובים של הפרויקט הקודם אכן הראו כי אלגוריתם זה משפר את ה-WA ביחס לאלגוריתם Greedy-GC תחת ההנחות שביצעו והצגנו לעיל.

אלגוריתם Generational GC:
הרעיון המרכזי מאחורי האלגוריתם Generational GC הוא שימוש בגיל של הדף הפיזי שיש לכתוב לזיכרון על מנת

לבחור את הבלוק אליו הוא ייכתב כך שהבלוקים יכילו דפים בעלי גיל דומה. לאור העובדה כי דפים אשר גילם קרוב זה לזה חיים ביחד ו"מתים" ביחד, כתיבתם לאותו הבלוק תורמת בשני מישורים: כל עוד הדפים חיים הם ימלאו את הבלוק ויתרמו לנצילות של הזיכרון, ומנגד ברגע שהדפים ימותו, הם ימותו בסמוך אחד לשני ובכך יתרמו להקטנת מספר הדפים הvalidים בבלוק, ובכך ימזערו את ה penalty-שיש לשלם על מחיקת הבלוק (יהיו פחות דפים validים אותם יש להעתיק).

האלגוריתם מממש את הרעיון הנ"ל ע"י חלוקת הדפים הפיזיים לדורות עפ"י גילם. באלגוריתם מוגדרים Tresholds המגדירים מלמטה ומלמעלה את תווך הגילים ששייכים לכל דור. לכול דור מוקצה בלוק, כך שעד אשר הוא מתמלא ניתן לכתוב אליו דפים שגילם מתאים לתווך של אותו הדור בלבד. כלומר קודם כתיבת דף לזיכרון, האלגוריתם מחשב את גילו, מוצא לאיזה דור הוא שייך עפ"י ה-Tresholds ואז כותב אותו לבלוק המוקצה לדור זה. ה-GC אינו רשאי לבחור לפינוי בלוקים אשר מוקצים לדורות השונים. כאשר בלוק של דור מתמלא, הוא מוסף לרשימת הבלוקים מהם ה-GC רשאי לבחור בלוק לפינוי ולאותו הדור מוקצה בלוק פנוי חדש (זאת אך ורק בניסיון הכתיבה הבא לאותו הדור).

האלגוריתם של ה-GC הוא האלגוריתם Greedy Lookahead שתואר לעיל.

הכותבים של העבודה הקודמת הראו כי אלגוריתם זה מביא לשיפור ב-WA ע"י שימוש ב-2 דורות, אך הראו כי מס' הדורות האופטימלי משתנה לכל ערך של OP. מציאת מס' דורות אופטימלי עבור OP מסוים הינה בעיה קשה ולא נפתרה בעבודה הקודמת. אם כן, בעבודה הקודמת שנעשתה פיתחו יוריסטיקה שמחשבת מס' דורות עבור OP שמביאה לתוצאות טובות במקרים השכיחים.

מטרת הפרויקט:

בפרויקט Garbage Collection Algorithms for Flash Memories הוצעו אלגוריתמים נהדרים שתחת הנחות מסוימות הביאו לשיפורים משמעותיים בביצועים של זיכרון SSD. אנו נרצה, תחת הנחות דומות, לפתח יותר את האלגוריתמים שהובאו בעבודה הקודמת על מנת לשפר אף יותר את הביצועים של זיכרון SSD.

Generational GCx:

רעיון כללי ומוטיבציה

ראינו כי בעבודה קודמת כי באלגוריתם Generational GC ה-GC היה נפרד לחלוטין מאלגוריתם הכתיבה. כלומר בעוד שאלגוריתם הכתיבה כתב דפים לבלוקים עפ"י דורות, הכתיבות שנעשו במסגרת ה-GC, כתיבות מחדש של דפים במצב valid שהיו בבלוק שנבחר לפינוי, נכתבו חזרה לאותו הבלוק שזה עתה נמחק. בלוק זה, שעתה יש בו מקומות פנויים נוסף חזרה לרשימה free list ממנה מוקצים בלוקים לדורות. כלומר, כאשר בלוק הוקצה לדור מסוים, הוא לא בהכרח היה פנוי ובסבירות גבוהה שהיו בו דפים valid-ים. למעשה הדפים הללו מפירים את הרעיון של אלגוריתם Generational שמטרתו היא שבבלוקים יהיו דפים ששייכים לאותו הדור כלומר בעלי גילים סמוכים זה לזה.

לכן נרצה שגם ה-GC יכתוב את הדפים אותם פינה מהבלוק שבחר עפ"י הדור <u>העדכני</u> אליו הם משתייכים, ושבלוק שמוקצה לדור יהיה בלוק ריק לחלוטין.

בעבודתו זו בהתבסס על האלגוריתם Generational GC, הכנסנו תיקון לאלגוריתם כך שב-GC בחירת הבלוק לפינוי נעשית אך ורק עפ"י האלגוריתם Greedy Lookahead וכתיבת הדפים לבלוקים, הן הכתיבה בפעם הראשונה והן הכתיבה במסגרת ה-GC נעשית אך ורק ע"י העיקרון של האלגוריתם Generational, כלומר הדף נכתב לבלוק המתאים לדור אליו הדף שייך. נשים לב כי גיל הדף בעת כתיבתו הראשונה לזיכרון, וגילו בכתיבות הבאות ב-GC הינו גיל שונה, שכן ככל שהזמן עובר גילו של הדף הולך וקטן.

נקרא לאלגוריתם המחודש בשם: Generational GCx.

ההשערה היא כי נראה שיפור ב-WA, שכן בעבודה הקודמת ראינו כי כאשר רק אלגוריתם הכתיבה עבד עפ"י דורות ה-WA ירד, אז כעת כאשר רעיון הדורות ישמר בצורה יותר טובה כך השיפור יהיה גדול יותר.

תיאור האלגוריתם:

מבני הנתונים שבשימוש האלגוריתם:

1. הרשימה gc_pages_to_rewrite מכילה דפים שה-GC פינה מבלוק טרם מחיקתו ועליהם להיכתב מחדש לזיכרון.
2. הרשימה free_list מכילה בלוקים ריקים.
3. המערך writing_sequence כפי שתיארנו קודם זהו רצף הכתיבות בגודל N המתקבל מהמשתמש.

נתאר את אלגוריתם הכתיבה של האלגוריתם Generational GCx :

4. i ←0
5. if i < N:
6. while pages_to_rewrite is not empty:
   a. current_page = pages_to_rewite.pop()
   b. write_generational(current_page)
7. write_generational(writing_sequence[i])
8. i ←i+1

כאשר הפונקציה write_generational המקבלת דף לכתיבה כארגומנט פועלת עפ״י האלגוריתם הבא :

1. יש לחשב את הדור אליו הדף שייך עפ״י גילו. נסמן את מס׳ הדור שחושב ב-j.
2. אם מוקצה בלוק לדור j, יש לכתוב את הדף לבלוק ולהמשיך לדף הבא ברצף הכתיבות.
3. אם לא מוקצה בלוק לדור j וגם קיים בלוק פנוי ברשימה free list, אזי יש להסיר בלוק מה-free_list, להקצות אותו לדור j ולחזור לשלב 2.
4. אם לא מוקצה בלוק לדור j וגם הרשימה free list ריקה, יש לקרוא ל-GC ולחזור לשלב 3.

עתה נתאר את אלגוריתם הפינוי של האלגוריתם Generational GCx :

1. בוחר בלוק לפינוי עפ״י אלגוריתם Greedy Lookahead (לא נרחיב כאן).
2. מעתיק את הדפים שבמצב valid לרשימה pages_to_rewite.
3. מוחק את הבלוק.
4. מוסיף את הבלוק המפונה לרשימה free_list.

הסבר - לפני שכותבים את הדף הבא לפי ה-writing sequence, קודם כל נבדוק אם ישנם דפים שה-GC פינה מבלוק ועליהם להיכתב מחדש לזיכרון. במקרה וישנם דפים כאלו, קודם כל הם יכתבו לזיכרון בדיוק באותו האופן בו דפים מה-writing_sequence היו נכתבים, עפ״י גילם העדכני. רק לאחר שמסיימים לכתוב את הדפים שהגיעו מה-GC נמשיך עם שאר הבקשות מה-writing sequence.

מימוש האלגוריתם:

שיפור האלגוריתם ממומש בסימולטור. הוראות ההפעלה נמצאות בקובץ md.README בדף ה Github-של הפרויקט.

ניסויים ותוצאות

ניסוי מס׳ 1 : בדיקת ביצועים Generational GCx

נרצה לבחון את ביצועי האלגוריתם Generational GCx אל מול אלגוריתם Generational GC .

תנאי הניסוי : מס׳ דורות – 2, $T = 96, Z = 256, N = 100000, page\_size = 4k$ ערכו של U משתנה על מנת לקבל ערכי OP שונים. ה- WA המוצג הינו ממוצע של 20 הרצות.

