פרויקט מסכם בקורס
תכנות מקבילי

ThreadScan & Forkscan

<u>מגישות:</u>

רואן סאדק - 315183723

אלכסנדרה סימונישווילי - 320568843

הקדמה

ניהול זיכרון במערכות מקבילות (Concurrent Memory Reclamation) מהווה אתגר משמעותי, במיוחד בתוכניות מרובות חוטים. כאשר חוט אחד משחרר אזור זיכרון, חוטים אחרים עלולים עדיין להחזיק אליו הפניות מבלי לדעת שהזיכרון כבר שוחרר. מצב זה עלול להוביל לשגיאות חמורות כמו Double Free, Dangling Pointers (שחרור כפול), ואף לקריסת המערכת.
המאמרים ThreadScan ו-Forkscan מציעים גישות שונות לפתרון בעיה זו תוך שמירה על ביצועים גבוהים ואוטומציה מקסימלית:
ThreadScan: מציג פתרון אוטומטי לניהול זיכרון המבוסס על מנגנון אותות (Signals). כאשר חוט מסמן אובייקט לשחרור, כל החוטים הפעילים מתבקשים לסרוק את מחסנית הקריאה (stack) והרגיסטרים שלהם כדי לבדוק אם הם מחזיקים הפניה לאותו אובייקט. רק לאחר שכל החוטים מאשרים שאין להם יותר הפניות, האובייקט משוחרר. גישה זו מציעה אוטומציה מלאה אך דורשת המתנה לכל החוטים לפני ביצוע השחרור, מה שעלול להשפיע על הביצועים בתרחישים מסוימים.
Forkscan: מציע גישה מבוססת על מנגנון Freeze-and-Fork, שבו נוצרת תמונת מצב של הזיכרון על ידי יצירת תהליך-משנה (fork). תהליך זה מבצע סריקה עצמאית של הזיכרון כדי לזהות הפניות פעילות, בעוד החוטים הראשיים ממשיכים לפעול כמעט ללא הפרעה. כתוצאה מכך, השחרור מתבצע בצורה אסינכרונית ויעילה, עם מינימום השפעה על ביצועי המערכת.
שני הפתרונות מציעים אלטרנטיבה לשיטות מסורתיות כמו מצביעי סכנה (Hazard Pointers) וספירת הפניות (Reference Counting), אשר דורשות מאמץ תכנותי ניכר ואינן תמיד סקלאביליות.
ThreadScan ו-Forkscan מספקים דרכים שונות להפחתת התקורה של ניהול זיכרון במערכות מקבילות, ומאפשרים שחרור בטוח של זיכרון ללא צורך בהתערבות ידנית מצד המתכנת.

סיכום

שני המאמרים עוסקים בבעיית שחרור זיכרון במערכות מקבילות, שבהן חוטים מרובים יכולים לגשת לזיכרון משותף. הם מציעים פתרונות יעילים המפחיתים את העומס על המתכנת באמצעות אוטומציה מלאה של תהליך ניהול הזיכרון. הפתרונות מתמקדים במניעת גישה לאזורים משוחררים (Dangling Pointers) ושגיאות כמו Double Free, תוך שמירה על ביצועים גבוהים במערכת מקבילית.

ThreadScan

הבעיה – במערכות מקבילות, כאשר תהליך משחרר זיכרון, ייתכן שחוטי ביצוע אחרים עדיין מחזיקים הפניות אליו. גישות מסורתיות כמו Hazard Pointers או Reference Counting דורשות מעקב ידני אחר הפניות, מה שמסבך את הפיתוח ועלול להוביל לשגיאות.

הפתרון – ThreadScan מציע גישה אוטומטית לחלוטין לניהול זיכרון באמצעות מנגנון אותות (Signals). כאשר חוט מסמן אובייקט לשחרור, כל החוטים הפעילים מתבקשים לסרוק את מחסנית הקריאה (stack) והרגיסטרים שלהם כדי לוודא שאין הפניות פעילות לאובייקט זה. רק לאחר שכל החוטים מאשרים שהאובייקט אינו בשימוש, הוא משוחרר בבטחה.

אוטומטיות וקלות שימוש – האלגוריתם אינו דורש מהמפתח מעקב ידני אחרי מצביעים, אלא מתממשק עם פעולות malloc ו-free הקיימות (פעולות בסיסיות להקצאה ושחרור הזיכרון באותה שפה).

האלגוריתם:

האלגוריתם ThreadScan מטפל בניהול זיכרון ומחיקת אובייקטים שאינם בשימוש. כאשר חוט רוצה למחוק אובייקט, הוא מוסיף מצביע שלו לבאפר מחיקה (delete buffer). כשהבאפר מתמלא, החוט שהוסיף את הפריט האחרון הופך לחוט ה- reclaimer ומתחיל את תהליך ThreadScan Collect. תהליך זה מערב את כל שאר החוטים כדי לבדוק האם עדיין יש הפניות לאובייקטים שבבאפר. בסוף התהליך, כל אובייקט במאגר מסומן כ- marked או unmarked. אובייקטים מסומנים עדיין נמצאים בשימוש ואינם נמחקים, בעוד שאובייקטים לא מסומנים משוחררים מהזיכרון.

תהליך ThreadScan Collect:

תהליך המחיקה מתבצע בשלבים, כאשר בכל שלב (reclamation phase) מזהים קבוצת אובייקטים שאינם נגישים ומוודאים שהמחיקה לא תגרום לקריסת המערכת.

השלבים:

1. מיון הבאפר – תחילה ממיינים את באפר המחיקה, כדי לייעל את תהליך הבדיקה.
2. שליחת אותות (signals) לכל החוטים – כל חוט מקבל בקשה לבצע סריקה של מחסנית הזיכרון שלו.
3. ביצוע TS-Scan – כל חוט סורק את הזיכרון שלו ומסמן אובייקטים שהוא עדיין מפנה אליהם.
4. שליחת אישור (ACK) לחוט המנקה (reclaimer) – כל חוט מאשר שסיים את הבדיקה.
5. מחיקת אובייקטים שאינם בשימוש – החוט המנקה משחרר מהזיכרון את האובייקטים הלא מסומנים.

פרטי מימוש ומגבלות:

- תקשורת בין חוטים – האלגוריתם משתמש באותות POSIX. כאשר חוט מקבל אות, מערכת ההפעלה מפסיקה את הפעולה שלו ומריצה את פונקציית הסריקה. אם חוט נמצא בקריאה למערכת (system call), הסיגנל יקטע את הקריאה ויחזיר שגיאת EINTR, מה שמצריך התמודדות מתאימה בקוד.
- התקדמות ThreadScan – (Progress) תלוי בתזמון של מערכת ההפעלה, מאחר שהוא משתמש באותות כדי לסרוק את מחסניות החוטים. עם זאת, במערכות כמו Linux, שבהן מתזמן המשימות מבטיח חלוקה הוגנת של זמן עיבוד, האלגוריתם מספק הבטחת התקדמות ללא חסימה.
- גבולות המחסנית (Stack Boundaries) – האלגוריתם מזהה את גבולות מחסניות החוטים באמצעות קריאה לפונקציה pthread_create. במקרים מסובכים יותר, כמו שימוש במחסניות דינמיות (כגון "Cactus stacks"), ייתכן שיידרשו שיפורים נוספים.
- ניהול באפר המחיקה – במקום להשתמש בבאפר אחד משותף שעלול לגרום לתחרות (contention) בין החוטים, ThreadScan משתמש בבאפרים מחיקה פרטיים לכל חוט.
- כאשר באפר של חוט מתמלא, הוא מתווסף לבאפר מרכזי לצורך סריקה, ולכן בחירת גודל מתאים עוזרת להפחית את כמות העומס על תהליכי הסנכרון.

הרחבת ThreadScan:

כדי להתמודד עם מצבים שבהם חוט מחזיק הפניות בזיכרון הערימה (heap), האלגוריתם מספק מנגנון הרחבה שמאפשר להגדיר אזורי זיכרון פרטיים לכל חוט.

שתי פונקציות חדשות נוספו:

TS_add_heap_block(start_addr, len) – מאפשרת להצהיר על אזור זיכרון שבו קיימות הפניות פרטיות.

TS_remove_heap_block(start_addr, len) – מסירה אזור זיכרון שהוצהר בעבר.

המימוש מחייב את המתכנת להגדיר מראש את האזורים הללו, אך האלגוריתם עצמו מבצע את תהליך הסריקה והמחיקה בצורה אוטומטית.

תקציר הוכחות ותוצאות:

המאמר מבסס את נכונות האלגוריתם של ThreadScan על שלוש הנחות יסוד עיקריות:

1. Shared References (הפניות משותפות): כל האובייקטים שנמצאים בבאפר המחיקה כבר אינם נגישים דרך הפניות משותפות או דרך מצביעים ב-heap. כלומר, אין מצביעים גלובליים או משתפים שיכולים לגרום לגישה לא תקינה לאחר שחרור האובייקט.

המשמעות: אם אובייקט כלשהו נמצא בבאפר המחיקה, אין אליו הפניות מהזיכרון המשותף, אלא לכל היותר הפניות מהמחסנית (stack) או הרגיסטרים של החוטים.

2. Reclamation Rate (קצב שחרור מוגבל): קיים גבול קבוע למספר אירועי השחרור שיכולים להתרחש במהלך ביצוע של כל שיטה. כלומר, לא ניתן לשחרר זיכרון בקצב אינסופי או לא מוגבל, מה שמאפשר בקרה על תהליך המחיקה.

