**הטכניון – מכון טכנולוגי לישראל**
**Technion – Israel Institute of Technology**

# ספריות הטכניון
## The Technion Libraries

---

**בית הספר ללימודי מוסמכים ע"ש ארווין וג'ואן ג'ייקובס**
*Irwin and Joan Jacobs Graduate School*

# ניהול זיכרון יעיל לשרתים

הראל פז

# ניהול זיכרון יעיל לשרתים

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר

דוקטור לפילוסופיה

הראל פז

הוגש לסנט הטכניון – מכון טכנולוגי לישראל

אלול תשס''ו חיפה אוגוסט 2006

המחקר נעשה בהנחיית ד"ר ארז פטרנק ובייעוצו של ד"ר הילל קולודנר

בפקולטה למדעי המחשב.

בראש בראשונה, ברצוני להביע את הערכתי הרבה למנחה שלי, דוקטור ארז פטרנק. ארז, בדרכו הנעימה, חשף אותי לעולם המחקר האקדמי וחלק עימי את הידע הנרחב שלו. ארז היה תמיד זמין, נגיש ומוכן לעזור, למרות לוח הזמנים הצפוף שלו. הייתה זו גם זכות וגם תענוג לעבוד עם ארז. למרות שעצוב להיפרד, אני מקווה שדרכינו יפגשו בעתיד.
ברצוני להודות ליועץ שלי, דוקטור הילל קולודנר, על שחלק עימי את תבונתו בדיונים הארוכים שערכנו. במהלך עבודתי, נעזרתי ברבים, מלבד ארז והילל, וזהו הזמן והמקום להביע את תודתי על עזרתם. חזי אזאצ'י עזר לי להיכנס לענייניים בתחילת לימודי המוסמכים. הדיונים המועילים שניהלתי עם יואב אוסייה, פרופ' מולי שגיב, דוקטור רן שחם, חיים קרמני ואורי זילברשטיין תרמו לי מאד. ברצוני להודות לפרופ' חגית עטייה על הערותיה מאירות העיניים על (גרסה מוקדמת של) עבודה זו, שעזרו לי לשפר את איכות הצגת העבודה. אני אסיר תודה לערן איסלר ולמקסים קובגן מהמעבדה למערכות מבוזרות של הפקולטה על תמיכה טכנית רבת שנים.
בנוסף, ברצוני להודות לבאים על תמיכתם המוראלית במהלך עבודה זו. במהלך לימודי המוסמכים, היה לי לעונג לחלוק חדר עם רועי מלמד. יחסי היום-יום עם רועי עזרו לי להתגבר על רגעים קשים. תודה אחרונה, אם כי (ממש) לא אחרונה בחשיבותה, ברצוני להוקיר למשפחתי, על תמיכתם, על סבלנותם ועל אהבתם.

אני מודה לקרן לזכרם של אהרון ומרים גוטווירת על מענק המחקר שהוענק לי.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי

# תוכן עניינים

# רשימת איורים

# רשימת טבלאות

# תקציר


זיכרון המוקצה בצורה דינאמית עלול להיהפך ללא ישיג. עצם שאינו חי אך גם אינו משוחרר נקרא זבל. ניהול זיכרון ידני צורך זמן (מהמתכנת) ומועד לטעויות כגון שחרור מוקדם של זיכרון, שחרור כפול של זיכרון ושכיחת שחרור זיכרון. איסוף זבל הוא שחרור אוטומטי של זיכרון שהוקצה בצורה דינאמית. איסוף זבל משחרר את המתכנת מנטל ניהול הזיכרון הידני ופותר את הבעיות שהוזכרו.

בעבודה זו אנו מתרכזים באיסוף זבל במערכות מרובות-מעבדים. מערכות מרובות-מעבדים הפכו לשכיחות ביותר כשרתים ואף חודרות יותר ויותר לפלח המחשבים השולחניים. רבים מאלגוריתמי איסוף הזבל שתוכננו עבור מחשבים בהם מעבד יחיד עובדים בתצורת *עצור-את-העולם*. אוסף זבל, העובד בתצורת עצור-את-העולם, עוצר (משעה) את כל חוטי התוכנית בזמן איסוף הזבל. אוספי זבל נאיביים משתמשים רק באחד מהמעבדים במערכות מרובות-מעבדים. בפרט, אוסף זבל נאיבי העובד בתצורת עצור-את-העולם במערכת מרובת-מעבדים, רץ על מעבד יחיד בעוד כל חוטי התוכנית מושעים. גישה זו מנצלת בצורה גרועה את משאבי המערכת (רק מעבד אחד פעיל בזמן איסוף הזבל), גורמת לכך שהתוכנית תרוץ לאט יותר ומשעה את חוטי התוכנית לפרק זמן ארוך יותר. לכן, שיטה זו אינה סקאלבילית.

מערכות מרובות-מעבדים המשתמשות בערימות גדולות מספקות אתגר לתכנון אוספי זבל יעילים. בפרט רצוי שתוכנית הרצה על שרת תושהה לפרק זמן קצר ככל האפשר כתוצאה מאיסוף זבל. לשם כך, אלגוריתמים בו-זמניים[1] הוצגו ונחקרו בעבר. בניגוד לגישת עצור-את-העולם, בו אוסף הזבל פועל רק כאשר כל חוטי התוכנית מושעים, אוסף-זבל בו-זמני פועל במקביל לריצת חוטי התוכנית. לכן, כאשר משתמשים באוסף-זבל בו-זמני, פרק הזמן בו חוטי התוכנית מושעים סימולטאנית קצר משמעותית מזמן ההשהיה של אוסף זבל העובד בתצורת עצור-את-העולם.

עם זאת, רב אוספי הזבל הבו-זמניים עדיין עוצרים את כל התוכנית בבת-אחת (בדרך כלל על-מנת להתחיל או לסיים את האיסוף). עצירת כל חוטי התוכנית סימולטאנית היא פעולה יקרה כיוון שבדרך-כלל, חוטים מסוגלים לעצור אך ורק בנקודות בתוכנית המכונות *נקודות בטוחות* (כלומר לא ניתן לעצור חוט בכל נקודת זמן שרירותית). בנקודות אלה אוסף הזבל מסוגל להסיק מהו גרף הקשירות של העצמים בתוכנית ולאסוף את הזבל בצורה בטוחה. המשמעות של נקודות בטוחות, כשמשתמשים באוסף זבל בו-זמני, היא שחוטי תוכנית שנעצרו (לשם האיסוף) חייבים לחכות לכל הפחות עד שכל החוטים האחרים יגיעו לנקודה בטוחה (ויעצרו) לפני שיוכלו לשוב לרוץ. תכונה זו מפחיתה מהסקאלביליות של אוספי זבל הבו-זמניים שכן ככל שמספר חוטי התוכנית גדל, מתארך שלב השהית התוכנית (הדרוש לשם עצירת כל החוטים בבת-אחת) בעת איסוף זבל. אם בנוסף אוסף הזבל איננו *מקבילי*, הרי שבזמן שחוטי התוכנית מושעים, רק אחד מהמעבדים במערכת מנוצל.

לכן, על מנת לקצר את זמן ההשהיה, ניתן להשתמש באיסוף זבל *תוך-כדי-מעוף*[2]. אוסף זבל תוך-כדי-מעוף הוא אוסף זבל בו-זמני שאינו עוצר את כל חוטי התוכנית סימולטאנית: כל חוט משתף פעולה עם אוסף הזבל (כלומר מושעה) בקצב שלו באמצעות מנגנון המכונה לחיצת-ידיים.

אפשרות נוספת לאיסוף זבל על מערכת מרובת-מעבדים היא לבצע את איסוף הזבל בצורה מקבילית על ידי מספר חוטי איסוף זבל העובדים במקביל. אלגוריתם מקבילי משיג תפוקה גבוהה יותר (גובה תקורה נמוכה יותר) מאלגוריתם בו-זמני, אולם הוא כופה זמן השהייה ארוך יותר (בדרך כלל, ארוך בלפחות סדר גודל). בעבודה זו התרכזנו באלגוריתמים בו-זמניים (ובפרט, באלגוריתמי תוך-כדי-מעוף) ולא באיסוף זבל מקבילי.

**ספירת מצביעים.** עבודה זו מתמקדת בעיקר (אך לא רק) באוספי זבל מבוססי אלגוריתם ספירת מצביעים. הרעיון בבסיס אלגוריתם זה הוא לשמור עבור כל עצם מונה הסופר את מספר המצביעים אליו. עצם נוצר עם מונה בעל ערך 1. בעת שינוי מצביע, מגדילים באחד את מונה העצם אליו נוספה ההצבעה ומפחיתים באחד את מונה העצם ממנו הוסרה ההצבעה. אם מונה של עצם מסוים מתאפס, ניתן לשחרר עצם זה כיוון שמובטח שהתוכנית לא תשתמש בו יותר. לפני שהעצם משוחרר, מפחיתים באחד את מוני העצמים המוצבעים ישירות ע"י עצם זה (מה שעשוי לגרום לשחרור רקורסיבי של עצמים נוספים).


---

[1] Concurrent באנגלית.

[2] On-the-fly באנגלית.

**מוטד**

מסורתית, אלגוריתמי ספירת מצביעים סבלו מהחסרונות המובנים הבאים: תקורה גבוהה, תקורת מקבול יקרה עוד יותר (פעולת עדכון מצביע דרשה לפחות פעולת סנכרון אחת) וחוסר היכולת לאסוף מעגלי זבל. חסרונות אלו הניעו חוקרים ומערכות מסחריות להתמקד בעיקר באלגוריתמים מסוג סריקת ערימה. לכן, המחקר והשימוש באוספי זבל מבוססי שיטת ספירת המצביעים (ובפרט ספירת מצביעים בו-זמנית), לא היה נרחב כמו באלה המבוססים על שיטת סריקת הערימה, ובפועל נוצר פער מחקרי של כשני עשורים בין שתי השיטות. מחקרים שבוצעו בשנים האחרונות בנושא אלגוריתמי ספירת מצביעים בסביבות מודרניות, הצליחו להקטין בצורה ניכרת חלק מהחסרונות המובנים הנ"ל. בפרט, לבנוני ופטרנק הצליחו להקטין בצורה ניכרת את העבודה הנדרשת מאלגוריתם ספירת מצביעים. בנוסף הם הראו כיצד ניתן להיפטר מרוב תקורת המקבול של אוסף זבל המבוסס על ספירת מצביעים (האלגוריתם שהציגו אנו דורש סנכרון בפעולת עדכון המצביע). כמו כן, שיטת ספירת המצביעים נראית מבטיחה לאיסוף זבל במערכות עתידיות בהן ישתמשו בערימות גדולות יותר. אלגוריתמים מבוססי סריקת ערימה חייבים לסרוק את כל העצמים החיים (בכל איסוף), ולכן עבודת אוספי הזבל גדלה ככל שמספר העצמים החיים גדל. בניגוד לכך, עבודת אוסף זבל המבוסס על ספירת מצביעים אינה תלויה במספר העצמים החיים אלא פרופורציונית לעבודה הנעשית ע"י התוכנית (בין שני איסופים) ולכמות הזיכרון שיש לאסוף.

המוטו המרכזי שלנו בעבודה זו הוא להפחית את הפער המחקרי שנוצר לרעת אלגוריתמי ספירת המצביעים, בייחוד מכיוון שאלגוריתמי איסוף זבל מבוססי ספירת מצביעים נראים מבטיחים לאיסוף זבל במערכות עתידיות. בפרט אנו מתרכזים בשיפור יעילות אוספי זבל העובדים בשיטת ספירת המצביעים על מערכות מרובות-מעבדים, ע"י תכנון ומימוש אלגוריתמים חדשים ומשופרים לניהול זיכרון לשרתים.

במהלך המחקר עבדנו על הפרוייקטים הבאים:

1. **אלגוריתם סריקת ערימה העובד תוך-כדי-מעוף**

   אנו מציעים אלגוריתם סריקת ערימה חדש העובד תוך-כדי-מעוף ומבוסס על אלגוריתם המבטים המחליקים של לבנוני ופטרנק. מימשנו את האלגוריתם הנ"ל על מכונת הג'אווה הוירטואלית המחקרית Jikes. הרצנו אותו על גבי שרת נטפיניטי והשוונו אותו לאלגוריתם הבו-זמני ולאלגוריתם המקבילי (העובד בשיטת עצור-את-העולם) אשר מסופקים עם Jikes. זמן ההשהיה המרבי שנמדד עבור תוכניות בודקות הביצועים היה לכל היותר 2 אלפיות שנייה. בכל המדידות, זמן ההשהיה היה לא רק קטן בשני סדרי גודל מזה של האלגוריתם העובד בשיטת עצור-את-העולם, אלא גם קטן מזמן ההשהיה של האלגוריתם הבו-זמני של Jikes. מבחינת התפוקה, האלגוריתם שלנו השיג תפוקה הגבוהה בעד כ-60% מזו של האוסף הבו-זמני של Jikes. כמצופה, האלגוריתם המקבילי (העובד בתצורת עצור-את-העולם) משיג תפוקה גבוהה יותר מהאלגוריתם שלנו (העובד תוך-כדי-מעוף) בעד-10%.

   בנוסף על היותו אלגוריתם סריקת ערימה (תוך-כדי-מעוף) העומד בפני עצמו, אוסף הזבל המוצע יכול לשמש כאוסף הגיבוי (האוסף מעגלי זבל) עבור האוסף של לבנוני ופטרנק העובד בשיטת ספירת מצביעים. שני האלגוריתמים הנ"ל מתאימים לחלוטין כיוון שהם משתמשים באותו מקצה זיכרון, אותם מבני נתונים וממשק מכונת ג'אווה וירטואלית זהה.

2. **אוספי זבל מוכווני-גילאים**

   אוספי זבל מבוססי דורות[3] ידועים ככאלה המשפרים את יעילות איסוף הזבל ומקצרים את זמן ההשהיה שכופה אוסף הזבל. בעבודה זו חקרנו מהי הדרך היעילה ביותר לשלב דורות עם אוספי זבל העובדים בשיטת ספירת מצביעים, בדגש על איסוף זבל בו-זמני. אנו מציעים גישה חדשה לאיסוף זבל, המכונה איסוף זבל מוכוון-גילאים, המנצלת את היפותזת הדורות החלשה לשם השגת שיפור ביעילות האיסוף. גישה זו שימושית במיוחד כאשר משתמשים באלגוריתם ספירת המצביעים על מנת לאסוף את דור הוותיקים. אוסף הזבל מוכוון-הגילאים הבו-זמני שהתקבל הינו יעיל במיוחד. המדידות שבצענו הראו שאוסף זבל זה עולה בביצועיו על אוסף זבל העובד בשיטת ספירת מצביעים על כל הערימה ועל אוסף זבל מבוסס דורות המשתמש באלגוריתם ספירת המצביעים לאיסוף את דור הוותיקים. אנו מסכמים

---

[3] Generational collectors באנגלית.

בהמלצה על אוסף הזבל מוכוון-הגילאים כדרך היעילה ביותר לאיסוף זבל בעזרת אלגוריתם העובד בשיטת ספירת מצביעים.

3. **איסוף מעגלי זבל בו-זמני בשיטת ספירת מצביעים**

אוסף זבל העובד בשיטת ספירת מצביעים אינו מסוגל לאסוף רכיבים מעגליים שאינם ישיגים משורשי התוכנית. לכן, אוסף זבל העובד בשיטת ספירת מצביעים מפעיל לעיתים רחוקות אלגוריתם סריקת ערימה (המשמש כגיבוי) או משתמש באוסף מעגלים ייעודי לאיסוף מבנים מעגליים. אנו מציעים אוסף מעגלים בו-זמני חדש.

אוסף המעגלים החדש משלב את אוסף המעגלים הסינכרוני של בייקון ורג'ן עם אלגוריתם המבטים המחליקים של לבנוני ופטרנק. שילוב זה משיג שני יתרונות. היתרון הראשון הוא ששילוב זה מפחית בצורה ניכרת את מספר העצמים המועמדים להיות חלק ממעגל זבל. לכן, העבודה הנדרשת לאיסוף מעגלי הזבל פוחתת אף היא ויעילות אוסף המעגלים משתפרת בצורה משמעותית. שנית, השילוב הנ"ל פותר את בעיית הסיום התאורתית של אוסף המעגלים הבו-זמני של בייקון ורג'ן, בו תנאי תחרות עלולים לדחות את איסופו של מעגל זבל לנצח. האלגוריתם החדש, הפועל על המבטים המחליקים, מבטיח שכל מבנה מעגלי, שאינו ישיג, ייאסף.

מימשנו את האלגוריתם הנ"ל עבור מכונת הג'אווה הוירטואלית המחקרית Jikes. בין יתר המדידות שבצענו, השווינו בין אלגוריתם סריקת ערימה המופעל כגיבוי (לאוסף זבל העובד בשיטת ספירת מצביעים) ובין אוסף המעגלים שבנינו. למיטב ידיעתנו, השוואה כזו, בין אלגוריתם סריקת ערימה המופעל כגיבוי ובין אוסף מעגלים, לא בוצעה (דווחה) בעבר.

4. **ספירת מצביעים בעזרת הבאה מראש**[4]

הפער בין מהירות הבאת נתונים מהזיכרון ובין מהירות המעבדים הולך וגדל. עקב כך, גישות לזיכרון מהוות פעמים רבות צוואר בקבוק של ריצת תוכניות. על מנת להפחית את הפער הנ"ל, משתמשים בהיררכיות של זיכרון מטמון, אך עדיין אפליקציות רבות נוטות לסבול מהשהיות לא זניחות הנגרמות מהחטאות בזיכרון המטמון. הבאת מידע מראש (prefetching) מסוגלת להפחית גם את ההשהיות הללו. בעבודה זו אנו מזהים מצבים בהם ניתן לחזות מראש את הגישות הבאות למידע, שיבצע אוסף זבל העובד בשיטה עדכנית של ספירת מצביעים. לכן, הכנסנו הוראות הבאה מראש לאוסף העובד בשיטת ספירת מצביעים של מכונת הג'אווה הוירטואלית המחקרית Jikes. מסתבר ששיטת ספירת המצביעים פחות רגישה להוראות הבאה מראש מאשר אוסף זבל העובד בסריקת ערימה. בעוד שאוסף זבל העובד בסריקת ערימה ניגש לכל עצם פעם אחת, אוסף העובד בשיטת ספירת מצביעים ניגש לעצמים שוב ושוב. למרות זאת, הוראות ההבאה מראש הפחיתו בממוצע כ-8.7% מתקורת ניהול הזיכרון וגרמו להאצת מהירות תוכניות הבדיקה ב-2.2% בממוצע.

5. **תבניות לשימוש יעיל במנהל הזיכרון**

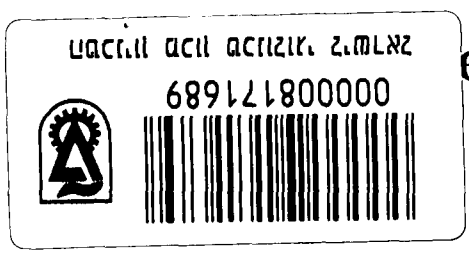
בעבודה זו בחנו כיצד לזהות שימוש לא יעיל במנהל הזיכרון והצענו דרכים לשימוש מושכל יותר במנהל הזיכרון. תחילה, הצענו מערכת אפקטיבית המאתרת אתרי הקצאות לא יעילים. לאחר מכן בחנו שלוש תבניות לשיפור יעילות במקרים הנ"ל, כאשר הדבר מתאפשר: עיעה"מ (עצם יחיד עבור הקצאות מרובות), מאגר עצמים ומאגר עצמים מורכב. מאגר עצמים מורכב הינו וריאציה מעניינת של מאגר עצמים סטנדרטי, המשפר יעילות בצורה ניכרת כאשר ניתן להשתמש בו. מצאנו שתוכניות סטנדרטיות כוללות פעמים רבות קטעי קוד המשתמשים באופן לא יעיל במנהל הזיכרון. קוד שכזה מאותר בקלות ע"י המערכת שבנינו וניתן לשנותו בעזרת התבניות שהצענו על מנת לגרום לתוכנית לרוץ מהר יותר. מימוש התבניות שהצענו, על מספר תוכניות בדיקה ומספר מכונות ג'אווה וירטואליות המשתמשות באוספי זבל שונים, גרם לשיפור של עד 22.8% בזמן הריצה של התוכנית כולה. בפרט, יעילות התבנית מאגר עצמים מורכב באה לידי ביטוי בתוכנית SPECjbb2000, בה השימוש בתבנית מאגר עצמים סטנדרטי לא השיג שום שיפור בביצועים, בעוד השימוש בתבנית מאגר עצמים מורכב השיג שיפור של 3.0-9.3%.

---

[4] Prefetch באנגלית.

# Efficient Memory Management for Servers

el Paz



# Efficient Memory Management for Servers

Research Thesis

Submitted in Partial Fulfillment of the

Requirements for the

Degree of Doctor of Philosophy



**Harel Paz**



Submitted to the Senate of

the Technion — Israel Institute of Technology

Elul, 5766 Haifa August 2006

The research thesis was done under the supervision of Dr. Erez Petrank
and the consultation of Dr. Hillel Kolodner
in the department of Computer Science.

First and foremost, I would like to express my most deep appreciation feelings to my supervisor, Dr. Erez Petrank. Erez, in his gentle way, has exposed me to the academic research world and shared his vast knowledge with me. Erez was always available and accessible, willing to assist, although having many other obligations. It was both a great pleasure and a privilege to work with Erez. It is sad to say good-bye, but I hope that our paths would cross in the future.

I would like to thank my advisor, Dr. Hillel Kolodner, for sharing his wisdom in the long discussions we had.

During the work on this thesis, I have been assisted by many, besides Erez and Hillel, and this is an appropriate place to express my gratitude for their help. Hezi Azatchi helped me getting into business in the early stages of my graduate studies. In addition, I have remarkably benefited from many helpful and fruitful discussions I had with Yoav Ossia, Prof. Mooly Sagiv, Dr. Ran Shaham, Haim Kermany, and Uri Silbershtein. I would like to thank Prof. Hagit Attiya for her useful comments on this manuscript that greatly improved its quality. I am also grateful to Eran Issler and Maxim Kovgan from the faculty's distributed systems laboratory for long years of technical assistance.

In addition to the help of the above, I would also like to thank the followings for their moral support. During my graduate studies, I had the pleasure of sharing an office with Roie Melamed. Daily interaction with Roie helped me overcome bad moments. Last but not least, I thank my family for the support, patience and love throughout my studies.

I would like to thank the Miriam and Aaron Gutwirth Memorial Fellowship for their financial help.

The generous financial help of the Technion is gratefully acknowledged.

# Contents

# List of Figures

# List of Tables

# Abstract

Modern SMP servers with large heaps provide new challenges for the design of suitable garbage collectors. Garbage collectors designed for client machines usually work in the so called stop-the-world (STW) manner. A STW collector stops all program threads while executing. Naive collectors employ only one of the available CPUs on an SMP. Naive STW collectors may lead to inefficient running times on servers, long pause times and poor CPU utilization. These problems can be solved by either executing the collector in a separate thread (or process) concurrently with the program threads, or by parallelizing the execution of the collection.

We have designed and implemented several alternative collectors for server platforms. These collectors attempt to improve the garbage collection process in SMP platforms. In particular, we have focused on concurrent collectors and reference-counting collectors.

We started by designing a mark-and-sweep on-the-fly algorithm based on the sliding-views mechanism of Levanoni and Petrank. This algorithm can also be used to infrequently collect garbage cycles in the reference-counting sliding-views collector. Next, we introduced the *age-oriented* collector, which exploits the generational hypothesis best when used with reference counting, to obtain highly efficient collection. In our third project, we proposed a new on-the-fly cycle collection algorithm to accompany the reference-counting sliding-views collector. In the fourth, we have inserted prefetch instructions into the reference-counting collector in order to hide (or decrease) cache miss stalls, and hence reduce the overhead imposed by a reference-counting collector. Finally, we studied ways to identify and eliminate wasteful use of the memory management subsystem by employing three patterns, when possible.

# Notations and Abbreviations

| | | |
|---|---|---|
| AO | - | Age Oriented |
| GC | - | Garbage Collection |
| JVM | - | Java Virtual Machine |
| MS | - | Mark and Sweep |
| RC | - | Reference Counting |
| RVM | - | Research Virtual Machine |
| SMP | - | Symmetric Multi-Processor |
| STW | - | Stop the World |
| SVMS | - | Sliding-Views Mark-and-Sweep |
| SVRC | - | Sliding-Views Reference-Counting |
| TIB | - | Type Information Block |
| ZCT | - | Zero Counts Table |

# A Glossary [1]

**Copying garbage collection**. A tracing garbage collection method that operates by relocating reachable objects and then reclaiming objects that are left behind, which must be unreachable and therefore dead.

**Floating garbage**. Garbage objects which are not being collected promptly after turning garbage. This subjective term usually denotes garbage that is generated during a collection cycle or garbage that a collection fails to reclaim even though eligible for collection.

**Garbage; garbage objects; dead objects**. Objects that are not reachable.

**Generational garbage collection**. A garbage collection method that makes use of the generational hypothesis that "most objects die young". Objects are gathered together in generations (set of objects of similar age). New objects are allocated in the youngest generation, and promoted to older generations if they survive long enough. Objects in older generations are collected less frequently, saving CPU time.

**Incremental garbage collection**. Garbage collection methods which perform garbage collection work in parts and not continuously (i.e., the collector may pause in the middle of a collection cycle while mutators continue). Hence, the collector work is done incrementally.

**Mark and sweep garbage collection**. A tracing garbage collection method that recursively traces and marks objects starting from the roots, then frees all non-marked objects.

**Mutators, mutator threads**. Threads that perform application work.

**Pause time, maximal**. The maximal duration a mutator has ever been stopped for by the collector.

**Reachable.** An object is reachable if a root refers to it, or another reachable object has a slot referencing it.

**Reference-counting collection**. A garbage collection method that determines reachability by counting the number of references to each object.

[1] Taken from [88, 63].

**Roots**. A collection of variables that may contain references which are immediately accessible to at least one mutator. Typical roots are the control stack, global variables, other static data, registers, intern tables, etc.

**Stop-the-world garbage collection**. Garbage collection methods in which the collector and mutators seldom operate concurrently. These methods are usually characterized by long interruptions to user processing, yet are simple to implement and often exert lower overheads for the collection.

**Throughput**. For a given benchmark, the number of operations completed per time unit.

**Tracing garbage collection**. A garbage collection method based on the fact that if an object is not reachable, mutators could never access it, and therefore it cannot be alive. In a collection cycle, a part or the entire objects graph is traced to find which objects are reachable. Those that were not reachable may be reclaimed.

**Write barrier**. Operations performed when a mutator stores an object reference from its local state into an object slot, if such operations are required.

# Chapter 1

# Introduction

## 1.1 Garbage collection

Garbage collection (GC) is the automatic recycling of dynamically allocated memory. It automatically determines what memory a program is no longer using, and recycles it for other use. Manual memory management is (programmer-) time consuming and error prone. GC avoids the need for the programmer to deallocate memory blocks explicitly, thus avoiding (or ameliorating) the following problems. First, the problem of premature frees where memory is released too early (while it is still reachable), causing a problem when this memory is accessed later. Memory leak is another (manual memory management) problem in which allocated memory is not released when no longer needed. This problem may cause shortage of memory in further stages of the application. An additional problem is erroneous attempts to release memory chunks that were already freed (double free problem).

By using GC, the burden on the programmer is reduced by not having to investigate such problems, thereby increasing productivity. GC can also dramatically simplify programs, chiefly by allowing modules to present cleaner interfaces to each other: the management of object storage between modules is unnecessary. Additionally, GC can reduce the amount of memory used because the interface problems of manual memory management are often solved by creating extra copies of data.

## 1.2 Servers

One of the urging needs that the Internet age has necessitated is the one for high-end, scalable, responsive, portable and rapidly developable servers. Nowadays, virtually every firm has to deploy Internet servers capable of serving anytime, any number–from a handful to millions–of customers and supply them with a prompt response, regardless of the current load the server has to handle. Thus, servers are required to be scalable and in particular, the response time of the server should be as independent as possible from the load on the server. In addition, due to rapidly emerging technologies and the users' demand for constantly increasing functionality, servers should be programmed using a portable language that allows code reuse and rapid development. In view of these factors, many server suppliers choose symmetric multiprocessor as their hardware platform and Java as their software platform.

A symmetric multiprocessor is a computer system possessing multiple CPUs and a shared memory. This allows maintaining performance as the number of users grows (up to a certain bound), by adding more processors to the system. This solution is usually cost-effective and it requires no alteration of existing code.