תוצאות הניסוי ומסקנות : בטבלה הבאה ניתן לראות את ערכי ה-WA עבור ערכי OP שונים שהתקבלו עבור כל אחד משני האלגוריתמים שברצוננו להשוות. ערכי ה-WA הינם ממוצע על פני 20 הריצות שביצענו :

OP	Generational GC	Generational GCx
0.066667	7.988795	4.3415355
0.103448	5.4313575	3.246884
0.142857	4.1236275	2.651138
0.185185	3.353043	2.275165
0.230769	2.805717	2.0133355
0.28	2.395226	1.814471
0.333333	2.060539	1.6657145
0.391304	1.809502	1.5460975
0.454545	1.628125	1.4516645
0.52381	1.4892315	1.3734205
0.6	1.3802415	1.307948
0.684211	1.2965215	1.2506775
0.777778	1.230225	1.20143

מתוצאות הניסוי ניתן להסיק כי האלגוריתם Generational GCx אכן מביא לשיפור משמעותי ביחס לאלגוריתם Generational GC הבסיסי, כאשר ככל שה-OP נמוך יותר השיפור משמעותי יותר. נרצה לראות את השיפור באחוזים:

OP	Improvement:
0.066667	45.65%
0.103448	40.22%
0.142857	35.71%
0.185185	32.15%
0.230769	28.24%
0.28	24.27%
0.333333	19.16%
0.391304	14.56%
0.454545	10.84%
0.52381	7.78%
0.6	5.24%
0.684211	3.54%
0.777778	2.34%

סה"כ אכן רואים שיפור משמעותי שמאשש את ההשערה שלנו כי באלגוריתם הקודם ה-GC הפר את עיקרון הדורות, וכאשר הפרה זו תוקנה הביצועים עלו.

ניסוי מס' 2: בדיקה כי הגורם המשפיע על השיפור הוא ה-OP ולא גורם אחר.

נרצה לבדוק כי השיפור שקיבלנו בניסוי מס' 1 ממשיך להתקבל גם עבור ערכים שונים של T, מס' בלוקים פיזיים, וערכי U מתאימים לקבלת אותם ערכי OP. למעשה אנו רוצים לבודד את ה-OP כגורם היחיד שמשפיע על קיום השיפור וגודלו.

תנאי הניסוי: מס' דורות – 2, $Z = 256, N = 100000, page\_size = 4k$ ערכו של U משתנה על מנת לקבל ערכי OP שונים. ה- WA המוצג הינו ממוצע של 20 הרצות.

OP	T=64		T=96		T=128		T=256	
	Generational GC	Generational GCx	Generational GC	Generational GCx	Generational GC	Generational GCx	Generational GC	Generational GCx
0.066667	6.531557	3.564221	7.988795	4.120323	7.106123	3.735837	7.471336	3.842759
0.103448	4.7258725	2.8082195	5.4313575	3.086675	5.0059195	2.9359115	5.1731375	3.040166
0.142857	3.728663	2.368184	4.1236275	2.5306795	3.88358	2.456161	3.97889	2.597176
0.185185	3.3791525	2.200526	3.353043	2.178944	3.3400135	2.2108575	3.333912	2.281786
0.230769	2.6158365	1.853105	2.805717	1.9333605	2.685209	1.9231925	2.726218	2.0211755
0.28	2.414956	1.7691075	2.395226	1.750985	2.383743	1.7888535	2.3712205	1.8663615
0.333333	2.076383	1.62559	2.060539	1.612622	2.0525725	1.6433445	2.0449875	1.704106
0.391304	1.7231435	1.4614995	1.809502	1.493166	1.758351	1.5021565	1.787741	1.575171
0.454545	1.55821	1.382021	1.628125	1.404766	1.588907	1.4166265	1.619946	1.472075
0.52381	1.4317955	1.315249	1.4892315	1.33188	1.460508	1.345805	1.4919965	1.398435
0.6	1.3807485	1.286049	1.3802415	1.2747505	1.3845935	1.2993885	1.4038535	1.3338355
0.684211	1.2938275	1.230729	1.2965215	1.21891	1.301638	1.245127	1.3236565	1.2806195
0.777778	1.1947115	1.1607545	1.230225	1.171305	1.2197305	1.18417	1.250626	1.22036

אכן ניתן לראות כי השיפור קיים לכל ערכי ה-T השונים. נראה את ערכי השיפור באחוזים:

OP	T=64	T=96	T=128	T=256
0.066667	45.43076	48.42372	47.42792	48.56664
0.103448	40.57776	43.16936	41.3512	41.23168
0.142857	36.48705	38.62977	36.75524	34.72612
0.185185	34.87935	35.01592	33.80693	31.5583
0.230769	29.15823	31.09211	28.37829	25.86156
0.28	26.7437	26.89688	24.95611	21.2911
0.333333	21.71049	21.73786	19.93732	16.66912
0.391304	15.18411	17.48194	14.57016	11.89042
0.454545	11.30714	13.71879	10.84271	9.128144
0.52381	8.139885	10.56595	7.853637	6.270893
0.6	6.858563	7.642938	6.153792	4.987557
0.684211	4.876887	5.986133	4.34153	3.251372
0.777778	2.842276	4.789368	2.915439	2.420068

בטבלה לעיל נח יותר לראות כי אכן עבור ערכי T שונים אך אותו הערך של OP קיים שיפור בשיעור דומה עבור האלגוריתם Generational GCx על פני האלגוריתם Generational GC.

ניסוי מס' 3: מציאת מס' דורות אופטימלי לכל OP תחת האלגוריתם Generational GCx.

לאחר שראינו כי עבור 2 דורות האלגוריתם Generational GCx בעל ביצועים טובים משמעותית מ-Generational GC יש לנו מוטיבציה להשתמש בו גם עבור מס' דורות גדול יותר, זאת במיוחד לאור בעובדה שבעבודה הקודמת נמצא כי עבור ערכי OP מסוימים מס' דורות גדול יותר מביא להורדת ה-WA. אולם מכיוון שמס' הדורות האופטימלי עבור כל OP שהוצע בעבודה הקודמת נעשה על בסיס יוריסטיקה וניסויים ספציפיים עבור האלגוריתם Generational GC, לא נוכל להשתמש במסקנות של העבודה הקודמת לאלגוריתם שלנו.

כפי שניתן לראות מתוצאות הניסויים והאנליזות שבוצעו בעבודה הקודמת, הבעיה הכללית של מציאת מס' הדור אופטימלי לכל קומבינציה של הפרמטרים T,Z,U היא בעיה קשה מאד. לכן בפרויקט זה עבור כל ערך של OP, הרצנו את האלגוריתם המשופר עם מספר דורות בטווח של 2-9, וראינו עבור כל OP מהו מספר הדורות שבו קיבלנו את הWA הגבוה ביותר.

תנאי הניסוי: מס' דורות – 2, $T = 96, Z = 256, N = 100000, page\_size = 4k$ ערכו של U משתנה על מנת לקבל ערכי OP שונים. ה- WA המוצג הינו ממוצע של 20 הרצות.

תוצאות ומסקנות:

OP	2 generations	3 generations	4 generations	5 generations	6 generations	7 generations	8 generations	9 generations
0.066667	4.352193	3.809791	3.530776	3.393577	3.386446	3.413093	3.533414	3.666802
0.103448	3.24004	2.87173	2.64904	2.536928	2.493882	2.458806	2.445174	2.503175
0.142857	2.653427	2.378004	2.204647	2.125496	2.077433	2.057109	2.038067	2.056041
0.185185	2.274866	2.060249	1.921792	1.863142	1.824809	1.80982	1.794362	1.808922
0.230769	2.011848	1.843938	1.734214	1.677339	1.653413	1.637319	1.626896	1.639461
0.28	1.813612	1.685823	1.589278	1.546913	1.516376	1.512147	1.504445	1.511413
0.333333	1.665261	1.559025	1.478495	1.440362	1.416076	1.412831	1.409007	1.412452
0.391304	1.547153	1.458564	1.389779	1.355928	1.33886	1.33378	1.333454	1.335065
0.454545	1.450728	1.378368	1.318303	1.289575	1.275212	1.272329	1.270459	1.267776
0.52381	1.373352	1.309463	1.257642	1.233991	1.223351	1.219499	1.216114	1.216413
0.6	1.30854	1.2489	1.202663	1.189965	1.178984	1.176285	1.171071	1.174713
0.684211	1.250998	1.201036	1.160345	1.151752	1.14302	1.140772	1.137236	1.140193
0.777778	1.201146	1.159035	1.126278	1.116155	1.114057	1.110119	1.108206	1.110388

הערכים המסומנים בטבלה הינם הערכים האופטימליים ביותר שהתקבלו בניסוי.

מניסוי חלקי זה אכן ניתן לראות כי מס' הדורות האופטימלי שהתקבל עבור כל OP תחת האלגוריתם Generational GCx שונה ממס' הדורות האופטימלי תחת האלגוריתם Generational GC ולא ניתן להכליל את היוריסטיקה שפותחה בעבודה הקודמת לעבודה שלנו.

ולכן כאשר מריצים את האלגוריתם Generational GCx בסימולטור אפשרנו למשתמש לבחור להריץ את הסימולציה כאשר מס' הדורות נבחר אוטומטית להיות מס' הדורות הטוב ביותר שנמצא עבור ערך ה-OP המתקבל. הסבר מפורט נמצא בדף ה- Github של הפרויקט.