המשמעות: האלגוריתם מבטיח שמספר אירועי השחרור במהלך זמן ריצה יהיה סופי, ולכן אין מצב שבו שחרור הזיכרון לא מסתיים או נכנס ללולאה אינסופית.

3. Matching References (זיהוי הפניות בזיכרון): כל המצביעים הם מיושרי-מילה (Word-Aligned), מה שמאפשר לזהות אותם בזיכרון בצורה אמינה.

המשמעות: האלגוריתם מניח שכל מצביעים חוקיים לזיכרון ניתנים לזיהוי בסריקת ה-stack והרגיסטרים, ואינו רגיש למצביעים "מוסווים" או מקודדים בצורה חריגה.

הוכחות נכונות האלגוריתם:

1. Safety Properties – האלגוריתם לעולם לא משחרר זיכרון שאינו מוכן למחיקה: טענה- אם כל האובייקטים שנמצאים בבאפר המחיקה הם במצב removed state, אזי ThreadScan לא ישחרר אובייקט שעדיין נגיש לאף חוט.

למה זה נכון?

- לפי הנחת הפניות משותפות (Shared References), אין הפניות משותפות מה-heap, ולכן כל מצביעים לאובייקטים אלה נמצאים רק במחסנית או ברגיסטרים של החוטים הפעילים.
- ThreadScan שולח אות (Signal) לכל החוטים, המחייב אותם לסרוק את המחסנית והרגיסטרים שלהם ולסמן הפניות פעילות.
- אם חוט כלשהו מחזיק הפניה פעילה, האובייקט לא ישוחרר, כיוון שהוא יסומן כעדיין בשימוש.
- אם אין הפניות – האובייקט משוחרר בבטחה.
- כתוצאה מכך, לא ייתכן מצב שבו ThreadScan משחרר זיכרון שנמצא עדיין בשימוש על ידי תהליכים אחרים.

2. Liveness Properties – האלגוריתם לא גורם לחסימה (Deadlock-Free): טענה- כל חוט שאינו מבצע free ימשיך לרוץ ללא השפעה משמעותית של האלגוריתם.

למה זה נכון?

- לפי הנחת קצב השחרור המוגבל (Reclamation Rate), כל חוט מבצע מספר מוגבל של פעולות במהלך תהליך השחרור.
- כאשר ThreadScan מופעל, כל חוט מקבל אות (Signal) שמבקש ממנו לבצע סריקה קצרה בלבד.
- כל חוט מגיב במהירות, מבצע את הבדיקה, שולח אישור, וממשיך בעבודתו.
- האלגוריתם אינו דורש המתנה או חסימה, אלא מוסיף מספר צעדים קבועים ומוגבלים בלבד.
- לכן, כל שיטה שלא מבצעת free משמרת את תכונות ההתקדמות שלה ולא מושפעת מהאלגוריתם.

ניסויים שנעשו ותוצאותיהם:

הניסויים במאמר נועדו לבדוק עד כמה ThreadScan יעיל בהשוואה לשיטות מסורתיות, תוך בחינת ביצועיו במבני נתונים שונים.

סביבת הניסוי: מעבד Intel Xeon בעל 40 ליבות, כל ליבה מריצה 2 חוטים (סה"כ 80 חוטים).
והאלגוריתם הופעל עם עד 1024 מצביעים לכל חוט.

מבני נתונים שנבדקו:

- רשימה מקושרת ללא נעילות (Lock-Free Linked List)
- טבלת גיבוב ללא נעילות (Lock-Free Hash Table)
- מבנה חיפוש (Skip List)

שיטות ניהול זיכרון שנבחנו:

- Leaky Implementation – ללא שחרור זיכרון כלל, משמש כבסיס להשוואה.
- Hazard Pointers – דורש מעקב ידני אחרי הפניות.
- Epoch-Based Reclamation – שחרור זיכרון לפי מחזורי זמן מוגדרים מראש.
- Slow Epoch – גרסה שבה חוטים איטיים מעכבים שחרור זיכרון.
- ThreadScan – השיטה החדשה שנבדקה.

תוצאות הניסויים:

ThreadScan משתווה בביצועים לשיטות היעילות ביותר:

- ThreadScan ו-Epoch-Based Reclamation הראו יכולת ביצוע טובה גם תחת עומס של 80 חוטים.
- Hazard Pointers עבד היטב בטבלאות גיבוב, אך פחות ברשימות מקושרות עקב הצורך בעדכונים רבים.
- ב-Skip List התקורה של ThreadScan עמדה על כ-25% עבור 200 חוטים, דומה לזו של Epoch-Based Reclamation.

ThreadScan מפחית עומס פיתוח משמעותי:

- בשיטות כמו Hazard Pointers, המתכנת נדרש לנהל הפניות ידנית, מה שמגביר את הסיכוי לטעויות.
- ThreadScan אינו דורש התערבות ידנית, מה שמקטין את המורכבות ומונע שגיאות תכנותיות.

ThreadScan אינו מושפע משמעותית מחוטים תקועים (Oversubscription Tests):

- שיטות כמו Slow Epoch נפגעו מאוד מחוטים איטיים שלא עדכנו את מונה הזמן שלהם בזמן. ThreadScan, לעומת זאת, המשיך לפעול היטב גם בתנאי עומס, מאחר שכל חוט מבצע סריקה מקומית בלבד.

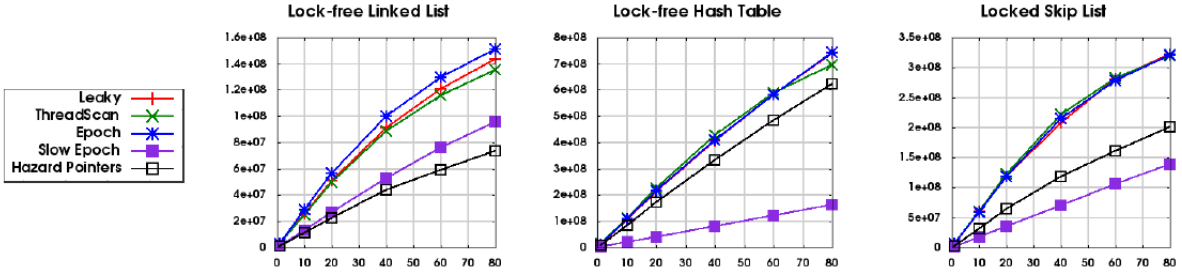


Figure 3: Throughput results for the lock-free linked list, lock-free hash table, and locked skip list: X-axis shows the number of threads, and Y-axis the total number of completed operations.

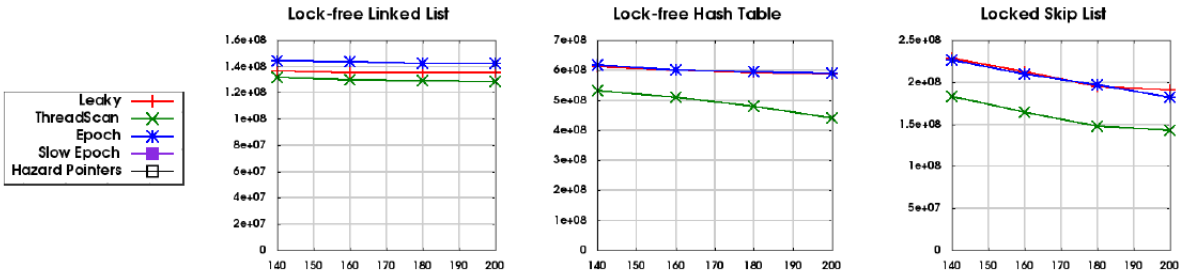


Figure 4: Throughput results for the oversubscribed system.

Forkscan

הבעיה – ניהול זיכרון במערכות מקבילות מהווה אתגר משמעותי בשפות כמו C ו-C++, מאחר ואין בהן מנגנון איסוף זבל אוטומטי (Garbage Collection). בעוד ששיטות מסורתיות לשחרור זיכרון, כגון Hazard Pointers או Epoch-Based Reclamation, דורשות מעקב ידני אחרי הפניות לזיכרון, Forkscan מציע גישה אוטומטית לחלוטין עם ביצועים תחרותיים.

הפתרון – Forkscan מציע מנגנון שחרור זיכרון מבוסס יצירת תמונת מצב (Snapshot) תוך שימוש בתכונות מערכת ההפעלה לשיפור היעילות. האלגוריתם פועל בשני שלבים מרכזיים: Freeze-and-Fork ו-Scan-and-Mark. תחילה, כאשר חוט Reclaimer מזהה שהצטברו מספר מספיק של מצביעים לשחרור, הוא שולח אות (Signal) לכל החוטים הפעילים, אשר כתגובה כותבים את מצב המחסנית והרגיסטרים שלהם ומאשרים חזרה. לאחר קבלת כל האישורים, נוצר תהליך-בן (Fork), היורש תמונת מצב עקבית של הזיכרון באמצעות Copy-on-Write. תהליך זה מבצע סריקה מקבילה של הזיכרון, מפרש מצביעים ומזהה הפניות פעילות. אובייקטים אשר אינם מוחזקים עוד על ידי חוטים משוחררים בבטחה. גישה זו מאפשרת לסרוק ולנהל את הזיכרון ללא הפרעה לפעילות השוטפת של התוכנית, מפחיתה משמעותית את התקורה, ומשפרת את סקלאביליות ניהול הזיכרון במערכות מקבילות.