Java is a shared-memory, multi-threaded, portable, object-oriented and garbage collected language. By virtue of being a shared-memory and multi-threaded, Java lets the programmer exploit the resources of the symmetric multiprocessor efficiently. Multi-threading also simplifies the design of servers that have to handle independent requests and it improves response latency. Being portable, object-oriented and garbage-collected, Java facilitates the development process, shortens it and guarantees a more reliable result.

The implementation of the garbage collection, however, must be designed specifically for servers with short response and service times, which are the ultimate goals of server scalability.

## 1.3 Garbage collection for servers

The general problem of garbage collection has drawn attention virtually from the first days of the general-purpose computer. However, most techniques were developed for uniprocessor client memory-constrained systems that executed non time-critical missions, such as text editing. The dominant characteristic of such garbage collection techniques was the favoring of space over speed. In particular, it is acceptable in such systems to suspend all execution threads while a centralized garbage collection process is compacting the heap. Since the

number of objects and threads is small relative to server applications, the pause is tolerable, in most cases, by the end user.

Modern SMP servers with large heaps provide new challenges for the design of suitable garbage collectors. Garbage collectors designed for client machines may lead to inefficient running times on servers and non-incremental collectors may lead to unacceptable pauses. A server application running on a symmetric multiprocessor should preferably never come to a complete halt due to garbage collection. Since the collector is usually single-threaded, when all user threads are stopped for garbage collection only a single processor is utilized, compromising the effectiveness of the multiprocessor system. In addition, as the number of threads increases, the time required to stop all threads simultaneously is increasing and the system becomes less responsive and non-scalable. This is especially the case when a thread can be stopped only at special safe-points in the code it is executing.

Thus, garbage collectors for server systems should be concurrent meaning that the garbage collector is executing concurrently with the user threads (also called mutator threads). It is preferable that synchronization between the garbage collector thread and the mutators be kept as low as possible. Concurrent garbage collectors are often extensions of sequential garbage collector algorithms. As such, they can be roughly categorized as concurrent mark and sweep (for example [32, 33]), concurrent copying collectors (see [17, 76, 52]) and concurrent reference counting (see [30]).

An *on-the-fly garbage collector* is a delicate concurrent collector that does not stop the program threads simultaneously to perform the collection, i.e., there is never more than a single mutator stopped (while a concurrent collector might halt program threads simultaneously for a short while). Instead, the collector executes on a separate thread (or process) concurrently with the mutator threads. On-the-fly collectors are useful for multithreaded applications running on multiprocessor servers, where it is important to fully utilize all processors and provide even response time, especially for systems in which stopping the threads is a costly operation. By avoiding the need of stopping all mutator threads simultaneously, on-the-fly collectors provide extremely short pauses.

### 1.3.1 Parallel garbage collection

A parallel collector could also be a suitable choice for collecting garbage in server systems. A parallel collector is a collector which executes the collection work in parallel by more than one collector thread, and hence achieves full CPUs utilization. Several parallel collectors have already been suggested [40, 57, 39, 75].

The advantage of a concurrent collector over a parallel collector is that a concurrent collector stops all the program threads only for a short period (to apply a short cooperation with the collector). A parallel collector executes the entire collection while the program threads are halted, thus incurring a longer pause time of 1-2 orders of magnitude. However, the scalability of concurrent collectors is limited: a single collector thread may not be able to supply enough free memory needed by mutators' allocations. If the collector falls behind the space requests of the mutators, the mutators would halt waiting for a collection cycle to terminate (and supply free space). In these cases the superiority of a parallel collector (over a concurrent collector) is expressed: the parallel collector always exploits all CPUs, while a program employing a concurrent collector uses only one until free space is supplied. Another advantage of the parallel collector is that it is invoked less frequently than a concurrent collector (and thus causes less overhead). A parallel collection should only be invoked when we ran out of memory, while a concurrent collection should be invoked sooner, as it tries to avoid an out of memory situation.

In order to gain both a better throughput and a short response time, a collector could be designed to be both parallel and concurrent (such as in [75]). In this work, however, we focus solely on concurrent garbage collectors.

### 1.3.2 An example

Figure 1.1 illustrates the difference between four different collector kinds: stop-the-world, parallel, concurrent and on-the-fly. The basic scenario refers to a symmetric multiprocessor with four processors. Collector threads are colored yellow and mutator threads are colored blue. All four scenarios are initiated similarly, where mutator threads run over all four processors. The difference between the collectors is introduced when a garbage collection cycle is invoked.

The stop-the-world garbage collector, in our example, is a naive stop-the-world collector, which stops all the mutator threads during a collection and operates over a single CPU. This collector inefficiently employs the hardware, as all but one CPU are left idle during the collection.

The parallel garbage collector introduced in Figure 1.1, is also a stop-the-world one, as mutators are halted during garbage collection work. However, garbage collection, in this case, is done simultaneously by multiple collector threads over all processors. Hence, it introduces a parallel stop-the-world collector. Since the collection work in done in parallel (and the four processors are exploited) the mutators are paused (interrupted) for a shorter

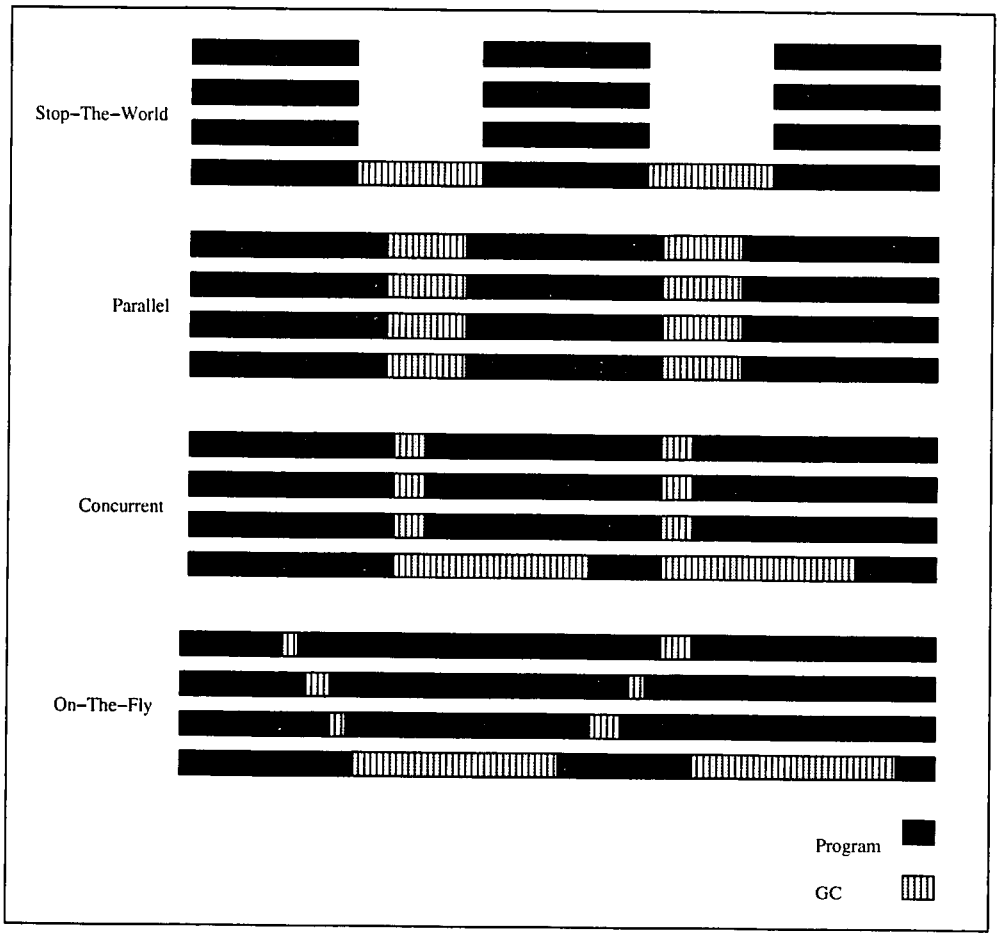



Figure 1.1: Example of a stop-the-world collector, a parallel collector, a concurrent collector and an on-the-fly collector.

period than with the naive stop-the-world. In addition, all processors are kept active during the application run.

The concurrent collector presented in the example initiates a collection cycle by simultaneously halting all mutators for a short period (for some synchronization). Then, it proceeds by resuming the mutator threads on three CPUs and running a dedicated collector thread on the fourth CPU. Hence, the collector thread runs concurrently with the mutator threads. When the collection cycle terminates, a mutator is resumed also on the fourth processor. With the concurrent collector, mutators are halted simultaneously only for a short period (at the beginning of the collection), yielding shorter pause times than the parallel garbage collector does.

Like the concurrent collector, the on-the-fly collector presented in Figure 1.1, works on the fourth processor concurrently with the program threads. However, instead of simultaneously halting all mutators threads in the collection beginning, each mutator is halted on its own

paste, and only then a collector thread is executed on the fourth processor. This yields an even shorter pause time (than the concurrent collector yields), as mutators are resumed after a short synchronization with the collector, independently of other mutators.

## 1.4 Research motto

Our work heavily relates to the field of reference-counting algorithms for server systems. Reference counting is one of the most intuitive methods for automatic storage reclamation. As such, systems using reference counting were implemented starting from the sixties [26]. The main idea is that we keep for each object a count of the number of references to the object. When this number becomes zero for an object $o$, we know that $o$ can be reclaimed.

Traditional reference-counting algorithms suffer from several flaws. First, when using a reference-counting collector, pointer manipulations impose a substantial computational overhead (in order to maintain the reference count of the involved objects). In addition, reference-counting collectors suffer from costly parallelism: the transformation of sequential reference counting into concurrent reference counting need cope with keeping the reference counting invariant (the number of references to an object equals the value of the reference count field of the object) in environments were arithmetic manipulation of memory locations is not atomic. Another major drawback is reference-counting inability to reclaim cyclic data-structures (strongly connected components in the objects graph).

These drawbacks have directed researchers and commercial systems to focus on tracing collectors. Hence, historically, the study of reference counting (and in particular concurrent reference counting) has not been as extensive and thorough as the study of tracing collectors, forming an approximate two decades research gap. Recent studies of reference-counting algorithms on modern platforms [82, 6, 62, 63] were able to mask some of these drawbacks. In particular, Levanoni and Petrank [62, 63] have drastically reduced the work required by a reference-counting collector and eliminated much of the parallelism overhead (by completely eliminating the need for synchronization operations in the write barrier).

In addition to these recent studies, reference counting seems appealing for future garbage collected systems, where very large heaps are used. Tracing collectors must traverse all live objects, and thus, the larger the usage of the heap (i.e., the amount of live objects in the heap), the more work the collector must perform. The amount of reference-counting work, however, is proportional to the amount of work done by the user program between collections plus the amount of space that is actually reclaimed. But it does not depend on the space

consumed by live objects in the heap.

In order to reduce reference-counting research gap and since reference counting seems appealing for future platforms, our central motto in this work was to try to advance research on reference counting to close the said gap. In particular, we focus on improving reference-counting collectors on SMP machines, by designing and implementing new and better algorithms for memory management on servers. During this work, we have focused on the following projects:

- Basing a mark-and-sweep on-the-fly algorithm on the sliding-views mechanism of Levanoni and Petrank [61, 63]. This algorithm can also be used to infrequently collect garbage cycles in the reference-counting sliding-views collector.
- Designing and implementing the *age-oriented* collector, which exploits the generational hypothesis (that "most objects die young") together with reference counting, to obtain highly efficient collection. We believe that the *age-oriented* collector forms the optimal manner of reference-counting use.
- Designing and implementing a new on-the-fly cycle collection algorithm. The new algorithm improves the efficiency and theoretical properties of concurrent cycle collection.
- Inserting prefetch instructions into the reference-counting collector in order to hide (or decrease) cache miss stalls, and hence reduce the overhead imposed by a reference-counting collector.
- Studying ways to identify and eliminate wasteful use of the memory management subsystem by employing three patterns, when possible: *SOfMA (single object for multiple allocations)*, *object pooling* and *compound object pooling*.

Next, we elaborate on each one of these projects.

## 1.5 An on-the-fly tracing collector

We propose a novel mark-and-sweep on-the-fly algorithm based on the sliding-views mechanism of Levanoni and Petrank (which was originally proposed for a reference-counting collector). We have implemented our collector on Jikes RVM running on a Netfinity multiprocessor and compared it to the concurrent algorithm and to the parallel stop-the-world

collector supplied with Jikes. The maximum pause time that we measured with our benchmarks over all runs was 2ms. In all runs, the pause times were smaller than these of the parallel stop-the-world collector by two orders of magnitude and they were also always shorter than the pauses of Jikes concurrent collector. Throughput measurements of the new garbage collector show that it outperforms Jikes' concurrent collector by up to 60%. As expected, the stop-the-world collector does better than the on-the-fly collectors with results showing about 10% difference.

On top of being an effective mark-and-sweep on-the-fly collector standing on its own, our collector may also be used as a backup collector (collecting cyclic data structures) for the Levanoni-Petrank reference-counting collector. These two algorithms perfectly fit sharing the same allocator, a similar data structure, and a similar JVM interface.

## 1.6 Age-oriented collectors

Generational collectors are well known as a tool for shortening pause times incurred by garbage collection and for improving garbage collection efficiency. In this work, we investigate how to best use generations with reference-counting collectors with an emphasis on concurrent collection. We propose a new collection approach, denoted *age-oriented* collection, for exploiting the generational hypothesis to obtain better efficiency. This approach is particularly useful when reference counting is used to collect the old generation. The resulting concurrent collector is highly efficient and non-obtrusive. Finally, an implementation is provided demonstrating how the age-oriented collector outperforms both the non-generational and the generational collectors' efficiency. We conclude by advocating the age-oriented approach as the best known state-of-the-art method to obtain the highest efficiency with a reference-counting collector.

## 1.7 Concurrent cycle collection in reference-counting collectors

A reference-counting garbage collector cannot reclaim unreachable cyclic structures of objects. Therefore, reference-counting collectors either use a backup tracing collector infrequently, or employ a cycle collector to reclaim cyclic structures. We propose a new *concurrent* cycle collector, i.e., one that runs concurrently with the program threads, imposing

negligible pauses (of around 1ms) on a multiprocessor.

Our new collector combines the state-of-the-art cycle collector [7] with the sliding-views collectors [62, 63, 4]. The use of sliding views for cycle collection yields two advantages. First, it drastically reduces the *number* of cycle candidates, which in turn, drastically reduces the *work* required to record and trace these candidates. Therefore, a large improvement in cycle collection efficiency is obtained. Second, it eliminates the theoretical termination problem that appeared in the previous concurrent cycle collector. There, a rare race may delay the reclamation of an unreachable cyclic structure forever. The sliding-views cycle collector guarantees reclamation of all unreachable cyclic structures.

The proposed collector was implemented on the Jikes RVM and we provide measurements including a comparison between the use of backup tracing and the use of cycle collection with reference counting. To the best of our knowledge, such a comparison has not been reported before.

## 1.8 Reference Counting using Prefetch

The performance gap between memory latency and processors' speed is increasing, making memory accesses a major performance bottleneck. Although cache hierarchies are used to reduce this gap, applications still tend to suffer considerable memory stall time due to cache misses. In this work we identify opportunities to prefetch predictable data accesses in advance in a modern reference-counting garbage collector. The proposed prefetch instructions were inserted into the Jikes reference-counting collector. Interestingly, reference counting turns out less susceptible to prefetching improvements than tracing collectors. Whereas tracing collectors touch each object once, reference-counting collectors touch objects repeatedly. Nevertheless, the inserted prefetch instructions reduce, on average, an 8.7% of the memory management overheads yielding a 2.2% overall application speedup.

## 1.9 Object pooling

In this project, we study ways to identify and eliminate wasteful use of the memory management subsystem. First, an effective profiler is proposed to spot inefficient use of allocations. Then, three patterns are studied to improve efficiency for these cases, when possible: *SOfMA (single object for multiple allocations)*, *object pooling* and *compound object pooling*. The latter

is an interesting tweak on standard object pooling, which improves performance significantly, when applicable. It turns out that various standard benchmarks include code that causes such extremely inefficient use of the memory manager. This code can be easily identified by our profiler and modified via the patterns that we propose to make the program run faster. Measuring the proposed modifications on various standard benchmarks with a couple of Java Virtual Machines (JVMs) and using various (standard) garbage collection algorithms, an improved application performance of up to 22.8% is achieved. The importance of the novel *compound object pooling* pattern, in particular, is highlighted by the SPECjbb2000 benchmark where naive object pooling yields no performance improvement. Nevertheless, *compound object pooling* is applicable and yields an overall throughput improvement of 3.0-9.3%.

## 1.10 Organization

The rest of the thesis is organized as follows. Section 2 presents related work on on-the-fly collectors, tracing collectors, generational collectors, reference-counting collectors and in particular the sliding-view reference-counting collector. Section 3 presents the mark-and-sweep sliding-views collector. Section 4 introduces the age-oriented collector. Our concurrent cycle collection algorithm is discussed in Section 5. Section 6 exhibits how using prefetch instructions can improve the efficiency of a reference-counting collector. In Section 7, we study ways to identify and eliminate wasteful use of the memory management subsystem by employing patterns.

# Chapter 2

# Related Work

## 2.1 On-the-fly collectors

Modern SMP servers with large heaps provide new challenges for the design of suitable garbage collectors, the so-called *stop the world* setting, that work while program threads are stopped. On multiprocessor platforms, it is not desirable to stop the program and perform the collection in a single thread on one processor, as this leads both to long pause times and poor processor utilization. A *concurrent collector* is a collector which runs concurrently with the program threads. The program threads may be stopped for a short time to initiate and/or finish the collection. An *on-the-fly collector* is a concurrent collector that does not need to stop the program threads simultaneously, not even for the initialization or the completion of the collection cycle. Hence on-the-fly collectors provide extremely short response time and are especially useful for systems in which stopping the threads is a costly operation.

The study of on-the-fly garbage collectors was initiated by Steele [91, 92] and Dijkstra, et al. [32] and continued in a series of papers [44, 10, 11, 59, 60, 34, 33, 62, 63, 35, 36, 6, 52, 4]. The advantage of an on-the-fly collector over a parallel collector and other types of concurrent collectors [8, 37, 71, 17, 38, 39, 75, 86, 58, 56], is that it avoids the operation of stopping all the program threads[1]. Such an operation can be costly, and it usually increases the pause times. Today, on-the-fly collectors achieve pauses as short as a couple of milliseconds, and even less [52]. On-the-fly collectors were mostly based on the mark-and-sweep algorithm, yet, an on-the-fly copying collector has appeared in [52] and on-the-fly reference-counting

[1]The collector of Nilsen et al. [72] incrementally scans the stack contents of each thread, but involves other overheads.

collectors were proposed in [6, 62, 63].

## 2.2 Reference counting

The traditional method of reference counting applicable in the realm of uniprocessing was first developed for Lisp by Collins [26]. In its simplest form, it allowed immediate reclamation of garbage in a localized manner, yet with a notable overhead for maintaining the space and semantics of the counters. Weizenbaum [100] showed how the delay introduced by recursive deletion (which is the only non-constant delay caused by classic reference counting) can be ameliorated for fixed sized objects, by distributing deletion over object creation operations. Deutsch and Bobrow [31] eliminated most of the computational overhead required to adjust reference counters in their method of deferred reference counting. According to the method, local references are not counted and thus it is unnecessary to track fetches, local pointer duplications and cancellations. Only stores into the heap need be tracked. However, the immediacy of reference counting is lost in a certain extent, since garbage may be reclaimed only after the mutator state is scanned and accounted for. Bacon et al. [6] and Levanoni and Petrank [62, 63] have extended the reference-counting algorithm to run concurrently with the program threads. Both achieve extremely low pause times (of around 2ms). The on-the-fly algorithm of Bacon et al. [6] employs a novel on-the-fly cycle detector, avoids the need of Deutsch and Bobrow's zero count tables, and shows how pointer updates could be done using a single compare-and-swap (instead of a central lock over all pointer updates). The on-the-fly algorithm of Levanoni and Petrank [62, 63] is described in Section 2.6.

## 2.3 Generational garbage collectors

Previous researchers have gathered considerable evidence to support the weak generational hypothesis that "most objects die young". The insight behind generational garbage collection is that storage reclamation can be made more efficient and less obtrusive by concentrating effort on reclaiming those objects most likely to be garbage, i.e., young objects. The generational strategy is to segregate objects by age into two or more regions of the heap called generations. Different generations can then be collected at different frequencies, with the youngest generation being collected frequently and older generations much less often.

Generational garbage collection was introduced by Lieberman and Hewitt [64], and the first published implementation was by Ungar [97]. Both algorithms were aimed to reduce

the running time of most collections by focusing on the young objects. Appel [2] presented a two-generation collector with variable young generation size: all its free space is devoted to the young generation. When the young generation becomes full (and thus both generations consume all usable memory), it collects the young generation, copying surviving objects to the older generation, and reducing the young generation size by this space. Major collections are executed only when the old generation occupies the entire heap. Demers, et al. [28] presented an algorithm for using generational collector without moving the objects. Their motivation was to adapt generations for conservative garbage collection. Instead of partitioning the heap physically and keeping the young objects in a separate place, it is suggested to partition the heap logically, and to keep a bit per object, indicating whether the object is young or old. Other interesting variants of generational collection include the Train algorithm [51], the older first collector [94], and Beltways [13].

Incorporations of generational collectors into on-the-fly collectors were done by Domani et al. [35], and by Azatchi and Petrank [5][2]. Both works employed fixed-sized young generation and both showed that combining generations with on-the-fly collectors may be useful. Domani, Kolodner and Petrank adopt the idea of Demers, et al. [28] to partition the young and the old objects logically (as objects cannot be moved). The Doligez-Leroy-Gonthier mark-and-sweep collector [34, 33] is used both for the collection of the young generation and the collection of the full heap. Azatchi and Petrank [5] showed how generational collection could interact with reference counting, by using reference counting for the old generation and mark and sweep for the young generation. They built on the reference-counting collector of Levanoni and Petrank [62] and on the mark-and-sweep collector of Azatchi et al. [4].

Blackburn and McKinley [14] implemented a uniprocessor stop-the-world generational collector with reference counting for the old generation and copying for the young. Using copying for the young generation is a natural choice for stop-the-world collection, as it makes better use of the high death rate (of objects belonging to the young generation). Adopting copying for an on-the-fly collector is rather difficult (the only known such construction appears in [52]). The goal in [14] was to use the generational collector to shorten the pauses a stop-the-world reference counting may incur, while obtaining good throughput. They used a clever mechanism to run part of the old generation collection together with the young collection in order to avoid the need for a full collection that requires a long pause. Indeed, their

[2] A partial incorporation of generations with an on-the-fly collector was used by Doligez, Leroy, and Gonthier [34, 33]. The whole scheme depends on the fact that many objects in ML are immutable. This is not true for Java and other imperative languages. Furthermore, the collection of the young generation is not concurrent.

collector demonstrated controlled pause times with good throughput. These pause times are larger than those obtained by on-the-fly collectors.

## 2.4 Reclaiming garbage cycles

Cyclic data structures (strongly connected components in the objects graph) are common, both in the application level and at the operating system level. The inability to reclaim garbage cyclic data structures is one of the severest disadvantage of reference counting. This inability was first noticed by McBeth [69]. Christopher [25] developed an algorithm whose primary method is reference counting, yet a tracing collector is called periodically to reclaim nodes in the heap that have a non-zero reference counts but are not externally reachable. The algorithm of Martinez et al. [68] reclaims cells, which were uniquely referenced when their count drops to zero, while when a pointer to a shared object is deleted, a local depth-first search is applied on it. This search subtracts reference counts due to internal pointers. If a collection of objects with zero reference counts is found, then a garbage cycle has been found, and is collected. Lins [65] extended this algorithm by postponing the above traversals while saving the values of the deleted pointer in a buffer (each such value is a candidate to be a root of a garbage cycle), and traverse the buffer at a suitable point. Delaying the traversal will decrease the number of over all candidates traversals since most buffer's values will be irrelevant by the time the buffer would be traversed (because their reference count would drop to zero or would be incremented). Bacon et al. [7] extended Lins algorithm to a concurrent cycle collection algorithm. They also improved Lins' algorithm by performing the tracing of all candidates and only then checking whether there are garbage cycles, instead of trace and check each candidate individually. However, due to race conditions there is no bound on the collection time of a garbage cycle, i.e., the algorithm does not guarantee that a garbage cycle would be collected until the i-th collection (from the time it became garbage). Lins [66] showed the algorithm can employ two graph traversals (instead of three) per candidate by using an extra data structure.

## 2.5 Mark and sweep

The mark-and-sweep garbage collector was first presented by [70]. A mark-and-sweep collector first marks any object which is directly reachable (either from a local or a global reference) and then recursively marks any object which is pointed to by a marked object.

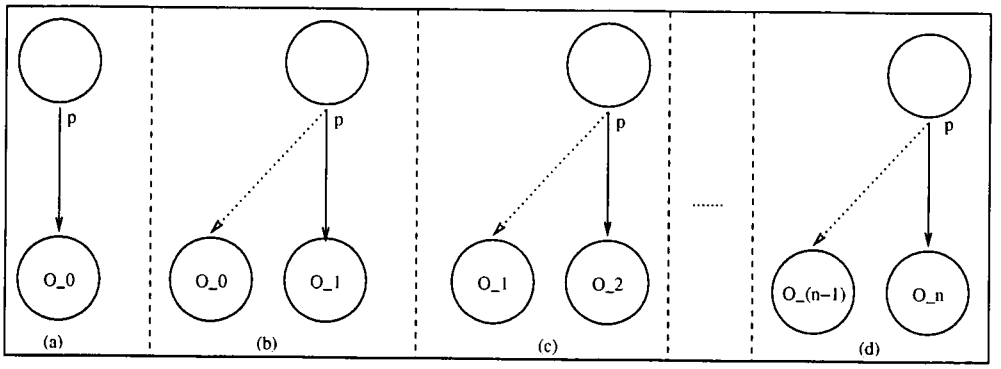



Figure 2.1: Example of reference-count processing savings when using the Levanoni-Petrank collector

Then, any object, which is not marked is *swept*, i.e., reclaimed. Much research and engineering effort has been put into this algorithm since. Mark-and-sweep algorithms that perform garbage collection using a snapshot of the heap appear in [103, 40]. Many mark-and-sweep concurrent collectors have been proposed, such as [17, 86, 75].

The study of mark-and-sweep on-the-fly garbage collectors was as initiated by Steele [91, 92] and Dijkstra., et al. [32] and continued in a series of papers [60, 59, 44, 10, 11] culminating in the Doligez-Leroy-Gonthier (DLG) collector [34, 33]. The DLG collector is considered the most advanced on-the-fly mark-and-sweep collector. It uses fine-grained synchronization, and it was used in a production JVM of IBM (see [35, 36]).

## 2.6 The sliding-views reference-counting collector

Some of our projects heavily rely on the Levanoni-Petrank sliding-views reference-counting collector [62, 63]. Hence, for completeness, we provide a review of the sliding-views reference-counting collector.

The Levanoni-Petrank collector [63] is an on-the-fly reference-counting collector that eliminates many of the reference-count updates by the following coalescing strategy. Consider a pointer slot $p$ that is assigned the values $o_0, o_1, o_2, \ldots, o_n$ between two garbage collections. Figure 2.1 introduces such scenario, where right after a collection (stage $a$) $p$ references $o_0$. Then at stage $b$ it is modified to reference $o_1$; next (at stage $c$) it is modified to reference $o_2$. Further modifications of $o$ are skipped until the last modification (before the next garbage collection invocation) at stage $d$ where $p$ is modified to reference $o_n$. In such case, all previous reference-counting collectors execute $2n$ reference-count updates for these assignments: RC($o_0$)$--$, RC($o_1$)$++$ (both for step $b$); RC($o_1$)$--$, RC($o_2$)$++$ (both for step

$c$); ..., RC($o_n$)++ (for step $d$). However, only two updates are required: RC($o_0$)−− and RC($o_n$)++.

Suppose the reference counts we have represent the heap view at the previous collection time, and we would like to update them for the current collection time. In light of the observation above, it suffices to do the following updates. For each pointer $p$ that was modified between the two collections:

1. find $p$'s referent in the previous collection time (corresponding to $o_0$ above) and decrement its reference count, and
2. find $p$'s referent in the current collection time (corresponding to $o_n$ above) and increment its reference count.

It remains to devise a mechanism that records all pointers that were modified after the previous collection. Furthermore, this mechanism should provide, for each such pointer, its referent at the previous collection time and its referent at the current collection time. To achieve this we employ a write barrier, a piece of code activated on each pointer assignment. Each program thread maintains a local buffer, denoted *Updates* buffer, in which all updated pointers are logged together with their pre-modification values. For efficiency (as explained by Levanoni and Petrank [63]), all pointers of an updated object are logged rather than each single updated pointer. To make sure that each object is logged only once, a *dirty* bit per object is employed to signify whether the object has already been logged. During a collection, all objects are marked not dirty. Then, at the first time a thread modifies an object, it marks the object dirty, it logs the modified object's address and it logs all its pointers' previous referents in the *Updates* buffer. Further modifications to the (dirty) object are not recorded. To deal with multithreaded programs, a carefully designed write barrier is presented in [63] allowing the above write barrier to operate on concurrent threads without requiring synchronized operations. When a new collection begins, the *Updates* buffer provides all the information required to update the reference counts: it lists all modified objects, and keeps a record of their referents before the first modification (these are the objects' referents in the previous collection time). In the current collection, the collector finds the current referents of these object's pointers.