עתה שמצאנו מס' דורות לכל OP שנותן WA אופטימלי ביחס לערכים שבדקנו, נוכל להשוות את ביצועי האלגוריתם Generational GCx לעומת Generational GC עבור המקרה של מס' דורות שאינו 2.

תנאי הניסוי: $T = 96, Z = 256, N = 100000, page\_size = 4k$ ערכו של U משתנה על מנת לקבל ערכי OP שונים. ה- WA המוצג הינו ממוצע של 20 הרצות.

תוצאות ומסקנות:

	Generational GC		Generational GCx		
OP	**number of generations**	**avrage WA**	**number of generations**	**avrage WA**	**improvement:**
0.066667	5	7.930496	6	4.3415355	45.26%
0.103448	5	5.415658	8	3.246884	40.05%
0.142857	5	4.1431605	8	2.651138	36.01%
0.185185	5	3.3627505	8	2.275165	32.34%
0.230769	5	2.843042	8	2.0133355	29.18%
0.28	4	2.4728915	8	1.814471	26.63%
0.333333	4	2.196501	8	1.6657145	24.17%
0.391304	4	1.980372	8	1.5460975	21.93%
0.454545	4	1.8098365	7	1.4516645	19.79%
0.52381	4	1.6709325	8	1.3734205	17.81%
0.6	3	1.556192	8	1.307948	15.95%
0.684211	3	1.4609085	8	1.2506775	14.39%
0.777778	3	1.380419	8	1.20143	12.97%

בטבלה לעיל אנו רואים כי השיפור של האלגוריתם Generational GCx ביחס לאלגוריתם בגרסתו הקודמת מביא לשיפר משמעותי גם עבור מס' דורות גדול מ-2 כאשר בשונה מ-2 דורות, שיעור השיפור עבור OP גבוהים גדול יותר.

מתוצאות ניסוי זה וכן מתוצאות הניסויים הקודמים ניתן להסיק כי אכן כדאי האלגוריתם Generational GCx עדיף מבחינת ביצועים על פני האלגוריתם הקודם Generational GC.

Generational GC's threshold:

באלגוריתם Generational GC, עבור 2 דורות, קיים threshold שערכו שווה ל-$\frac{U \cdot Z}{2}$, כך שדף לוגי שגילו קטן מה-threshold משויך לדור 1 ואילו דף שגילו גדול מה- threshold משויך לדור 2. רעיון זה מוכלל עבור מס' דורות גדול יותר באופן הבא: עבור k מס' הדורות, דף ישויך לדור i, $1 \leq i < k$, אם גילו אינו קטן מ-$(i-1) \cdot \frac{U \cdot Z}{k}$ וכן גילו קטן מ-$i \cdot \frac{U \cdot Z}{k}$, ודף ישויך לדור k אם גילו גדול או שווה ל-$(k-1) \cdot \frac{U \cdot Z}{k}$.

הבחירה של ה-threshold מבוססות על ההנחה כי תחת התפלגות יוניפורמית של הדפים לכתיבה, גיל של דף בהסתברות גבוהה יהיה לכול היותר קירוב של Z*U, ולכן על מנת שהעומס על כל דור יהיה דומה היה הגיוני לחלק את תווך הגילים בין 0 לבין Z*U למס' הדורות. אולם הנחה זו לא נבדקה בפרויקט הקודם. ולכן נרצה לבדוק אם ההנחה שנעשתה אכן עומדת במבחן המציאות, או אולי קיים threshold שונה שייתן תוצאות טובות יותר.

ניסוי מס' 4 – Thresholds Comparison:

לצורך ניסוי זה שינינו את הסימולטור כך שכאשר מריצים אותו עבור אלגוריתם Generational GC הוא מקבל ארגומנט נוסף, threshold, אשר קובע את ה-thresholds של האלגוריתם השמורים במערך k_bounds באופן הבא:

```
for (int i = 0;  i < number_of_generation -1;  i++) {

        k_bounds[i] = threshold * (i + 1);

}
```

כלומר עתה הפונקציה getGeneration(page_index_in_writing_sequence) אשר קובעת את הדור של דף תבצע:

```
page_age = getPageAge(page_index_in_writing_sequence)

for (int i = 0;  i < num_of_gens - 1;  ++i) {
   if (page_score < k_bounds[i])
      return i + 1;
}

 return num_of_gens;
```

נבחן את ערכי ה-WA

המתקבלים עבור thresholds שונים.

תנאי הניסוי: מס' דורות – 2, $T = 96, Z = 256, N = 100000, page\_size = 4k$ ערכו של U משתנה על מנת לקבל ערכי OP שונים. ה- WA המוצג הינו ממוצע של 20 הרצות.

OP	U*Z	U*Z*0.9	U*Z*0.7	U*Z*0.5	U*Z*0.3
0.066667	5.05087	4.87284	4.510113	4.338042	4.521316
0.103448	3.53765	3.42524	3.27249	3.25073	3.418687
0.142857	2.77934	2.717192	2.637952	2.650705	2.804723
0.185185	2.324229	2.292695	2.252133	2.275741	2.401049
0.230769	2.027136	2.001989	1.986159	2.010302	2.122756
0.28	1.817701	1.802351	1.787031	1.814721	1.915764
0.333333	1.654645	1.641074	1.637801	1.666121	1.751861
0.391304	1.522645	1.517139	1.5171	1.546299	1.621808
0.454545	1.426507	1.424427	1.425751	1.451347	1.519221
0.52381	1.35192	1.348557	1.350309	1.373417	1.432579
0.6	1.285677	1.282176	1.285418	1.307801	1.359319
0.684211	1.225932	1.223879	1.230089	1.250723	1.295037
0.777778	1.174484	1.173661	1.180875	1.201061	1.240343

הערכים המודגשים בטבלה הינם הערכים האופטימליים ביותר שהתקבלו בניסוי. נשווה את התוצאות שקיבלנו בניסוי זה לתוצאות קודמות:

OP	old threshold:	best found threshold:	Improvement:
0.066667	4.3415355	4.3380415	0.08%
0.103448	3.246884	3.2507295	-0.12%
0.142857	2.651138	2.6379515	0.50%
0.185185	2.275165	2.2521325	1.01%
0.230769	2.0133355	1.986159	1.35%
0.28	1.814471	1.787031	1.51%
0.333333	1.6657145	1.6378005	1.68%
0.391304	1.5460975	1.5170995	1.88%
0.454545	1.4516645	1.4244265	1.88%
0.52381	1.3734205	1.3485565	1.81%
0.6	1.307948	1.282176	1.97%
0.684211	1.2506775	1.223879	2.14%
0.777778	1.20143	1.1736605	2.14%

אכן ניתן לראות כי ה- threshold הקודם אינו אופטימלי וכי ניתן לקבל WA נמוך יותר ע"י שימוש ב-thresholds שונים. אמנם השיפור שהתקבל עבור חלק מערכי ה-OP אינו משמעותי ומסתכם בשיפור של עד 2.3%. בנוסף, נראה כי עבור ערכי OP נמוכים, שאלו הם המקרים המעניינים יותר, ה- threshold הקודם נותן את התוצאות הטובות ביותר על פני ה- thresholds שנבדקו. כלומר לאור תוצאות הניסוי נראה כי הבחירה של ה-threshold במקרה של 2 דורות להיות $\frac{U \cdot Z}{2}$ הייתה בחירה טובה אך לא אופטימלית.

עם זאת, מעניין לראות בתוצאות הניסוי כי קיים קשר בין ערך ה-threshold לערך ה-OP כך שעבור OP נמוך יותר נעדיף להגדיל את תווך הגילים של דור 2 על חשבון דור 1. נחזור לנקודה זו בהמשך.

ניסוי מס' 5 –Median threshold:

בניסוי קודם הגענו למסקנה כי הבחירה של ה- threshold באלגוריתם Generational GC אינה אופטימלית. מכאן המוטיבציה לנסות ולמצוא threshold שישפר את ביצועי האלגוריתם.

הבחירה ב-threshold נעשתה במטרה לגרום לכך שהעומס על הדורות השונים יהיה דומה עד כמה שניתן. אך מכיוון שבהנחות הפרויקט אנו למעשה יודעים את כל הכתיבות העתידיות שלנו לזיכרון, אין צורך להשתמש ב-threshold

שיביא לכך שבקירוב או באופן הסתברותי העומס על הדורות השונים יהיה דומה, אלא נוכל להשתמש בעובדה "שאנו יודעים את העתיד" על מנת לגרום לכך שהעומס על הדורות השונים יהיה בקירוב טוב מאוד שווה. עבור 2 דורות, נעשה זאת ע"י קביעת ה- threshold כחציון הגילים של הדפים בתור הכתיבות. עבור מס' דורות גדול יותר ניתן להכליל את הרעיון של חציון בקלות.

<u>הערה:</u> מכיוון שגיל דף אינו חח"ע ויכולים להיות מס' רב של דפים בעלי אותו הגיל, החלוקה לדורות עפ"י חציון אינה מחלקת בפועל את העומס הלוגי לחצי בדיוק. לדוגמא עבור רצף הדפים הבא: A,B,A,B,A,B,C הגילים של הדפים ברצף יהיו בהתאמה: 2,2,2,2,3,2,1. הגיל החציוני של רצף זה יקבע להיות 2. וכך כל הדפים שגילם קטן מ-2 יכתבו לדור הצעיר, שזה למעשה רק דף יחיד לעומת 6 דפים שיכתבו לדור המבוגר. כלומר העומס על הדורות אינו שווה. אך מבדיקות שביצענו ראינו כי העומס אכן מתחלק בקירוב טוב מאוד לחלוקה של 50-50.