שימור ביצועים – בניגוד לאיסוף זבל קלאסי (Garbage Collection), Forkscan מצמצם את ההשהיה שנגרמת בשלבי מחיקת הזיכרון, מה שהופך אותו ליעיל במיוחד במערכות עתירות ביצועים.

האלגוריתם:

אלגוריתם Forkscan פועל בשיטה אוטומטית לניהול ושחרור זיכרון במערכות מקבילות. הוא מחלק את התהליך לשלושה שלבים עיקריים:

1. הקפאה ופיצול (Freeze-and-Fork):

בשלב זה המערכת יוצרת תמונת מצב עקבית של הזיכרון כדי לאפשר סריקה בטוחה של המצביעים. התהליך מתבצע כך:

- איסוף המידע מהבאפרים של המחיקה – לכל חוט קיים באפר של מצביעים המיועדים למחיקה (delete buffer). כאשר באפר זה מתמלא, הוא מועבר לחוט Reclaimer אשר מרכז את המידע מכל החוטים ויוזם תהליך שחרור.
- שליחת אות לכל החוטים הפעילים – החוט האחראי משדר אות (Signal) לכל שאר החוטים, אשר מגיבים על ידי כתיבת גבולות המחסנית (Stack Boundaries) ותוכן הרגיסטרים שלהם, לאחר שכל החוטים מסיימים לשמור את המידע, הם שולחים אישור חזרה לחוט ה-Reclaimer ורק כאשר מתקבלים כל האישורים, מתבצע השלב הבא (הזיכרון מוקפא עד קבלת כל האישורים).
- יצירת תהליך בן (Fork) – לאחר שכל החוטים מאשרים את קבלת האות, החוט Reclaimer יוצר תהליך בן חדש (Child Process). תהליך זה יורש תמונת מצב עקבית של הזיכרון בעזרת מנגנון Copy-on-Write של מערכת ההפעלה, ומבצע את הסריקה מבלי לשבש את הפעולה של החוטים הראשיים.

- שחרור כל החוטים לפעולה רגילה – לאחר יצירת תהליך הבן, כל החוטים הראשיים חוזרים לפעילות רגילה בעוד שהתהליך החדש אחראי על זיהוי המצביעים הפעילים וביצוע השחרור.

2. סריקה וסימון (Scanning and Marking):

לאחר יצירת תהליך הבן (fork), מתבצעת סריקה של הזיכרון על מנת לזהות הפניות פעילות ולסמן אובייקטים הניתנים לשחרור. הסריקה מחולקת לשני שלבים עיקריים:

- מציאת שורשים (Find Roots): תהליך הבן מזהה את טווחי הזיכרון שצריך לסרוק ומחלק אותם ל-M תתי-טווחים בלתי תלויים. כל תת-טווח מועבר לתהליכים אחים (Sibling Processes) כך שהסריקה מתבצעת במקביל לשיפור היעילות. תהליך זה סורק את הזיכרון כדי למצוא הפניות מחוץ לבאפר המחיקה (Delete Buffer) אל אובייקטים שבתוכו. כאשר נמצאות הפניות כאלו, הן מסומנות כפעילות כדי למנוע שחרור מוקדם של האובייקטים. בנוסף, באפר המחיקה משותף בין כל תהליכי הסריקה, כך שכל שינוי בו נראה לכל התהליכים.

- סימון רקורסיבי (Recursive Marking): לאחר זיהוי השורשים, המערכת מחלקת את באפר המחיקה לחלקים קטנים יותר, וכל תהליך אחראי לבדוק את ההפניות הנוספות שבתוכו. תהליכי הבן (Sibling Processes) סורקים את המצביעים המסומנים ומבצעים חיפוש רקורסיבי אחר הפניות נוספות. בתום שלב זה, כל האובייקטים הגלויים לתוכנית סומנו, אובייקטים לא מסומנים נחשבים כלא בשימוש וניתן לשחרר אותם בבטחה. תהליך זה מאפשר גם זיהוי מחזורי הפניות (Cycles) – אם יש אובייקטים שמצביעים אחד על השני אך אינם נגישים מבחוץ, הם יכולים להשתחרר.

- שליחת נתוני הסריקה חזרה: לאחר שהסריקה מסתיימת, התהליך האחרון מבין תהליכי הבן (Sibling Processes) שולח הודעה חזרה לתהליך האב ומסיים את פעולתו.

3. מחיקת אובייקטים (Deletion Phase):

לאחר זיהוי המצביעים הלא פעילים, מתבצע תהליך מחיקת הזיכרון.

- מניעת ביצועי יתר של free() – קריאות מרובות ל-free() עלולות לגרום לעיכובים, ולכן Forkscan דוחה את המחיקה ומשלב אותה בקריאות הקצאה עתידיות (malloc()), כך ששחרור הזיכרון יתבצע יחד עם הקצאות חדשות.

- ספירת הפניות בבאפר המחיקה – Forkscan משתמש בספירת הפניות כדי להבטיח שאובייקט לא ישתחרר מוקדם מדי. כאשר ספירת ההפניות מגיעה לאפס, האובייקט נחשב מוכן לשחרור.

- חלוקת תהליך המחיקה על פני זמן ריצה – בניגוד לאיסוף זבל מסורתי שגורם לעצירות פתאומיות, Forkscan מפזר את פעולות המחיקה לאורך זמן, מה שמונע השפעות שליליות על הביצועים.

תקציר הוכחות ותוצאות:

Forkscan מתבסס על ארבע הנחות יסוד כדי להבטיח את נכונותו ויעילותו:

1. **(No False Negatives) – אין שלילות שגויות:** כל המצביעים בזיכרון מיושרי מילה (Word-Aligned) וניתנים להשוואה לבלוקים מוקצים. Forkscan מסיר שלושה ביטים נמוכים מכל כתובת שנבדקת, כדי לחשוף מצביעים מוסתרים (Pointer-Hiding) הנמצאים בשימוש במבני נתונים מסוימים. משמעות הדבר היא שכל מצביע לגיטימי יזוהה, ולא יהיו החמצות של הפניות קיימות.

2. **(No Thread Crashes) – אין קריסות של חוטים:** מניחים כי אין קריסות חוטים במהלך ביצוע האלגוריתם. כלומר, כל החוטים יגיבו לקריאת האות (Signal) בזמן הסריקה, וישמרו על עקביות.

3. **(Bounded Allocation Rate) – קצב הקצאה מוגבל:** קיים גבול סופי לקצב ההקצאות של הזיכרון בתוכנית. ההנחה הזו מבטיחה ש-Forkscan לא יתמודד עם עומס בלתי מוגבל, מה שיכול היה לפגוע בביצועים.

4. **(No False Positives) – אין חיוביות שגויות:** מילים אקראיות בזיכרון אינן מזוהות בטעות ככתובות של מצביעים חוקיים. משמעות הדבר היא שכל הכתובות שנמצאות משויכות באמת לאובייקטים שהוקצו, ולא מתרחשת מחיקה שגויה של נתונים.

הוכחות נכונות האלגוריתם:

בהתבסס על ההנחות לעיל, Forkscan מבטיח את הדברים הבאים:

- לא ניתן לשחרר בלוק זיכרון שנמצא עדיין בשימוש.
- כל בלוק זיכרון שאינו נגיש בסופו של דבר ישתחרר.
- קיים גבול סופי לכמות הזיכרון שהתוכנית משתמשת בו בכל זמן נתון.

<u>הוכחה (סקיצה):</u> נניח כי T הוא הזמן שבו קריאת ה-fork() מסתיימת, כלומר, כאשר נוצר תהליך הבן.

עיקרון מרכזי: כל אובייקט שהוקצה ואינו מופיע במחסניות או ברגיסטרים של החוטים בזמן T, לא יכול להופיע מאוחר יותר, ולכן הוא בלתי נגיש (Unreachable). תהליך הבן מקבל תמונת מצב עקבית של כל הזיכרון, ולכן לא יכולות להתרחש החמצות או טעויות בזיהוי מצביעים פעילים.

לפי הנחה 1, כל אובייקט נגיש יכיל ספירת הפניות חיובית בסיום הסריקה, ולכן לא ניתן למחוק אותו בטעות.

לפי הנחה 2, כל אובייקט שאינו נגיש לא יהיה לו ספירת הפניות, ולכן Forkscan יבצע מחיקה בטוחה.

לפי הנחה 3, קיים גבול סופי לכמות ההקצאות, ולכן אין מצב שבו הזיכרון יוצא משליטה.

<u>משמעות ההוכחה:</u> בלוקים פעילים לא ישתחררו מוקדם מדי (מניעת שגיאות גישה לזיכרון), כל בלוק שאינו נגיש באמת ישתחרר (שחרור יעיל של זיכרון) וקיים גבול לכמות הזיכרון שהתוכנית תצרוך, מה שמבטיח יציבות לאורך זמן.

ניסויים ותוצאות:

<u>מבני נתונים שנבדקו:</u>

- רשימה מקושרת ללא נעילות (Lock-Free Linked List)
- טבלת גיבוב ללא נעילות (Lock-Free Hash Table)
- Skip List ללא נעילות

שיטות שנבדקו בהשוואה:

- Leaky – גישה ללא שחרור זיכרון כלל.
- BDW-GC – איסוף זבל של Boehm-Demers-Weiser.
- Hazard Pointers – שמירה ידנית על הפניות חוקיות.
- Forkscan.