A special case of modified objects are newly created objects. Such objects do not have referents at the previous collection time as they did not exist then. Newly created objects are created dirty (to prevent logging in the *Updates* buffer) and are logged (upon creation) in a special buffer, denoted the *YoungObjects* buffer. The collector increments the reference

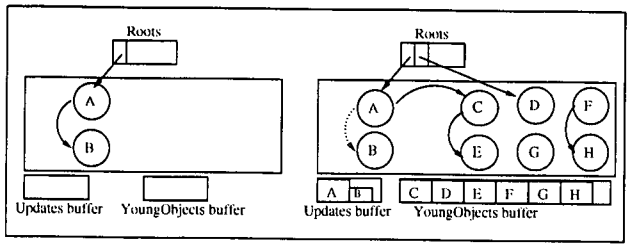



Figure 2.2: An example: heap and buffers view in 2 subsequent collections

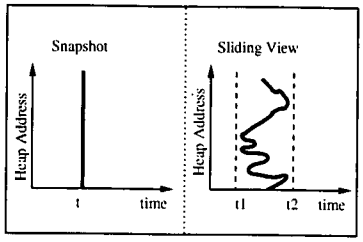



Figure 2.3: A snapshot view at time $t$ vs. a sliding view at interval $[t1,t2]$

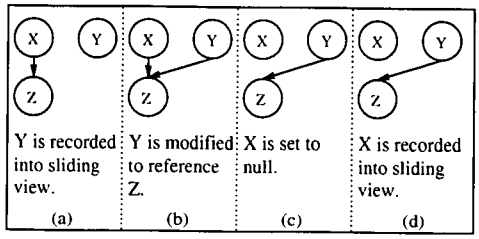



Figure 2.4: An example in which the reachability of object $Z$ is missed by a sliding view

counts of their referents at the current collection time, but does not need to do any related decrements.

An example appears in Figure 2.2. It depicts the heap and the buffers in two subsequent collections, where the view of the former collection appears on the left side. The *YoungObjects* buffer contains the six objects that were created after the last collection. Between the two collections a pointer in $A$ was modified to reference $C$. Hence, $A$ was logged in the *Updates* buffer, together with its previous referent $B$ (which appears next to $A$ in a smaller font). The collector uses this information in the following way. It iterates over the objects logged in the *Updates* buffer and finds $A$. It decrements the reference count of $B$, which is $A$'s descendant in the previous collection, and it increments the reference count of $C$, $A$'s descendant at the current collection time. It then iterates over the six objects in the *YoungObjects* buffer. It increments the reference counts of their descendants at the current collection time. For example, for the object $F$ the reference count of $H$ is incremented.

The above algorithm uses an implicit snapshot of the heap. A snapshot at time $t$ is a copy

of the content of each heap object at time $t$. The implicit snapshot is obtained by stopping all program threads simultaneously to read their local buffers, their roots, and un-dirty all the objects. However, stopping all threads simultaneously increases the pause time. To get an on-the-fly collector, program threads are not stopped simultaneously, and thus a snapshot cannot be used. Instead, a collection works with a *sliding view* of the heap. A sliding view of the heap is associated with a time interval $[t_1,t_2]$ (rather than a single point in time). It provides the content of each heap object at some arbitrary time $t$, satisfying $t_1 \leq t \leq t_2$. In contrast to a snapshot, objects are not all viewed at the same time. Figure 2.3 depicts the difference between a sliding view and a snapshot. Using a sliding view for collection introduces a correctness danger: the view may not reflect correctly objects reachability. Figure 2.4 shows such example, where the reachability of $Z$ is missed in the sliding view, although it is reachable. The example takes place during the time interval $[t_1,t_2]$. During step (a), the collector samples (the descendants of) object $Y$. In step (b) object $Y$ is modified to reference object $Z$. As the collector has already sampled object $Y$, it is not aware of this modification. In stage (c) object $X$ is modified to stop referencing object $Z$, and then at stage (d) object $X$ is sampled. Although object $Z$ was reachable throughout the entire duration of the example, the sliding-view misses the references to object $Z$ and hence it may be reclaimed (if it is not referenced by another object). A solution to this problem is a *snooping* mechanism. The *snooping* mechanism records (via write barrier) any object to which a new reference is created in the heap during the time interval $[t_1,t_2]$. Snooped objects are considered (as referenced by) roots, and are not reclaimed in the current collection cycle. Returning to the example presented in Figure 2.4, as the reference from object $Y$ to object $Z$ is created during the time interval $[t_1,t_2]$, object $Z$ would be snooped and hence would not be collected.

### 2.6.1 The collector phases

A collection begins by taking a sliding view of the heap. To achieve cooperation between the collector and the program threads, handshakes are used. During a handshake, each thread is halted (separately, not simultaneously) for a short pause to cooperate with the collector. During a halt, data may be exchanged between the collector and the program threads. The Levanoni-Petrank collector employs four handshakes during the collection cycle.

The collection starts with the collector raising the thread local $Snoop_i$ flag, signaling to the mutators that it is about to start computing a sliding view. During the first handshake, the mutators are given new empty local buffers and the old buffers are transferred to the

collector. There are two such buffers for each mutator. $YoungObjects_i$ contains all objects created since the last collection by Mutator $i$ and $Updates_i$ contains all objects modified by Mutator $i$ since the last collection together with their previous sliding-view non-null pointer slot values. Next, the dirty flags of the objects listed in the buffers (both the *YoungObjects* and the *Updates* buffer) are cleared while the mutators are running. Most of the clearing operations in this stage, clear dirty bits created during the previous collection cycle, as intended. But some clearing is done to (a small number of) objects that have been dirtied and logged in the buffers (by the running program threads) concurrently with the collector clearing process. Such dirty bits should not have been cleared and hence the collector proceeds by reinforcing these bits, i.e., by fixing these dirty bits. It reads the new local buffers from all mutators in a second handshake and sets the dirty bits for the relevant objects. A third handshake with no specific operation is executed to make sure that the reinforced dirty bits are visible to all mutators. A fourth handshake is used to reset the thread local $Snoop_i$ flag, to scan threads' local states and to mark objects directly reachable from the roots as *Roots*.

After the fourth handshake the collector adjusts $rc$ fields due to differences between the sliding views of the previous and current cycle. Each object which is logged to one of the mutator's local buffers has been modified since the previous collection cycle, thus we need to decrement the $rc$ of its children (as reflected by its pointer slot values) as appearing in the previous sliding view and increment the $rc$ of its slots values in the current sliding view. The $rc$ decrement operation of each modified object is done using the objects' information obtained from the retrieved local buffers. This information contains the object non-null pointer slots' value at the previous sliding view.

In order to decide which objects' $rc$ to increment, one should determine what are the children of an object $O$ in the current sliding view. If $O$ has not been modified since the beginning of this collection then its pointer slots values may be read from the heap. If it has been modified, the values of its pointer slots in the current sliding view must be obtained from the new *Updates* buffers currently being written by the threads.

A collection cycle ends with reclamation which recursively frees any object with zero $rc$ field which is not marked as *Roots* and is not snooped.

# Chapter 3

# An On-the-Fly Mark-and-Sweep Garbage Collector Based on Sliding Views

## 3.1 Introduction

In this chapter, we present the design and implementation of a new efficient and non-intrusive garbage collector suitable for Java and C# running on modern SMP's and using large heaps[1]. Our algorithm is a non-moving mark-and-sweep collector based on a "relaxed" snapshot of the heap (the sliding views). It is suitable for modern SMP's running concurrent programs. It is fully concurrent (on-the-fly) allowing short pause times. Namely, each thread is stopped for a short while to cooperate with the collector, but the threads never need to be stopped at the same time. Finally, our collector may be used with conservative JVM's.

The algorithm presented in this work is inter-operable with the reference-counting sliding-views algorithm of Levanoni-Petrank meaning that they share the same allocator, and their data structure may be united so that it is possible to decide on a (collection) cycle by (collection) cycle basis which algorithm should be invoked. Hence, our sliding-views mark-and-sweep (SVMS) collector may also be used as a backup tracing algorithm for the sliding-views reference-counting collector of Levanoni and Petrank [62, 63]. Any reference-counting collector may need to use a tracing collector to reclaim cyclic garbage. It is advantageous for

[1]This work was presented in [4].

the reference-counting collector to have a tracing collector that may use a similar allocator and a similar JVM interface. A preliminary version of this mark-and-sweep algorithm was implemented for that purpose and was used to produce the results of the reference-counting collector reported in [62, 63]. However, the tracing sliding-view algorithm has not been reported previously and its properties have not yet been investigated. In this work, we present a mature version of this collector accompanied by an implementation and measurements.

### 3.1.1 The main algorithmic ideas

The basic mark-and-sweep algorithm operates by stopping all program threads, *marking* any object which is directly reachable (either from a local or a global reference) and then recursively marking any object which is pointed to by a marked object. Then, any object, which is not marked is *swept*, i.e., reclaimed. Finally, program threads are resumed.

To simplify the presentation of our new collector, we start with a simple concurrent mark and sweep that uses a snapshot. Concurrent mark-and-sweep collectors perform some, or all, of the above steps concurrently with the program threads (the mutators). *Snapshot at the beginning* [101, 40] mark-and-sweep collectors exploit the fact that a garbage object remains garbage until the collector recycles it, i.e., being garbage is a stable property. Thus, (naive) snapshot at the beginning operates by:

1. stopping the mutators,
2. taking a snapshot (replica) of the heap and roots,
3. resuming the mutators,
4. tracing the replica,
5. sweeping all objects in the original heap whose replicated counterparts are unmarked. These reclaimed objects must have been unreachable at the time the snapshot was taken and hence they remain unreachable until the collector eventually frees them.

The problem with this approach is that making a snapshot of the heap is not realistic. It requires too much space and time. However, a useful property of today's benchmarks is that even if they employ a large heap, only a small part of it is modified at a time.

Loosely speaking, our algorithm works as follows. Assume first that the heap does not change at all (which is not correct) and traverse the heap concurrently with the program

activity. However, the heap *is* modified by the program threads and we cannot ignore it. Our solution is to record objects states before they are first modified by a mutator. Later, we trace according to the recorded state. A mechanism that remembers the object state before it is first modified by a mutator has been developed in [62, 63] for monitoring changes in reference counts. We employ that mechanism for our algorithm.

In turns out that a substantial portion of this mechanism's overhead could be saved when incorporated into a mark-and-sweep collector. The first saving is obtained by the fact that recording is necessary only when the collector is active. The second saving is due to the fact that we need to record an object $A$ only if the object $A$ is modified after the collector started and before $A$ is traced. This seldom happens. In particular, during a collection we create new objects as marked. Thus, updates to new objects, which are most frequent, do not require recording the values of the new object. Finally, and similarly to the reference-counting saving, we need to record objects only once: the first time they get modified after the collection starts. All these savings make the write barrier very efficient. Usually, it only employs a fast path running only a couple of if statements. The long path of actually recording the object's state is taken infrequently (see the measured statistics in Section 3.6 below). The modified write barrier ,that we propose, maintains the good properties of the original write barrier of Levanoni and Petrank [63]. In particular, it allows concurrent threads to collect the information with no extra synchronization. More details appear in Section 3.2.1.

Finally, the algorithm described so far needs to stop all threads at the same time in order to determine the snapshot time. This is a must with a multiprocessor since we need to determine one specific time at which no thread is in the middle of an update operation or in the middle of creating a new object and at which all threads "know" that a snapshot time has been set. Such wide mutator synchronization increases the pause time, as all mutator threads must come to a halt together. In order to eliminate this synchronization, we let the collector determine the snapshot time for each mutator asynchronously at its own pace. This reduces the pause time to the level reported in this work but requires some care to assure correctness. In particular, we get a fuzzy snapshot, called a "sliding view" of the heap. This view is not an accurate snapshot, but we can use it for collection with an additional aiding mechanism called "snooping". During the (short) time that the sliding view is determined, the write barrier records all pointer assignments. When marking the roots is over, snooping is stopped and all pointer slots recorded by the snooping mechanism are traced as if they were roots. This may lead to a small amount of floating garbage but it is required for correctness. Details appear in Section 3.2.2.

In our implementation, we employ state-of-the-art engineering tricks such as block oriented allocator [17, 36] and color toggle [60, 50, 30, 53, 35], allowing a simple coloring of allocated objects, and saving some of the sweep work.

### 3.1.2 Comparison with the Levanoni-Petrank collector

This work is based on the *sliding-views* concept from [62, 63]. The collector presented here is a different collector by nature since it is a mark-and-sweep collector. For today's benchmarks, the tracing collector runs faster. Our contribution here is in presenting the tracing collector, implementing it on Jikes RVM and measuring its performance against two collectors supplied with Jikes: the on-the-fly reference-counting collector and the stop-the-world tracing collector.

The algorithmic contribution in this work is in the composition of several algorithmic ideas into one optimized collector. We start with the snapshot mark-and-sweep collector employing ideas from [40, 103]. We then modify this collector to make it suitable for stock hardware: instead of using the operating system copy-on-write feature, we let the mutators record modified objects via a write barrier. But now, we may borrow the mechanism of [62, 63] for keeping track of modified objects and obtain a fast and non-intrusively collector. Once we make this connection, we note that optimization may be used on the combined modified collector. The write barrier must record a modified object only if the collector is tracing (rather than always as in [62, 63]). Furthermore, recording is only required if the object being modified has not yet been traced. These two restrictions allow frequent use of a fast path for the write barrier and only infrequent actual recording of an object state.

### 3.1.3 Implementation and results

The sliding-views mark-and-sweep collector is implemented in Jikes RVM [1], a JVM system written entirely in Java (with some primitives for manipulating raw memory). The system was run on a 4-way IBM Netfinity server. We used the SPECjbb2000 benchmark and the SPECjvm98 benchmark suites. These benchmarks are described in detail in SPEC's web site [90].

It turns out that our algorithm is non-intrusive. The maximum pause time measured for all the run benchmarks was 2 ms, which is two orders of magnitude shorter than the pauses of the stop-the-world collector, but is even shorter than the concurrent Jikes collector. This pause time is the time it takes to scan the roots of a single thread. The rest of our handshakes

are much faster. In Jikes concurrent collector, the pause time is larger since (in addition to scanning the roots in each collection) it sometimes runs some allocator maintenance while threads wait. This is not required by our collector.

As for efficiency, our on-the-fly collector is slower than the stop-the-world collector by around 5-10%, which is "normal" for concurrent collectors. Comparing with the concurrent collector supplied with Jikes RVM (see [6]), we obtained a throughput improvement of up to 60% (for the SPECjbb2000 benchmark).

**Memory consistency.** We start by describing our collector for a sequentially consistent memory. In Section 3.4 below, we provide modifications that allow the algorithm to run on platforms, which are not sequentially consistent. From our experience with the Netfinity (running the Pentium III Xeon processor), the modifications were not required for all the benchmarks that we used.

**The DLG collector.** Although our collector comes from an advanced synergy of [103, 40] with [62, 63], the outcome collector should be also compared to the collector of Doligez-Leroy-Gonthier [34, 33], which is the most advanced on-the-fly mark-and-sweep collector. The DLG collector also uses fine-grained synchronization, and it was used in a production JVM of IBM (see [35, 36]). Unfortunately, we are not aware of an implementation of that algorithm that is available for academic research (and in particular, it is not implemented on Jikes). Therefore, it is not possible to show a direct comparison of throughput and latency. We expect the pause times to be similar as in both algorithms the longest pause emanates from marking the roots. In terms of efficiency, although the tracing algorithms are somewhat different, we do not see any theoretical comparison factors that may be stated without actually running the collectors. With respect to the write barrier more may be said. Ignoring the short interval in which the roots are marked and both collectors use an extended write barrier, the DLG collector marks gray the ex-target of any modified pointer. This means that the write barrier forces the mutator to touch a different object, whereas our write barrier touches only the modified object, copying the non-null pointers *at the first time the object is modified* and *only before it is traced.* Thus, our write barrier may take the short path more frequently and it may impose a better cache behavior. However, the actual answer must be done by a comparison of our collector with a serious and well-thought implementation of the DLG collector (which is not available for us). In any case, we believe that it is important to propose a (good) alternative to the state-of-the-art on-the-fly mark-and-sweep collector.

**Chapter organization.** We start with an overview of the collector algorithm in Section 3.2 below. We provide the algorithmic details and pseudo-code in Section 3.3. In Section

3.4 we explain how to adapt the algorithm to platforms that do not provide sequentially consistent memory. We say a few words on the implementation for Java in Section 3.5 and in Section 3.6 we present performance results. We conclude in Section 3.7.

## 3.2 Collector Overview

In this section we describe our new collector. For clarity of presentation, we start with an intermediate concurrent algorithm called *the snapshot algorithm.* In Section 3.2.2, we extend this intermediate algorithm making it on-the-fly.

### 3.2.1 Snapshot based algorithm

We start with an intermediate algorithm called *the snapshot algorithm.* This is a concurrent collector that requires a synchronization point in which all mutators are halted together to determine a snapshot time in which no mutator is in the middle of an update operation or in the middle of creating a new object. Most of the ideas presented with this simpler collector apply to our on-the-fly collector. Note that the length of the pause for this algorithm is short, but it requires synchronizing all application threads, which might mean longer pauses, especially for operating systems that do not support an efficient suspension of all application threads.

The idea, as presented in Section 3.1.1, is to perform the marking phase after taking a snapshot of the heap. Once the heap is frozen in a snapshot, the marking phase may proceed on the snapshot view while the mutators go on modifying the real heap. At the end of the trace, unmarked objects may be safely reclaimed since dead objects can not be touched or modified by the mutators.

Since taking a real snapshot is too costly, our algorithm takes the following approach. In the beginning of the collection all mutators are stopped and implicitly agree on a snapshot time. At the same stop, their roots are being marked and all threads resume. From that moment on, the mutators use the following write barrier for each pointer modification. If the collector is still tracing, and if the modified object has not been traced yet, and if the modified object is not dirty, then the object becomes dirty and the values of its pointers are saved (copied) to a local buffer. The write barrier does the logging (and dirtying) only for non-dirty objects. Thus, actual logging of the object state is only required infrequently: when the collector has started, but has not yet traced the modified object, and when the

object is modified for the first time. In that case, the saved values are the values of non-null pointers as existed during the snapshot time. Mostly, the write barrier runs the short path (i.e., performs the pointer assignment without any logging) and finishes quickly. As an object is only saved once during a collection cycle, the number of objects that need to be saved is the number of objects that get modified during the collection. We ignore for a moment the possibility that mutators modify the same object concurrently. We will show later that the write barrier works well also in this case without requiring explicit synchronization.

Given the operations described above, the collector may trace the objects as if it has a heap snapshot. Non-dirty objects may be read from the heap, because they were not modified. The state of dirty objects at the time of the snapshot may be obtained from the local buffers. To finish the collection cycle, the mutators are notified that the write barrier is not required anymore, and sweep is run to reclaim unmarked objects. Finally, all the dirty marks on the objects that appear in the buffers are cleared so that they become ready for the next collection.

We now return to the race issue raised above. What happens if two mutators modify the same object concurrently? Are the recorded values correct? Our write barrier is taken from the Levanoni-Petrank reference-counting collector and is especially designed to handle such races without employing costly synchronization operations. A simplified version of the write barrier pseudo-code appears in Figure 3.1. (We study this simplified version since it clarifies all the relevant points. The actual write barrier is more efficient and it appears in Section 3.3 below.)

Two mutators that invoke the update barrier concurrently to modify the same location do not fool the collection. We remark that in normal benchmarks (and programs) mutators do not race over writing to the same location without synchronization. Such races rarely appear in programs (they do appear in programs that try to implement a lock, or programs that trust the various threads to write the same value, etc.). Such races usually appear when the program contains a bug. Either way (and even if the program contains a bug) we would like our collector to handle the situation properly and not fail during program execution. Our first analysis of this write barrier is based on sequential consistency. However, simple modifications may settle this issue and make the collector run correctly on weakly consistent platforms at a negligible throughput penalty. This issue is discussed in Section 3.4 below.

Looking at the write barrier pseudo-code we split the analysis into two cases. First, suppose one of the updating threads sets the dirty flag of an object before any other thread reads the dirty flag. In this case, only one thread records this object and the records properly reflect the pointer values at the snapshot time. The other case is when more than one thread

```
Procedure Update(o: Object, s: Slot, new: Object)
begin
1.    if TraceOn and o.color=white then
2.       local old := read(o)
3.       // was o written to since the snapshot time ?
4.       if ¬ Dirty(o) then
5.          // ... no;. keep a record of the old values.
6.          Buffer[CurrPos] := ⟨o, old⟩
7.          CurrPos := CurrPos + length(o)
8.          Dirty(o) := true
9.    write(s, new)
end
```

Figure 3.1: SVMS mutator code: A simplified update operation

finds the dirty bit clear. We will show that in this (rare) case, more than one mutator may log the value of an object, but it is guaranteed that all logs will reflect the same (correct) value corresponding to the object's state during the snapshot time.

Looking at the code, each thread starts by recording the old value of the object, and only then it checks the dirty bit. On the other hand, the actual update of $o$ occurs after the dirty bit is set. Thus, if a thread detects a clear dirty bit, then it is guaranteed, since sequential consistency is assumed, that the value it records is the value of $o$ before any of the threads has modified it. So while several threads may record the object $o$ in their buffers, all of them must record the same (correct) information. To summarize, in case a race occurs, it is possible that several threads record the object $o$ in their local buffers. However, all of them record the same correct value of $o$ at the snapshot time. When using the information for the tracing, each of these records may be used. We conclude that even when races occur, the content of any heap pointer during the snapshot time can be obtained. The value of this pointer has either not been modified since the snapshot or it appears in the records taken by the mutators.

### 3.2.2 Using sliding views

The snapshot-based algorithm manages to execute a major part of the collection while the mutators run concurrently with the collector. The main disadvantage of this algorithm is the halting of the mutators in the beginning of the collection. During this halt all threads are stopped while the local roots are marked. This halt hinders both efficiency, since only one processor executes the work and the rest are idle, and scalability, since more threads will cause more delays. While efficiency can be enhanced by parallelizing the local marking phase, scalability calls for eliminating complete halts from the algorithm. This is indeed the case with our sliding-views algorithm, which avoids grinding halts completely.

A handshake [34, 33] is a synchronization mechanism in which each thread stops at a time to perform some transaction with the collector. Our algorithm uses four handshakes and avoids using stronger synchronization mechanism between threads. Thus, mutators are only suspended one at a time. The suspension duration is short and depends on the size of the mutator's local state.

In the snapshot algorithm we had a fixed point of time at which we perform the trace. It was the time when all mutators were stopped. Namely, the snapshot algorithm is guaranteed to trace the same objects as if it had done the trace while keeping the mutators suspended. By dispensing with the complete halting of threads we no longer have this fixed point of time. Rather, we have a fuzzier picture of the system, formalized by the notion of a *sliding view*, which is essentially a non-atomic picture of the heap. We show how sliding views can be used instead of atomic snapshots in order to devise a collection algorithm. This approach has been taken from the reference-counting collector of Levanoni and Petrank [62, 63] and is similar to the way snapshots are taken in a distributed setting. Each mutator at a time will provide its view of (the modifications in) the heap, and special care will be taken by the system to make sure that while the information is gathered, concurrent modifications of the heap do not fool the collection.

Instead of stopping all mutators together for initiating a collection and marking their local roots, we stop one mutator at a time. The problem with such a relaxation is that the various threads start using the write barrier at different times. Furthermore, the scanning of the stacks is not done simultaneously and thus, a reference may be missed because it is moved from one location to another during the time we mark the threads' local roots.

Therefore we take a rather extreme, yet required, measure. Before we start marking the roots, we raise a snoop flag for each mutator. The local snoop flag is cleared when the local roots of the mutator are marked. Thus, throughout the time we mark local roots, the

threads use a *snooping* mechanism via their write barrier. During this interval of time, all pointer updates are monitored. For each pointer update $p = O$ we add the object $O$ to a local snooping buffer. All objects recorded in this manner will later be traced during the mark phase as if they were roots.

The snooping mechanism may lead to some floating garbage as we conservatively do not collect objects which have been recorded by the snooping mechanism (have been snooped), although such objects may become garbage before the cycle ends. However, if a snooped object becomes unreachable, it is guaranteed to be collected in the next cycle.

## 3.3 The Garbage Collector Details

### 3.3.1 The LogPointer

One important choice that we made in our implementation affects the algorithmic details concerning the dirty bit. Each object must have a dirty bit signifying whether a pointer in the object has been modified since the sliding view started. Instead of using a single dirty bit per object we chose to dedicate a full word for the task. Indeed, this consumes space, but it allows keeping information about the dirty object. In particular, we use this word to keep a pointer to the location in the thread's local buffer where the object's pointers have been logged. A zero value (a null pointer) signifies that the object is not dirty (and not logged). We call this word the *LogPointer*.

Paying the extra price of allotting a whole word for the flag and transforming it into a pointer that identifies the logged contents of an object, rather than using a boolean bit-sized flag, enables an efficient tracing mechanism. Our tracing procedure does not need to "search" all local buffers to find out the recorded information about the object's state as recorded in the local buffers. Instead, it follows the pointer in the object header. Thus, the tracing procedure can always proceeds immediately after accessing the object's *LogPointer* field, either as dictated by the current objects' contents or according to the previous state of the object, as recorded in the log entry (pointed by the *LogPointer* field). Which of the two routes is taken is determined by the value of *LogPointer* (whether it is null or not).

```
Procedure Update(o: Object, offset: int , new: Object)
begin
1.    if TraceOn and o.color=white then
2.      if o.LogPointer=NULL then // object not dirty
3.        TempPos := CurrPos
4.        foreach field ptr of o which is not NULL
5.          Buffer[++TempPos] := ptr
6.        // is it still not dirty?
7.        if o.LogPointer=NULL then
8.          // add pointer to object
9.          Buffer[++TempPos] := address of o
10.         //committing values in buffer
11.         CurrPos := TempPos
12.         // set dirty
13.         o.LogPointer =
14.               address of Buffer[CurrPos]
15.   write( o, offset ,new)
16.   if Snoop and new != NULL then
17.     Snooped := Snooped ∪ { new }
end
```

Figure 3.2: SVMS mutator code: Update operation

### 3.3.2 Mutator cooperation

The mutators need to execute garbage-collection related code on three occasions: when updating an object (accomplished by the Update Procedure presented in Figure 3.2), when allocating a new object (accomplished by the New Procedure introduced in Figure 3.3) and during handshakes (accomplished by the handshake mechanism). The Update and New operations never interleave with a handshake. Namely, cooperation with a handshake waits until a currently executed Update or New operation finish.

In what follows we sometimes use the standard terminology of denoting a marked object *black* and an unmarked object *white*.

**Write barrier.** Procedure **Update** (Figure 3.2) is activated at pointer assignment and its main task is to record the object whose pointer is modified. We stress that the write barrier (the Update protocol) is only used with heap pointer modification. Modifications of local pointers in the registers or stack are not monitored. The logging should be done for a limited period: from the time local roots are marked till the tracing is done. The variable *TraceOn* is local to the mutator but is controlled by the collector. It tells the mutator whether the logging should be done. Thus, the first check is whether executing the write barrier is at all required. Next, we check whether the object is colored black. If it is the case, then the object is either new (i.e., this object was created during the current collection), or has already been traced during the current collection. In both cases, there is no need to log its old values (since this object won't be traced). Going through the pseudo-code, we see that each object's *LogPointer* is optimistically probed twice (lines 2 and 7) so that if the object is dirty (which is often the case), then the write barrier is extremely fast. If the object was not logged (i.e., the *LogPointer* of an object is NULL) then after the first probe, the objects values are recorded into the local *Buffer* (lines 3-5). The second probe at line 7 ensures that the object has not yet been logged (by another thread). If *LogPointer* is still NULL (in the second probe), then the recorded values are committed (line 9) and the buffer pointer is modified (line 11). In order to be able to distinguish later between objects and logged values, in line 9 we actually log the object's address with the least significant bit set on (while values are logged with least significant bit turned off). Then, the object's *LogPointer* field is set to point to these values (lines 13-14). After logging has occurred, the actual pointer modification happens. Finally, while marking the roots of the mutators, the snoop flag is on. At that time, the new target of the pointer assignment is recorded in the local snooped buffer. This happens in lines 16-17. The variables *Buffer*, *CurrPos*, *Snoop* and *Snooped* are local to the thread.