לשם כך שינינו את הסימולטור כך שניתן לבחור להריץ סימולציה של האלגוריתם Generational GC עם threshold שהוא החציון (או הכללה של חציון עבור מס' דורות גדול מ-2).

תנאי הניסוי: מס' דורות – 2, $T = 96, Z = 256, N = 100000, page\_size = 4k$ ערכו של U משתנה על מנת לקבל ערכי OP שונים. ה- WA המוצג הינו ממוצע של 20 הרצות.

OP	best previous bound:	median bound:
0.066667	4.3380415	4.36294
0.103448	3.2507295	3.235257
0.142857	2.6379515	2.63687
0.185185	2.2521325	2.259828
0.230769	1.986159	1.996166
0.28	1.787031	1.800322
0.333333	1.6378005	1.650656
0.391304	1.5170995	1.529297
0.454545	1.4244265	1.435369
0.52381	1.3485565	1.3599165
0.6	1.282176	1.2937975
0.684211	1.223879	1.237828
0.777778	1.1736605	1.188819

מהתוצאות ניתן לראות כי הלכה למעשה הרעיון של חציון אינו עומד במבחן המציאות והוא מביא לתוצאות פחות טובות מה-thresholds שמצאנו בניסוי קודם.

אולם אם ננתח את אופן פעולת החציון, למעשה הדור אליו ישתייך דף כלשהו מושפע מכל הדפים שהולכים להיכתב בעתיד המאוד רחוק או מהדפים שנכתבו בעבר המאוד רחוק כאשר במציאות לרב אין השפעה של דפים שנכתבים בהפרשים כל כך גדולים האחד על השני. ולכן נרצה לנסות להשתמש ברעיון של חציון, אך במקום לחשב את החציון על פני כל הדפים בתור הכתיבות, נעבוד באינטרוולים, כך שבתחילת כל אינטרוול נחשב את ה- thresholds בשיטת החציון ונכתוב את הדפים אשר באינטרוול זה לזיכרון על פי thresholds אלו. באופן זה נדאג שבקירוב בכול נקודת זמן של הזיכרון העומס על הדורות יהיה שווה.

כלומר עבור האלגוריתם Generational GC עם threshold דינאמי בשיטת החציון כתיבת הדפים בתור הכתיבות תעשה באופן הבא:

```
for (n = 0; n < writing_sequence_len / interval; n++) {
   calcKBounds(n * interval, interval);
   for (i = n * interval; i < n * interval + interval; ++i) {
      generation = getGeneration(i, num_of_gens);
      writeGenerational(i, generation, writing_sequence);
   }
}
```

כאשר הפונקציה calcKBounds(base_index, interval_size) מחשבת את "החציון המוכלל" של הגילים של הדפים באינטרוול שמתחיל ב-base_index וגודלו interval_size ומאתחלת בהתאם את המערך k_bounds.

ניסוי מס' 6 –Dynamic-Median threshold :

נרצה לבדוק אם האלגוריתם Generetional GC עם thresholds הנקבעים באופן דינאמי, כלומר משתנים לאורך האלגוריתם, בשיטת החציון יביא לירידה של ה-WA ביחס לתוצאות הקודמות שקיבלנו.

לשם כך שינינו את הסימולטור כך שעבור Generetional GC עם thresholds בשיטת החציון מתקבל ארגומנט נוסף מהמשתמש, interval, שקובע את גודל האינטרוול עליו האלגוריתם יחשב את החציון (או הכללתו).

תנאי הניסוי : מס' דורות – 2, $T = 96, Z = 256, N = 100000, page\_size = 4k$ ערכו של U משתנה על מנת לקבל ערכי OP שונים. ה- WA המוצג הינו ממוצע של 20 הרצות.

OP	previous results:	interval size:			
		N	U*Z	U*Z*0.5	U*Z*0.2
0.066667	4.338042	4.36294	4.187368	4.168608	4.1661395
0.103448	3.25073	3.235257	3.110019	3.1069955	3.086675
0.142857	2.637952	2.63687	2.553265	2.5291885	2.5306795
0.185185	2.252133	2.259828	2.20858	2.187364	2.184831
0.230769	1.986159	1.996166	1.951971	1.9400575	1.938343
0.28	1.787031	1.800322	1.760455	1.758327	1.760114
0.333333	1.637801	1.650656	1.619665	1.623496	1.6178125
0.391304	1.5171	1.529297	1.511922	1.5026895	1.5021565
0.454545	1.424427	1.435369	1.424453	1.4185915	1.415647
0.52381	1.348557	1.359917	1.342848	1.344776	1.3434605
0.6	1.282176	1.293798	1.285274	1.2806915	1.2830985
0.684211	1.223879	1.237828	1.23333	1.230376	1.228322
0.777778	1.173661	1.188819	1.180513	1.181886	1.1801435

הערכים המסומנים בטבלה הינם הערכים האופטימליים ביותר שהתקבלו בניסוי. אנו רואים כי עבור האינטרוול הקטן ביותר שנבדק מתקבלות התוצאות הטובות ביותר עבור רב ערכי ה-OP, בהתאם להנחה כי כתיבות רחוקות אינן משפיעות על הכתיבה הנוכחית.

עבור ערכי ה-OP עבורם התקבלו תוצאות אופטימליות באינטרוול 0.5*U*Z, ההבדל ב-WA האופטימלי לזה שהתקבל באינטרוול 0.2*U*Z הינו הבדל קטן, ולכן לצורך השוואת התוצאות שהתקבלו, נשתמש בתוצאות של האינטרוול 0.2*U*Z.

נשווה את התוצאות שקיבלנו בניסוי זה לתוצאות הטובות ביותר שקיבלנו בניסוי מס' 4 :

OP	best previous bound:	dynamic median bound with intervals:	imrovement:
0.066667	4.3380415	4.1661395	3.96%
0.103448	3.2507295	3.086675	5.05%
0.142857	2.6379515	2.5306795	4.07%
0.185185	2.2521325	2.184831	2.99%
0.230769	1.986159	1.938343	2.41%
0.28	1.787031	1.760114	1.51%
0.333333	1.6378005	1.6178125	1.22%
0.391304	1.5170995	1.5021565	0.98%
0.454545	1.4244265	1.415647	0.62%
0.52381	1.3485565	1.3434605	0.38%
0.6	1.282176	1.2830985	-0.07%
0.684211	1.223879	1.228322	-0.36%
0.777778	1.1736605	1.1801435	-0.55%

ניתן לראות כי עבור ערכי ה-OP הקטנים יותר ישנו שיפור ב-WA, וכן ניתן לזהות מגמה כללית כך שככל שה-OP גדל אחוז השיפור קטן כאשר החל מ-OP של 0.6 מתקבלות תוצאות פחות טובות מה-threshold הקודם שנבדק.

כלומר ההנחה כי עומס זהה על 2 הדורות יביא לתוצאות אופטימליות מתבררת כהנחה שגויה.

מניסוי זה וכן מניסוי מס׳ 3 נוכל להסיק כי יש קשר בין ה-OP לבין ה- threshold האופטימלי. כלומר לא בהכרח שתמיד נרצה כי העומס על הדורות יהיה שווה, וגודל העמוס על כל דור יכול להיות תלוי ב-OP של הזיכרון.

ננסה לנתח את השפעת ערך ה-OP על ה-threshold האופטימלי. למעשה ערך ה-threshold קובע את חלוקת העומס בין הדורות השונים. עבור ערך OP נמוך, ה-GC נקרא לעיתים תכופות יותר, כלומר בסבירות גבוהה מס׳ הדפים החוקיים (valid) שיש לכתוב מחדש בעת פינוי בלוק גדול יותר. השפעה אפשרית של עובדה זו יכולה להיות הגדלת עומס על הדורות הנמוכים (הצעירים), שכן גילו של דף שנכתב מחדש לזיכרון קטן ביחס לגילו המקורי בזמן כתיבתו לראשונה לזיכרון. לעומת זאת עבור OP גבוה, ה-GC מופעל לעיתים רחוקות יותר כך שפחות דפים נכתבים מחדש לזיכרון, ולכן ייתכן כי העמסה על הדורות הצעירים במקרה זה ע״י ה-GC אינה משמעותית כמו במקרה הקודם. כמו כן מכיוון שעובר זמן רב יותר בין 2 קריאות סמוכות ל-GC, יותר דפים הופכים להיות לא חוקיים (invalid) עד הקריאה הבאה ל-GC ואולי משום כך בבלוקים של הדורות הצעירים מס׳ הדפים שהם בלתי חוקיים גדול יותר, שכן גילם קטן יותר. במקרה זה ייתכן שניתן להגדיל את העומס על הדורות הצעירים ביחס לדורות המבוגרים ובכך הבלוקים של הדורות הצעירים יתמלאו מהר יותר ולכן כאשר ה-GC ייקרא יהיו לו יותר בלוקים ששייכים לדורות הצעירים לבחור מתוכם בלוק לפנות כאשר ההנחה היא שבבלוקים אלו יהיה מס׳ דפים חוקיים קטן יותר אותם יש לכתוב מחדש.