תוצאות מרכזיות:

בבדיקות שבוצעו על Forkscan במערכת מרובת ליבות, נמצא כי האלגוריתם מציג ביצועים טובים יותר מ-Garbage Collection מסורתי כמו BDW-GC, עם השהיות נמוכות משמעותית (פי 242 פחות). בהשוואה לשיטות ניהול זיכרון ידניות כמו Hazard Pointers, Forkscan דרש פחות התערבות מצד המתכנת ושמר על ביצועים תחרותיים. בתרחישים של עומס גבוה, Forkscan הצליח להתמודד עם מיליוני צמתים בטבלאות גיבוב ורשימות מקושרות, אך הגיע לנקודת שבירה בעומס של 80 חוטים עקב bottleneck בתהליך יצירת תמונת הזיכרון (snapshot). בסביבות בהן היה יותר זיכרון פנוי, Forkscan הציג ביצועים משופרים והפחית משמעותית את זמני ההשהיה, במיוחד ב-Skip List.

ממצאים נוספים:

Forkscan הראה כי תוספת זיכרון פנוי משפרת את הביצועים ומפחיתה את זמני ההשהיה, בניגוד ל-BDW-GC שלא הציג שיפור משמעותי במצב זה. כמו כן, נמצא כי הכפלת מספר החוטים מ-40 ל-80 לא תרמה משמעותית לביצועים, מאחר שצוואר הבקבוק נגרם בעיקר משלב הקפאת הזיכרון ולא מעיבוד הנתונים. ב-Skip List, Forkscan שמר על איזון בין ניצול משאבים לביצועים, אך כאשר היה עומס חוטים גבוה, Hazard Pointers הציג ביצועים יציבים יותר. בסך הכל, Forkscan מציע איזון מוצלח בין ביצועים לנוחות תכנות, תוך שמירה על ניהול זיכרון אוטומטי עם תקורה נמוכה יחסית.

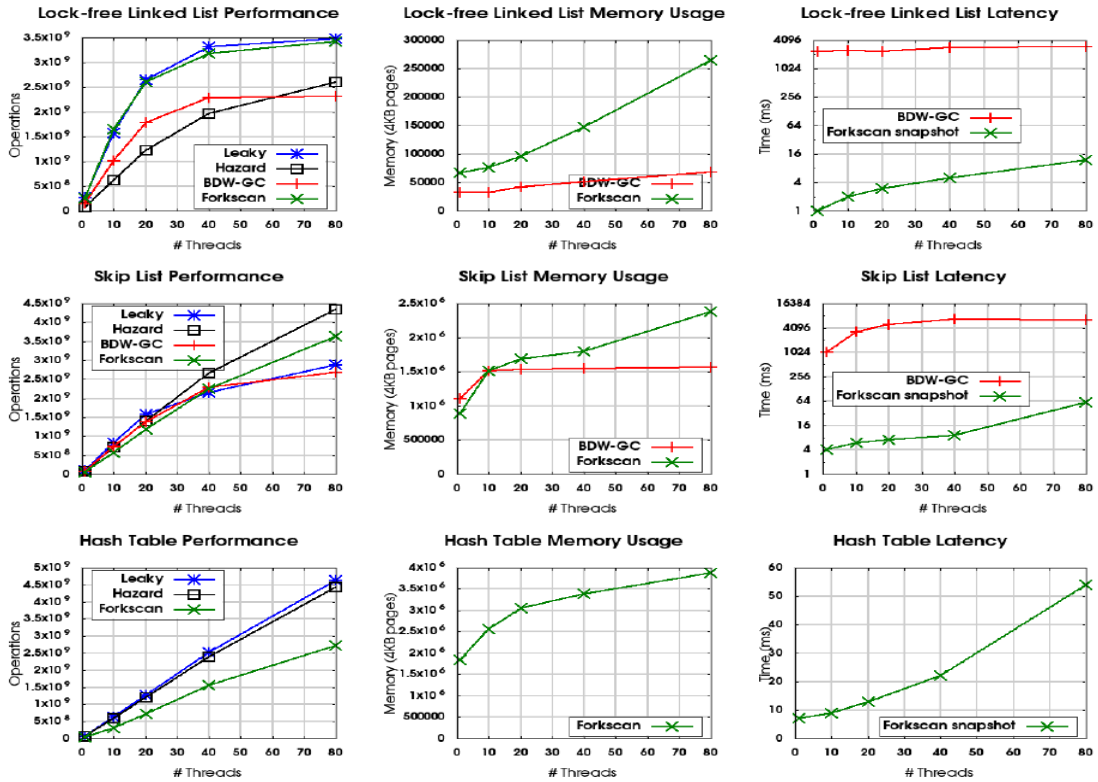


Figure 2. Performance results on the linked list, skiplist, and hash table data structures. For each: total operations, memory usage of the application, and average latency per reclamation iteration (logscale for the first two structures, to compare Forkscan with BDW-GC).

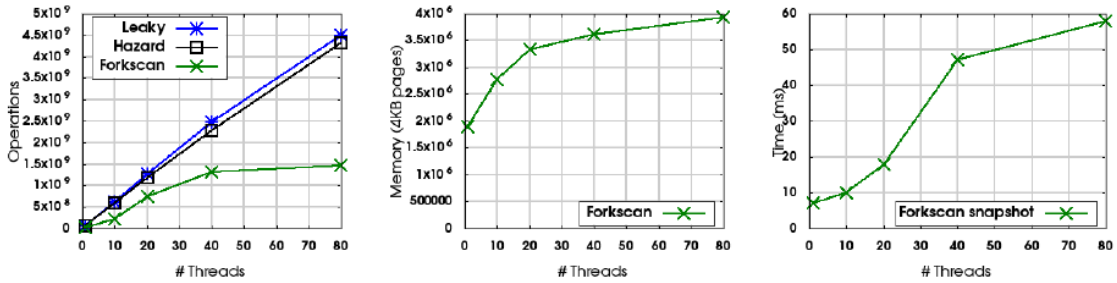


Figure 3. Performance results for a hash table with 40% update operations.

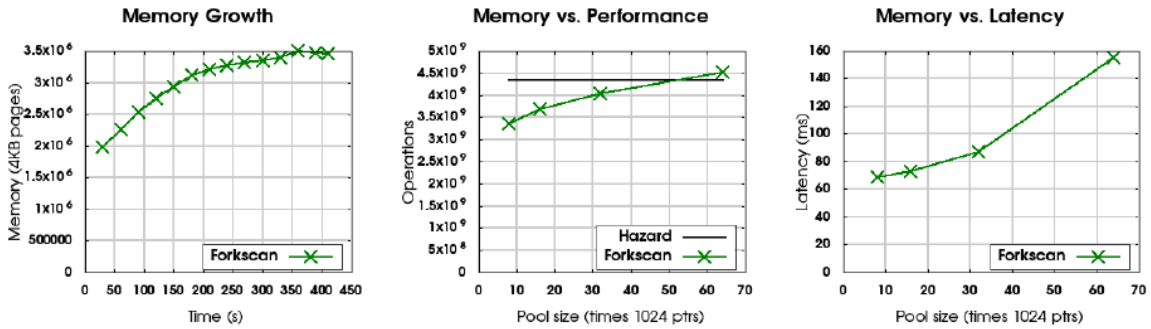


Figure 4. Forkscan memory usage and memory/latency vs. performance tradeoffs.

המשותף בין המאמרים:

שני המאמרים עוסקים בפתרון בעיית שחרור זיכרון בטוח במערכות מקבילות, שבהן חוטים שונים עשויים להחזיק מצביעים לאובייקטים שכבר אינם בשימוש. שני האלגוריתמים מציעים גישה אוטומטית לניהול זיכרון, ללא צורך במעקב ידני מצד המתכנת, מה שמונע שגיאות כמו Dangling Pointers ו-Double Free. עם זאת, הם פועלים בדרכים שונות: ThreadScan משתמש באותות (Signals) כדי לסנכרן חוטים ולסרוק את המחסנית והרגיסטרים שלהם בזמן ריצה, בעוד Forkscan יוצר תמונת זיכרון עקבית (Snapshot) באמצעות Fork ו-Copy-on-Write, ומבצע את הסריקה בתהליך נפרד, ללא השפעה ישירה על החוטים הראשיים. שני הפתרונות מספקים ביצועים גבוהים, אך מתאימים לתרחישים שונים: ThreadScan עדיף במערכות הדורשות תגובה מהירה עם תקורה נמוכה, מאחר שהוא מבצע סריקה ישירה של המחסנית, בעוד Forkscan מתאים למערכות בעלות נפח זיכרון גדול ומורכבות גבוהה, מאחר שהוא מפזר את עומס ניהול הזיכרון לאורך זמן בעזרת תהליך ייעודי. עם זאת, לכל פתרון יש מגבלות משלו – ThreadScan אינו אידיאלי למערכות עם שימוש כבד בזיכרון, מאחר שהוא דורש אינטראקציה תכופה עם החוטים הפעילים, בעוד Forkscan עשוי לגרום לעומס יתר תחת עומסים גבוהים מאוד, בשל העלות הכרוכה ביצירת תהליך הבן. לכן, הבחירה בין שני הפתרונות תלויה בדרישות הביצועים והזיכרון של המערכת ובאופן השימוש שלה במשאבי עיבוד וניהול זיכרון.