In our prototype we did not treat large objects in a special manner. However, buffering objects of substantial size, that contain a large amount of pointers, may exceed the 2ms pause time reported. To make sure this does not happen, one may associate dirty bits with areas smaller than object sizes. For example, the heap may be partitioned into cards of fixed size and each of them may be associated with a dirty bit (or a log pointer). Another possibility is to modify the write barrier and collector treatment of only large objects, so that they, only, are split into cards.

**Creating a new object.** Procedure **New** (Figure 3.3) is used when allocating an object. After the object is allocated, it is given a color (dictated by the collector), according to the allocation color. The allocation color is set by the collector during the various collection

```
Procedure New(size: Integer, o:Object)
begin
1.  Obtain an object o of size size from the allocator.
2.  o.color := AllocColor
3.  return o
end
```

Figure 3.3: SVMS mutator code: Allocation operation

steps.

**The handshake mechanism.** Our handshake mechanism is the same as the one employed by the Doligez-Leroy-Gonthier collector [34, 33]. The mutator threads are never stopped together for cooperating with the collector. Instead, threads are suspended one at a time for the handshake. The stopping of the thread is not allowed while it is executing the write barrier or while it is creating a new object. While a thread is suspended, the collector executes the relevant actions for the handshake and then the thread is resumed. The collector repeats this process until all threads have cooperated. At that time, the handshake is completed.

### 3.3.3 Phases of the collection

The collector algorithm runs in phases as follows.

- **First handshake:** during this handshake each mutator is stopped and the `Snoop` local flag, which activates the snooping mechanism, is set.
- **Second handshake:** during this handshake each mutator is stopped and the `TraceOn` local flag, which activates the logging mechanism, is set.
- **Third handshake:** during this handshake each mutator is stopped and the local roots of each mutator are marked. Also, the *Snoop* local flag is cleared.
- **Tracing:** after the third handshake is done, the collector traces the heap from the marked objects and from all snooped objects.
- **Fourth handshake:** during this handshake each mutator is stopped and the local flag `TraceOn` is cleared, so that the mutators stop recording updates in the buffers.

```
Procedure Tracing-Collection-Cycle
begin
1.  Initiate-Collection-Cycle   // 1st and 2nd handshake
2.  Get-Roots                   // 3rd handshake
3.  Trace-Heap
4.  Sweep                       // 4th handshake
5.  Prepare-Next-Collection
end
```

Figure 3.4: SVMS collector code: Tracing algorithm

- **Sweep:** the collector sweeps the heap and reclaims allocated unmarked objects.
- **Clear dirty marks:** The collector clears the dirty marks of all objects previously recorded in the buffers.

### 3.3.4 Collector code

Collector's code for cycle $k$ is presented in **Procedure Tracing-Collection-Cycle** (Figure 3.4). Let us briefly describe each of the collector's procedures.

**Procedure Initiate-Collection-Cycle** (Figure 3.5) runs the first two handshakes. During the first handshake the *Snoop* flag is raised, signaling to the mutators that they should start snooping all stores into heap slots. During the second handshake the *TraceOn* flag is raised, signaling to the mutators that they should start logging old pointer values of objects modified for the first time. For correctness, it is important to separate the two handshakes. When any mutator starts logging values in its local buffer, all mutators should be already snooping. The modifications are done via handshake to make sure that on a multiprocessor each mutator sees its value properly.

**Procedure Get-Roots** (Figure 3.6) carries out the third handshake during which the *Snoop* flag is turned off and the thread local roots are accumulated into the *Roots* (global) buffer. Next, the *Snooped* buffer of each thread (containing snooped objects), is accumulated into *Roots*, and then cleared (for the next collection). In this procedure a color toggle is executed switching the values of black and white. The color toggle mechanism avoids races between the Sweep and the New procedures, and it avoids some redundant work of the Sweep Procedure

```
Procedure Initiate-Collection-Cycle
begin
1.    // first handshake
2.    for each thread T do
3.       suspend thread T
4.       Snoop := true
5.       resume T
6.    // second handshake
7.    for each thread T do
8.       suspend thread T
9.       TraceOn := true
10.      resume T
end
```

Figure 3.5: SVMS collector code: Initiate-Collection-Cycle Procedure

(see [60, 50, 30, 53, 35]). Note that it is correct to mark new objects black during the collection, since they are alive and have no children at the time of creation. The *AllocColor* variable of each thread is then set so that new objects are created black.

**Procedure Trace-Heap** (Figure 3.7) implements marking the roots and tracing the heap. (The threads are not stopped for this stage.)

**Procedure Trace** (Figure 3.8) traces a single object. It gets an object as input. This object is traced only if its color is *white*. If it is white, the collector tries to determine the object content (in particular, its children) as reflected in the sliding view of the cycle. If the object has changed since the sliding view was taken (line 9), then its sliding-view value is obtained from the relevant *Buffer* by checking the location pointed by *LogPointer* (line 10). Otherwise, the object has not changed since the sliding view was taken. In this case, we make a copy of the object and trace the copy so that tracing will not be affected by further concurrent execution of the program. Note, that the object is marked *black* only after determining the object's sliding-view content (recall that the update Procedure does not log *black* objects).

**Procedure Sweep** (Figure 3.9) starts with the fourth handshake, which turns off the *TraceOn* flag. As of this time, pointer values will not be recorded anymore. This is fine

```
Procedure Get-Roots
begin
1.   black := 1-black
2.   white := 1-white
3.   // third handshake
4.   for each thread T do
5.     suspend thread T
6.     AllocColor := black
7.     Snoop := false
8.     Roots := Roots ∪ State // copy thread local state.
9.     resume thread T
10.  for each thread T do
11.    // copy and clear snooped objects set
12.    Roots := Roots ∪ Snooped
13.    Snooped := ⊘
end
```

Figure 3.6: SVMS collector code: Get-Roots Procedure

since tracing has completed. Next, all *white* objects are returned to the allocator and made *blue*, to signify that they have been reclaimed. Note that by the end of the sweep all objects are black or blue. The color toggle makes use of this fact. One may think of black as white and continue to use the same color for allocation. During the next mark, the meaning of black is switched with white and the next collection starts.

**Procedure Prepare-Next-Collection** (Figure 3.10) clears all dirty marks (i.e., all *o.LogPointer*s) that were set by mutators during this collection cycle. Clearing runs concurrently with program run. The global *Roots* buffer and the local *Buffer* of each thread are also cleaned.

## 3.4 Weak memory consistency

Modern SMP's do not always guarantee sequential consistency. Thus, it is important to check which modifications are required by our collector to make it work on a weakly consistent platform. In this section we provide the required modifications and discuss their cost.

```
Procedure Trace-Heap
begin
1.    for each object o ∈ Roots do
2.       push o to MarkStack
3.    while MarkStack is not empty
4.       obj = pop(MarkStack)
5.       Trace(obj)
end
```

Figure 3.7: SVMS collector code: Trace-Heap Procedure

Before going through the required modifications, we would like to stress that suspending a thread implies a synchronization barrier. Thus, a handshake serves implicitly as a synchronization barrier among all threads, guaranteeing, for example, that the setting of the snoop flag is visible to all processors before the second handshake.

**Dependency 1:** in the write barrier, the reads and writes of the *LogPointer* (serving as the dirty flag) and the pointer slot must be executed in the order stated in the algorithm, so that several mutators do not race and write inconsistent data into the local buffers. To solve this dependency, we note that the write barrier begins with a check whether the collector is on and whether the object is not dirty. We need to add a synchronization barrier after setting the *LogPointer* and before modifying the pointer. This is done only if both checks are validated, i.e., the collector is on and the object is not dirty.

**Cost:** The measures in 3.6 show that the write barrier rarely needs to actually log an object. Thus, the vast majority of the pointer updates require no cost for handling the first dependency with weakly consistent platform.

**Dependency 2:** Another interaction that relies on the order of operations is the interaction between the mutators running the write barrier and the tracing collector. There are two problems here.

The first problem occurs when the collector discovers that the object is dirty and it then reads the buffer entry associated with the object. However, if sequential consistency is not guaranteed, the buffers may not yet contain the updated values (even though the *LogPointer* has already been set). The second problem occurs when the collector copies the object contents and then reads the *LogPointer* to find it null. The collector assumes that it has an unmodified copy of the object, as it was when the sliding view was taken. However, when

```
Procedure Trace(o: Object)
begin
1.    if o.color = white then
2.       if o.LogPointer = NULL then // if not dirty
3.          temp := o // getting a replica
4.          // is still not dirty?
5.          if o.LogPointer = NULL then
6.             for each slot s of temp do
7.                v := read(s)
8.                push v onto MarkStack
9.       else // object is dirty
10.         BufferPtr := getOldObject(o.LogPointer)
11.         for each slot s of BufferPtr do
12.            v := read(s)
13.            push v onto MarkStack
14.      o.color := black
end
```

Figure 3.8: SVMS collector code: Trace Procedure

sequential consistency is not guaranteed, it is possible that the collector read the contents of the object after it was modified, but because of memory access reordering the setting of the *LogPointer* flag has not yet become visible to the collector.

The idea for solving the first problem is to run the tracing in phases. First trace all objects that have not been modified and keep a list of all those objects that have been modified and still need to be traced. After this phase is done, the collector runs a handshake with the mutators to obtain their local buffers and provide them with new buffers. Now, a new phase begins in which we may trace through objects whose contents are recorded in the obtained buffers and through all objects that have not yet been modified. We run such phases again and again until the tracing is done. Checking the conditions that trigger the run of a new phase, one may check that one or two phases normally suffice for a typical benchmark. In particular, an object cannot be traced after the first handshake if it is *not* modified before the handshake, it is *not* traced before the handshake, and it *is* modified just after the first handshake (before it is traced). Such an event is rare in practice.

```
Procedure Sweep
begin
1.    // fourth handshake
2.    for each thread T do
3.       suspend thread T
4.       TraceOn := false
5.       resume T
6.    Let swept point to the first object in the heap
7.    while swept does not point pass the heap do
8.       if swept.color = white then
9.          swept.color := blue
10.         return swept to the allocator
11.      advance swept to the next object
end
```

Figure 3.9: SVMS collector code: Sweep Procedure

To solve the second problem, we use a "buffering" solution. Recall that because of the first dependency the mutators are running a synchronization barrier after setting the *LogPointer* and before modifying the pointer. Depending on some parameter $m$, the collector starts by making copies of $m$ objects that appear to be not dirty. Next, it performs a synchronization barrier. Then, the *LogPointer* of each of the $m$ copied objects is probed. If it is still null, then the copy of the object may be traced. Otherwise, the object is dirty and its content should be obtained from local buffers. The parameter $m$ determines the frequency of running the synchronization barrier, and in this sense the larger $m$ the better. However, a large $m$ implies a large buffer for copying objects, and also a somewhat increased probability that the copied object has been modified during the (longer) time interval between the time it was copied and the time its *LogPointer* was checked.

**Cost:** Running a couple of additional handshakes for each collection cycle is of negligible cost compared to the overall running time of the collection cycle (and to the running time of the program). Running a synchronization barrier once for every $m$ collector operations is negligible for $m$ large enough.

We remark that we have not implemented these modifications, but we have not witnessed any problem caused by reordering instructions on the Intel platform.

```
Procedure Prepare-Next-Collection
begin
1.    Roots := ⊘
2.    for each thread T do
3.       // clear all LogPointers
4.       foreach object o in Buffer
5.          o.LogPointer := NULL
6.       // clear objects buffer
7.       Buffer := ⊘
end
```

Figure 3.10: SVMS collector code: Prepare-Next-Collection Procedure

## 3.5 An Implementation for Java

We have implemented our algorithm in Jikes RVM [1] (running on Linux Red-Hat 7.2). The entire system, including the collector itself, is written in Java (extended with unsafe primitives available only to the Java Virtual Machine implementation to access raw memory). Jikes uses *safe-points*: rather than interrupting threads with asynchronous signals, each thread periodically checks a bit in a condition register that indicates that the runtime system wishes to gain control. This design significantly simplifies implementing the handshakes of the garbage collection. In addition, rather than implementing Java threads as operating system threads, Jikes multiplexes Java threads on *virtual-processors*, implemented as operating-system threads. Jikes establishes one virtual processor for each physical processor.

**Memory allocator.** Our implementation employs the non-copying allocator of Jikes, which is based on the allocator of Boehm, Demers, and Shenker [17]. This allocator is well suited for collectors that do not move objects. Small objects are allocated from per-processor segregated free-lists build from 16KB pages divided into fixed-size blocks. Large objects are allocated out of 4KB blocks with first-fit strategy. This allocator keeps the fragmentation low and allows efficient reclamation of objects.

## 3.6 Measurements

**Platform and benchmarks.** We have taken measurements on a 4-way IBM Netfinity 8500R server with 550MHz Intel Pentium III Xeon processors and 2GB of physical memory. The benchmarks we used were the SPECjvm98 benchmark suite and the SPECjbb2000 benchmark. These benchmarks are described in detail in SPEC's web site [90]. We feel that the multithreaded SPECjbb2000 benchmark is more interesting, as the SPECjvm98 are more appropriate for clients and our algorithm is targeted at servers.

**Testing procedure.** We used the benchmark suite using the test harness, performing standard automated runs of all the benchmarks in the suite. Our standard automated run runs each benchmark five times for each of the JVM's involved (each implementing a different collector). To get additional multithreaded benchmarks, we have also modified the _227_mtrt benchmark from the SPECjvm98 suite to run on a varying number of threads. We measured its run with 2, 4, 6, 8 and 10 threads. Finally, to understand better the behavior of our collector under tight and relaxed conditions, we tested it on varying heap sizes. For the SPECjvm98 suite, we started with a 24MB heap size and extended the sizes by 8MB increments until a final large size of 96MB. For SPECjbb2000 we used larger heaps, starting from 256MB heap size and extending by 64MB increments until a final large size of 704MB.

**The compared collectors.** We tested our concurrent collector against 2 collectors: Jikes concurrent collector and Jikes parallel load-balancing non-copying mark-and-sweep collector. Both collectors are distributed with Jikes RVM.

The concurrent collector is a modern on-the-fly pure reference-counting collector developed at IBM and reported in Bacon et al. [6]. It has similar characteristics to our collector, namely, the mutators are only very loosely synchronized with the collector, allowing very low pause times. This collector is denoted hereafter *Jikes concurrent collector*. We chose this collector, as it is the only on-the-fly collector that is available for comparison.

The stop-the-world collector associates a collector thread for each processor. This is a modern stop-the-world mark-and-sweep parallel collector initiated when an allocation fails. We refer to this collector later as *Jikes STW (stop-the-world) collector*. We chose this collector as a representative efficient stop-the-world collector.

### 3.6.1 Pause times

The maximum pause times for the runs of the SPECjvm98 benchmarks and the SPECjbb2000 benchmark are reported in table 3.1. The SPECjvm98 benchmarks were run with a 64MB heap size and the SPECjbb2000 (with 1,2,3 warehouses) were run with a 256MB heap size. In these measurements, the number of program threads is smaller than the number of CPU's. Note that if the number of threads exceeds the number of processors, then large pause times appear because threads lose the CPU to other mutators or the collector. The length of such pauses depends on the operating system scheduler and is not relevant to the collector. Hence we report only settings in which the collector runs on a separate spare processor.

Our maximum pause time measured for all the run benchmarks was 2.04 ms. Our pause times are smaller than these of Jikes concurrent collector for all tested benchmarks. One may wonder why these pause times are shorter than the ones reported for Jikes concurrent collector. Usually, the longest pause time for an on-the-fly collector is the time required for scanning the roots of a single thread, which is the same for both collectors. We discovered that the longest pauses in Jikes concurrent collector are due to freeing blocks for the allocator that is sometimes executed in addition to scanning the roots. For our collector the operation of scanning the roots is the longest pause. Other pauses are an order of magnitude shorter than the root-scanning handshake. Thus, our collector obtains shorter pauses than Jikes concurrent collector.

As expected the maximum pause times measured for our collector were much smaller than these of Jikes STW collector. In fact, the measurements show that the maximum pause times of Jikes STW collector are larger by a factor of at least 200!

Note that pause measurement for the _222_mpegaudio benchmark is not included for the STW collector, since it has low allocation activity and no collection is executed during its run (using the STW collector).

### 3.6.2 Server performance

**Comparison against Jikes concurrent collector.** Our major benchmark is the SPECjbb2000 benchmark. SPECjbb2000 requires multi-phased run with increasing number of warehouses. Each phase lasts for two minutes with a ramp-up period of half a minute before each phase. The benchmark provides a measure of the throughput and we report the throughput ratio improvement. Note that a larger number is better, and we report the ratio between our collector and the compared collector. Thus, the higher the ratio, the better our collector

| Benchmarks | Maximum pause time (milliseconds) | | |
|---|---|---|---|
| | Sliding Views | Jikes concurrent | Jikes STW |
| jess | 1.3 | 2.77 | 261 |
| db | 0.66 | 1.84 | 193 |
| javac | 2.04 | 2.81 | 645 |
| mpegaudio | 0.54 | 0.8 | - |
| jack | 0.91 | 1.66 | 226 |
| mtrt | 0.91 | 1.80 | 376 |
| jbb-1 | 0.6 | 1.79 | 324 |
| jbb-2 | 0.73 | 2.6 | 422 |
| jbb-3 | 0.93 | 3.15 | 517 |

Table 3.1: SVMS maximum pause time in milliseconds

behaves, and any ratio larger than 1 implies that our collector outperforms the compared collector.

The design point for Jikes concurrent collector was for one collector CPU to be able to handle 3 mutator CPU's, so that for four-processor chip multiprocessors one CPU would be dedicated to collection. Thus, when comparing to Jikes concurrent collector in this subsection, we also let the collector run on a separate spare processor and the results show mainly the ability of the concurrent collector to run concurrently without interfering with mutators work.

The measurements are reported for a varying number of warehouses and varying heap sizes in Figure 3.11. We can see that with small number of warehouses, both collectors act similarly with our collector doing a little better. When the number of warehouses is three and up, all 3 mutators' CPUs are in use, and the efficiency of the collector becomes more important. We can see that in this case, our collector outperforms Jikes concurrent collector and obtains a performance improvement of up to 60%.

The SPECjvm98 benchmarks and the modified _227_mtrt benchmark provide a measure of the elapsed running time, which we report. Here, the smaller the better. In Figure 3.12 we report the running time ratio of our collector and the compared collector. For clarity of

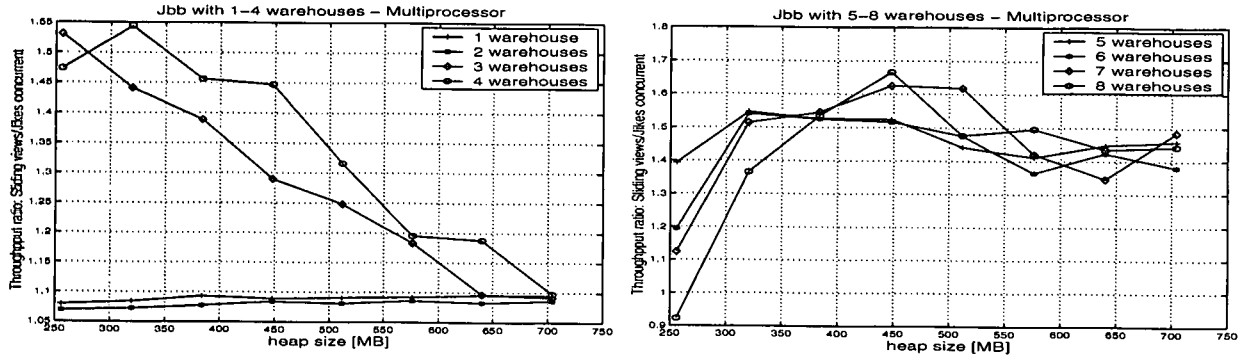



Figure 3.11: SPECjbb2000 on a multiprocessor: SVMS throughput ratio compared to Jikes concurrent collector

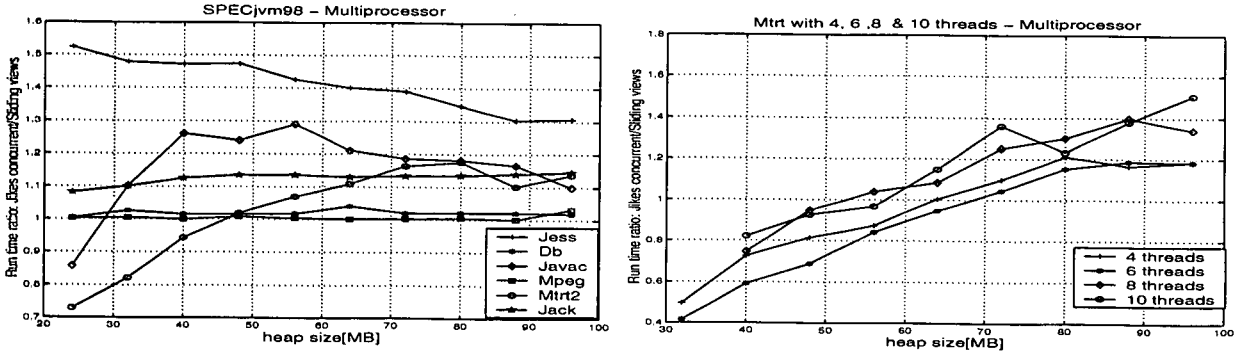



Figure 3.12: SPECjvm98 and modified _227_mtrt on a multiprocessor: SVMS run-time ratio compared to Jikes concurrent collector

presentation, we report the inverse ratio, so that higher ratios still show better performance of our collector, and ratios larger than 1 imply our collector outperforming the compared collector.

As before, when running the SPECjvm98 benchmarks on a multiprocessor, we allow a designated processor to run the collector thread. Results are reported in Figure 3.12. Here again the collector runs concurrently with the program thread and good concurrency is the main factor in the comparison. Mostly, the collectors perform similarly with our collector usually slightly winning. The picture changes for _213_javac and _202_jess with which our collector does much better. Indeed the compared collector is known to perform badly on these benchmarks (see [6]).

Note that the cases in which Jikes concurrent collector wins with SPECjvm98 as well as with the modified _227_mtrt measurements presented below, is when the heap is tight. The reason for worse results on small heaps is that during these runs, we get short in memory (on both collectors), and so mutators are sometimes halted waiting for a collection cycle

to terminate (and supply free space). These measurements demonstrate the superiority of reference counting (employed by Jikes concurrent collector) for such settings. When frequent collections are performed, the tracing collector still has to trace the whole heap and sweep it, whereas the reference-counting collector only needs to run over the latest modifications (in order to update the reference counts) and free the unreachable space. Note however, that this phenomena occurs only in highly stressful conditions. Normally, mutators are halted only in order to perform handshakes.

We do not include results for the _201_compress benchmark since its allocation activity is not significant.

Next, we report the measurements for the modified _227_mtrt benchmark. We modified it to work with a varying number of threads (4, 6, 8, 10 threads) and the resulting throughput measures are reported in (the right picture of) Figure 3.12. Note that a run with two threads appear with the SPECjvm98 measurements (reported as mtrt2 at the left picture of Figure 3.12). Once more we allow a designated processor to run the collector thread, however since all 3 mutator CPU's are in use, the collector's efficiency plays the major factor in these measurements. Here, again, we can see that with small heaps the compared collector wins. As before, this happens because of the superiority of reference counting in a setting where frequent collections are required.

**Comparison against Jikes STW collector.** We have compared our collector performance over the SPECjbb2000 benchmark and SPECjvm98 benchmarks also against Jikes STW collector. However, when comparing against Jikes STW collector with four and up mutators (on our 4-way machine), our collector did not run on a spare processor but rather shared a processor with the program threads. Note, nevertheless, that we gave the collector (in this case) the highest priority, so that when a collection is triggered the collector would always get enough CPU.

The measurements of the SPECjbb2000 benchmark are reported for a varying number of warehouses and varying heap sizes in Figure 3.13. We can see that with a small (1-3) number of warehouses (when our collector runs on a dedicated processor), both collectors have similar throughput, except for 3 warehouses for small heap sizes, where Jikes STW collector is slightly better.

When running 4-8 warehouses over small heap, Jikes STW collector outperforms our collector. This is the expected cost of running concurrently with program threads and using a write barrier. However, on large enough heap sizes, the compared collector is only slightly (3%-10%) better than our collector. The reason for the bad results over small heap sizes is that on these sizes our collector sometimes get short in memory, and so mutators are

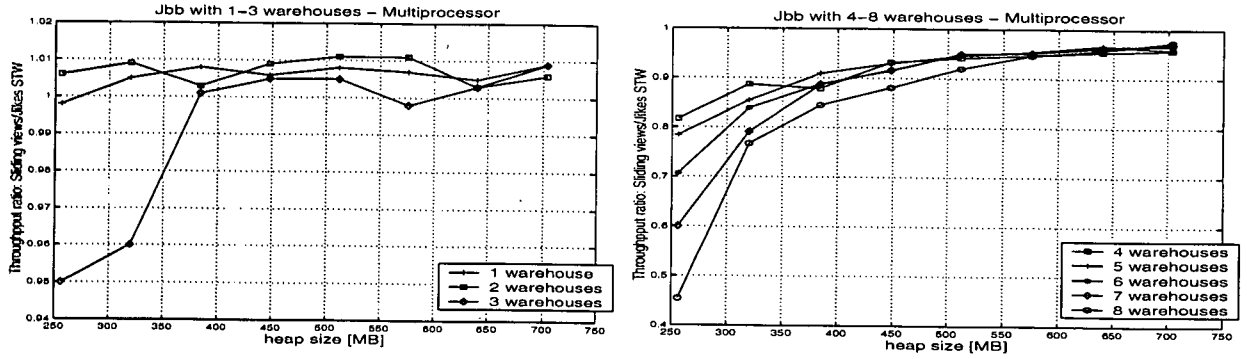



Figure 3.13: SPECjbb2000 on a multiprocessor: SVMS throughput ratio compared to Jikes STW collector

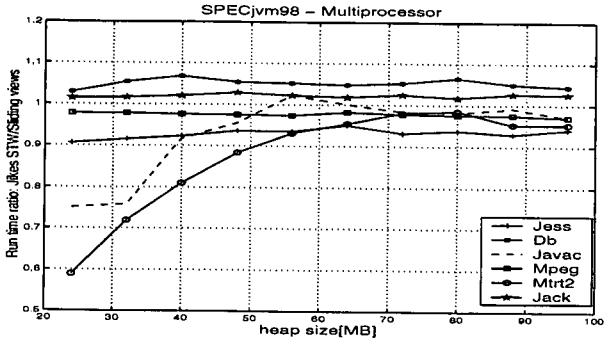



Figure 3.14: SPECjvm98 on a multiprocessor: SVMS run-time ratio compared to Jikes STW collector

sometimes halted waiting for a collection cycle to terminate (and supply free space). In these cases the superiority of a parallel collector (over a concurrent collector) is expressed: the parallel collector always exploits all 4 CPUs, while our on-the-fly collector uses only one until free space is supplied.

The measurements of the SPECjvm98 benchmark are reported in Figure 3.14. Here, our collector thread runs on a designated processor (i.e., the number of virtual processors is one more than the number of threads used by the benchmarks). Jikes STW collector runs on the same number of CPU's (gaining efficiency from running the collector in parallel on them all). It can be seen that usually the collectors perform similarly. When running _213_javac and _227_mtrt with smaller heap sizes, our collector performs worst, for the same reasons described above: utilizing only one of two CPUs (three in case of _227_mtrt) when mutators are stucked due to lack of free space.

| Benchmarks | percent trace is on | percent not traced | percent not dirty | fraction of logging |
|---|---|---|---|---|
| compress | 2.9 | 86.4 | 4.5 | 1/894 |
| jess | 5.9 | 3.3 | 3.8 | 1/13210 |
| db | 1.92 | 0.56 | 4.24 | 1/219354 |
| javac | 17.1 | 11.0 | 33.3 | 1/160 |
| mpegaudio | 0.04 | 86.0 | 4.6 | 1/64099 |
| jack | 4.2 | 10.6 | 1.4 | 1/16572 |
| mtrt2 | 13.2 | 3.4 | 5.4 | 1/4116 |
| jbb-1 | 2 | 7 | 8.6 | 1/8336 |
| jbb-2 | 6.1 | 17.8 | 8.8 | 1/1033 |
| jbb-3 | 23.3 | 17 | 8.5 | 1/299 |

Table 3.2: SVMS Write barrier: Fraction of write-barrier executions that take the long path (on average)

### 3.6.3 Collector characteristics

**Write-barrier measurements.** The write barrier (Figure 3.2) minimizes the number of object logging by using 3 filters. Table 3.2 shows the effect of each of these filters. Only write barrier executions that pass these 3 filters would actually log non-null pointers of the modified object. The measurements were taken while the collector ran on a separate spare processor. The SPECjvm98 benchmarks were run with a 64MB heap size and SPECjbb2000 (with 1,2,3 warehouses) was run with a 256MB heap size.