כדי לבדוק את נכונות ההשערות שהעלנו לעיל, נחזור על ניסוי מס׳ 3 כאשר הפעם נבדוק מהו אחוז הכתיבות הלוגיות לכול דור ומהו אחוז הכתיבה הפיזית לכול דור. כאשר אחוז הכתיבות הלוגיות לדור i יחושב ע״י:

$$\frac{number\ of\ logical\ writes\ to\ generetaion\ i}{N} \cdot 100$$

ואילו אחוז הכתיבות הפיזיות לדור i יחושב ע״י:

$$\frac{number\ of\ physical\ writes\ to\ generetaion\ i}{number\ of\ physical\ writes} \cdot 100$$

אחוזי הכתיבה המוצגים בטבלה הינם ממוצע על פני 20 הריצות.

OP	Treshold 0.9*U*Z			Treshold 0.7*U*Z			Treshold 0.5*U*Z		
	WA	Logical writes to gen 1	Physical writes to gen 1	WA	Logical writes to gen 1	Physical writes to gen 1	WA	Logical writes to gen 1	Physical writes to gen 1
0.066667	4.87	67.70%	89.13%	4.51	58.34%	83.65%	4.34	46.30%	75.37%
0.103448	3.43	67.50%	85.34%	3.27	58.08%	78.92%	3.25	46.11%	69.79%
0.142857	2.72	67.23%	82.23%	2.64	57.83%	75.28%	2.65	45.85%	65.65%
0.185185	2.29	66.91%	79.67%	2.25	57.53%	72.36%	2.28	45.63%	62.36%
0.230769	2	66.66%	77.49%	1.99	57.27%	69.91%	2.01	45.43%	59.54%
0.28	1.8	66.38%	75.70%	1.79	57.02%	67.78%	1.81	45.16%	57.13%
0.333333	1.64	66.08%	73.96%	1.64	56.73%	65.87%	1.67	44.98%	55.11%
0.391304	1.52	65.77%	72.44%	1.52	56.45%	64.21%	1.55	44.70%	53.32%
0.454545	1.42	65.52%	71.19%	1.43	56.24%	62.76%	1.45	44.47%	51.75%
0.52381	1.35	65.21%	70.05%	1.35	55.97%	61.45%	1.37	44.23%	50.39%
0.6	1.28	64.91%	68.90%	1.29	55.68%	60.21%	1.3	44.00%	49.19%
0.684211	1.22	64.62%	67.85%	1.23	55.35%	59.17%	1.25	43.78%	47.94%
0.777778	1.17	64.39%	66.83%	1.18	55.15%	58.05%	1.2	43.52%	46.93%

*בטבלה מוצגים אחוזי הכתיבה לדור 1 בלבד, אחוזי הכתיבה לדור 2 משלימים כמובן ל-100%.

**התוצאות המסומנים בכחול אלו התוצאות אופטימליים.

מסקנות מתוצאות הניסוי:

1. אכן האלגוריתם של ה-GC מגדיל את העומס הפיזי על הבלוקים ששייכים לדור הצעיר – דור 1, כך שככל שה-OP קטן יותר, העומס הפיזי על הדור הצעיר גדול יותר. זאת כפי שציפינו.
2. ה-WA האופטימלי שנמצא עבור ערך OP מסוים מתקבל עבור עומס פיזי על הדור הצעיר הנע בין 64%-75%, כלומר בכול המקרים זהו עומס פיזי שגדול משמעותית מ-50%. סה"כ ההנחה הראשונית שלנו כי עומס שווה על הדורות יביא לתוצאות טובות יותר היא אכן הנחה שגויה. (בתוצאות שקיבלנו עבור גדלי אינטרוולים נוספים ושלא הבאנו כאן על מנת שלא להעמיס, התקבלו מקרים של עומס פיזי קרוב מאוד ל-50% -50% על שני הדורות, וה-WA אכן היה גבוה יותר מזה שמתקבל בטבלה לעיל).
3. נראה כי יש בה אמת בהנחה כי ככל שה-OP גדול יותר כך עדיף להעמיס יותר על הדור הצעיר, אולם התוצאות אינן חד-משמעיות.

לאור המסקנות הללו הוספנו אפשרות להריץ את האלגוריתם Generational GC כך שיחלק את העומס הלוגי בין הדורות עפ"י ארגומנטים המתקבלים מהמשתמש וכך ניתן לשלוט באופן עקיף על העומס הפיזי על הדורות. חלוקת העומס נעשית באופן דומה לשיטת החציון באינטרוולים. גם את גודל האינטרוול מקבלים מהמשתמש.

הוראות הפעלה ניתן לראות בקובץ README.md שבדף ה-Github של הפרויקט.

ניסוי מס' 7 – Load-Balancing Generational GC:

נריץ עתה את האלגוריתם עם אחוז עומס לוגי משתנה על הדורות במטרה למצוא את העומס הלוגי האופטימלי עבור ערכי OP שונים.

תנאי הניסוי: מס' דורות – 2, $T = 96, Z = 256, N = 100000, page\_size = 4k$ ערכו של U משתנה על מנת לקבל ערכי OP שונים. גודל האינטרוול הוא $U \cdot Z \cdot 0.2$, עבורו קיבלנו תוצאות טובות יותר בניסוי מס' 5 עם החציון. ה-WA המוצג הינו ממוצע של 20 הרצות.

op	Logical load on gen 1				
	45%	50%	55%	60%	65%
0.066667	4.120323	4.16614	4.270132	4.425318	4.63572
0.103448	3.088701	3.086675	3.117452	3.188493	3.298047
0.142857	2.543575	2.53068	2.538878	2.571659	2.63351
0.185185	2.200436	2.184831	2.178944	2.189931	2.22586
0.230769	1.955763	1.938343	1.933361	1.93839	1.957898
0.28	1.775672	1.760114	1.750985	1.753363	1.764703
0.333333	1.631668	1.617813	1.612622	1.612799	1.620533
0.391304	1.516869	1.502157	1.495475	1.493166	1.499157
0.454545	1.427422	1.415647	1.407784	1.404766	1.406057
0.52381	1.352944	1.343461	1.33672	1.33188	1.333567
0.6	1.29244	1.283099	1.276427	1.274751	1.275532
0.684211	1.236585	1.228322	1.222019	1.21891	1.219852
0.777778	1.18865	1.180144	1.17432	1.17173	1.171305

התוצאות המסומנים בטבלה אלו הם ערכי ה-WA האופטימליים שהתקבלו בניסוי עבור ערך OP מסוים.

אכן ניתן לראות בצורה יפה כי ככל שערך ה-OP גדול יותר נרצה שהעומס הלוגי על דור 1 יהיה גדול יותר על מנת לקבל WA נמוך יותר.

נשווה את תוצאות הניסוי האחרון לתוצאות ניסוי מס 4' ונבחן את השיפור שקיבלנו באחוזים:

op	Exp #4 results:	Exp #7 results:	Improvement:
0.066667	4.338042	4.120323	5.02%
0.103448	3.25073	3.086675	5.05%
0.142857	2.637952	2.53068	4.07%
0.185185	2.252133	2.178944	3.25%
0.230769	1.986159	1.933361	2.66%
0.28	1.787031	1.750985	2.02%
0.333333	1.637801	1.612622	1.54%
0.391304	1.5171	1.493166	1.58%
0.454545	1.424427	1.404766	1.38%
0.52381	1.348557	1.33188	1.27%
0.6	1.282176	1.274751	0.58%
0.684211	1.223879	1.21891	0.40%
0.777778	1.173661	1.171305	0.20%

אנו יכולים לראות כי אכן בחירה טובה של העומס הלוגי על דור 1 יכולה להביא לשיפור ב-WA ביחס לתוצאות הטובות ביותר שקיבלנו עבור ערכי threshold סטטיים שונים. אולם לאכזבתנו השיפור אינו משמעותי.

כמסקנה מניסוי זה הוספנו לסימולטור אפשרות לבחור את העומס הלוגי האופטימלי עפ"י הניסוי האחרון בהתאם לנתוני הזיכרון האחרים שמתקבלים מהמשתמש. הוראות הרצה ניתן לראות בקובץ README.md בדף Github של הפרויקט.