בעיות נוספות והצעה לשיפור

במצבים שבהם יש מעט תהליכים פעילים או שהמערכת פועלת בתנאי עומס נמוך (למשל, כאשר שינויים בזמן אמת או מערכות מושעות), האלגוריתמים של Forkscan עשויים להיות פחות יעילים בשל מנגנון Copy-On-Write, אשר דורש משאבים נוספים כדי לייצר תמונת זיכרון יציבה. מנגנון זה אינו בהכרח מתאים לכל תרחיש, מאחר שהוא מעכב את תהליך השחרור בכך שהוא מצריך יצירת עותק של זיכרון, גם כאשר הפעילות נמוכה יחסית. ייתכן שבמצבים כאלה, היה עדיף להשתמש במנגנון אחר המאפשר ניהול זיכרון דינמי ללא הצורך בהקפאת מצב החוטים.

כפי שמתואר במאמר, Forkscan אינו מבצע שחרור מידי של הזיכרון אלא רק לאחר שהוא מאתר מצביעים לאובייקטים שאינם בשימוש. גישה זו מבטיחה דיוק גבוה אך עשויה לגרום לעיכובים כאשר כמות התהליכים הפעילים נמוכה או כאשר יש עומס בלתי מאוזן. במקרים כאלה, משך הסריקה מתארך, ועשויה להיווצר תקורה לא רצויה בשל הצורך לבצע מחזורי סריקה נוספים כדי לוודא שהזיכרון משוחרר בצורה בטוחה.

הרעיון הוא לפתח גרסה "קלת משקל" של ForkScan, המייעלת את תהליך ניהול הזיכרון וממקדת את משאבי המערכת בתרחישים קריטיים. לדוגמה, במקום לבצע סריקה על כל הזיכרון לאחר כל מחזור סריקה, ניתן לזהות אזורי זיכרון רלוונטיים ולבצע עדיפות סלקטיבית, כך שחלקי הזיכרון החשובים ייסרקו בעדיפות ראשונה, בעוד אזורים פחות פעילים יידחו לשלבים מאוחרים יותר ובנוסף לשלב ניהול דינמי של תהליך הסריקה על ידי התאמה אוטומטית של זמני הריצה, כך שאזורי זיכרון קריטיים יסרקו מוקדם יותר במקרים של פעילות גבוהה.

כדי לפתור בעיות נוספות, אנו מציעים שילוב בין תהליכים עתירי זיכרון ותהליכים קלי זיכרון עם פתרון עדיפות הדרגתי (Decay Priority), אשר מאפשר לנהל באופן חכם את עומס הסריקה, כך שהמערכת תוכל לתעדף משימות בהתאם לרמת הדחיפות ולניצול המשאבים בזמן אמת.

1. **שימוש בדגימה (Sampling) לחיסכון במשאבים:**

כיצד זה עובד?

במקום לספור כל גישה לזיכרון, המערכת תשתמש במנגנון דגימה חכמה (Sampling) שיבדוק חלק מהגישות בלבד ויחלץ מהן מידע סטטיסטי על השימוש בזיכרון.

על ידי מעקב חלקי בלבד, האלגוריתם יוכל לקבל תמונה מייצגת של הדפים הפעילים ביותר, תוך הפחתת התקורה של ניהול גישות לכל דף זיכרון. דגימה זו יכולה להתבצע באמצעות טבלת עקיבה קומפקטית, אשר תשמור מספר קטן של דפים בעלי תדירות גבוהה ותעדכן את הדגימה רק אחת לכמה מחזורים.

למה זה עדיף?

- **חיסכון במשאבים:** במקום לעקוב אחר כל גישה לזיכרון, ניתן לקחת מדגם אקראי או מבוסס זמן ולחשב ממנו מגמות.
- **הקטנת התקורה של ניטור:** האלגוריתם לא צריך לנהל טבלאות ענקיות של כל גישה, אלא לשמור מידע על קבוצה קטנה של הדפים הפעילים ביותר.
- **תמונה אמינה של הזיכרון:** מכיוון שהדגימה מתעדכנת מחזורית, ניתן לקבל מידע עדכני ורלוונטי על הפעילות בזיכרון ללא בזבוז משאבים מיותר.
- **אפשר לבחור תדירות דגימה –** אפשר לקבוע תדירות דגימה לפי צרכי המתכנת (למשל דגימה אחת לכל ms100 לעומס נמוך, או כל ms10 אם יש צורך בתגובה אמינה יותר

לשינויים. ניתן להשתמש בדגימה אדפטיבית – תדירות הדגימה תעלה כאשר ForkScan מזהה פעילות חריגה בזיכרון.

פסאדו-קוד אפשרי למימוש של ForkScan עם מודל דגימות:

(נלקח מפסאודו קוד של ForkScan מהמאמר + תוספות שהוספנו למודל Sampling)

```
function CONSOLIDATE_PTRS
  // Called when a user thread pool is full.
  // Aggregate pointers from all threads.
  delete-buffer ← Ø
  for th ϵ threads do
    delete-buffer U = GET_PTRS_POOL(th)
  SORT(delete-buffer)
  SIGNAL-CONDITION-VAR(reclaimer-conditional)

function RECLAIMER_THREAD(ctx)
  while 1 do
    WAIT-ON-CONDITION-VAR(reclaimer-conditional)
  \\Signal all threads to pause
    for th ϵ threads do
      SIGNAL(th, snapshot)
    wait for ACK from all threads
  \\At this point, the system is "frozen," so we fork
    pid ← FORK()
    if pid = 0 then
  \\The child scans the memory snapshot
      SCAN(ctx)
      EXIT()
  \\ This is the parent
    resume all threads via signal
  \\ Child got snapshot
    wait for child to finish
    PUSH-BACK(delete-buffer)
  \\ Free memory

\\============== Memory Sampling (Tracks Access Frequency) ======= Added
function SAMPLE_MEMORY_USAGE(ctx)
  memory-ranges ← GET_MAPPED_RANGES()
  access-counters ← HASHMAP()
  \\Maps memory range -> access count
  for each range ϵ memory-ranges do
```

```
    access-counters[range] ← 0
  \\ Initialize access counters
  while program is running do
    sleep(SAMPLING_INTERVAL)
  \\ Sample periodically
    sampled-ranges ← RANDOM_SUBSET(memory-ranges, SAMPLE_SIZE)
  \\ Avoid full scan
    for range in sampled-ranges do
      if IS_PAGE_ACCESSED(range) then
        access-counters[range] += 1
  \\ Increment counter
    RESET_PAGE_TRACKING(sampled-ranges)
  \\ Clear access tracking
  return access-counters
  \\ Return stats for scan prioritization

\\============== Scanning Memory (Prioritizes Hot Regions) ======= Updated
function SCAN(ctx)
  memory-ranges ← GET_MAPPED_RANGES()
  access-stats ← SAMPLE_MEMORY_USAGE(ctx)
  \\ Get sampling data
  SORT(memory-ranges, DESCENDING_BY(access-stats))
  \\ Prioritize high-use pages
  Split memory-ranges into M partitions
  for id ϵ [1..M-1] do
    pid ← FORK()
    if pid = 0 then
      SCAN_FOR_REFS(memory-ranges[id])
      EXIT()
  SCAN_FOR_REFS(memory-ranges[0])
  Wait for all children to finish

function SCAN_FOR_REFS(memory-ranges)
  for each word ϵ memory-ranges do
  \\ Check if word is a reference to some object
    i ← BINARY-SEARCH(word, delete-buffer)
    if i ≠ 0 then
  \\ Found a reference -> record it
      SET-LOW-BIT(delete-buffer[i])
      SCAN_FOR_REFS(delete-buffer[i])

function SNAPSHOT_SIGNAL_HANDLER(ctx)
```

```
\\ Executed by a thread on receiving snapshot signal
Spill registers and stack boundaries to stack
Send ACK to reclaimer-thread
Wait for resume signal
\\ freeze point for snapshot
```

2. **הפחתת עדיפות הדרגתית (Decay Priority) לאזורים קריטיים בתדירות נמוכה**

כיצד זה עובד?

- לכל אזור זיכרון יש מדד עדיפות (Priority Score) שקובע באיזו תדירות הוא ייסרק.
- אזורים פעילים מאוד יקבלו עדיפות גבוהה ויסרקו לעיתים קרובות יותר.
- אזורים שאינם נגישים לעיתים קרובות יקבלו עדיפות נמוכה יותר עם הזמן – אלא אם כן הם מסומנים כחשובים.
- אם אזור לא נגיש במשך מספר מחזורי סריקה, האלגוריתם יקטין בהדרגה את תדירות הסריקה שלו, כדי לא לבזבז עליו משאבים.

כיצד זה מונע איבוד מידע חשוב?

- אם אזור לא נסרק למשך זמן רב מידי, האלגוריתם יוודא שהוא עדיין ייסרק אחת לכמה מחזורים, גם אם עדיפותו ירדה.
- מבנים קריטיים (כגון טבלאות האש או רשימות קישורים ארוכות-חיים) לא יאבדו עדיפות לחלוטין, גם אם הם בתדירות נמוכה.
- האלגוריתם מאזן בין יעילות ובין שמירה על אזורים קריטיים, כך שאף אזור חשוב לא יוזנח.

מה היתרונות בגישה זו?