Recall that logging should be done only from the time local roots are marked till the tracing is done. The second column shows the percentage of write-barrier executions that occur during this time. These executions would pass the first filter (TraceOn flag was on). One can see that usually, as the number of mutators increases, the percentage of write-barrier executions that occur during this time increases, since memory is consumed faster making the collector run on a larger fraction of the overall time.

As objects that were already traced during the collection should not be logged, the third column shows the percentage of write-barrier executions in which the object (to be modified)

| Benchmarks | Heap size | update buffers | mark stack | snoop buffers | overall overhead |
|---|---|---|---|---|---|
| jess | 64 | 0.26 | 0.05 | 0.12 | 0.43 |
| db | 64 | 0.28 | 0.15 | 0.07 | 0.5 |
| javac | 64 | 0.73 | 0.22 | 0.11 | 1.06 |
| jack | 64 | 0.13 | 0.55 | 0.07 | 0.75 |
| mtrt | 64 | 0.15 | 0.55 | 0.14 | 0.84 |
| jbb-1 | 256 | 0.07 | 0.02 | 0.02 | 0.11 |
| jbb-2 | 256 | 0.17 | 0.02 | 0.05 | 0.24 |
| jbb-3 | 256 | 0.34 | 0.02 | 0.12 | 0.48 |
| jbb-4-8 | 256 | 0.36 | 0.02 | 0.13 | 0.51 |

Table 3.3: SVMS: Space overhead as a percentage of heap size

was not yet traced, thus, this percentage of write barrier executions pass the second filter (given that it passed the first filter). Normally, a large fraction of pointer updated are initializations of newly allocated objects. As can be seen, for most benchmarks a vast majority of the objects (on which the write barrier is executed) were already traced. This can be explained by the fact that new objects are created black.

Since any object is logged at most once per collection, the fourth column shows the percentage of write-barrier executions in which the object was actually logged, given that it was not traced yet and the collector is currently tracing. This is the fraction of objects that pass the third filter (out of these which passed the first and second filter). The low percentage indicates that objects are usually modified many times. The write barrier makes sure that only one of these modifications take the long path of the write barrier.

The fifth column shows the fraction of write barriers that run the long path out of the number of all write barriers executed during the run. The measurements show that each one of the 3 filters is essential for making the long path write-barriers executions rare.

**Write-barrier buffers' size.** The space overhead consumed by the thread local buffers depends on the behavior of the application. In this section we provide some measurements providing some insight on this overhead for the benchmarks we ran. In table 3.3 we present the space consumed by these size-varying structures for each of the benchmarks. The numbers reported are the maximum sizes required throughout the execution. The second column

| Benchmarks | percent get roots | percent trace | percent sweep | percent prepare next |
|---|---|---|---|---|
| jess | 0.97 | 39.7 | 57.39 | 1.91 |
| db | 0.53 | 40.48 | 56.73 | 1.94 |
| javac | 0.77 | 57.71 | 39.13 | 2.36 |
| mpegaudio | 2.19 | 84.38 | 12.76 | 0.66 |
| jack | 0.9 | 34.85 | 63.02 | 1.22 |
| mtrt2 | 0.7 | 54.28 | 43.89 | 1.1 |
| jbb-1 | 0.29 | 25.03 | 74.13 | 0.55 |
| jbb-2 | 0.26 | 28.45 | 70.21 | 1.08 |
| jbb-3 | 0.37 | 49.55 | 47.62 | 2.45 |

Table 3.4: SVMS: Percent time spent on each collection phase

presents the maximum overhead of the write-barrier buffers, the third column presents the maximum overhead of the *markStack* used for the traversal of the heap and the forth columns presents the maximum overhead of the *snoop* buffer. The last column summaries the total buffers' overhead.

The size of the buffers depends on application behavior. Specifically, the write-barrier buffers' size depends on the time consumed by the tracing phase (since the write barrier is active only during the tracing phase), and on the number of processors used to run mutators during the tracing phase (if more processors are used to run mutators, then more objects are logged). For multithreaded benchmarks we report the overall space used for all buffers by all mutators. The measurements show that the space overhead is negligible compared to the heap size. Note that with SPECjbb2000, when the number of warehouses (mutators) go up, the volume of activity goes up and so does the space overhead of the buffers. Since we use a 4-way machine, only 3 mutators may run concurrently with the tracing operation, thus, above 3 warehouses, this overhead remains steady.

**Profiling measurements.** Our collector comprises of 4 phases: getting roots, tracing, sweeping and preparing for the next collection. Table 3.4 shows the percentage of time that the collector spends at each one of these phases. As can be seen, at least 97% of the collector work is spent on tracing and sweeping, while the other 2 phases are minor. The distribution

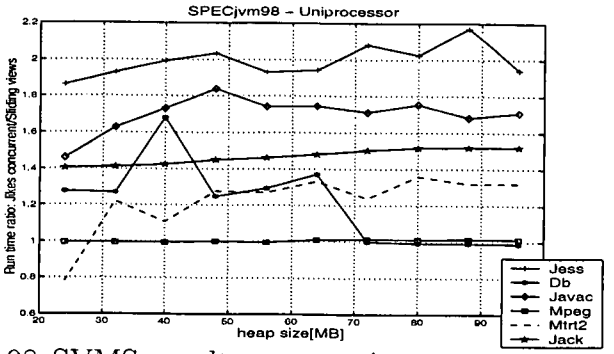



Figure 3.15: SPECjvm98 SVMS results on a uniprocessor compared to Jikes concurrent collector

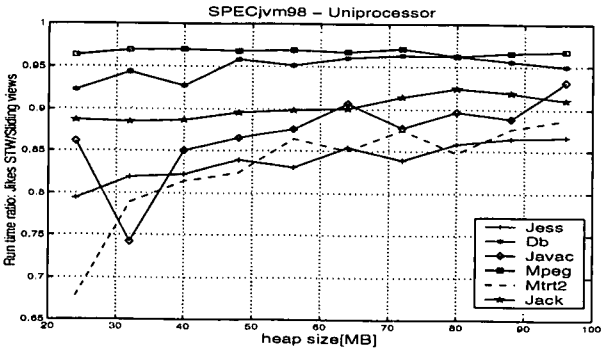



Figure 3.16: SPECjvm98 SVMS results on a uniprocessor compared to Jikes STW collector

between the tracing and the sweeping phases differs among the different benchmarks. It depends on the size of the live objects' graph and the amount of objects' freeing.

The measurements were taken while the collector ran on a separate spare processor. The SPECjvm98 benchmarks were run with a 64MB heap size and the SPECjbb2000 benchmark (with 1,2,3 warehouses) was run with a 256MB heap size.

### 3.6.4 Client performance

Although our collector is targeted at servers running on SMP platforms, as a sanity check, we also measured its performance against Jikes concurrent collector and Jikes STW collector on a uniprocessor. The behavior of the collector on a uniprocessor may demonstrate its efficiency. We measured our collector on a uniprocessor with the SPECjvm98 benchmark suite and the results appear in Figures 3.15 and 3.16. It turns out that our algorithm is better than Jikes concurrent collector in almost all tests, and that its throughput does not fall below 80% of Jikes STW collector's on most of the tests. These measurements do not serve much more than a sanity check since the compared collectors are also not targeted at running on a client machine.

## 3.7 Conclusions

We presented a novel on-the-fly mark-and-sweep garbage collector with low latency and high throughput. We have implemented our collector on Jikes Research JVM running on a 4-way IBM Netfinity server and compared the behavior of our collector with Jikes stop-the-world collector and Jikes concurrent collector (both supplied with Jikes JVM package). Comparisons to Jikes stop-the-world collector show that the pauses have been reduced by a factor of at least 200. The longest pause measured between all runs of our collector was 2ms. When comparing the throughput with the stop-the-world collector, we see an anticipated reduction of throughput of around 10%. Comparing to Jikes concurrent collector, we see that the pauses became shorter and the throughput has improved in almost all cases.

# Chapter 4

# Age-Oriented Concurrent Garbage Collection

## 4.1 Introduction

Dynamic memory management and garbage collection is arguably a key factor in supporting fast and reliable large software products. However, naive garbage collection algorithms may have undesirable effects on program behavior, most notably long pauses and reduced throughput. Generational garbage collection [64, 97] ameliorates both problems by reducing the average pause times and increasing efficiency. The basic assumption underlying generational collectors design is the weak generational hypothesis: "most objects have short lifetimes". Given this hypothesis, it was suggested to concentrate the effort on young objects which are most likely to be unreachable. Generational collectors segregate objects according to their age into two or more groups called generations, and run frequent collections of the young generation. Keeping the young generation small yields frequent short collections that make room for further allocations. The older generation (or the entire heap) is collected infrequently when space is exhausted. Full heap collections require long pauses, but are infrequent.

If the generational hypothesis is indeed correct, we get several advantages. First, reducing pauses is achieved for most collections because the collections are short. Second, collections are more efficient since they concentrate on the young part of the heap where a high percentage of garbage is found. Finally, the working set size is smaller both for the program (as

it repeatedly reuses the young area) and for the collector (as most of the collections trace over a smaller portion of the heap).

In this work, we propose a new way, called *age-oriented* collection, to better exploit the generational hypothesis with concurrent and on-the-fly garbage collectors, especially when a reference-counting collector is employed[1]. Concurrent collectors already achieve short pause times and therefore the main interest in using the generational hypothesis is to try and improve the application throughput. An age-oriented collector is defined as follows.

**Definition 1:** *An* age-oriented *collector is a collector that*

1. *always collects the entire heap (unlike generational collectors),*
2. *during a collection it treats each generation differently (like generational collectors).*

Age-oriented collectors differ from generational collectors because the entire heap is always collected (infrequently). Like the generational framework, an age-oriented collector may be instantiated in various ways, depending on the choice of collector for the young generation and the choice of collector for the old generation. Assuming that the generational hypothesis holds, reasonable instantiations should handle the young generation with a collector that is efficient with a high death rate, and handle the old generation with a collector that is efficient with lower death rates. In particular, our flagship instantiation of the generic age-oriented collector employs reference counting for the old generation and mark and sweep for the young generation. The complexity of reference counting is proportional to the number of pointer updates and the amount of unreachable space. Therefore, it can handle huge live spaces efficiently. Mark-and-sweep benefits from a high death rate since its complexity bottleneck is the scanning of the live objects. Avoiding the sweep by using copying collectors may be even better for the young generation, but concurrent versions of copying collectors are not easy to obtain. Hence our collector is a non-moving one, in which generation membership is logical (not physical), i.e., we keep a bit per object, indicating whether the object is young or old.

One other instantiation that we have tried (and is now included in JMTk) is a parallel age-oriented collector denoted *copyMS*, employing mark and sweep for the old generation, and copying for the young generation. In this work, we focus on the use of *concurrent* reference-counting age-oriented collectors, which was most successful in practice.

We build on three previous on-the-fly collectors.

[1]This work was presented in [81].

1. The on-the-fly reference-counting collector of Levanoni and Petrank [62, 63].

2. The on-the-fly mark-and-sweep collector of Azatchi et al. [4].

3. The *generational* on-the-fly collector of Azatchi and Petrank [5] that uses collector (2) for the young generation and collector (1) for full heap collections.

The third (generational) collector incorporated the first two collectors in a generational manner, and it outperformed the original collectors. In this work, we also employ the first and second collectors, but we combine them in an age-oriented manner.

The measurements (in Section 4.6) show that the age-oriented collector significantly outperforms not only the original Levanoni-Petrank collector, but also the more efficient generational collector [5]. This indicates that an age-oriented collector is the most efficient way known today to employ reference counting.

**Chapter organization.** In Section 4.2, we introduce the age-oriented framework and our proposed instantiation. An overview of the age-oriented collector algorithm is introduced in Section 4.3. The pseudo-code of the age-oriented collector appears in Section 4.4. Implementation and results are given in Sections 4.5 and 4.6. We conclude in Section 4.7.

## 4.2 Age-Oriented Collection: Motivation and Overview

Generational collectors reduce pauses and improve efficiency by frequently invoking young generation collections. Frequent initiation of young generation collections also imposes some overhead, as it repeatedly involves synchronization with the program threads, marking of all the objects referenced by roots, etc. Using a large young generation implies less frequent collections and better throughput, but also longer pauses (for young generation collections). In particular, as noted Blackburn et al. [13] the generational collector presented by Appel [2], where all the free space is devoted to the young generation, is the best performing generational configuration. Hence, the size of the young generation determines the trade-off between the collector's efficiency and the length of its pause time.

Previous concurrent generational collectors [35, 5] have used a fixed sized young generation. Using a small fixed sized young generation is useful for the stop-the-world framework as it shortens most pause times. However, the size of the young generation does not determine the pause times with concurrent collectors[2]. Concurrent collectors run concurrently with

[2]We normally measure pauses induced by concurrent collectors when the number of program threads is

the program threads and induce very short pauses. Thus, the motivation for incorporating generations is focused at improving the throughput. Hence, we would like to use the largest possible young generation in order to achieve best throughput.

An observation regarding reference counting follows. There is a difference between using tracing and using reference counting to collect the old generation. Tracing collection work is proportional to the number of reachable objects, hence there is a (relatively) fixed cost for each full collection. Delaying a tracing collection of an old generation as far as possible is desirable as it decreases the accumulated garbage collection work. However, reference-counting work is proportional to the mutators' work and to the number of dead objects. This work is accumulative. Thus, delaying a reference-counting collection does not decrease the overall garbage collection work (it only delays and accumulates it).

Putting the above together, we get that a good way to use reference counting with the old generation is via an age-oriented collector. First, when running a collection on-the-fly we look for improving efficiency, and thus for the largest young generation. Second, whereas a generational collector collects the young generation repeatedly in order to defer as much as possible the collection of the old generation, an age-oriented collector does not make such a deferring attempt. Therefore, the largest young generation is obtained. When reference counting is used, not delaying the collection of the old generation does not hurt the throughput.

To summarize this motivational discussion with an overview, we instantiate the age-oriented generic collector by choosing reference counting for the old generation and mark and sweep for the young generation. We build on the previous generational collector of [5]. The underlying techniques come from [62, 63].

## 4.3 The Age-oriented collector

This section presents our instantiation of the age-oriented collector. The pseudo-code is provided in Section 4.4. Our age-oriented collector extends the reference-counting collector of [62, 63] (reviewed on Chapter 2.6) by using it for the old generation and adding the sliding-views tracing collection (presented in Chapter 3) for the young generation.

The original reference-counting collector of [63] iterates over all the young objects recorded

smaller than the number of CPU's. If the number of threads exceeds the number of processors, then large pauses are induced by threads losing the CPU to one another. The lengths of such pauses depend on the operating system scheduler and are not attributable to the garbage collector.

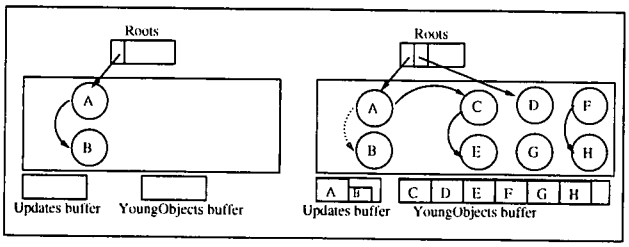



Figure 4.1: An example: heap and buffers view in 2 subsequent collections.

in the *YoungObjects* buffer, incrementing the reference counts of their descendants, only to find out later that most of them are dead (assuming the weak generational hypothesis). Thus, it then decrements the reference counts of all their descendants (before deleting them). The source of this inefficiency is that the collector does not know in advance which of the young objects are dead, and which are reachable. The age-oriented collector avoids this problem by detecting the roots of the young generation and tracing only the small number of reachable young objects, updating the reference counts of reachable young objects and their descendants during the trace. The surviving young objects are promoted to the old generation.

The main phases of the age-oriented collector (ignoring on-the-fly issues, such as the fact that the objects graph is modified during the collector's work) are presented in Figure 4.2, while these of the original reference-counting collector are presented in Figure 4.3. The difference between the two collectors is reflected from theses figures. As with generational collectors, one needs to identify all young objects directly referenced by the program roots and by old objects. We denote these objects *youngGenerationRoots*. The age-oriented collector obtains these roots from the information collected by the original collector:

1. Young objects referenced by the program roots and young snooped objects are obtained when iterating through the roots and through the snooped buffer at the beginning of each collection.

2. An old object that references a young object must have been modified after the previous collection, as the young object did not exist earlier. All modified objects are logged in the *Updates* buffer. Hence, young objects directly referenced by old objects are detected while adjusting reference counts that are due to the *Updates* buffers: each young object whose $rc$ is incremented is added to the *youngGenerationRoots*.

After locating the roots, the tracing of the young generation uses the current sliding views as explained in Chapter 3: the descendants of a traced object are determined according to the current sliding view. If the object is not dirty, then we may read the heap to find its

```
1.  Roots := programRoots ∪ SnoopedObjects
2.  youngGenerationRoots := YoungObjects ∩ Roots
3.  for each object logged in Updates do
4.      - decrement rc of its previous sliding-view descendants
5.      - increment rc of its current sliding-view descendants, while adding
          young objects whose rc is incremented into youngGenerationRoots
6.  trace young objects reachable from youngGenerationRoots, while
7.      incrementing the rc of each object traced
8.  reclaim young objects with zero rc which do not belong to Roots
9.  reclaim old objects with zero rc which do not belong to Roots recursively
```

Figure 4.2: Age-Oriented: Collection Cycle

```
1.  Roots := programRoots ∪ SnoopedObjects
2.  for each object logged in Updates do
3.      - decrement rc of its previous sliding-view descendants
4.      - increment rc of its current sliding-view descendants
5.  for each object logged in YoungObjects do
6.      increment rc of its current sliding-view descendants
7.  reclaim objects with zero rc which do not belong to Roots recursively
```

Figure 4.3: Sliding views: Collection Cycle

old descendants. However, if it was modified since the beginning of the current collection (and is dirty) we use its recorded slots in the update buffers, representing its state in the current sliding view. For each traced object, the $rc$ of its descendants is incremented, and each descendant that is young and not yet traced, is traced recursively. When the trace terminates, each object marked as *Roots* or having a non-zero $rc$ is considered live, and we can safely reclaim each (young or old) object which does not fulfill either of these conditions. Dead young objects are freed via sweep on the *YoungObjects* buffer and dead old objects are freed as usual by recursive freeing of the reference-counting algorithm.

**Example.** We use Figure 4.1 to present the principles of the age-oriented collector. The previous sliding view is depicted on the left side, and the current sliding view is depicted on the right. The roots are depicted above the heap and the old generation (containing $A$ and $B$)

is visibly separated on the left side of the heap from the young generation, which is depicted on the right side of the heap. When the age-oriented collector scans the objects logged in the *Updates* buffer (line 3 in the pseudo-code of Figure 4.2), it finds $A$. It decrements the reference count of $B$, its descendant in the previous sliding view (line 4 in the pseudo-code), and increments the reference count of $C$, its current sliding-view values (line 5). The incremented values that belong to the young generation ($C$) are considered roots for the young generation tracing (line 5). An additional young generation root is $D$ which is directly referenced by the program roots (line 2). Hence, the age-oriented collector traces the young generation from $C$ and $D$ (line 6). In comparison, the original reference-counting collector would have iterated over the six young objects incrementing the reference counts of their current sliding view, only to find out later that the work spent on $F$, $G$, and $H$ was wasted.

### 4.3.1 Properties

The age-oriented collector employs the original algorithm's write barrier with no modifications, as it fits naturally to the new age-oriented collector. No extra treatment of inter-generational pointers is required during the program run. The roots for the collection (of the young generation) are young objects that are referenced by old objects (as reflected by the *Updates* buffer) and young objects which are marked *Roots*.

As with any reference-counting collector, this age-oriented algorithm cannot reclaim cyclic data structures in the old generation (cyclic structures in the young generation are collected immediately). To reclaim such structures, the tracing sliding-view algorithm of [4] is run infrequently on the full heap. Its use is rare since cyclic structures in the young generation are collected immediately. An alternative approach is presented by Paz et al. [80].

Since the collectors we build on [62, 63, 4] do not move objects, the partitioning to young and old generations is logical (as in [28, 35, 5]). A bit per object indicates whether the object is young or old. Our promotion policy is naive: any young object that survives a collection is considered old in the next collection.

The new collector retains the characteristics of the original collector. In particular, it is adequate for a multithreaded environment and a multiprocessor platform, and it retains the short pauses of the original collectors. The measurements (in Section 4.6) show that the age-oriented collector outperforms the original Levanoni-Petrank collector. Comparison against a similar generational collector [5] demonstrates superiority in this case too. This provides a good indication that the age-oriented framework is effective.

### 4.3.2 A race condition

The use of the above strategy in the Levanoni-Petrank on-the-fly collector is susceptible to a race condition. To form the sliding views without stopping the mutators simultaneously, the collector reads the mutators' buffers (*YoungObjects* buffers and *Updates* buffers) while the mutators run. Thus, while the buffers are being read by the collector from one thread, objects are being created and modified by other (program) threads. The problem happens during the first handshake, when a new object is created by a mutator during the time the collector is collecting the local buffers from the mutators one by one.

Here is the problematic scenario. Suppose a collection is started and the collector reads the *YoungObjects* (and *Updates*) buffers of mutator $T_1$. After that, any new object created by $T_1$ is considered young for the next collection, and not for the current one. There is no danger in erroneously reclaiming this object; however, a specific race may cause its *descendants* to be erroneously reclaimed in the *next* collection. Suppose that $T_1$, after delivering its buffers to the collector in the first handshake, creates a new object $O_1$ and logs it in its $YoungObjects$ buffer. This log will be read only in the next garbage collection. Suppose also that another thread $T_2$, whose buffers were not yet read by the collector, modifies an old object $O_2$ to reference $O_1$. The record of this modification is logged into $T_2$'s *Updates* buffer and is read (processed) in the *current* collection (this record will not appear in the next collection). Note the source of confusion: because $T_2$'s buffers are read after $T_1$'s buffers are read, $O_1$ is new in the next collection but the creation of a pointer to it from an old object is processed by the collector in the current collection. During the current collection, the reference count of $O_1$ will be incremented and it will not be collected until this reference count is properly decremented. However, we expect newly created objects that are referenced from the old generation to be traced. Tracing is normally triggered by the appearance of a created object which is referenced by an object logged in the *Updates* buffer, signifying an inter-generational pointer. But with the race described, the record that was created in the *Updates* buffer is already processed in the current collection and will not appear again in the *Updates* buffer when the next collection starts and the object $O_1$ will be processed as a new object. Due to this race the descendants of object $O_1$ will not be traced in the next collection, which may lead to a premature reclamation of objects. Consider, for example, a scenario where an object $O_3$ is created after the current collection is over (and before the next collection begins), and then $O_1$ is modified to reference $O_3$. Since $O_1$ is dirty (new objects are created dirty), it will not be logged onto the *Updates* buffer. The current collection does not deal with $O_3$ as it does not exist during the collection. In addition, the next collection will not reach $O_3$ during the young generation tracing (assuming it is solely referenced by $O_1$), since

due to the described race condition $O_1$ will not be traced in the next collection. Hence $O_3$ would be improperly reclaimed in the next collection.

To ensure that all necessary tracing is executed properly (so that no live object is collected), we add a specific treatment for objects with the above properties. These objects are identified by being young objects with a non-zero reference count in the beginning of a collection (due to a race in the previous collection). Note that this can only happen because of this race. Normally, a young object has a zero reference count as its reference count was not updated in previous collections. Thus, during the traversal of the *YoungObjects* buffer, which clears the dirty bits of the young objects (during the Clear-Dirty-Marks procedure presented in Section 4.4.5), we check if a young object has a non-zero reference count and if so, it will be traced later during the collection. From correctness perspective this check may be run at any time after the race is no longer possible (i.e., after the second handshake of the previous cycle) and before updating rc's of young objects in the current collection cycle.

## 4.4 The Garbage Collector Details

In this section, pseudo-code and explanations regarding the new age-oriented collector are provided. In order to stress the additions to the original reference-counting collector, we add an asterisk to any line in the age-oriented algorithm code that differs from the original Levanoni-Petrank collector [62, 63].

### 4.4.1 The LogPointer

The original reference-counting algorithm, as well as the new age-oriented collector, requires maintaining a dirty bit signifying whether an object has been modified since the most recent collection started. During the first modification of an object in a cycle, its pointers are recorded in the *updates* buffer and its dirty bit is set. As detailed in Chapter 3.3.1, we chose to dedicate a full word to keep the dirty bit. This word serves as a pointer into the thread's local buffer where this object's pointers have been logged. We call this word the *LogPointer*. A zero value (a null pointer) signifies that the object is not dirty (and not logged). A non-zero *LogPointer*'s value signifies that the object is dirty; in this case, the *LogPointer*'s value references to the log entry of this object (containing the object's sliding-view pointers' values). That way, when the collector needs to know the sliding-view slot values of a dirty object, it does not have to perform a sequential search of this object in the *updates* buffer:

```
Procedure Update(obj: Object, offset: int, new: Object)
begin
1.   if obj.LogPointer = NULL then     // OBJECT NOT DIRTY
2.      TempPos := CurrPosU_i
3.      // TAKE A TEMPORARY REPLICA OF THE OBJECT
4.      foreach field ptr of obj which is not NULL
5.         Updates_i[TempPos++] := ptr
6.      if obj.LogPointer = NULL then     // IS IT STILL NOT DIRTY?
7.         // ADD POINTER TO OBJECT AND AN IDENTIFYING BIT
8.         Updates_i[TempPos++] := address of obj | 0x1
9.         CurrPosU_i := TempPos     // COMMITTING THE REPLICA
10.        obj.LogPointer := address of Updates_i[CurrPosU_i]     // SET DIRTY
11.  write( obj, offset ,new)
12.  if Snoop_i and new != NULL then
13.     Snooped_i := Snooped_i ∪ { new }
end
```

Figure 4.4: Mutator code: Update Operation

it simply follows the *LogPointer* reference (see for example Figure 4.12, which is described later).

### 4.4.2 Main data structures

The following pseudo-code employs the following buffers:

- The threads' local $Updates_i$ buffers holding all the (non-young) objects which have been modified for the first time (since the previous collection) by a thread. The modified objects are logged together with their previous sliding-view pointers' values (i.e., the values before the object's first modification). During a collection, all the local $Updates_i$ buffers are accumulated into the collector's *Updates* buffer.

- The threads' local $YoungObjects_i$ buffers holding all the objects which have been created been by a thread since the previous collection. During a collection, all the local $YoungObjects_i$ buffers are accumulated into the collector's *YoungObjects* buffer.

```
Procedure New(size: Integer, obj: Object)
begin
1.  Obtain an object obj of size size from the allocator.
2.  YoungObjects_i[CurrPosY_i++] := address of obj
3.  obj.LogPointer = address of YoungObjects_i[CurrPosY_i]
4.  return obj
end
```

Figure 4.5: Mutator code: Allocation Operation

- The threads' local $Snooped_i$ buffers holding all the objects snooped by a thread.
- The threads' local $State_i$ buffers holding the objects referenced by a thread's roots.
- The collector's *Roots* buffer holding the objects referenced by a collection's roots.
- The collector's *ClearConflictSet* buffer holding objects whose *LogPointer* may have cleared by mistake (due to race conditions). We'll elaborate on this buffer later.
- The collector's *markStack* buffer used for the tracing of the young generation.
- The collector's *ZCT* (zero-count table) buffer holding the objects whose reference count field reached zero during the collection. After all reference-count adjustments are made, the collector traverses this buffer, reclaiming all objects which still have a zero reference-count field (and are not referenced by the program roots).

In addition, each object posses the following fields:

- The *LogPointer* field, described in Section 4.4.1.
- The *rc* field, holding the object's reference count.
- The boolean *live* field, which indicates whether an object is currently considered by the collector as live.

Other than that, some other variables are used in the following pseudo-code:

- The thread's local boolean $Snoop_i$ flag, indicating whether the write barrier should log objects, to which a new reference is added, into the $Snooped_i$ buffer.

```
Procedure Collection-Cycle
begin
1.   Initiate-Collection-Cycle      // 1ST HANDSHAKE
2.   Clear-Dirty-Marks
3.   Reinforce-Clearing-Conflict-Set      // 2ND AND 3RD HANDSHAKE
4.   Mark-Roots    // 4TH HANDSHAKE
5.   Update-Old-Reference-Counters
*6.  Trace-Young
*7.  Reclaim-Young-Garbage
8.   Reclaim-Old-Garbage
9.   Prepare-Next-Collection
end
```

Figure 4.6: Age-oriented collector code- Collection Cycle

- Each $Updates_i$ buffer maintains a $CurrPosU_i$ counter and each $YoungObjects_i$ buffer maintains a $CurrPosY_i$ counter. These counters indicate the number of addresses already logged into the buffer.
- A temporary variable, *TempPos*, used by the write barrier.