סיכום ביניים:

עד כה מצאנו 2 שיפורים לאלגוריתם Generational GC המקורי, הראשון היה עדכון האלגוריתם של ה-GC עצמו כך שעבור הדפים ה-valid-ים שנאספו מבלוק שנמחק, כתיבתם מחדש נעשית עפ"י אלגוריתם הכתיבה לפי דורות, שיפור לו

קראנו Generational GCx. עדכון זה של האלגוריתם הביא לשיפור המשמעותי ביותר ב-WA בפרויקט עד כה. כמו כן הוספנו אפשרות למשתמש לבחור את העומס הלוגי על הדורות כך שבחירה טובה של עומס לוגי על הדורות יכולה גם היא להביא לשיפור ב-WA, אם כי לא משמעותי עפ״י הניסויים שביצענו. לשילוב 2 השיפורים יחד נקרא Generational GCx with Load balancing. נבצע השוואה של התוצאות הטובות ביותר שקיבלנו עבור 2 השיפורים לעיל למול התוצאות הטובות ביותר שהתקבלו בפרויקט הקודם אותו אנו ממשיכות:

op	Generational GC:	Generational GCx with Load Balancing:	Improvement:
0.066667	7.988795	4.120323	48.42%
0.103448	5.4313575	3.086675	43.17%
0.142857	4.1236275	2.5306795	38.63%
0.185185	3.353043	2.178944	35.02%
0.230769	2.805717	1.9333605	31.09%
0.28	2.395226	1.750985	26.90%
0.333333	2.060539	1.612622	21.74%
0.391304	1.809502	1.493166	17.48%
0.454545	1.628125	1.404766	13.72%
0.52381	1.4892315	1.33188	10.57%
0.6	1.3802415	1.2747505	7.64%
0.684211	1.2965215	1.21891	5.99%
0.777778	1.230225	1.171305	4.79%

סה״כ הצלחנו להביא לשיפור יפה, כאשר עבור ערכי OP נמוכים השיפור משמעותי מאוד ואלו הם המקרים המעניינים יותר.

ניסוי מס׳ 7 – Results-Validation:

בדומה לניסוי 2 נרצה לוודא את עצמנו ולראות כי הגורם היחיד על השיפורים הוא ערך של OP ולא מספר הבלוקים הפיזי או הלוגי שנבחרו.

תנאי הניסוי: מס׳ דורות – 2, $N = 100000, Z = 256, page\_size = 4k$, ערכו של T משתנה ועל פיו נקבע ערכו של U על מנת לקבל את ערך ה-OP עבורו אנו בודקים את תוצאות האלגוריתם שלנו. ערכי ה-WA בטבלה הינם ממוצע של 20 ריצות.

OP	T=64		T=96		T=128		T=256	
	OLD	NEW	OLD	NEW	OLD	NEW	OLD	NEW
0.066667	6.531557	3.564221	7.988795	4.120323	7.106123	3.735837	7.471336	3.842759
0.103448	4.725873	2.80822	5.431358	3.086675	5.00592	2.935912	5.173138	3.040166
0.142857	3.728663	2.368184	4.123628	2.53068	3.88358	2.456161	3.97889	2.597176
0.185185	3.379153	2.200526	3.353043	2.178944	3.340014	2.210858	3.333912	2.281786
0.230769	2.615837	1.853105	2.805717	1.933361	2.685209	1.923193	2.726218	2.021176
0.28	2.414956	1.769108	2.395226	1.750985	2.383743	1.788854	2.371221	1.866362
0.333333	2.076383	1.62559	2.060539	1.612622	2.052573	1.643345	2.044988	1.704106
0.391304	1.723144	1.4615	1.809502	1.493166	1.758351	1.502157	1.787741	1.575171
0.454545	1.55821	1.382021	1.628125	1.404766	1.588907	1.416627	1.619946	1.472075
0.52381	1.431796	1.315249	1.489232	1.33188	1.460508	1.345805	1.491997	1.398435
0.6	1.380749	1.286049	1.380242	1.274751	1.384594	1.299389	1.403854	1.333836
0.684211	1.293828	1.230729	1.296522	1.21891	1.301638	1.245127	1.323657	1.28062
0.777778	1.194712	1.160755	1.230225	1.171305	1.219731	1.18417	1.250626	1.22036

בטבלה התייחסנו לתוצאות שהתקבלו מאלגוריתם Generational GC כ-OLD ואילו לתוצאות שהתקבלו עבור האלגוריתם Generational GCx with Load Balancing כ-NEW. מתוצאות הניסוי ניתן לראות כי אכן Generational GCx with Load Balancing מביא לתוצאות משופרות ביחס ל-Generational GC ללא תלות במס' הבלוקים הפיזיים. נראה כי שיעור השיפור מושפע מערך ה-OP בלבד.

בטבלה הבאה ניתן לראות כי אחוזי השיפור של האלגוריתם Generational GCx with Load Balancing ביחס לאלגוריתם Generational GC עבור ערך OP קבוע וערכים משתנים של T ו-U הינם יחסית דומים זה לזה:

OP	T=64	T=96	T=128	T=256
0.066667	45.43076	48.42372	47.42792	48.56664
0.103448	40.57776	43.16936	41.3512	41.23168
0.142857	36.48705	38.62977	36.75524	34.72612
0.185185	34.87935	35.01592	33.80693	31.5583
0.230769	29.15823	31.09211	28.37829	25.86156
0.28	26.7437	26.89688	24.95611	21.2911
0.333333	21.71049	21.73786	19.93732	16.66912
0.391304	15.18411	17.48194	14.57016	11.89042
0.454545	11.30714	13.71879	10.84271	9.128144
0.52381	8.139885	10.56595	7.853637	6.270893
0.6	6.858563	7.642938	6.153792	4.987557
0.684211	4.876887	5.986133	4.34153	3.251372
0.777778	2.842276	4.789368	2.915439	2.420068

ניסוי מס' 8 – מציאת חלוקת עומס טובה כאשר מס' הדורות גדול מ-2:

בניסוי 3 מצאנו לכל ערך OP כי מס' דורות גדול מ-2, נע בין 6 דורות ל-8, מביא תוצאות טובות יותר ב-WA.

נרצה לבדוק אם המסקנות שהסקנו עבור 2 דורות תקפות במקרה של מס' דורות גדול יותר. באופן מובן, במקרה זה יהיה קשה יותר לבדוק מהי חלוקת העומס הטובה ביותר או לזהות מגמה ברורה או לשלול מגמות אחרות, מכיוון שקיימות הרבה אפשרויות לחלק את העומס כשמס' הדורות גדול יותר.

ולכן נרצה לבצע ניסוי חלקי ושאינו מקיף ואולי לקבל תחושה כיצד כדאי לחלק את העומס בין הדורות במקרה זה.

לשם כך נבצע ניסויים עם חלוקות עומס שונות ונראה מתי הביצועים משתפרים ומתי הם נהיים גרועים יותר.

כמו על לפשט את הניסוי נריץ עבור כל ערכי ה-OP את הסימולציה עם 8 דורות, שכן שהו הערך שהביא לתוצאות הטוב ביותר עבור רב גדול של ערכי ה-OP שבדקנו.

תנאי הניסוי- $T = 96, Z = 256, N = 100000, page\_size = 4k$ ערכו של U משתנה על מנת לקבל ערכי OP שונים. גודל האינטרוול הוא $U \cdot Z \cdot 0.2$, ה- WA המוצג הינו ממוצע של 20 הרצות.

בניסוי נשווה בין ערכי ה-WA המתקבלים עבור 4 חלוקות העומס הבאות:

Load-Balancing#1:

generation1	generation2	generation3	generation4	generation5	generation6	generation7	generation8
10%	10%	10%	10%	15%	15%	15%	15%

Load-Balancing#2:

generation1	generation2	generation3	generation4	generation5	generation6	generation7	generation8
8%	8%	12%	12%	15%	%15	15%	15%

Load-Balancing#3:

generation1	generation2	generation3	generation4	generation5	generation6	generation7	generation8
5%	5%	8%	8%	15%	15%	22%	22%

Load-Balancing#4:

generation1	generation2	generation3	generation4	generation5	generation6	generation7	generation8
5%	8%	10%	%12	12%	15%	15%	23%

בחרנו את החלוקות לעיל בעקבות המסקנה אליה הגענו כי ה-GC של האלגוריתם Generational GCx מעמיס הרבה יותר על הדורות הצעירים, ולכן חלוקת עומס לוגית טובה היא כזו שמורידה את העומס הלוגי על הדורות הצעירים כך שהפער בעומס הפיזי עליהם לא יהיה גדול מידיי.

ניתן לראות את התוצאות המלאות עם אחוזי העומס הלוגי והפיזי בנספח.

כמו כן הרצנו כל מיני חלוקות ובדקנו את העומס הפיזי שמתקבל על הדורות השונים וכן באופן כללי איזו חלוקה הביאה תוצאה טובה יותר, ולפי התוצאות של ההרצות הראשונות בחרנו את חלוקת העומס של הריצות הבאות.