- **הפחתת בזבוז משאבים:** אין צורך להמשיך לסרוק דפים ישנים שלא ניגשו אליהם זמן רב.
- **התאמה דינמית למערכת:** כאשר פעילות הזיכרון משתנה, ForkScan יכול להגיב במהירות ולשנות עדיפויות בהתאם.
- **שמירה על מבני נתונים קריטיים:** אפילו אזורים בתדירות נמוכה יקבלו סריקה תקופתית מובטחת, מה שמונע דליפות זיכרון בלתי צפויות.

פסאדו-קוד אפשרי למימוש של ForkScan עם מודל הפחתת עדיפות הדרגתית:

(נלקח מפסאודו קוד של ForkScan מהמאמר + תוספות Decay Priority + Sampling)

```
function CONSOLIDATE_PTRS
  // Called when a user thread pool is full.
  // Aggregate pointers from all threads.
  delete-buffer ← Ø
  for th ϵ threads do
    delete-buffer U = GET_PTRS_POOL(th)
  SORT(delete-buffer)
  SIGNAL-CONDITION-VAR(reclaimer-conditional)
```

```
function RECLAIMER_THREAD(ctx)
  while 1 do
    WAIT-ON-CONDITION-VAR(reclaimer-conditional)
  \\Signal all threads to pause
    for th ϵ threads do
      SIGNAL(th, snapshot)
    wait for ACK from all threads
  \\At this point, the system is "frozen," so we fork
    pid ← FORK()
    if pid = 0 then
  \\The child scans the memory snapshot
      SCAN(ctx)
      EXIT()
  \\ This is the parent
    resume all threads via signal
  \\ Child got snapshot
    wait for child to finish
    PUSH-BACK(delete-buffer)
  \\ Free memory

\\============== Memory Sampling (Tracks Access Frequency) ======= Added
function SAMPLE_MEMORY_USAGE(ctx)
  memory-ranges ← GET_MAPPED_RANGES()
  access-counters ← HASHMAP()
  \\Maps memory range -> access count
  priority-scores ← HASHMAP()
  \\ Maps memory range -> priority score
  for each range ϵ memory-ranges do
    access-counters[range] ← 0
  \\ Initialize access counters
    priority-scores[range] ← DEFAULT_PRIORITY
  \\ Default priority can be adjusted
  while program is running do
    sleep(SAMPLING_INTERVAL)
  \\ Sample periodically
    sampled-ranges ← RANDOM_SUBSET(memory-ranges, SAMPLE_SIZE)
  \\ Avoid full scan
    for range in sampled-ranges do
      if IS_PAGE_ACCESSED(range) then
        access-counters[range] += 1
  \\ Increment counter
```

```
        priority-scores[range] ← CALCULATE_NEW_PRIORITY(priority-scores[range],
access-counters[range])
  \\ Adjust priority scores
      RESET_PAGE_TRACKING(sampled-ranges)
  \\ Clear access tracking
   return access-counters, priority-scores
  \\ Return stats for scan prioritization

\\============== Priority Score Decay ======================== Added
function CALCULATE_NEW_PRIORITY(current_priority, access_count)
   DECAY_FACTOR ← 0.95
  \\ Priority decay factor to gradually reduce priority of low-access regions
   MAX_PRIORITY ← 100
   MIN_PRIORITY ← 1
 \\ Increase priority for frequently accessed regions
   new_priority ← current_priority + (access_count * ACCESS_WEIGHT)
 \\ Apply decay and make sure the priority stays within bounds
   new_priority ← new_priority * DECAY_FACTOR
   if new_priority > MAX_PRIORITY then
      new_priority ← MAX_PRIORITY
   else if new_priority < MIN_PRIORITY then
      new_priority ← MIN_PRIORITY
   return new_priority

\\============== Critical Memory Regions Allocation ========= Added
function ALLOCATE_CRITICAL_REGIONS(memory-ranges)
   critical-regions ← Ø
  \\ Initialize a list to hold critical regions
   for each range ϵ memory-ranges do
      if IS_CRITICAL(range) then
  \\ If the region is critical based on program requirements
         priority-scores[range] ← MAX_PRIORITY
  \\ Allocate high priority to critical regions
         critical-regions += range
   return critical-regions
  \\ Return list of critical regions with max priority

\\============== Scanning Memory (Prioritize High-Priority Regions) == Updated
function SCAN(ctx)
   memory-ranges ← GET_MAPPED_RANGES()
   access-stats, priority-scores ← SAMPLE_MEMORY_USAGE(ctx)
  \\ Get sampling data
```

```
    critical-regions ← ALLOCATE_CRITICAL_REGIONS(memory-ranges)
   \\ Allocate critical regions as high priority
    for each range ϵ critical-regions do
      priority-scores[range] ← MAX_PRIORITY
   \\ Merge critical regions with regular memory ranges, preserving their priority
   \\ Ensure critical regions stay prioritized
    SORT(memory-ranges, DESCENDING_BY(priority-scores))
   \\ Prioritize by calculated priority scores
    Split memory-ranges into M partitions
    for id ϵ [1..M-1] do
      pid ← FORK()
      if pid = 0 then
        SCAN_FOR_REFS(memory-ranges[id])
        EXIT()
    SCAN_FOR_REFS(memory-ranges[0])
    Wait for all children to finish

  function SCAN_FOR_REFS(memory-ranges)
    for each word ϵ memory-ranges do
   \\ Check if word is a reference to some object
      i ← BINARY-SEARCH(word, delete-buffer)
      if i ≠ 0 then
   \\ Found a reference -> record it
        SET-LOW-BIT(delete-buffer[i])
        SCAN_FOR_REFS(delete-buffer[i])

  function SNAPSHOT_SIGNAL_HANDLER(ctx)
   \\ Executed by a thread on receiving snapshot signal
    Spill registers and stack boundaries to stack
    Send ACK to reclaimer-thread
    Wait for resume signal
   \\ freeze point for snapshot
```

שיפור ForkScan באמצעות דגימה חכמה (Sampling) לזיהוי אזורים פעילים יחד עם הפחתת עדיפות הדרגתית (Decay Priority) יאפשר למערכת לעבוד בצורה יעילה יותר, לצמצם בזבוז משאבים, ולהתאים את הסריקה לתנאי השימוש.

במקום לבצע סריקה מיותרת על כל הזיכרון, ForkScan יתמקד במה שבאמת רלוונטי, תוך שמירה על מבנים קריטיים גם אם הם בתדירות נמוכה.

הוכחת נכונות

רקע על נכונות אלגוריתמים לניהול זיכרון במערכות מקבילות:

כדי להוכיח שהאלגוריתם המשופר שומר על נכונות, עלינו להבטיח שהוא משחרר רק אובייקטים שאינם נגישים ובמקביל אינו משאיר זיכרון לא מנוצל באופן משמעותי. בנוסף, האלגוריתם חייב לספק הבטחות שלמות (completeness) כלומר, לוודא שכל הזיכרון שאינו נגיש משוחרר בסופו של דבר.

תנאי נכונות בסיסיים לניהול זיכרון מוצלח:

תנאי 1: אין שחרור מוקדם של זיכרון (No Premature Freeing), כל אובייקט נשמר בזיכרון עד שכל החוטים יסיימו את השימוש בו.

תנאי 2: אין זיכרון שגוי שנשאר מוקצה (No Memory Leak Guarantee), אם אין הפניות לאובייקט כלשהו, הוא ישוחרר בתוך זמן סופי.

תנאי 3: שחרור יעיל (Efficiency in Reclamation), יש להימנע מזיכרון לא מנוצל על ידי שחרור מוקדם של דפים עם עדיפות נמוכה, אך תוך שמירה על מינימום סריקות מיותרות.

הוכחת נכונות – דגימה (Sampling) לזיהוי אזורים בתדירות גבוהה:

- ✓ האלגוריתם משתמש במנגנון דגימה הסתברותית (Sampling), שמאפשר לו לזהות באופן סטטיסטי אילו דפים נגישים בתדירות גבוהה.
- ✓ גם אם לא כל הגישות מתועדות, חוק המספרים הגדולים (Law of Large Numbers) מבטיח שהמדגם עדיין מייצג באופן קרוב את פעילות הזיכרון בפועל.
- ✓ מכיוון שהדגימה כל הזמן מתעדכנת, אזורי זיכרון שניגשים אליהם לעיתים קרובות ימשיכו לקבל עדיפות גבוהה.

הוכחת שלמות של דגימה הסתברותית:

נניח שיש n דפים בזיכרון, ומתבצעת k דגימות לכל מחזור.

הסתברות שתדגום דף זיכרון שהגישה אליו נמוכה היא p=k/n, אם p קטן מאוד, עדיין נבטיח שדפים קריטיים ייבדקו דרך עדיפות דינמית (Decay Priority).

מכאן נובע שהמדגם מתכנס לערך מייצג ככל שהזמן עובר.

הוכחת נכונות – הפחתת עדיפות הדרגתית (Decay Priority):

- ✓ כל אזור בזיכרון מתחיל עם עדיפות ברירת מחדל ומקבל עדיפות גבוהה יותר אם ניגשו אליו בתדירות גבוהה.
- ✓ אם אזור לא נגיש לאורך זמן, העדיפות שלו יורדת באופן הדרגתי—אך אף פעם לא נעלמת לחלוטין.
- ✓ אזורים שמסומנים כקריטיים יקבלו תמיד עדיפות מינימלית מסוימת כדי שלא יוזנחו בטעות.

הוכחת שלמות של Decay Priority:

נסמן את העדיפות של אזור זיכרון P_t בזמן t.

הגדרת האלגוריתם היא: $P_{t+1} = ((w \cdot A_t) + P_t) * 0.95$

כאשר:

0.95 – פקטור דעיכה (Decay Factor).