### 4.4.3 Mutator cooperation

The mutators need to execute garbage-collection related code on three occasions: when updating an object, when allocating a new object and during handshakes. This is accomplished by the Update (Figure 4.4) procedure, the New (Figure 4.5) procedure and the handshake mechanism, respectively. The Update and New operations never interleave with a handshake. Namely, cooperation with a handshake waits until a currently executed Update or New operation finishes. The Update and New operations are exactly equal to those used in the original sliding-views reference-counting algorithm [63].

**Procedure Update** (Figure 4.4) is activated at pointer assignment and its main task is to record the object whose pointer is modified (i.e., log objects' values at the sliding views). We stress that the write barrier (the Update protocol) is only used with heap pointer modification. Modifications of local pointers in the registers or stack are not monitored. Going through the pseudo-code, we see that each object's *LogPointer* is optimistically probed

```
Procedure Initiate-Collection-Cycle
begin
1.    for each thread T_i do
2.       Snoop_i := true
3.    for each thread T_i do      // FIRST HANDSHAKE
4.       suspend thread T_i
5.       Updates := Updates ∪ Updates_i     // COPY (WITHOUT DUPLICATES).
6.       Updates_i := ∅      // CLEAR BUFFER.
7.       YoungObjects := YoungObjects ∪ YoungObjects_i
8.       YoungObjects_i := ∅      // CLEAR BUFFER.
9.       resume thread T_i
end
```

Figure 4.7: Age-oriented collector code- Initiate-Collection-Cycle

twice (lines 1 and 6) so that if the object is dirty (which is often the case), the write barrier is extremely fast. If the object was not logged (i.e., the LogPointer of an object is NULL) then after the first probe, the objects' values are recorded into the local $Updates_i$ (lines 3-5). The second probe at line 6 ensures that the object has not yet been logged (by another thread). If *LogPointer* is still NULL (in the second probe), then the recorded values are committed as the buffer pointer is modified (line 9). In order to be able to distinguish later between objects and logged values, in line 8 we actually log the object's address with the least significant bit set on (while values are logged with least significant bit turned off). Then, the object's *LogPointer* field is set to point to these values (line 10). After logging has occurred, the actual pointer modification happens. Finally, from the time a collection begins until marking the roots of the mutators, the snoop flag is on. At that time, the new target of the pointer assignment is recorded in the local $Snooped_i$ buffer (lines 12-13). The variables $Updates_i$, $CurrPosU_i$, $Snoop_i$ and $Snooped_i$ are local to the thread.

The Update protocol described above may cause a noticeable pause when applied to objects containing a large number of pointers. To bound the length of such pauses, a large object could be divided into fixed-size cards, so that each card would posses its own LogPointer. Upon write-barrier activation over such a large object, only the pointers related to the relevant card should be copied into the relevant $Updates_i$ buffer. That way the write-barrier length could be controlled. However, we have not implemented such solution.

```
Procedure Clear-Dirty-Marks
begin
1.    for each object obj ∈ Updates do
2.       obj.LogPointer := NULL
3.    for each object obj ∈ YoungObjects do
4.       obj.LogPointer := NULL
*5.      if obj.rc > 0 then
*6.         obj.live := true
*7.         push obj onto markStack
end
```

Figure 4.8: Age-oriented collector code- Clear-Dirty-Marks

We do not further elaborate on the properties of the write barrier, on why it works in a multithreaded environment, etc. We focus on the modifications required to obtain an age-oriented collector. A thorough discussion of the write barrier appears in [63].

**Procedure New** (Figure 4.5) is used when allocating an object. After Thread $T_i$ creates an object, the object's address is logged into $YoungObjects_i$, a (mutator) local buffer. This buffer contains the addresses of the objects which would be considered as young objects in the next collection cycle. This logging will tell us which objects are young. For the reference counting this means that reference counts of descendants of these objects (if they survive the next collection) must be updated. There is no need to record their children slot values as they are all null at creation time. The LogPointer of a newly allocated object is modified to record its log address (so that future assignments to this object won't activate the write barrier and log it again in the *updates* buffer).

**Our handshake mechanism** is the same as the one employed by the Doligez-Leroy-Gonthier collector [34, 33]. The mutator threads are never stopped simultaneously for cooperating with the collector. Instead, threads are suspended one at a time for the handshake. The stopping of the thread is not allowed while it is executing the write barrier or while it is creating a new object. While a thread is suspended, the collector executes the relevant actions for the handshake and then the thread is resumed. The collector repeats this process until all threads have cooperated. At that time, the handshake is completed.

```
Procedure Reinforce-Clearing-Conflict-Set
begin
1.   ClearConflictSet := ∅
2.   for each thread T_i do      // SECOND HANDSHAKE
3.     suspend thread T_i
4.     ClearConflictSet := ClearConflictSet ∪ Updates_i[1 ... CurrPosU_i]
5.     resume thread T_i
6.   for each object obj ∈ ClearConflictSet do
7.     if obj.LogPointer = NULL then      // NEED TO REINFORCE
8.       obj.LogPointer := address of obj's replica in Updates_i
9.   for each thread T_i do      // THIRD HANDSHAKE- EMPTY HANDSHAKE
10.    suspend thread T_i
11.    resume thread T_i
end
```

Figure 4.9: Age-oriented collector code- Reinforce-Clearing-Conflict-Set

### 4.4.4 Phases of the collection

The collector algorithm runs in phases as follows.

- **Start snooping:** raising the $Snoop_i$ local flag of each mutator, which activates the snooping mechanism.
- **First handshake:** during this handshake each mutator is stopped and its log buffers ($YoungObjects_i$ and $Updates_i$), are accumulated by the collector.
- **Clear dirty marks:** The collector clears the dirty marks of all objects previously recorded in the buffers.
- **Second handshake:** during this handshake each mutator is stopped in order to reinforce logs (to its current $Updates_i$ buffers) which were cleared during the clear dirty marks phase.
- **Third handshake:** no operation (empty) handshake to make sure that the proper dirty marks are visible by all mutators.

```
Procedure Mark-Roots
begin
1.    for each thread T_i do      // FOURTH HANDSHAKE
2.       suspend thread T_i
3.       Snoop_i := false
4.       Roots := Roots ∪ State_i      // COPY THREAD LOCAL STATE.
5.       resume thread T_i
6.    for each thread T_i do
7.       // COPY AND CLEAR SNOOPED OBJECTS SET
8.       Roots := Roots ∪ Snooped_i
9.       Snooped_i := ∅
*10.  for each object obj ∈ Roots do
*11.     if obj.live = false then
*12.        obj.live := true
*13.        push obj onto markStack
end
```

Figure 4.10: Age-oriented collector code- Mark-Roots

- **Fourth handshake:** during this handshake each mutator is stopped and the objects reachable from the local roots of each mutator are marked. Also, the $Snoop_i$ flag is cleared.
- **Update reference counts of old objects:** the collector adjusts the $rc$ fields of old objects' descendants.
- **Trace young objects:** the collector traces the live young objects (while incrementing their $rc$).
- **Reclaim young garbage:** the collector reclaims dead young objects. Reclamation in this phase is more efficient than a reference-counting reclamation, as no recursive deletion is required.
- **Reclaim old garbage:** the collector reclaims old objects (and their descendants using recursive deletion) which have a zero $rc$ and which are not referenced by the system roots.

```
Procedure Update-Old-Reference-Counters
begin
1.    for each object obj whose replica rep in Updates do
2.      // DECREMENT PREVIOUS REFERENT OF THE OBJECT obj
3.      for each slot s in the replica of rep do
4.        previous-value := read(s)
5.        previous-value.rc--
6.        if previous-value.rc = 0 then
7.          add previous-value to ZCT
8.      // INCREMENT REFERENCE COUNT OF SLIDING-VIEW DESCENDANTS
9.      Increment-Descendants-RC(obj)
end
```

Figure 4.11: Age-oriented collector code- Update-Old-Reference-Counters

- **Prepare next collection:** prepares the buffers for the next collection.

### 4.4.5 Collector code

Collector's code for cycle $k$ is presented in **Procedure Collection-Cycle** (Figure 4.6). Let us briefly describe each of the collector's procedures.

**Procedure Initiate-Collection-Cycle** (Figure 4.7) raises first the (local to mutator) $Snoop_i$ flag, signaling the mutators that they should start snooping all stores into heap slots. Then the first handshake is carried out to gather the local buffers of the threads.

**Procedure Clear-Dirty-Marks** (Figure 4.8) clears all dirty marks set by mutators prior to responding to the first handshake. This stage takes place while the mutators are running. When handling the *YoungObjects* buffer, each object with a non-zero *rc* is also pushed onto *markStack*. Objects pushed onto *markStack* would be traced later. A young object can have a non-zero *rc* due to a race condition (as explained in 4.3.2). Other young objects (which have a zero *rc*) are optimistically considered dead (and thus are not marked). Lines 5-7 (marked with asterisk) are relevant only to the age-oriented collector, as the original algorithm treats young objects similarly to old objects.

**Procedure Reinforce-Clearing-Conflict-Set** (Figure 4.9) implements the reinforcement

```
Procedure Increment-Descendants-RC(obj: Object)
begin
1.    if obj.LogPointer = NULL then     // IF OBJECT HAS BEEN MODIFIED
2.       // NO - READ ITS DESCENDANTS FROM HEAP.
3.       replica := copy(obj)
4.       // CHECK AGAIN IF COPIED REPLICA IS VALID.
5.       if obj.LogPointer != NULL then
6.          // OBJECT HAS BEEN MODIFIED WHILE BEING READ.
7.          // GET REPLICA FROM BUFFERS.
8.          replica := getOldObject(obj.LogPointer)
9.    else     // OBJECT WAS MODIFIED. USE BUFFERS TO OBTAIN REPLICA.
10.      replica := getOldObject(obj.LogPointer)
11.   for each slot s in replica of obj do     // INCREMENT DESCENDANTS' RC
12.      curr := read(s)
13.      curr.rc++
*14.     // IF DESCENDANT IS YOUNG, IT SHOULD BE TRACED.
*15.     if curr.live = false then
*16.        curr.live := true
*17.        push curr onto markStack
end
```

Figure 4.12: Age-oriented collector code- Increment sliding-view values

step and assures that it is visible to all mutators. A second handshake takes place, during which thread buffers are read into $ClearConflictSet$. Then, $LogPointers$ of objects logged into $ClearConflictSet$ are reinforced to point to their current position in $Updates_i$. Finally, the third handshake of the cycle takes place with no action in it. The reason for that handshake is that a thread can fall behind another thread by at most one handshake[3]. Thus, threads that have responded to the fourth handshake will not be interfered by operation carried out by threads during the clearing or reinforcement stages.

**Procedure Mark-Roots** (Figure 4.10) carries out the fourth and last handshake during

[3]In real implementation on Jikes this scenario cannot happen as explained in [5], and thus, we do not have this empty handshake in our Jikes implementation.

```
Procedure Trace-Young
begin
*1.  while markStack is not empty
*2.      obj := pop(markStack)
*3.      Increment-Descendants-RC(obj)
end
```

Figure 4.13: Age-oriented collector code- Trace-Young

```
Procedure Reclaim-Young-Garbage
begin
*1.  for each object obj ∈ YoungObjects do
*2.      if obj.live = false then
*3.          return obj to the general purpose allocator.
end
```

Figure 4.14: Age-oriented collector code- Reclaim-Young-Garbage

which the local $Snoop_i$ flag is turned off and the objects referenced by thread local roots are accumulated into the *Roots* (global) buffer (the set of objects directly reachable from thread $T_i$ is denoted $State_i$). Next, the $Snooped_i$ buffer of each thread (containing snooped objects), is accumulated into *Roots*, and then cleared (for the next collection). Thus, during this procedure, the true root set of this collection cycle is being marked.

Lines 10-13 present a code for marking roots of the young generation. It is relevant only to the age-oriented collector: each young object belonging to *Roots*, which was considered until now as dead (i.e., has a zero *rc*, and thus was not marked *live* in procedure Clear-Dirty-Marks), is marked *live* (as it belongs to the roots set and thus should be treated as live) and is pushed onto the *markStack* (so it would be traced later). The *live* mark also indicates that the object has been promoted into the old generation.

**Procedure Update-Old-Reference-Counters and procedure Increment-Descendants-RC** (Figures 4.11-4.12) adjust reference counts corresponding to the modified objects. It examines the *Updates* buffer's objects which are all objects modified since the previous collection cycle. The *rc* of their children slot values in the previous sliding view are decremented, whereas the *rc* of their current children slot values are incremented. During *rc* adjustments, every object's whose *rc* is decremented to 0, is inserted into the *ZCT* (zero-count table).

```
Procedure Reclaim-Old-Garbage
begin
1.    for each object obj ∈ ZCT do
2.       if obj.rc > 0 ∨ obj ∈ Roots then
3.          ZCT := ZCT − {obj}
4.    for each object obj ∈ ZCT do
*5.      obj.live := false
6.       Collect(obj)
7.    ZCT := ∅
end
```

Figure 4.15: Age-oriented collector code- Reclaim-Old-Garbage

The procedure Increment-Descendants-RC (Figure 4.12) performs $rc$ increments of current sliding-view children slot values of a given object. An object may be modified by mutators while the replica is taken. In such case, its children slots at the current sliding view can be found by looking at the current collection cycle log entry which is pointed by the dirty flag (the *LogPointer* points to the logging location of this object). In lines 14-17 (related only to the age-oriented algorithm), each object (whose $rc$ was just incremented) which is not marked *live*, is marked *live* and pushed onto *markStack*. These are young object for which we have found evidence of being live only now.

**Procedure Trace-Young** (Figure 4.13) traces the live young objects. The roots of this tracing are the young objects, located in *markStack*, for which we have found evidence of being live. These objects are traced, while we also increment the $rc$ of their current sliding-view slots values (using the procedure Increment-Descendants-RC presented in Figure 4.12). No decrement is needed as these objects were created since the last collection.

This procedure is related only to the age-oriented algorithm, as the original algorithm processes the increments related to all the young objects.

**Procedure Reclaim-Young-Garbage** (Figure 4.14) releases the young objects not marked *live*. Note that releasing these objects is more efficient than a reference-counting reclamation, since there is no need to decrement the $rc$ of theirs descendant as done in traditional reference-counting systems (and thus no recursive deletion is needed).

**Procedure Reclaim-Old-Garbage** (Figure 4.15) releases unreachable old objects (and

```
Procedure Collect(obj : Object)
begin
1.    if obj.LogPointer != NULL then
2.          // obj WAS MODIFIED WHILE THE SLIDING-VIEW WAS TAKEN
3.       replica := obj.LogPointer
4.       Recursive-Deletion(replica)
5.       Invalidate-Log-Entry(obj.LogPointer)
6.          // SO IT WILL BE IGNORED IN THE NEXT COLLECTION
7.    else
8.       Recursive-Deletion(obj)
9.    return obj to the general purpose allocator.
end
```

Figure 4.16: Age-oriented collector code- Collect

their descendants if needed). At first, each $ZCT$ object which has a positive $rc$ field or is marked as *Roots* is deleted from the $ZCT$. Otherwise, its *live* flag is turned-off and it is collected by the **Collect procedure** and the **Recursive-Deletion procedure** (Figures 4.16-4.17), which decrement the $rc$ of an object's sliding-view values before releasing it (and performs recursive releases if necessary). After reclaiming the garbage, the $ZCT$ is cleared.

Both line 3 in Figure 4.14 and line 9 in Figure 4.16 return an object to the general purpose

```
Procedure Recursive-Deletion(obj : Object)
begin
1.    for each slot s of obj do
2.       curr := read(s)
3.       curr.rc--
4.       if curr.rc = 0 ∧ curr ∉ Roots then
5.          Collect(curr)
end
```

Figure 4.17: Age-oriented collector code- Recursive deletion

```
Procedure Prepare-Next-Collection
begin
1.   for each object obj in Roots
2.     if obj.rc = 0 then
3.       add obj to ZCT
4.   // CLEAR BUFFERS FOR NEXT COLLECTION
5.   Roots := ∅
6.   Updates := ∅
7.   YoungObjects := ∅
end
```

Figure 4.18: Age-oriented collector code- Prepare-Next-Collection

allocator (upon an object's reclamation). The meaning of this action is implementation dependant. In particular, it may include the zeroing of the space returned (as requested for Java) and/or zeroing a certain bitmap bit signifying to the allocator that the relevant chunk is now available for allocation.

**Procedure Prepare-Next-Collection** (Figure 4.18) inserts objects referenced solely by *Roots* (and thus having a zero *rc*), into the *ZCT*. In the next collection, these objects would be examined while iterating through the *ZCT*. Next, the procedure cleans the global *Roots*, *YoungObjects* and *Updates* buffers.

## 4.5 Implementation for Java

The age-oriented collector was implemented in Jikes RVM (research virtual machine) [1], a research Java virtual machine. The entire system, including the collector itself is written in Java (extended with unsafe primitives available only to the Java Virtual Machine implementation to access raw memory). In this section, we would like to point out some implementation choices that we made.

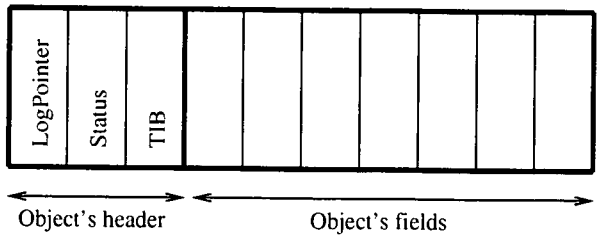



Figure 4.19: The age-oriented object model

### 4.5.1 Memory Allocator

Our implementation employs the non-copying non-coalescing allocator of Jikes, which is based on the allocator of Boehm, Demers, and Shenker [17]. This allocator is well suited for collectors that do not move objects. Small objects are allocated from per-processor segregated free-lists build from 16KB pages divided into fixed-size blocks. Large objects are allocated out of 4KB blocks with first-fit strategy. This allocator keeps the fragmentation low and allows efficient reclamation of objects.

### 4.5.2 Object-Headers

The object layout of our memory manager, within the Jikes RVM version we've used, is displayed in Figure 4.19. Jikes' basic object header contains two words[4]. One word of the header is a *status* word supporting memory management, synchronization, and hashing. The second word of the header holds a reference to the *Type Information Block (TIB)* for the object's class (which serves as Jikes' virtual method table).

In order to support reference-counting collection, our implementation employs an additional word in an object's header, which holds the *LogPointer* of an object (detailed in Section 4.4.1). In addition, we employ an additional bitmap table holding the objects' reference counts. This bitmap uses two bits to hold the reference count of an object. In case of overflow, i.e., when the reference count of an object reaches three, the reference count gets stuck so it would not be incremented or decremented until the next mark-and-sweep collection. A mark-and-sweep collection resets this bitmap, and while tracing the objects' graph it increments the reference counts of all objects encountered during the graph traversal.

[4]An array object's header includes an additional length field.

### 4.5.3 Triggering

In a stop-the-world garbage collector setting, mutators halt during a garbage collection. However, with concurrent collectors, mutators run during a collection and hence also consume memory. Therefore, when triggering a concurrent collection, our goal is on one hand to trigger a collection as late as possible so that we get as few collections as possible (to avoid garbage collection overheads). On the other hand, we would like to trigger early to ensure that the collections will complete their work before the mutators consume all available memory (otherwise, the mutators would halt waiting for the collector thread to free memory). Our triggering mechanism keeps an estimation of the work the next collection would have to deal with and an estimation of the amount of free memory available. This work estimation is based on the number of objects created since the last collection and the number of (old) objects modified since the last collection. Whenever the ratio between the amount of work and the estimated available memory goes below a certain threshold, a collection is triggered.

### 4.5.4 Root set

The root set of our collector includes the object referenced from global variables, the object referenced from static variables, the object referenced from each thread's run-time stack and the snooped objects. The pseudo-code presented in Section 4.4 refers explicitly only to the object referenced from the threads stack and to the snooped objects. In practice, we give a special treatment to global roots (the global variables and the static variables) by using a designated write barrier with them. The write barrier presented in Figure 4.4 is not invoked upon global roots modification because they are scanned on each cycle regardless of whether they have been modified or not. Therefore the designated write barrier does not mark slots dirty when they change. However, it does invoke the snooping mechanism. Thus, each new reference, written into to a global root while the snoop flag is set, makes the referent snooped. Such an object cannot be collected during the current cycle. Finally, when local states are checked during the fourth handshake, objects reachable from global roots are also marked as *Roots*.

Note that in practice our implementation does not mark the roots explicitly. Instead, in a beginning of a collection, when determining the root set (and adding the addresses of objects referenced by the roots into the *Roots* buffer), the collector incremented the reference-count field of each root object, so that it would not be reclaimed in the current collection. At the end of a collection, the collector re-traversed the root set (i.e., objects referenced by the *Roots* buffer) decrementing the reference-count field of each root object.

## 4.6 Measurements

**Platform and benchmarks.** We have taken measurements on a 4-way IBM Netfinity 8500R multiprocessor with a 550MHz Intel Pentium III Xeon processors and 2GB of physical memory. The benchmarks used were the SPECjvm98 benchmark suite and the SPECjbb2000 benchmark (described in [90]). The multithreaded SPECjbb2000 benchmark is more important, as SPECjvm98 benchmarks are mostly single-threaded and our algorithm, being on-the-fly, is targeted at multithreaded programs running on multiprocessors[5]. In this work, as well as in other recent work (see for example [6, 35]) SPECjbb2000 is the only representative of large multithreaded applications.

**Testing procedure.** We used the benchmark suite using the test harness, performing standard automated runs of all the benchmarks in the suite. Our standard automated run runs each benchmark five times for each of the JVM's involved (each implementing a different collector). Finally, to understand better the behavior of our collector under tight and relaxed conditions, we tested it on varying heap sizes. For the SPECjvm98 suite, we started with a 24MB heap size and extended the sizes by 8MB increments until a final large size of 96MB. For SPECjbb2000 we used larger heaps, starting from 256MB heap size and extending by 64MB increments until a final large size of 704MB.

**The compared collectors.** The age-oriented collector was tested against 3 collectors. First, against the original reference-counting collector [62, 63], denoted *the original collector*. Second, against the generational collector of [5], denoted *the generational collector*. And finally, against the Jikes parallel stop-the-world mark-and-sweep collector. Recall that the second (generational) collector of [5] is a collector that builds on exactly the same two collectors of [62, 63, 4], but it combines them in the standard generational manner: it performs minor concurrent collections frequently while major concurrent collections are run infrequently (whereas the age-oriented collector always collects the entire heap at each collection). Note that we have included the original collector in our measurements, to demonstrate that the age-oriented outperforms the generational collector, which is already a substantial improvement over the original collector.

### 4.6.1 Comparison with Related On-the-Fly Collectors

We start with the SPECjbb2000 benchmark, which requires multi-phased run with increasing number of warehouses (threads). We report two throughput ratios improvements: the first

[5]We also feel that there is a dire need in academic research for more multithreaded benchmarks.

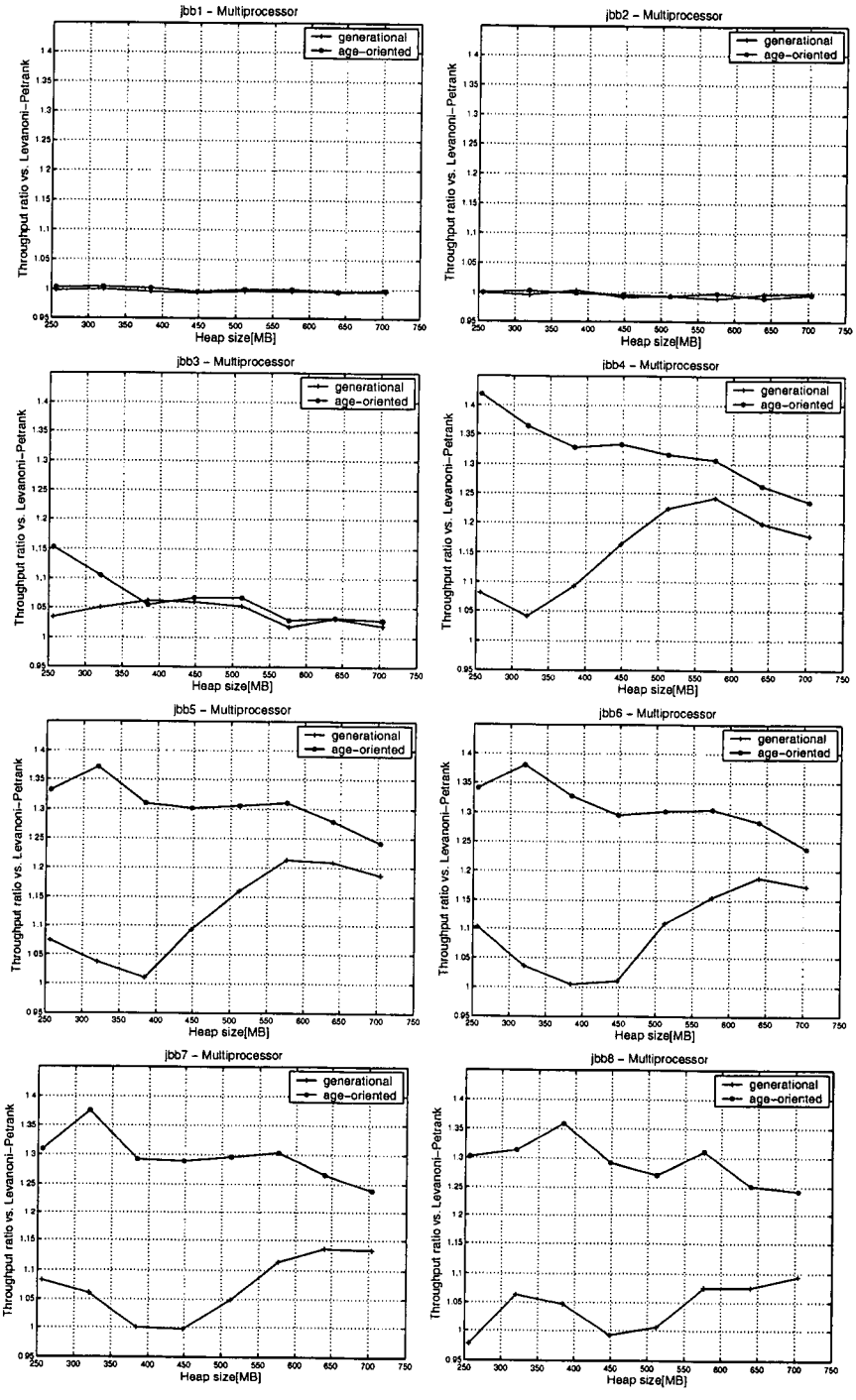



Figure 4.20: SPECjbb2000 on a multiprocessor: throughput ratio of the generational and the age-oriented collector for 1-8 warehouses (compared to the original reference-counting collector)

is the ratio of the age-oriented collector and the original Levanoni-Petrank collector, while the second is the ratio between the generational collector and the original collector. The measurements are reported for a varying number of warehouses and varying heap sizes in Figure 4.20. In each figure, we report the measurement for a specific number of warehouses, where jbb$i$ stands for running with $i$ terminals, i.e., $i$ program threads. In each figure, the higher the ratio, the better the measured collector performs compared to the original reference-counting collector. The behavior of the collectors should be separated into two cases.

The first case is with 1-3 warehouses. In this case, since our machine has four processors, any of the three on-the-fly collectors runs on a spare processor. In this setting, the collectors do not differ much. If the collectors could handle all their work while mutators are running (except for handshakes), all 3 collectors would have achieved the same throughput (as they share similar allocator and write barrier). This is indeed what we get with 1-2 warehouses. A throughput improvement (of usually 5%) begins only with 3 warehouses and especially on tight heaps, where both the generational and the age-oriented collectors outperform the original collector. The age-oriented does a little better. The reason for this improvement is that the original collector must deal now with work supplied by 3 mutators (i.e., more work for the collector), and thus mutators sometimes halt waiting for the collector to terminate its work.

The second case refers to 4-8 warehouses, where collectors do not run on a spare processor but rather share a processor with the program threads. Note nevertheless, that we gave the collector (in this case) the highest priority, so that when a collection is triggered the collector would always get a dedicated processor. Thus, when the number of warehouses is four and up, the efficiency of the collector becomes more important: a collector should not only be able to handle all its work while mutators are running, but also as the collector becomes more efficient, a collection would consume less time, thus letting mutators use a larger fraction of the fourth processor (and therefore increasing the throughput). The results show that the age-oriented collector substantially outperforms the generational collector, which already performs better than the original collector. It can be seen that the age-oriented collector usually obtains a performance improvement of 25%-40% over the original collector. As with 3 warehouses, the superiority of the age-oriented collector is usually higher with (relatively) small heaps where the collector efficiency is more significant, as more garbage collections are required. The generational collector is noticeably less efficient on tight heap, since full collections cannot be postponed much. The improvements of the age-oriented are less visible with larger heaps simply because there are fewer collections, and less time spent

on collections.