תוצאות ומסקנות :

OP	Load-Balancing #1:	Load-Balancing #2:	Load-Balancing #3:	Load-Balancing #4:
0.066667	2.920386	2.796637	2.747775	2.744681
0.103448	2.050701	2.026881	2.083283	2.040366
0.142857	1.713768	1.706083	1.782007	1.749787
0.185185	1.517441	1.512969	1.598939	1.562447
0.230769	1.389319	1.389677	1.464329	1.439588
0.28	1.304275	1.301887	1.370408	1.346798
0.333333	1.238983	1.235725	1.294652	1.276483
0.391304	1.190997	1.187036	1.235143	1.2197
0.454545	1.15301	1.150897	1.187101	1.176851
0.52381	1.119136	1.11804	1.146523	1.139333
0.6	1.097896	1.096329	1.116344	1.111492
0.684211	1.077636	1.077532	1.092259	1.087971
0.777778	1.060665	1.06186	1.072236	1.06907

התוצאות שקיבלנו טובות יותר מהתוצאות עבור Generational GCx עם Treshold הסטטי של האלגוריתם Generational GC. כמו כן בדומה ל-2 דורות, גם כאן ניתן לראות כי קיים קשר דומה בין ערך ה-OP לבין חלוקת העומס האופטימלית, כך שככל שה-OP קטן עדיף להעמיס לוגית פחות על הדורות הצעירים ועבור OP גדול מספיק ניתן להגדיל את העומס הלוגי על הדורות הצעירים (החריגה עבור OP שווה ל- 0.230769 הינו זניח). זה לא מפתיע, שכן התופעה גם במקרה של מס' דורות גדול מ-2, ככל שה-GC מופעל יותר כך העומס הפיזי על הדורות הנמוכים גדל, ובאופן משמעותי ביותר על הדור הצעיר ביותר (כפי שניתן לראות בנספח).

בטבלה הבאה ניתן להעריך את גודל השיפור ביחס ל-Threshold הסטטי בו נעשה שימוש באלגוריתם Generational-GC :

OP	Generational GCx without Load-Balancing	Generational GCx with Load-Balancing	improvement:
0.066667	4.3415355	2.7446805	36.78%
0.103448	3.246884	2.0268805	37.57%
0.142857	2.651138	1.706083	35.65%
0.185185	2.275165	1.512969	33.51%
0.230769	2.0133355	1.3893185	30.99%
0.28	1.814471	1.3018865	28.25%
0.333333	1.6657145	1.2357245	25.81%
0.391304	1.5460975	1.1870355	23.22%
0.454545	1.4516645	1.1508965	20.72%
0.52381	1.3734205	1.1180395	18.59%
0.6	1.307948	1.096329	16.18%
0.684211	1.2506775	1.0775315	13.84%
0.777778	1.20143	1.060665	11.72%

ובהשוואה לתוצאות של האלגוריתם Generational GC השיפור כמובן גדול ומשמח אף יותר:

OP	Generational GC	Generational GCx with Load-Balancing	improvement:
0.066667	7.930496	2.7446805	65.39%
0.103448	5.415658	2.0268805	62.57%
0.142857	4.1431605	1.706083	58.82%
0.185185	3.3627505	1.512969	55.01%
0.230769	2.843042	1.3893185	51.13%
0.28	2.4728915	1.3018865	47.35%
0.333333	2.196501	1.2357245	43.74%
0.391304	1.980372	1.1870355	40.06%
0.454545	1.8098365	1.1508965	36.41%
0.52381	1.6709325	1.1180395	33.09%
0.6	1.556192	1.096329	29.55%
0.684211	1.4609085	1.0775315	26.24%
0.777778	1.380419	1.060665	23.16%

סיכום:

בעבודה הקודמת לנו בנושא Garbage Collection Algorithms for Flash Memories פותח אלגוריתם Generational GC שתחת הנחות מסוימות הביא לשיפור טוב ביחס לאלגוריתם המוכר Greedy GC, שללא חלק מן ההנחות נחשב כאופטימלי. אנחנו החלטנו להמשיך את עבודה הזו ולפתח אותה. התמקדנו ב-2 דברים:

1. תיקון ה-GC כך שכתיבה מחדש של דפים לזיכרון תיעשה בהתאם לאלגוריתם הדורות.
2. חקירת ה-Thresholds שקובעים את תווך הגילים עבור כל דור ובכך קובעים את העומס הלוגי על כל דור.

עבור תיקון ה-GC קיבלנו את השיפור המשמעותי ביותר עבור המקרה של 2 דורות וגם עבור יותר דורות. ובכך הסקנו כי בזיכרון Flash שמקיים את ההנחות שלנו האלגוריתם Generational GCx יביא לתוצאות טובות משמעותית ביחס לאלגוריתם האחרים שהזכרנו.

במהלך החקירה שלנו את ה-Thresholds של אלגוריתם הכתיבה Generational GCx, הסקנו כי חלוקת עומס שווה בין הדורות אינה מביאה לתוצאות אופטימליות, וזאת מ-2 סיבות אותן פירטנו במהלך הפרויקט. וכתוצאה מכך, ע"י ניסויים, ראינו כי ע"י tuning של חלוקת העומס בין הדורות השונים ניתן לשפר את הביצועים של האלגוריתם. עבור 2 דורות, המשחק של חלוקת העומס על הדורות לא הביא לשיפורים משמעותיים. אולם עבור מס' דורות גדול יותר השיפור כבר היה משמעותי ביותר. כלומר סה"כ הראנו כי שליטה בחלוקת העומס הלוגי בין הדורות השונים, ובאופן

עקיף שליטה בעומס הפיזי עליהם, יכול להיות כלי שימושי לשיפור הביצועים של זיכרון Flash המקיים את ההנחות של הפרויקט.

את האלגוריתם Generational GCx עם Threshold סטטי וכן את האלגוריתם Generational GCx with Load-Balancing מימשנו על גבי הסימולטור של העבודה הקודמת בנושא. כך שניתן לסמלץ את שני האלגוריתמים הללו ולשחזר את התוצאות שקיבלנו והצגנו בפרויקט.

נספח:

תוצאות מלאות לניסוי מס' 8 :

Results for load_balancing = (0.1, 0.1, 0.1, 0.1, 0.15, 0.15, 0.15, 0.15):																	
OP	**AVG_WA**	**logical0**	**logical1**	**logical2**	**logical3**	**logical4**	**logical5**	**logical6**	**logical7**	**physical0**	**physical1**	**physical2**	**physical3**	**physical4**	**physical5**	**physical6**	**physical7**
0.066667	2.920386	9.68%	9.62%	9.64%	9.60%	14.46%	14.52%	14.39%	14.87%	52.01%	9.05%	6.32%	5.19%	6.96%	6.80%	6.78%	6.88%
0.103448	2.050701	9.79%	9.68%	9.78%	9.76%	14.61%	14.65%	14.62%	15.09%	39.31%	9.73%	7.37%	6.62%	9.17%	9.18%	9.22%	9.40%
0.142857	1.713768	9.89%	9.78%	9.87%	9.79%	14.84%	14.82%	14.80%	15.13%	32.42%	9.63%	7.86%	7.18%	10.73%	10.65%	10.60%	10.92%
0.185185	1.517441	9.96%	9.88%	9.90%	9.90%	14.93%	14.85%	14.93%	15.18%	27.40%	9.48%	8.23%	7.85%	11.76%	11.65%	11.71%	11.92%
0.230769	1.389319	9.97%	9.88%	9.94%	9.87%	14.98%	14.90%	14.94%	15.33%	23.55%	9.28%	8.54%	8.31%	12.56%	12.46%	12.48%	12.82%
0.28	1.304275	9.98%	9.87%	9.90%	9.96%	14.98%	14.93%	14.93%	15.28%	20.89%	9.22%	8.73%	8.71%	13.08%	13.01%	13.01%	13.35%
0.333333	1.238983	9.96%	9.87%	9.88%	9.94%	14.86%	14.99%	14.81%	15.22%	18.63%	9.20%	8.97%	9.01%	13.45%	13.55%	13.40%	13.79%
0.391304	1.190997	9.88%	9.76%	9.82%	9.85%	14.80%	14.81%	14.78%	15.20%	16.79%	9.19%	9.19%	9.19%	13.83%	13.81%	13.80%	14.19%
0.454545	1.15301	9.80%	9.71%	9.70%	9.80%	14.64%	14.65%	14.63%	15.06%	15.23%	9.38%	9.32%	9.42%	14.05%	14.05%	14.10%	14.45%
0.52381	1.119136	10.01%	9.86%	10.00%	9.93%	14.96%	15.01%	14.93%	15.28%	13.84%	9.48%	9.57%	9.51%	14.31%	14.37%	14.28%	14.63%
0.6	1.097896	9.82%	9.71%	9.73%	9.83%	14.69%	14.65%	14.81%	15.07%	13.02%	9.55%	9.56%	9.65%	14.44%	14.39%	14.57%	14.82%
0.684211	1.077636	9.93%	9.81%	9.86%	9.90%	14.79%	14.87%	14.79%	15.26%	12.27%	9.65%	9.69%	9.71%	14.53%	14.61%	14.55%	14.99%
0.777778	1.060665	9.93%	9.78%	9.89%	9.90%	14.91%	14.88%	14.79%	15.42%	11.59%	9.66%	9.76%	9.77%	14.71%	14.69%	14.59%	15.23%