A_t – מספר הפעמים שהדף נגיש במחזור הנוכחי.

w – משקל הגישה.

במערכת יציבה, אם אזור אינו נגיש לאורך זמן ($A_t = 0$) אזי P_t מתכנס לערך מינימלי ולא לאפס.

מכיוון שאזורי זיכרון קריטיים מסומנים מראש, גם אם יש דעיכה, עדיין תהיה להם עדיפות כבוהה יחסית ככה שלא ייתכן שזיכרון חשוב יוזנח.

לפיכך, כל אזור זיכרון רלוונטי ייסרק בתוך זמן סופי.

<u>התאמה למודל הזיכרון של ForkScan:</u>

כדי לבדוק שהמודל תואם את ForkScan המקורי, נבחן שלושה תרחישים אפשריים:

תרחיש 1- תהליכונים ניגשים לזיכרון בתדירות גבוהה:

- ✓ Sampling יזהה אותם במהירות.
- ✓ Decay Priority תגדיל את חשיבותם באופן דינמי.
- ✓ סריקות יבוצעו בתדירות גבוהה יותר.

תרחיש 2- אזורי זיכרון ישנים שאינם בשימוש יורדים בתדירות:

- ✓ Decay Priority יקטין בהדרגה את עדיפותם.
- ✓ Sampling לא יתעד אותם יותר אם לא מתבצעת גישה אליהם.
- ✓ פחות סריקות מיותרות – שיפור ביצועים.

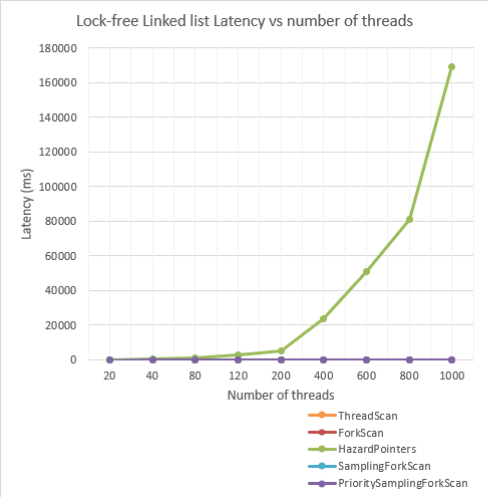
תרחיש 3- אזור בעל גישה נדירה אך קריטית:

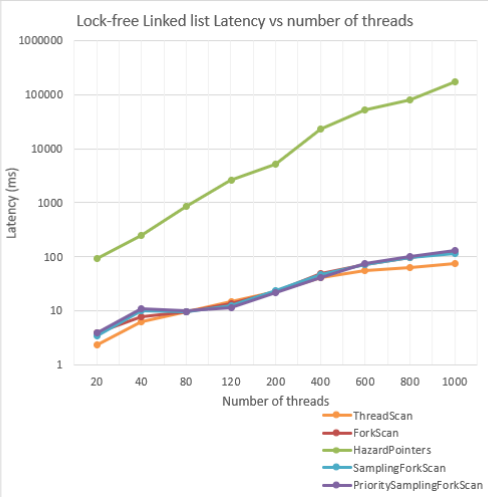
- ✓ אם אזור זה מסומן כקריטי, העדיפות שלו תישמר.
- ✓ Decay Priority ישמור אותו בסריקות מחזוריות, גם אם אין בו שימוש שוטף.

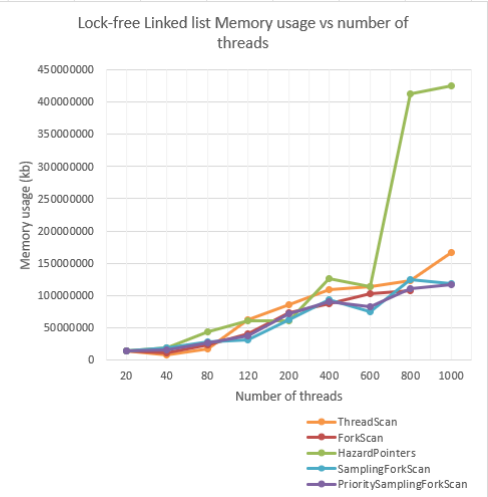
מסקנה – האלגוריתם נכון ושומר על יעילות גבוהה:

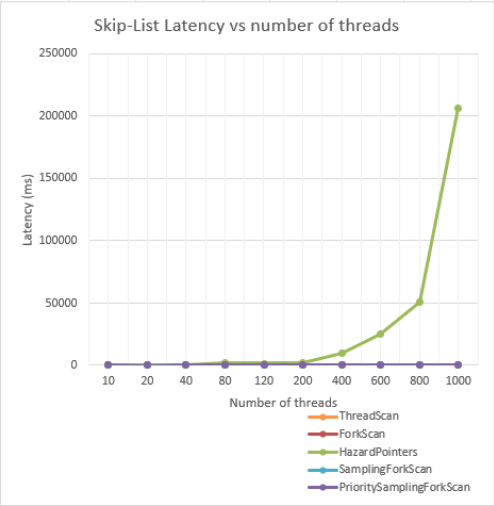
- ✓ האלגוריתם אינו משחרר אובייקטים שעדיין נגישים, כי: Sampling מזהה אזורים נגישים בסטטיסטיקה מדויקת ו- Decay Priority שומר על זיכרון קריטי בסריקות גם אם הגישה אליו נמוכה.
- ✓ האלגוריתם משחרר כל זיכרון שאינו נגיש בזמן סופי, כי: אזורים ללא גישה מאבדים בהדרגה עדיפות ונמחקים ו- מבנים קריטיים תמיד נסרקים בפרקי זמן מוגדרים.
- ✓ האלגוריתם יעיל יותר מ-ForkScan הרגיל, כי: הסריקות מתמקדות רק באזורים רלוונטיים, במקום לבצע סריקות מיותרות והעומס על Copy-on-Write קטן, מה שמפחית תקורה על המערכת.

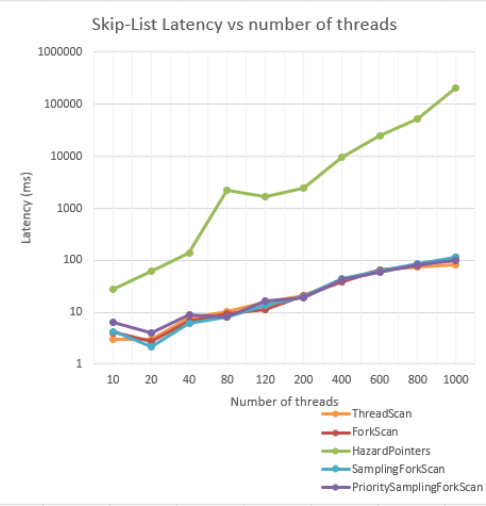
ביצועים עם ריבוי חוטים (20-1000)