**SPECjvm98 measurements.** Figure 4.21 presents comparison of the age-oriented collector with the original collector and with the generational collector over the suite's benchmarks[6]. Here again, the higher the ratio, the better the measured collector performs compared to the original reference-counting collector. When running SPECjvm98 benchmarks on a multiprocessor, we allow a designated processor to run the collector thread. Here again the collector runs concurrently with the program thread and good concurrency is the main factor in the comparison. Results show that age-oriented collector performs slightly better (usually wins by few percentages) than both the original collector and the generational collector. However, we would like to stress that such measurements are in favor of the original collector, since it does not show that our collector consumes much less resources of the spare processor than the original collector. As the age-oriented collector is much more efficient the original collector, it collects the garbage much faster, and hence employs the dedicated processor less than the original collector. Similar measurements on a uniprocessor look much different (as will be discussed below).

Table 4.1 addresses this exact issue. We have measured for both original and age-oriented collectors the time each one actually works, i.e., the amount of time in which the spare processor is in use. Our results demonstrate that the original collector works 2.41-18.67 times more than the age-oriented collector. Since SPECjvm98 runs use a designated processor, these numbers influence the throughput mainly if the original collector could not perform all its work concurrently with the mutators (causing the mutators to stop while waiting for memory space). That is why when running SPECjbb2000 with 4 warehouses or more, our graphs show a much substantial throughput superiority: in these cases more CPU time for collector means less CPU time for mutators (on the CPU in which the collector is ran). Results of SPECjvm98, where the collector does not have a dedicated processor, are provided in subsection 4.6.5, where SPECjvm98 benchmarks are run on a uniprocessor. These results support our claim, as it shows that our collector performs much better than the original.

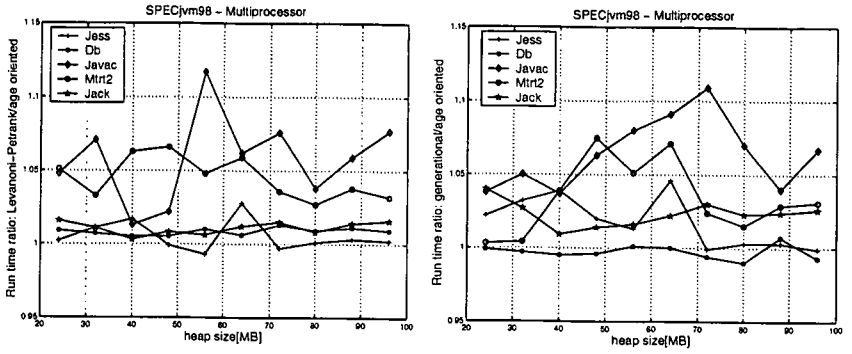



Figure 4.21: SPECjvm98 on a multiprocessor: run-time ratio of the age-oriented collector compared to the original collector (left) and compared to the generational collector (right)

| Maximum pause time(ms) | jess | db | javac | mtrt | jack |
|---|---|---|---|---|---|
| Age-oriented | 18.67 | 5.47 | 2.41 | 8.26 | 13.84 |

Table 4.1: Collector work ratio: work time ratio between the age-oriented collector and the original collector.

### 4.6.2 Comparison to a Stop-the-World Collector

Using an on-the-fly collector leads to extremely short pause times, but has a throughput cost. To measure this cost, we have compared the performance of the age-oriented collector against the Jikes parallel stop-the-world mark-and-sweep collector. In this comparison, the multithreaded SPECjbb2000 was run on a 4-way platform, and SPECjvm98 benchmarks were run on a uniprocessor. The results, appearing in Figure 4.22, show that unless the heap is tight (and then the mutators exhaust the heap before the concurrent collector is done) the overhead incurred by running the collector concurrently is up to 10%. Obtaining short pauses normally require a pay in the throughput. A 10% throughput reduction is considered a small cost for a two orders of magnitude reduction in the pause times (see pause time measurements in Section 4.6.3 below). The tight conditions highlight the advantage of parallel collectors in this setting. Parallel collectors always exploit all CPUs, while our on-the-fly collector uses only one processor while all program threads wait for free space to allocate. An exception is seen with the _213_javac benchmark. This benchmark creates cycles that are promoted to

[6]Measurements of _222_mpegaudio and _201_compress are not presented. _222_mpegaudio does not perform meaningful allocation activity. _201_compress heavily depends on a tracing collector as it creates substantial garbage cycles, so its measurements are not relevant for a comparison to a reference-counting collector.

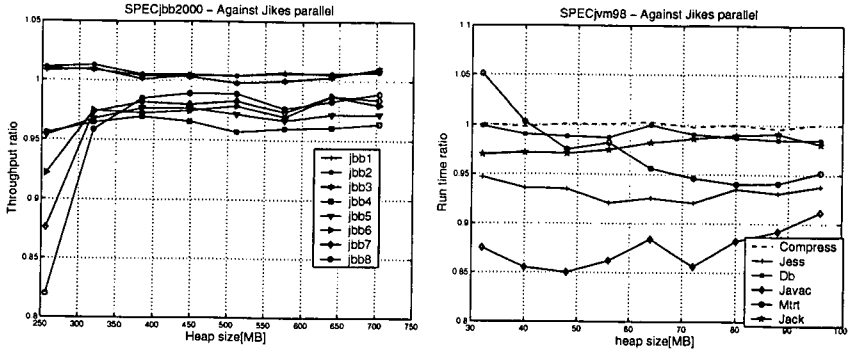



Figure 4.22: SPECjbb2000 on a multiprocessor (left) and SPECjvm98 on a uniprocessor (right): age-oriented comparison against Jikes parallel mark-and-sweep collector

| Maximum pause time | compress | jess | db | javac | mtrt | jack | jbb-1 | jbb-2 | jbb-3 |
|---|---|---|---|---|---|---|---|---|---|
| Age-oriented | 1.0 | 1.7 | 1.1 | 2.1 | 1.4 | 1.2 | 1.1 | 1.4 | 1.9 |
| Jikes Parallel | 195 | 261 | 188 | 643 | 225 | 376 | 322 | 417 | 511 |

Table 4.2: Age-oriented maximum pause time in milliseconds

the old generation and die there. Since the age-oriented collector employs reference counting with the old generation, it does not collect these garbage cycles (until the backup tracing collector is used), causing frequent garbage collection invocations.

### 4.6.3 Pause times

Table 4.2 presents the maximum pause times of the age-oriented collector and Jikes parallel collector. Pauses were measured with a 64MB heap for SPECjvm98 benchmarks, and a 256MB heap for SPECjbb2000 with 1, 2, and 3 warehouses. For this number of threads, no thread gets swapped out, and so pauses are due to the garbage collection only. If we run more program threads, large pause times (whose lengths depend on the operating system scheduler) appear because threads lose the CPU to other threads.

The maximum pause time of 2.1ms, measured for the age-oriented collector, is two orders of magnitude shorter than that of Jikes parallel collector. The length of the age-oriented pause time is dominated by the time it takes to scan the roots of a single thread (occurring in one of the handshakes). This operation also dominates the pause time of the previous on-the-fly collectors [62, 5], and thus their pause times are similar. Hence, the age-oriented

| Collector phases-<br>Levanoni-Petrank | jess | db | javac | mtrt | jack | jbb |
|---|---|---|---|---|---|---|
| Clear-Dirty-Marks and<br>Reinforce-Clearing-Conflict-Set | 3.1% | 4.2% | 5.1% | 6.5% | 4.1% | 4.3% |
| Mark-Roots | 0.9% | 0.9% | 2.9% | 1.6% | 3.8% | 1.2% |
| Update-Reference-Counters-Young | 42.3% | 46.0% | 40.2% | 41.6% | 37.4% | 40.5% |
| Update-Reference-Counters-Old | 0.4% | 0.9% | 8.2% | 0.7% | 0.4% | 1.5% |
| Reclaim-Garbage-Young | 51.9% | 47.1% | 38.4% | 48.0% | 48.7% | 49.6% |
| Reclaim-Garbage-Old | 0.9% | 0.4% | 3.5% | 0.9% | 3.4% | 2.2% |
| Prepare-Next-Collection | 0.5% | 0.5% | 1.7% | 0.7% | 2.2% | 0.7% |

Table 4.3: Profiling of the Levanoni-Petrank collector.

collector achieves a significant throughput improvement over the original reference-counting collector and over the generational collectors, while retaining the short pause times.

It is important to note that the pauses induced by the collector do not happen frequently. If pauses of 2ms occurred once every 3ms, then pause times would loose their meaning and we should look at mutator's minimum utilization (MMU). However, in our case, the pauses form a negligible part of the collection cycle, and are split far apart from each other.

### 4.6.4 Profiling measurements

We have profiled the different phases of both the original Levanoni-Petrank algorithm and the age-oriented algorithm. The measurements were taken while the SPECjvm98 benchmarks were run with a 64MB heap size and the SPECjbb2000 benchmark (with 1,2,3 warehouses) was run with a 256MB heap size. Table 4.3 presents the profiling measurements of the original Levanoni-Petrank algorithm. These measurements indicate that:

- a very large portion of the collector work (usually around 90%) is dedicated to the handling of the $YoungObjects$ buffer, which includes incrementing the $rc$ of the sliding-view values of this buffer's objects and the release of the objects included in this buffer which turned out to be garbage (i.e., had a zero $rc$) after all reference-counting adjustments were made.

| Collector phases- Age-Oriented | jess | db | javac | mtrt | jack | jbb |
|---|---|---|---|---|---|---|
| Clear-Dirty-Marks and Reinforce-Clearing-Conflict-Set | 53.0% | 27.7% | 11.5% | 37.9% | 57.6% | 41.3% |
| Mark-Roots | 15.1% | 7.0% | 7.1% | 5.7% | 9.8% | 11.4% |
| Trace-Young | 13.8% | 51.3% | 42.9% | 47.8% | 17.6% | 22.0% |
| Update-Reference-Counters-Old | 6.6% | 4.3% | 19.1% | 4.2% | 5.2% | 7.5% |
| Reclaim-Garbage-Young | 1.3% | 5.9% | 1.3% | 0.5% | 1.4% | 0.5% |
| Reclaim-Garbage-Old | 2.3% | 0.4% | 13.7% | 0.6% | 1.5% | 10.3% |
| Prepare-Next-Collection | 7.9% | 3.4% | 4.4% | 3.3% | 6.9% | 7.0% |

Table 4.4: Profiling of the age-oriented collector.

- the *Updates* buffer handling (*rc* adjustments of objects logged in the *Updates* buffer, and the release of such objects) usually consumes negligible resources.

The clear throughput superiority of the age-oriented collector is better understood after seeing these profiling measurements: our age-oriented collector has focused on reducing the overhead of the 2 major collector phases (adjusting *rc* of young objects' sliding-view slots and releasing young objects). Table 4.4 summarizes the profiling measurements of the age-oriented algorithm. Since the two major phases (which comprise 78%-95% of the original collector work) were dramatically improved, the other phases, which comprise 5%-22% of the original collector work, now comprise of 42%-85% of the original collector work. That actually shows how well the age-oriented collector does.

Note also that applying the reference-counts updates of the old generation in each collection of the age-oriented collector usually consumes a small percentage of the entire collection (less than 8% in 5 of out the 6 benchmarks).

### 4.6.5 Client performance

Although the age-oriented collector is targeted at multiprocessors running on SMP platforms, we also measured its performance against the original algorithm on a uniprocessor. These measurements were also run on the Netfinity multiprocessor, as with Jikes one can control the

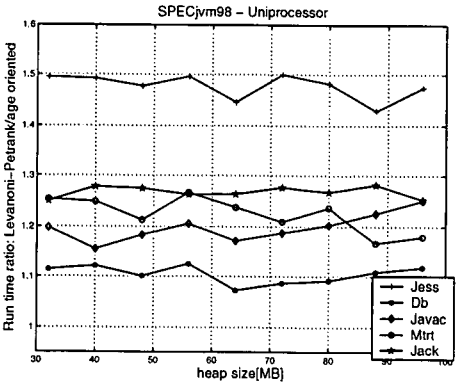



Figure 4.23: SPECjvm98 on a uniprocessor: age-oriented run-time ratio compared to the Levanoni-Petrank collector

number of CPUs the benchmark may use (including the collector thread). The behavior of the collector on a uniprocessor may demonstrate its efficiency. We measured the age-oriented collector on a uniprocessor with the SPECjvm98 benchmark suite and the results appear in Figure 4.23. Our measurements show that the age-oriented algorithm is substantially better than the original one in all tests as throughput ratios are shows improvement of 10% to 50%. Note that on uniprocessor, the age-oriented collector achieves a better throughput ratio than on multiprocessor (Figure 4.21), because runs on uniprocessor measure also the collections time, where the age-oriented collector is substantially more efficient than the original one.

## 4.7 Conclusions

We have proposed a framework of garbage collectors called age-oriented collectors, which we advocate as a state-of-the-art method of using reference counting. These collectors exploit the generational hypothesis in a different manner than standard generational collectors. Instead of running frequent young generation collections, the entire heap is collected infrequently, but young objects are treated differently from old objects. An age-oriented collector allows using the largest possible young generation, and thus obtains high efficiency. The most fitting use of age-oriented collectors is with concurrent collectors where pauses do not matter and particularly when the old generation is collected via reference counting, and thus deferring its collection is not beneficial.

We have designed and implemented an instantiation of an age-oriented collector, based on the reference-counting collector of [62, 63] for the old generation, and the tracing collector of [4] for the young objects. This age-oriented collector was implemented on the Jikes

RVM. Our measurements show that this collector maintains the short pauses of the original collectors and significantly outperforms both the original reference-counting collector as well as the generational variant. We conclude by advocating the use of an age-oriented collector for best exploiting the advantages of reference counting today.

# Chapter 5

# An Efficient On-the-Fly Cycle Collection

## 5.1 Introduction

Reference counting is a classical garbage collection algorithm. Systems using reference counting were implemented starting from the 1960s [26]. However, reference-counting garbage collectors cannot reclaim cyclic structures of objects. Thus, reference-counting collectors must be accompanied either by a backup mark-and-sweep collector (run infrequently to collect unreachable cyclic structures) or by a cycle collector.

Attempts to avoid developing and maintaining an additional mark-and-sweep collector on the reference-counting collected system led to design of a cycle collector [15, 25, 68]. This effort culminated in the state-of-the-art on-the-fly cycle collector of Bacon and Rajan [7].

### 5.1.1 The challenge

Bacon and Rajan [7] propose two cycle collectors. The simpler *synchronous* collector is the most efficient cycle collector known today. It runs in a stop-the-world context. Their more involved *asynchronous* collector is the only on-the-fly cycle collector known today. However, the asynchronous collector requires a lot of overhead in order to make the collection safe in the presence of concurrent program threads.

To understand why this happens, one should note that a typical cycle collector traces cycle

candidates repeatedly to discover which cycles are unreachable. (Typically, each candidate structure is traced two or three times.) A crucial problem with repeated scanning arises when concurrent program threads modify the objects graph during the scan. This means that the collector cannot trust a scan to repeat the very same structure that a previous scan has traversed. Furthermore, as modifications occur concurrently with the scan, each specific one scan cannot be guaranteed to view a consistent snapshot of the objects graph at any specific point in time. Such problems are the source of the two drawbacks of Bacon and Rajan's on-the-fly cycle collector: a practical drawback and a theoretical one.

The practical problem is that in order to achieve safety, the algorithm in [7] makes many repeated scans over the candidates. This reduces the overall efficiency of the reference-counting collector. The theoretical drawback is that completeness cannot be guaranteed[1]. A rare race condition may prevent an unreachable cyclic structure from ever being reclaimed.

### 5.1.2 The solution

In this work, we propose an algorithm for on-the-fly cycle collection that solves these drawbacks [2]. We do this by employing the sliding-views techniques recently developed for concurrent garbage collection in [62, 63, 4] and using them with the cycle collector of [7]. The main idea is to virtually fix the graph processed by the cycle collector. Suppose first that we stopped the threads and took a replica of the heap snapshot. Running the *synchronous* (more efficient) algorithm of [7] on this snapshot efficiently detects any cyclic structure. Of course, taking a replica of the heap is not realistic. However, a virtual snapshot of the heap may be taken using the ideas in [63]. Furthermore, if we use a sliding view instead of a snapshot (as in [63]) and make the appropriate adjustment to use a sliding view to scan the objects graph (as described in Chapter 3), then we obtain an on-the-fly cycle collector with the same short pauses of recent on-the-fly collectors ([6, 62, 63, 4]).

The theoretical liveness problem is immediately solved. If an unreachable cyclic structure is generated by the program before the snapshot, or before the start of the interval in which the sliding view is read, then the garbage cycle may be easily identified in this view. When a cycle collection is executed on top of this sliding view, this cycle is guaranteed to be reclaimed.

Unfortunately, the solution described above does not work. Oddly, the problem stems

[1] *Completeness* of a concurrent garbage collector is equivalent to the standard *liveness* property in distributed computing. A collector is complete if all unreachable objects are eventually reclaimed.

[2] This work was presented in [80].

from the celebrated efficiency of the sliding-views reference-counting collector. All previous cycle collectors required as input a list of all decrements of reference counts in order to work correctly. Missing decrements may lead to missed garbage cycles that will never be reclaimed. However, the sliding-views reference-counting collector does not keep track of all decrements. A large fraction of all reference-count updates are ignored by the sliding-views reference-counting collector and it is shown in [63] that objects may be correctly reclaimed even when only a small fraction of the reference-count updates are recorded and executed. To solve this mismatch, we extend the analysis of the sliding-views collector to show that the cycle collector may base its candidates on the decrements that are being recorded plus a special treatment of newly created objects. This is an interesting new property of the sliding-views collector, which is magically applicable to cycle collection.

From the practical point of view, the use of the simpler synchronous algorithm implies more efficient execution. Furthermore, making only a small fraction of the decrements (because of using the sliding-views reference-counting collector) implies recording fewer candidates for cyclic structures, which, in turn, means less work on traversing these candidates. This yields a substantial reduction in the cycle collector work. In addition, we suggest further improvements to the synchronous algorithm in [7], making it run even faster. This is done via a better scheduling strategy and new filtering techniques that further reduce the number of traced objects.

The behavior of the cycle collector was measured on two different configurations. The first configuration was the standard reference counting setting, which uses reference counting and cycle collection on the entire heap. As observed in previous work [5, 14, 81], reference counting is not effective on new objects (that tend to die fast). Previous work recommends using generations and employing reference counting to collect the old generation only. One such effective collector is the age-oriented collector presented in Chapter 4. So, if reference counting should be used on the old generation only, an interesting question that arises is how effective the cycle collector is when used on the old generation only. We resolve this quandary by also incorporating the new cycle collector into the age-oriented collector and examining its effectiveness when run on the old generation only.

Cycle collectors spend a large fraction of their time working on cycle candidates among newly allocated objects. By concentrating on the old objects, the age-oriented collector eliminates a large part of the cycles as well as a large fraction of the cycle collector's work, while tracing the young objects.

### 5.1.3 Implementation, measurements, discussion

We implemented the new cycle collector with the Levanoni-Petrank reference-counting collector [63] and with the (more efficient) age-oriented collector (presented in Chapter 4). The implementation was done on the Jikes research virtual machine [1] and compared against the original cycle collector of Bacon and Rajan [7]. We measured various features of these collectors showing that the amount of work decreased significantly with the new cycle collector. We also got good results on the entire program run, but felt that comparing the throughput is irrelevant in this case, since the two cycle collectors are built on two different reference-counting collectors (see [6, 63]).

We used the SPECjbb2000 benchmark and the SPECjvm98 benchmark suites. These benchmarks are described in detail in SPEC's web site [90].

One of the more interesting measurements we provide is the first comparison of cycle collection to backup tracing collection. This comparison is important since these are the two main options provided to an implementer of a reference-counting algorithm. Unfortunately, there is no prior report comparing these two alternatives. We measure the throughput of a JVM that uses the cycle collector with a JVM that uses a backup tracing collector to collect unreachable cyclic structures.

It turns out that backup tracing is more effective than the proposed cycle collector when reference counting is used to collect the entire heap. However, in the more relevant approach, when reference counting is used on the old generation only (which is the reference counting use that we advocate), the new cycle collector is more effective as its performance equals to the backup tracing solution and even outperforms it on tight heaps. Note that in both cases we compared apples to apples: the same scenario was run once with a backup tracing collector and once with a cycle collector. Detailed measurements are provided in Section 5.6. We conclude with an even stronger recommendation to use reference counting with cycle collection on the old generation only.

**Discussion.** The best way to use reference counting today is to run it on the old generation only, as discussed in [5, 14, 81]. In such a case, running cycle collection with reference counting is the right choice. Tracing collectors need to trace the live objects in the heap and their complexity relates to the size of this set. Reference counting needs only to account for reference-count updates and reclaiming dead objects. In a sense, tracing collectors pay a type of property tax whereas reference-counting collectors pay an income tax. Future applications may employ larger heaps in which they will maintain large areas of long-lived seldom-modified objects residing in an old generation. If this happens, reference

counting may become the best collector for the old generation. In fact, the work of [14] has already demonstrated the superiority of an appropriate generational reference-counting collector over tracing collectors. If generational reference counting becomes the collector of choice, than a companion cycle collector will be required. In this case, the cycle collector proposed here is an efficient companion and we expect it to outperform a backup tracing collector.

**Chapter organization.** We start by providing an overview of the existing collectors in Section 5.2. A synopsis of the new cycle collector, stressing the main new ideas, is provided in Section 5.3. The details of the cycle collector including pseudo-code are given in Section 5.4. Implementation and results are given in Sections 5.5 and 5.6. We conclude in Section 5.7.

## 5.2 Review of previous cycle collectors

In this section, we review previous cycle collectors. We use the term *cycle* or *cyclic structure* when referring to a strongly connected component in the objects graph. A strongly connected component is a maximal subgraph of a directed graph such that for every pair of vertices u, v in the subgraph, there exists a directed path from u to v and a directed path from v to u.

### 5.2.1 Collecting cycles on a uniprocessor

We start with the synchronous cycle collector of [7] (building on [68, 65]) that runs in a stop-the-world manner on a uniprocessor. Garbage cycles can only be created when a reference count is decremented to a non-zero value ([68, 65]). The reference-counting collector records all objects whose reference count is decremented to a non-zero value. The cycle collector uses this list as a set of candidates that may belong to a garbage cycle. Three colors are used to mark the state of objects. The initial color of all objects is black. A possible member of a garbage cycle is marked gray. White signifies an object that is identified as part of an unreachable cycle. The cycle collector runs up to three traversals on all objects reachable from the candidate set as follows.

- **The mark stage:** traces the graph of objects reachable from the candidates, subtracting counts due to internal references and marking traversed nodes gray. At the end of this traversal, all nodes of each unreachable cyclic structure have zero reference counts, whereas each reachable cyclic structure has at least one node with a positive reference count.

- **The scan stage:** scans the subgraph of (gray) objects reachable from the candidates. All objects reachable from external pointers (those with positive reference counts) and all their descendants are marked black. Also, reference counts are restored to reflect all outgoing pointers from black objects. All other nodes in the subgraph are colored white (these objects are identified as forming a garbage cycle).
- **The collect stage:** scans the subgraph again and reclaims all white objects.

### 5.2.2 Collecting cycles on-the-fly

The first concurrent cycle collection algorithm was proposed in [7]. Their algorithm consists of two phases, each running several scans on the sub-graph reachable from the candidates. In the first phase, a variant of the above synchronous algorithm is used, but instead of reclaiming the white nodes these nodes are recorded as potential unreachable cyclic structures. However, due to concurrent mutator activity, some of the white objects may have been incorrectly identified and may actually be reachable. The second phase is executed in the next (reference-counting) collection and it then re-examines the potential cycles, identifying which are indeed unreachable and reclaiming their objects.

**Restoring the reference counts.** In a concurrent setting the subgraph of the candidate objects cannot be re-traversed, because pointers may have been modified since a previous traversal. Thus, temporarily modifying reference counts and then restoring them correctly becomes impossible. Therefore, the algorithm uses an additional reference-counting field denoted *CRC* (cyclic reference count). Temporary calculations are executed in this additional field and the main reference-count field is used only by the reference-counting collector. We will also use the *CRC* additional count in our algorithm.

**Two disadvantages.** The above concurrent garbage collector has a theoretical drawback and a practical drawback. A garbage collector is called *complete* if it eventually collects all unreachable objects. The first problem of this cycle collector is that it is not complete. Rare race conditions may prevent it from collecting garbage cycles. An example appears in [7]. The second problem is practical. The algorithm traces the candidate cycles a couple of times in the second phase to ensure that no false garbage cycle is reclaimed. These extra scannings reduce efficiency, especially for (typical) benchmarks that contain many garbage cycles or many false cycle candidates. Moreover, the concurrent cycle collection algorithm enforces additional overhead on the execution of the reference-counting algorithm as it must fix sub-graphs that were left gray or white due to improper re-traversals.

## 5.3 Cycle collector overview

In this section we provide an overview of the new collector, emphasizing its main ideas. A full description including the pseudo-code is provided in Section 5.4.

Both drawbacks of the asynchronous collector in [7] stem from the fact that the concurrent cycle collector cannot rely on being able to re-trace the same graph. In this work, we propose a new concurrent cycle collector that eliminates the disadvantages of the previous cycle collector yielding a non-intrusive, efficient cycle collector that guarantees completeness. The idea is to use a snapshot of the heap or a sliding view of the heap ([63]). Given a fixed view of the heap (as reflected by a snapshot or the sliding-view mechanism), it is possible to eliminate much of the redundant tracing and to guarantee completeness. Hence, we employ the reference-counting system of [63] (an overview appears in Chapter 2.6) to reclaim the acyclic garbage; and in order to trace the sub-graph reachable from the candidates, we use the sliding-views tracing mechanism proposed in Chapter 3.

We start by describing the new cycle collection algorithm assuming two inputs: a snapshot of the heap, and a list of all objects whose reference count has been decremented to a positive value since the last cycle collection. A first observation is that given these two inputs, we may apply the *synchronous* algorithm of [7] on the given snapshot and correctly identify the garbage cycles in the heap as viewed in the snapshot. Now, combining the fact that the synchronous algorithm is efficient and the fact that being a garbage cycle is a stable property, i.e., program activity cannot make an unreachable object reachable, we get an efficient identification of garbage cycles. After explaining how to use a snapshot (which achieves a concurrent collector), we will move to using a sliding view. A sliding view is a bit harder to use, but provides an on-the-fly collector.

Next, we need to specify how to obtain the inputs efficiently. We first concentrate on the first input: the snapshot. The second input cannot be obtained efficiently, but we will find ways to use a restricted version of it.

### 5.3.1 Obtaining a snapshot (or a sliding view)

The cycle collector uses the snapshot by repeatedly traversing several subgraphs of the snapshot heap. To obtain a snapshot that can be traversed, we may use the sliding-view mark-and-sweep mechanism described in Chapter 3 above. We also employ the write barrier of [63]. Then, to traverse an object according to its pointer values as existed at snapshot time, we scan each object in the following manner. First, the dirty bit of the object is examined.

If the object is not dirty (no pointer in the object has been modified since the snapshot was taken), then its current state in the heap is equal to its state during the snapshot and the collector may trace it by reading its pointers from the heap. Otherwise, the object has been modified since the snapshot time and it is marked dirty. In this case, the collector traces its snapshot values as recorded in the threads' local buffers. This way, objects are traced according to their state at the snapshot time, and as a consequence, repeated traces are bound to trace the same graph each time.

In terms of completeness, this means that once a garbage cycle is created, it must exist in the next snapshot, and thus is bound to be collected by the next execution of the synchronous cycle collection. In terms of efficiency, this means that we may use the efficient synchronous algorithm and get rid of inefficiencies originating from the need to insure correctness in spite of program-collector races. For example, the entire second phase of the asynchronous algorithm of [7] is redundant: there is no need to *store* identified garbage cycles and validate them during the next garbage collection.