Results for load_balancing = (0.08, 0.08, 0.12, 0.12, 0.15, 0.15, 0.15, 0.15):																	
OP	**AVG_WA**	**logical0**	**logical1**	**logical2**	**logical3**	**logical4**	**logical5**	**logical6**	**logical7**	**physical0**	**physical1**	**physical2**	**physical3**	**physical4**	**physical5**	**physical6**	**physical7**
0.066667	2.796637	7.75%	7.67%	11.52%	11.62%	14.47%	14.49%	14.43%	14.83%	46.25%	9.83%	8.74%	6.65%	7.26%	7.08%	7.03%	7.16%
0.103448	2.026881	7.83%	7.75%	11.71%	11.68%	14.68%	14.59%	14.70%	15.04%	35.36%	9.61%	9.48%	8.07%	9.37%	9.24%	9.34%	9.53%
0.142857	1.706083	7.91%	7.83%	11.81%	11.82%	14.84%	14.74%	14.86%	15.10%	29.13%	9.15%	9.93%	8.83%	10.75%	10.63%	10.69%	10.90%
0.185185	1.512969	7.97%	7.90%	11.88%	11.92%	14.94%	14.82%	14.82%	15.28%	24.48%	8.63%	10.18%	9.53%	11.81%	11.70%	11.64%	12.03%
0.230769	1.389677	7.97%	7.86%	11.90%	11.98%	14.95%	14.97%	14.86%	15.34%	21.12%	8.13%	10.37%	10.13%	12.52%	12.51%	12.43%	12.80%
0.28	1.301887	7.98%	7.85%	11.95%	11.94%	14.94%	14.96%	14.82%	15.40%	18.43%	7.85%	10.66%	10.48%	13.08%	13.06%	12.93%	13.51%
0.333333	1.235725	7.96%	7.87%	11.86%	11.90%	14.95%	14.85%	14.85%	15.29%	16.26%	7.68%	10.82%	10.81%	13.58%	13.49%	13.46%	13.90%
0.391304	1.187036	7.92%	7.84%	11.82%	11.81%	14.82%	14.78%	14.72%	15.18%	14.48%	7.62%	11.08%	11.05%	13.88%	13.89%	13.76%	14.24%
0.454545	1.150897	7.83%	7.72%	11.69%	11.72%	14.65%	14.60%	14.68%	15.11%	13.05%	7.59%	11.25%	11.28%	14.10%	14.04%	14.14%	14.54%
0.52381	1.11804	8.00%	7.89%	11.87%	11.98%	15.04%	14.95%	14.97%	15.29%	11.72%	7.67%	11.37%	11.49%	14.42%	14.32%	14.36%	14.65%
0.6	1.096329	7.87%	7.77%	11.70%	11.79%	14.71%	14.69%	14.59%	15.18%	10.89%	7.71%	11.52%	11.62%	14.48%	14.46%	14.37%	14.95%
0.684211	1.077532	7.92%	7.80%	11.85%	11.84%	14.78%	14.92%	14.85%	15.26%	10.23%	7.73%	11.63%	11.64%	14.52%	14.66%	14.57%	15.01%
0.777778	1.06186	7.95%	7.82%	11.93%	11.86%	14.89%	14.89%	14.80%	15.36%	9.71%	7.74%	11.76%	11.68%	14.69%	14.69%	14.60%	15.14%

Results for load_balancing = (0.05, 0.05, 0.08, 0.08, 0.15, 0.15, 0.22, 0.22):																	
OP	**AVG_WA**	**logical0**	**logical1**	**logical2**	**logical3**	**logical4**	**logical5**	**logical6**	**logical7**	**physical0**	**physical1**	**physical2**	**physical3**	**physical4**	**physical5**	**physical6**	**physical7**
0.066667	2.747775	4.83%	4.76%	7.68%	7.74%	14.50%	14.44%	21.22%	21.59%	36.64%	8.95%	8.96%	5.73%	9.59%	7.82%	11.02%	11.28%
0.103448	2.083283	4.90%	4.83%	7.79%	7.77%	14.67%	14.68%	21.37%	21.99%	27.82%	8.81%	8.61%	6.40%	11.08%	9.49%	13.77%	14.02%
0.142857	1.782007	4.94%	4.88%	7.82%	7.91%	14.81%	14.83%	21.57%	22.13%	22.93%	8.43%	8.31%	6.97%	11.53%	10.62%	15.40%	15.80%
0.185185	1.598939	4.97%	4.88%	7.92%	7.94%	14.85%	14.90%	21.64%	22.44%	19.60%	7.94%	8.11%	7.14%	11.89%	11.49%	16.57%	17.26%
0.230769	1.464329	5.00%	4.91%	7.93%	7.92%	14.93%	14.96%	21.75%	22.42%	17.06%	7.32%	7.97%	7.14%	12.37%	12.24%	17.72%	18.18%
0.28	1.370408	4.99%	4.90%	7.93%	7.96%	14.87%	15.00%	21.86%	22.33%	15.11%	6.86%	7.98%	7.09%	12.74%	12.75%	18.52%	18.95%
0.333333	1.294652	4.97%	4.90%	7.88%	7.97%	14.86%	14.87%	21.84%	22.25%	13.55%	6.45%	7.78%	7.15%	13.11%	13.12%	19.24%	19.61%
0.391304	1.235143	4.95%	4.85%	7.86%	7.94%	14.81%	14.72%	21.72%	22.05%	12.05%	6.09%	7.64%	7.27%	13.52%	13.45%	19.84%	20.14%
0.454545	1.187101	4.90%	4.81%	7.77%	7.79%	14.64%	14.74%	21.33%	22.01%	10.65%	5.74%	7.55%	7.36%	13.83%	13.92%	20.14%	20.81%
0.52381	1.146523	4.99%	4.89%	7.93%	7.94%	14.96%	15.00%	21.85%	22.41%	9.31%	5.47%	7.63%	7.52%	14.13%	14.17%	20.62%	21.13%
0.6	1.116344	4.93%	4.84%	7.77%	7.90%	14.59%	14.75%	21.53%	22.00%	8.44%	5.28%	7.62%	7.68%	14.20%	14.37%	20.97%	21.45%
0.684211	1.092259	4.96%	4.86%	7.80%	7.93%	14.79%	14.90%	21.63%	22.34%	7.68%	5.12%	7.64%	7.74%	14.42%	14.51%	21.11%	21.78%
0.777778	1.072236	4.96%	4.84%	7.90%	7.84%	15.00%	14.83%	21.82%	22.31%	6.97%	5.02%	7.77%	7.68%	14.73%	14.55%	21.40%	21.88%

Results for load_balancing = (0.05, 0.08, 0.1, 0.12, 0.12, 0.15, 0.15, 0.23):																	
OP	**AVG_WA**	**logical0**	**logical1**	**logical2**	**logical3**	**logical4**	**logical5**	**logical6**	**logical7**	**physical0**	**physical1**	**physical2**	**physical3**	**physical4**	**physical5**	**physical6**	**physical7**
0.066667	2.744681	4.83%	7.67%	9.63%	11.58%	11.55%	14.50%	14.54%	22.48%	37.15%	14.08%	8.40%	7.31%	6.09%	7.61%	7.62%	11.76%
0.103448	2.040366	4.90%	7.75%	9.76%	11.69%	11.69%	14.67%	14.70%	22.83%	28.05%	12.86%	8.85%	8.23%	7.70%	9.60%	9.68%	15.03%
0.142857	1.749787	4.94%	7.79%	9.89%	11.83%	11.78%	14.84%	14.76%	23.07%	23.47%	11.86%	8.78%	8.89%	8.62%	10.82%	10.82%	16.75%
0.185185	1.562447	4.97%	7.85%	9.95%	11.87%	11.97%	14.87%	14.89%	23.16%	19.75%	11.01%	8.78%	9.48%	9.48%	11.76%	11.63%	18.10%
0.230769	1.439588	5.00%	7.89%	9.93%	11.91%	11.98%	14.93%	14.91%	23.29%	17.15%	10.31%	8.73%	9.97%	9.93%	12.41%	12.28%	19.21%
0.28	1.346798	4.99%	7.89%	9.89%	12.03%	11.87%	14.91%	14.95%	23.30%	15.03%	9.67%	8.81%	10.46%	10.29%	12.86%	12.85%	20.03%
0.333333	1.276483	4.97%	7.87%	9.91%	11.88%	11.89%	14.87%	14.97%	23.16%	13.40%	9.08%	8.98%	10.63%	10.62%	13.28%	13.34%	20.67%
0.391304	1.2197	4.95%	7.81%	9.85%	11.83%	11.82%	14.82%	14.83%	22.98%	11.91%	8.56%	9.13%	10.93%	10.92%	13.66%	13.68%	21.21%
0.454545	1.176851	4.90%	7.70%	9.76%	11.71%	11.67%	14.71%	14.62%	22.92%	10.59%	8.22%	9.29%	11.14%	11.11%	13.98%	13.88%	21.80%
0.52381	1.139333	4.99%	7.89%	9.94%	11.93%	11.97%	15.00%	14.98%	23.28%	9.23%	8.15%	9.45%	11.34%	11.35%	14.22%	14.21%	22.05%
0.6	1.111492	4.92%	7.77%	9.78%	11.72%	11.75%	14.75%	14.68%	22.92%	8.32%	8.03%	9.56%	11.45%	11.48%	14.43%	14.33%	22.40%
0.684211	1.087971	4.96%	7.78%	9.89%	11.86%	11.82%	14.88%	14.83%	23.19%	7.51%	7.92%	9.67%	11.59%	11.57%	14.56%	14.51%	22.67%
0.777778	1.06907	4.96%	7.83%	9.83%	11.94%	11.89%	15.00%	14.71%	23.34%	6.84%	7.92%	9.65%	11.75%	11.68%	14.74%	14.47%	22.95%