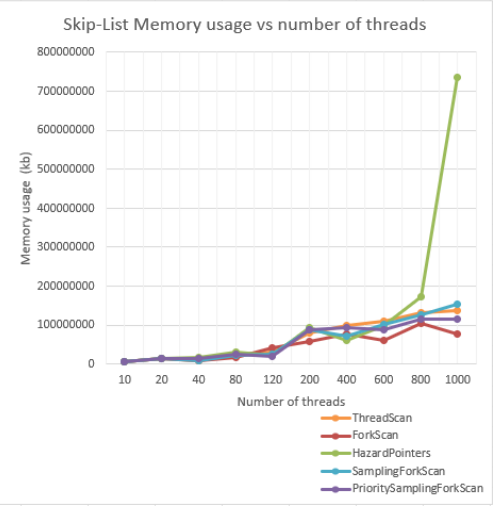


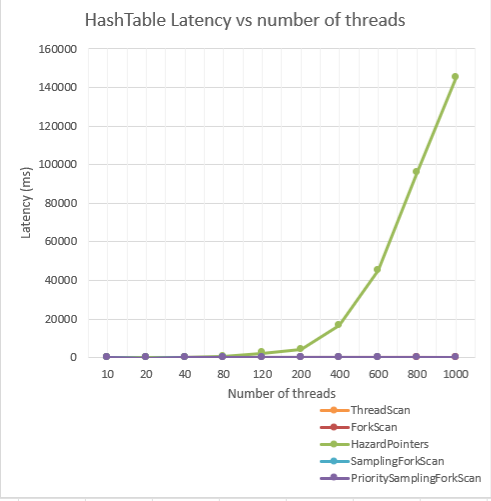


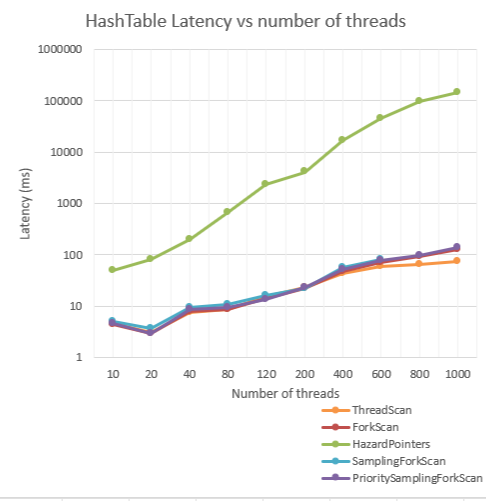


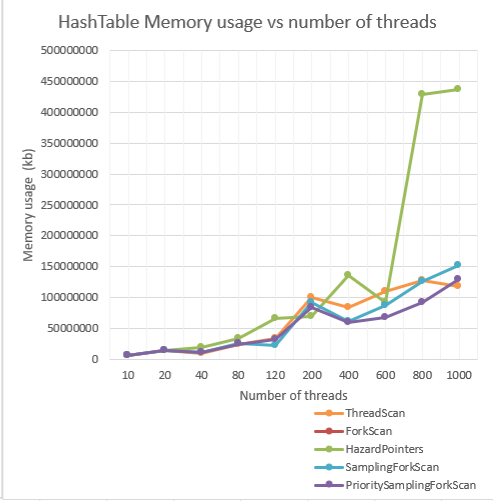




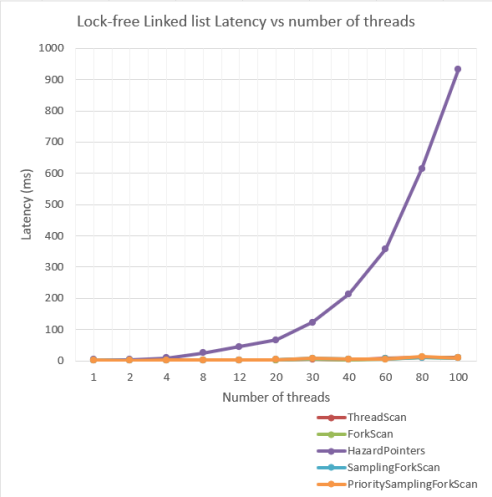


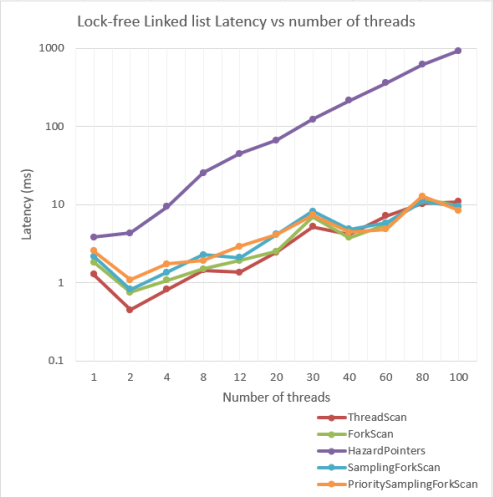


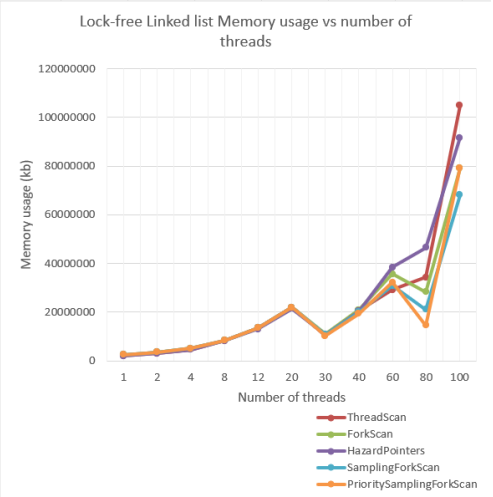


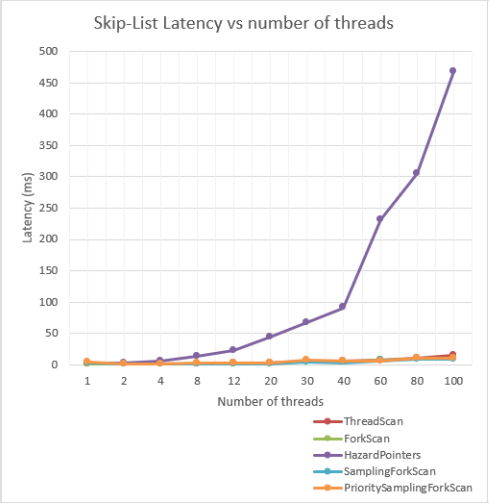


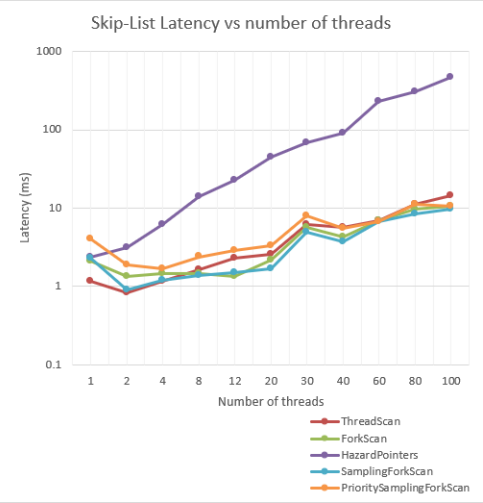
ביצועים עם מעט חוטים (1-100)

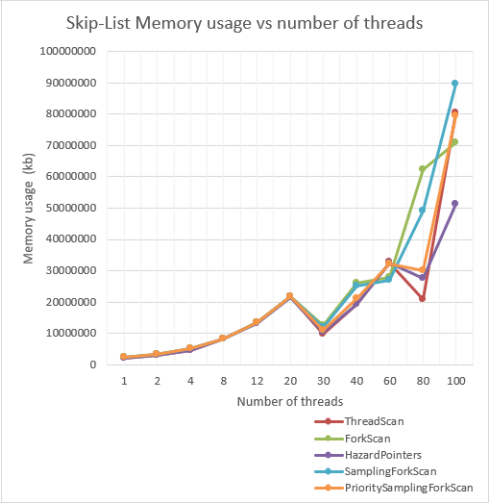


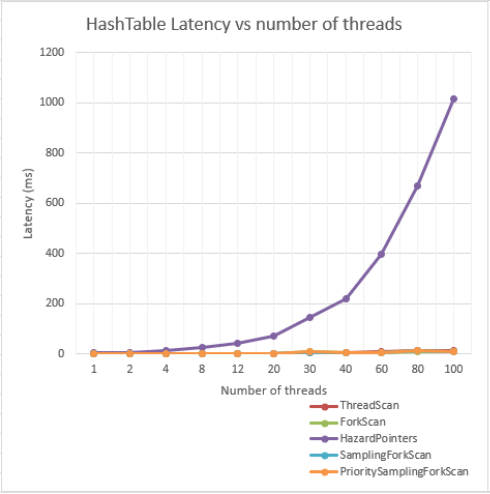


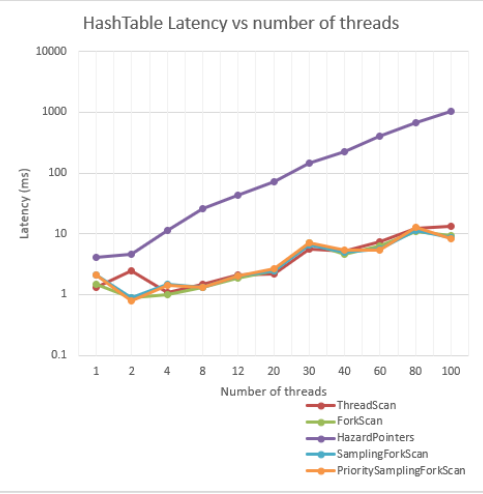


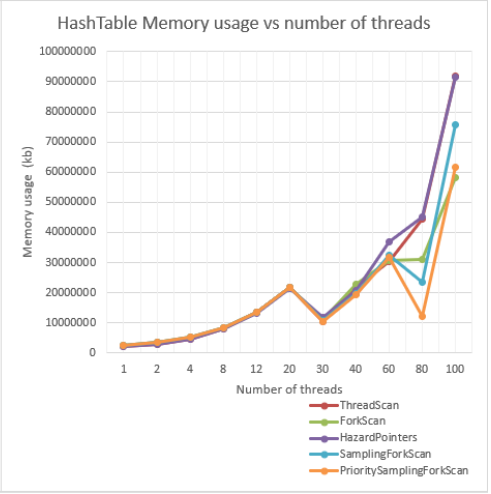












מקורות:

[1] Alistarh, Dan et al. “ThreadScan: Automatic and Scalable Memory Reclamation.” Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA 2015), June 13-15, 2015, Portland, Oregon, USA, Association for Computing Machinery (ACM), June 2015 © 2015 Association for Computing Machinery (ACM). https://dl.acm.org/doi/abs/10.1145/3201897

[2] Alistarh, Dan et al. "Forkscan: Conservative Memory Reclamation for Modern Operating Systems." EuroSys '17: Proceedings of the Twelfth European Conference on Computer Systems, April 2017, Belgrade, Serbia, Association for Computing Machinery, April 2017 © 2017 The Authors. https://dl.acm.org/doi/abs/10.1145/3064176.3064214

[3] Mitzenmacher, M., & Upfal, E. (2017). "Probability and Computing: Randomized Algorithms and Probabilistic Analysis".

רלוונטיות: מסביר כיצד טכניקות דגימה הסתברותית מאפשרות חיזוי מדויק של דפוסי גישה לזיכרון במערכות דינמיות.

[4] Michael, M. M. (2002). "Safe memory reclamation for dynamic lock-free objects using atomic reads and writes". ACM Transactions on Parallel Computing.

רלוונטיות: מציג טכניקות למניעת שחרור מוקדם של זיכרון, התואמות לרעיונות של Decay Priority .

[5] Maged M. Michael (2004) “Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects”.