We now proceed to explain how sliding views are used instead of snapshots. The goal is to eliminate the need for a simultaneous halt of all program threads and to obtain extremely short pauses. A sliding view is a distorted snapshot. It may yield a fuzzy picture of the heap, but it is still one that may be obtained without stopping all mutators simultaneously. The cycle collector remains the same, except that it uses a sliding view of the graph rather than a snapshot. As in the previous sliding-views collectors, a sliding view may find an object unreachable because the view does not represent the heap at a consistent point in time. However, the snooping mechanism (see Section 2.6) makes sure that these objects are not reclaimed, guaranteeing the safeness of the procedure. For the cycle collector, this means that a set of objects may be incorrectly identified as being an unreachable cyclic structure. How can this happen? Inaccuracies of reference counts due to the sliding view are discussed in [61, 63]. Intuitively, if no pointer is written to the heap at the beginning of the collection (when all mutators are halted one by one), then the sliding view represents a snapshot of the heap taken at the time the first mutator is stopped; denote this time by $t_1$. However, as pointers are being written in the heap, this snapshot gets distorted. In particular, the view may contain values of pointers that were updated after $t_1$. If such a modified pointer creates a false unreachable garbage cycle in the view, then a pointer must have been added to this cycle during the interval in which the sliding view is taken. In this case, it is guaranteed that the object that falsely seems unreachable in the sliding view will be snooped and therefore, we will not reclaim the cyclic structure that contains it. Thus, the safety of the cycle collector may be reduced to the safety of the sliding-view mark-and-sweep collector.

With respect to completeness, it holds that any unreachable cyclic structure that is formed before the collection begins, must be collected. The reason is that these objects are not modified during the time the sliding view is taken, and in particular, no new pointers are being written to objects in this cycle. Thus, none of the objects in the cyclic structure is snooped and the view of all pointers into and in between these objects appears in the sliding view exactly as it would have appeared had we taken a real snapshot at time $t_1$. Thus, such an unreachable cyclic structure will be reclaimed.

### 5.3.2 Obtaining the list of candidates

It remains to obtain the second input to the synchronized cycle collector of [7]. This collector described above and all previous collectors used a candidate set consisting of all newly created objects plus all objects whose reference count is reduced to a positive value by any pointer modification since the previous cycle collection. However, the sliding-views reference-counting collector of [63] does not maintain such a list. In fact, it is oblivious to most of the pointer updates and this is what guarantees its efficiency. A naive solution is to add the recording of such a list to the reference-counting collector of [63]. This is not acceptable as it would undermine the efficiency of the reference counting. Instead, we analyze what is really required to collect cycles and find out that the reduced set of candidates suffices. This preserves the efficiency of the reference-counting collector and also significantly improves the efficiency of the cycle collector as fewer candidates need to be recorded and less work is required to traverse their descendants.

**Newly created objects.** Let us review one technicality that also exists in prior literature. The assertion that it is enough to consider only reference-count decrements as candidates is accurate but not applicable to modern collectors. The reason is that reference counts are not updated for root pointer modifications (as suggested by [31]). Thus, all known cycle collectors use, as candidates, more than the set of objects whose reference count was decremented to a positive value. The set of candidates also includes all objects created since the last collection and all objects referenced directly from the roots during the previous collection.

Consider, for example, two new objects that point to one another (forming a cycle) and a root pointer that points to one of them. If the root pointer is modified, then a cycle of garbage is formed, but it is not noted by reference-count decrements. The extended candidate set as above is enough to detect any such garbage cycle. We do not elaborate on this, as this solution is used by all previous collectors.

**Obtaining the candidates.** The sliding-views collector can yield almost for free a list of newly created objects and a list of objects that were referenced by the roots during the previous collection. We now concentrate on the more problematic set of objects whose reference count was decremented.

The Levanoni-Petrank reference-counting collector [63] uses a shortcut to reduce a large fraction of the reference-count updates. The idea is that when a pointer $p$ takes the values $o_0, o_1, o_2, \ldots, o_n$ between two sliding views, the only required reference-count updates are a decrement to $rc(o_0)$ and an increment to $rc(o_n)$. However, the fact that not all increments and decrements of the objects $o_1, o_2, \ldots, o_{n-1}$ are executed might prevent noting that one of the decrements creates a new unreachable cycle.

We now claim that we are able to collect all garbage cycles, even though we record and consider many fewer objects as candidates, i.e., those supplied by the Levanoni-Petrank reference-counting collector. To be more precise, when a pointer $p$ takes the values $o_0, o_1, o_2, \ldots, o_n$ between two collections, only $o_0$ is considered as a candidate (if its reference count is decremented to a non-zero value) by the new cycle collector. The objects $o_1, o_2, \ldots, o_{n-1}$ that may have been considered by previous collectors as candidates are ignored. Additional relevant decrements are decrements that are executed by the reference-counting collector itself. When an object is reclaimed, the collector decrements the reference counts of all its descendants. These decrements may also produce candidates (if the descendant's reference count is not decremented to zero).

To show that the collector does not miss a garbage cycle, we divide the argument into two cases: garbage cycles composed solely of old objects and garbage cycles containing at least one young object, where a young object is an object that has been created after the previous sliding view (or snapshot) has been taken. We show that each of these two cases are properly handled.

The easy case is when a garbage cycle includes a young object. As mentioned earlier in this section, all young objects (surviving the reference-counting collection) are considered candidates. Thus, this cycle will not be missed.

The more interesting part is to note that garbage cycles containing only old objects (these created before the previous sliding view) are not missed. If this cycle was reachable during the previous sliding view and is unreachable in the current sliding view, then there exists a pointer to one of the cycle's objects in the previous sliding view, but this pointer does not exist in the current sliding view. If this was a root pointer, then the cycle is considered by the fact that all root pointers from previous collection are candidates. Otherwise, this is a heap pointer that has been modified during the time interval between the two sliding views.

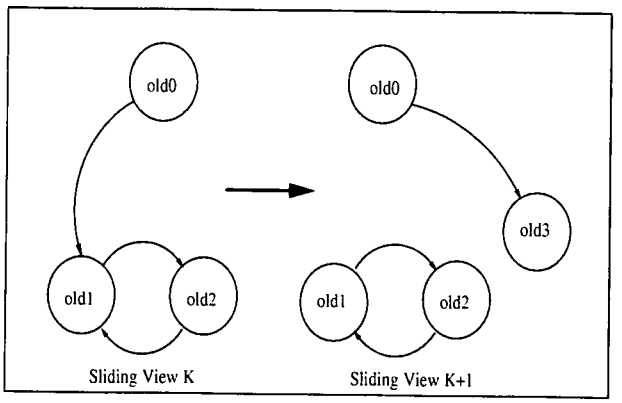



Figure 5.1: An example: The creation of a garbage cycle composed solely of old objects

This scenario is depicted in Figure 5.1 (and is explained below). The pointer modification results either from the application modification of the pointer (as in Figure 5.1), or because the object containing this pointer was reclaimed and then the memory manager deleted the pointer. In the first case, the change of this pointer is logged in a local buffer, causing a decrement to the reference count of the object previously referenced. In the latter case, the delete operation of the collector implies a similar reference-count decrement. In each of these cases, this object becomes a candidate for cycle collection. Hence, cycles containing only old objects are accounted for properly.

Figure 5.1 introduces an example of a garbage cycle composed solely of old objects is created between the $K^{\text{th}}$ and the $K+1^{\text{st}}$ sliding views. In this example, the cycle was reachable from *old0* and it became unreachable because *old0* was modified. Since *old0* is modified between the sliding views, *old0* (and in particular, the pointer to *old1* in it) must be logged to a local buffer that is later used by the reference-counting collector. Therefore, the reference count of *old1* gets decremented in the $K+1^{\text{st}}$ collection, and it is then considered as a candidate.

As explained above, when a pointer $p$ takes the values $o_0$, $o_1$, $o_2$, ..., $o_n$ between two collections, only $o_0$ is considered as a candidate (if its reference count is decremented to a non-zero value) by the new cycle collector. Note, however, that the reference deletion from objects $o_1$, $o_2$, ..., $o_n - 1$ could create a garbage cycle. Figure 5.2 shows an example that demonstrates how we are able to collect all garbage cycles even though we record considerably fewer candidates. The example is divided into six steps (numbered 1 through 6). At Step 1 of Figure 5.2, the sliding-view picture of the last collection is shown. At Step 2, object $P$ is modified to reference object *O2*. Since $P$ is modified for the first time since the previous sliding view, it is set as dirty and logged to the *Updates* buffer together with its previous value *O1*. Next, in Step 3 object *O2* is set to reference object $B$. At Step 4, object $B$ is set

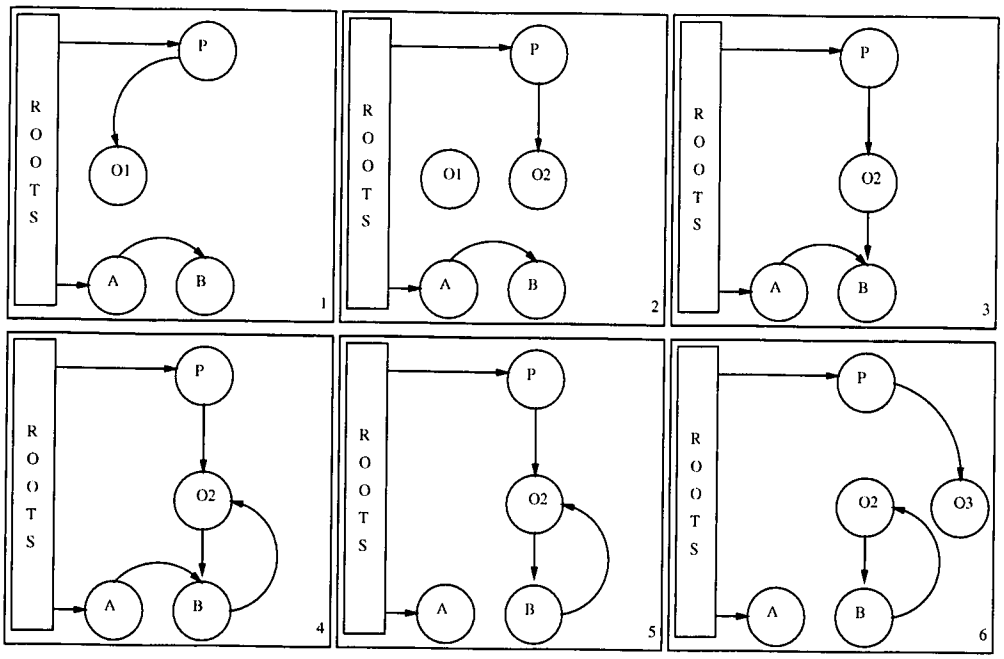



Figure 5.2: Example showing that all garbage cycles are collected even though recording considerably fewer candidates.

to reference object *O2*. At Step 5, object $A$ is modified to stop referencing object $B$. As $A$ is modified for the first time since the previous sliding view, it is set as dirty and logged to the *Updates* buffer together with its previous values $B$. At Step 6, object $P$ is modified to reference object *O3*. Since $P$ is already dirty, $P$ will not be logged again to the *Updates* buffer. Note that the deletion of the reference from $P$ to *O2* creates a garbage cycle composed of objects $B$ and *O2*. However, *o2* did not became a candidate in this stage, as the write barrier did not log it in the *Updates* buffer and hence the collector would not decrement its reference-count value[3]. The garbage cycle would be collected since object $B$ was logged as the previous value of $A$ in Step 5. Hence the collector would decrement its reference count later, and since the reference count of $B$ would not reach 0 (as it is referenced by *O2*), it would considered as a candidate. Hence the above garbage cycle is bound to be collected.

To summarize, we may employ the efficient write barrier of Levanoni-Petrank and collect cycles correctly using as candidates all objects whose reference count is decremented to a non-zero value, as well as all young objects and all objects that were referenced by the roots during the previous collection.

[3]As described earlier in this section, if *O2* is a newly created object, it would be considered as a candidate. Here, we assume that *O2* is not a newly created object.

### 5.3.3 Making a stop-the-world collector on-the-fly

We now explain in brief how we made it possible to run the stop-the-world collector of [7] on-the-fly. The main purpose of stopping the threads is to make the object graph fixed for the collector. This fixing lets the collector examine the reachability graph without experiencing pointer modifications during the examination. A standard way to let the collector examine a fixed graph while the mutators go on modifying it is to let the collector work on a snapshot of the heap. Creating a real snapshot is too costly, but snapshots can be implicitly created using specialized write barriers. Letting a stop-the-world collector work on a snapshot (with the necessary modifications required to let the collector work on the created snapshot view) only achieves a concurrent collector. Namely, the obtained collector must stop all mutators simultaneously for a single point in time that will create the snapshot time.

In order to make the collector on-the-fly, sliding views must be used. To create sliding views, the mutators are not required to stop simultaneously. It is enough that each cooperate with the collector at will. Sliding views do not provide a real snapshot, but a distorted approximation of it. However, a properly adjusted stop-the-world collector may use a sliding view in a similar way to a snapshot in order to reclaim the unreachable objects appearing in the sliding view. The correctness is based on a snooping mechanism. The snooping mechanism makes sure that we do not reclaim objects to which a new pointer is installed while the mutators are cooperating with the collector. Given this mechanism, the stop-the-world collector may work on the sliding view to reclaim all unreachable objects. Note that the obtained collector is on-the-fly. The mutators only cooperate non-simultaneously with the collector to create the sliding view. The collector then goes on running concurrently with program run, while using the obtained sliding view. Since the sliding view is a static picture of the heap, the stop-the-world collector may work on it according to its original algorithm.

### 5.3.4 Behavior with the age-oriented collector

The cycle collector was incorporated into the full heap reference-counting collector of [63]. However, previous work shows that reference counting is most effective when used with the old generation only [5, 14, 81]. It is, therefore, crucial to examine the behavior of the cycle collector in such a setting, when reference counting is employed on old objects only. Furthermore, from the description above, it seems that newly created objects add a substantial burden on the cycle collector. Our measurements show that this is indeed correct. Hence, we also applied the proposed cycle collector with a collector that runs concurrent reference counting and cycle collection on the old generation only. We chose the

age-oriented collector (described in Chapter 4), which is an altered generational collection that is specially targeted at reference counting for the old generation.

It is important to note that the age-oriented collector is an efficient collector. In particular, it is more efficient than the reference-counting algorithm as a stand-alone. Therefore, it is more relevant to check its performance with a cycle collector. Using the age-oriented collector it was possible to eliminate a large fraction of the cycles as well as a large fraction of the cycle collector's work since it does not need to consider the young objects as candidates. Indeed, cycle collection was more effective in this setting.

### 5.3.5 Reducing the number of traced objects

Next, several methods are proposed to further reduce both the number of *candidates* and the number of *objects traced* during a candidate traversal. First, we make use of the strategy of Bacon and Rajan [7] to reduce tracing. They proposed to ignore acyclic objects (objects that cannot be a part of a cycle, e.g., an array of scalars). Such objects are statically determined and are never considered as candidates. Any acyclic object reached during the algorithm traversals is ignored. We adopt the schema of Bacon and Rajan, which considers as acyclic each object whose class contains only scalars, references to classes that are both acyclic and final, and arrays of the previous classes. Additional elimination strategies are proposed next.

**Examining only mature candidates.** The new collector runs cycle collection with each garbage collection. However, it lets the candidates "mature" before actually testing them for membership in an unreachable cyclic structure. While candidates "wait" to be examined, many of them are removed from the candidate list (as explained below). The collector examines only the candidates that were accumulated $k$ collections ago and not filtered out of the list. Technically, instead of having one large candidate buffer, we employ $k+1$ smaller candidate buffers, each containing candidates accumulated in different collections. In the current cycle collection we use the oldest buffer. One reason for removal from the candidate list is that objects are simply reclaimed by the reference-counting collector before reaching the oldest buffer. A second filtering technique employed is to remove any object that is added to the most recent candidate buffer, from any older buffer in which it appears. This means that an object appears only in one buffer and that if its reference count is decremented several times, it will be examined only once (if not reclaimed earlier). (The removal from older lists is executed in a short processing of the buffers at the end of each collection.)

The $k+1$ buffers method allows structured control over candidates maturity and allows tracing only candidates that have not been filtered out (and have not died) throughout

the last $k$ collections. Previous collectors run the cycle collector each $k$ collections on all candidates. Thus, they handle candidates that were recently discovered and have not yet matured.

**Ignoring identified live objects.** Next, we try to reduce the number of candidates and also the amount of executed tracing. Several objects are known to be alive at the beginning of a collection. These include objects that were directly reachable from the roots, objects that were snooped and objects that were marked dirty because they were modified after the sliding view was taken. Many such objects are traced by previous cycle collectors. The proposed collector treats these objects differently. First, if the cycle collector reaches one of these objects during the first (mark) stage, it ignores it. It does not decrement its reference count, nor does it trace its subgraph. By avoiding the explorations of these objects, we do not lose any garbage cycles, since if an object is known (or assumed) to be alive, so are all the objects reachable from it.

To see that this modification does not foil correctness, we first note that avoiding the trace of such objects affects only their descendants, which are alive and should not be reclaimed during the current collection. Thus, we do not mind not handling these objects during the cycle collection. If they remain black, they will not be reclaimed. The only remaining concern is that this modification will prevent identifying live descendant objects within some candidate cyclic structure. However, this cannot happen, since such descendants have an outer reference (their ancestor) and so their *CRC* cannot decrease to zero and they cannot be reclaimed.

**Saving the double scan stage.** To save more scanning time, we add an additional stage, the *mark live black* stage, between the mark stage and the scan stage. In this stage we preprocess the list of objects that are known to be alive and trace them to mark their subgraph black. If we do not do this, the scan stage may sometimes color an object white together with its subgraph, only to find out later that this object was referenced from another (gray) externally referenced object. In this situation, the object (and subgraph) would be re-traversed and colored black during a second traversal of the same subgraph. Such repeated traversals are eliminated by the new stage.

Not all objects that are known to be alive can be identified when collection commences. In particular, an object may be modified (and thus become dirty) by a mutator after being traced during the mark stage of the cycle collector. The subgraph of such an object may be traced and colored gray (during the mark stage) and only later will it become evident that the tracing was redundant. Thus, we also add a simple check to the second (scan) stage. When reaching a gray object in the scan stage, we check whether it is dirty. If it is, its

subgraph is colored black immediately in order to save further scans. This modification is safe since this object was alive when the sliding view was taken.

One disadvantage of using dirty objects for the above filtering methods is that we must use the additional *CRC* (cyclic reference-count) field, as in the asynchronous cycle collector of Bacon and Rajan, in order to correctly restore the "real" reference counts. The reason that the *CRC* field is needed is that the set of identified live objects (actually, only the subset of dirty objects) is not fixed during the collection. Recall that we do not scan dirty objects, implying that we also do not decrement their descendants' reference counts for internal references. However, since objects become dirty concurrently, we may decrement their reference counts several times before noting that they are alive. It is not possible later to tell how many times decrements were applied on dirty objects (before they became dirty) and thus it is not possible to restore the original reference counts. This necessitates the use of the *CRC* field. The choice is whether to use the dirty bit to identify reachable objects or to avoid using the *CRC* field. We chose to consider dirty objects and use the *CRC* field.

## 5.4 The Garbage Collector Details

In this section, pseudo-code and details of the new cycle collection algorithm are provided.

### 5.4.1 The LogPointer

The original reference-counting algorithm requires maintaining a dirty bit signifying whether an object has been modified since the most recent collection started[4]. During the first modification of an object in a cycle, its pointers are recorded in the *Updates* buffer and its dirty bit is set. We follow [62, 63, 4] by choosing to dedicate a full word to keep the dirty bit. Indeed, this consumes space, but it allows keeping information about the dirty object. In particular, this word is used to keep a pointer into the thread's local buffer where this object's pointers have been logged. A zero value (a null pointer) signifies that the object is not dirty (and not logged). We call this word the LogPointer.

Tracing of a sliding view exploits the LogPointer field extensively. When an object is scanned, the LogPointer is checked. If it is null, then the current state of the object may be

[4]To simplify the presentation, the original collector in [62] is described as using a dirty bit per slot. However, as discussed in the full paper [61, 63] and as implemented in subsequent work [4, 5, 81], a bit per object was already used.

used. Otherwise, it provides a pointer to the log entry where the state of the object in the sliding view is recorded.

**Procedure Update** (Figure 5.3) describes the write barrier that is activated at pointer assignment. Its main task is to record the object whose pointer is modified (i.e., log objects' values at the sliding views). We stress that the write barrier (the Update protocol) is only used with heap pointer modification. Modifications of local pointers in the registers or stack are not monitored. Going through the pseudo-code, we see that each object's LogPointer is optimistically probed twice (lines 1 and 5) so that if the object is dirty (which is often the case), then the write barrier is extremely fast. If the object was not logged (i.e., the LogPointer of an object is NULL), then after the first probe, the object's values are recorded into the local $Updates_i$ (lines 2-4). The second probe at line 5 ensures that the object has not yet been logged (by another thread). If LogPointer is still NULL (in the second probe), then the recorded values are committed as the buffer pointer is modified (line 7). In order to be able to distinguish later between objects and logged values, in line 6 we actually log the object's address with the least significant bit set "on" (while values are logged with the least significant bit turned off). Then, the object's LogPointer field is set to point to these values (line 8). After logging has occurred, the actual pointer modification happens. Finally, from the time a collection begins until marking the roots of the mutators, the snoop flag is on. At that time, the new target of the pointer assignment is recorded in the local $Snooped_i$ buffer. This happens in lines 10-11. The variables $Updates_i$, $CurrPos_i$, $Snoop_i$ and $Snooped_i$ are local to the thread.

The write barrier works safely with a multithreaded application without requiring explicit synchronization, as thoroughly discussed in [63].

### 5.4.2 General issues

**Candidate objects status.** In order to process the candidate buffers, we keep a state with each object. An object is allocated in a *non-buffered* state. When it is first buffered, it is marked *newly-buffered*. During each collection, all buffers are processed. Each *newly-buffered* object is removed from all older buffers. An object that is a member of the oldest buffer (the buffer that is currently being checked for cycles) is marked *old-buffered*. All other buffered objects (in buffers that are not the youngest or oldest buffers) are marked *mid-buffered*. The candidate buffers are denoted, in a corresponding manner, *newCandidatesBuffer*, *mid-CandidatesBuffer* and *oldCandidatesBuffer*. Note that there may be several buffers of type *midCandidatesBuffer*.

```
Procedure Update(obj: Object, offset: int, new: Object)
begin
1.   if obj.LogPointer = NULL then     // OBJECT NOT DIRTY
2.      TmpPos := CurrPos_i
3.      foreach field ptr of obj which is not NULL
4.         Updates_i[TmpPos++] := ptr
5.      if obj.LogPointer = NULL then     // IS IT STILL NOT DIRTY?
6.         Updates_i[TmpPos++] := tag(address of o) // ADD A TAGGED POINTER TO OBJECT
7.         CurrPos_i := TmpPos     // COMMITTING THE REPLICA
8.         obj.LogPointer := address of Updates_i[CurrPos_i]     // SET DIRTY
9.   write( obj, offset ,new)
10.  if Snoop_i and new != NULL then
11.     Snooped_i := Snooped_i ∪ { new }
end
```

Figure 5.3: SVRC mutator code: Update Operation

**Assumed procedures.** In the pseudo code we assume the existence of some simple methods. These include:

- is-Acyclic: checks whether an object is inherently acyclic. This check is executed according to the rules set in Section 5.3.5.
- is-Buffered-Not-Old: checks whether an object is buffered but not in the oldest buffer.
- is-Released: checks whether an object has been reclaimed in the current collection (by either the reference-counting collector or the cycle collector)[5]. As this procedure is called by the collector, the *current collection* is well defined[6].

[5]Our original reference-counting collector implementation employs Jikes segregated-free-list allocator [102]. This allocator divides the heap into blocks, where each block is partitioned into equal-sized slots. In addition, a "used" bit is associated with each such slot. This bit indicates whether the corresponding slot is in use (i.e., contains an object) or not. Whenever a slot is allocated its bit is marked as taken, and whenever an object is freed its bit is marked as free. The is-Released method returns the opposite of this bit.

[6]In our implementation we did not allocate a released object's memory until after the cycle collector is

```
1.  Roots := programRoots ∪ SnoopedObjects
2.  increment the rc of all Roots     // SO IT WON'T BE COLLECTED THIS COLLECTION
3.  for each object logged in Updates do
4.        - decrement rc of its previous sliding-view descendants
5.        - increment rc of its current sliding-view descendants
6.  for each object logged in YoungObjects do
7.        increment rc of its current sliding-view descendants
8.  reclaim objects with zero rc recursively
9.  Process-Cycles     // COLLECT GARBAGE CYCLES
10. decrement the rc of all Roots     // AS IT WAS INCREMENTED IN LINE 2
```

Figure 5.4: SVRC: Sliding views reference counting with cycle collection

### 5.4.3 Cooperation with the reference-counting collector

Next we elaborate on the cooperation between the reference counting collector and the cycle collector. Full pseudo-code of the sliding-view reference-counting collector and the age-oriented collector are in [79].

The cycle collection algorithm is activated on each collection cycle after the reference-counting collector has finished collecting the acyclic garbage. The entire collection including cycle collection is described in Figure 5.4.

During the reference-counting collector run, it detects the objects that become candidates. These candidates are accumulated into the newest candidate buffer. The candidates are accumulated in the following collector stages:

- During the traversal of the *Roots* buffer, each object is accumulated to the newest candidate buffer (it is not considered as a candidate in this cycle collection as it is obviously not garbage).
- During the traversal of the *Updates* buffer, each object whose reference count is decremented but does not reach zero is added to the candidate buffer.

entirely done. We could have instead used a bitmap for marking the released objects (which the is-Released method could probe), so that a released object could be immediately allocated. Such a bitmap should be cleared at the end of each collection.

```
Procedure Add-Candidate(cand: object)
begin
1.    if cand.status != newly-buffered ∧ !is-Acyclic(cand) then
2.       cand.status := newly-buffered
3.       push cand into newCandidatesBuffer
end
```

Figure 5.5: Cycle collector: Add-Candidate Procedure

- The *YoungObjects* buffer keeps a list of all young objects for use by the reference-counting collector. After the reference-counting collector reclaims all dead objects, a special traversal of this buffer puts all live objects (with non-zero reference count) into the candidate buffer.

These three kinds of candidates are accumulated into the (same) candidate buffer. The accumulation of candidates is done by Procedure Add-Candidate (Figure 5.5). This procedure is called by the reference-counting collector in order to insert an object into the *newCandidatesBuffer*. If this object is not already buffered in the *newCandidatesBuffer*, and it is not acyclic, it is buffered into the *newCandidatesBuffer* after its state is modified into *newly-buffered*. Note that by modifying the object's status to *newly-buffered*, this procedure actually invalidates the object appearance in an older candidate buffer if it existed (as explained in Section 5.3.5).

An interface in the opposite direction is the one allowing the cycle collector to call Procedure RC-Free, which performs the recursive deletion of an object. RC-Free is invoked by Procedure Reclaim of the cycle collector, described in Figure 5.15.

**Objects' colors.** Note that before invoking Procedure Process-Cycles (line 9 of Figure 5.4), all objects are black. The cycle collector relies on this property. This property is achieved by the following:

- Objects are allocated black.
- The cycle collector is the only one to color objects in gray or white. In addition, the cycle collector maintains the property that all objects surviving the previous cycle collections are black by the end of a cycle collection.

**The LogPointer.** Independently of the cycle collection, the reference counting must clear all dirty bits (i.e., nullifies the LogPointers) at the beginning of a collection cycle.

```
Procedure Process-Cycles
begin
1.  Mark-Candidates
2.  Mark-Live-Black
3.  Scan-Candidates
4.  Collect-White
5.  Process-Buffers
end
```

Figure 5.6: Cycle collector: Process-Cycles Procedure

Actually, this task is non-trivial for the on-the-fly collectors (see [63]). The cycle collector relies on this property, and considers each non-dirty object as an object that was modified after the current collection has began. This is used to trace objects as in the trace of the sliding-view mark-and-sweep collector.

### 5.4.4 Cycle algorithm code

The cycle algorithm's code for cycle $k$ is presented in **Procedure Process-Cycles** (Figure 5.6). This procedure is applied in every cycle collection after the reference-counting collector is done collecting the non-cyclic garbage. It consists of the following stages:

- **Mark-Candidates stage:** traces the graph of relevant candidates, subtracting counts (of the $CRC$ field) due to internal references and marking nodes as potentially reclaimable (by coloring them gray).
- **Mark-Live-Black stage:** colors black the objects that the **Mark-Candidates** stage has identified as alive. Its purpose is to save redundant traversals as described in Section 5.3.5.
- **Scan-Candidates stage:** scans the subgraph of relevant candidates, and re-colors black objects that are reachable from external pointers. All other nodes in the subgraph are colored white.
- **Collect-White stage:** scans the (white) subgraphs again and reclaims all garbage (white) objects.

```
Procedure Mark-Candidates
begin
1.   for each cand in oldCandidatesBuffer do
2.     if is-Released(cand) ∨ cand.status = newly-buffered then
3.       remove cand from oldCandidatesBuffer
4.     else if cand.color = black then      // NOT PREVIOUSLY TRAVERSED
5.       Mark(cand,true)
6.     else      // GRAY OBJECTS, I.E., PREVIOUSLY TRAVERSED
7.       cand.status := non-buffered
8.       remove cand from oldCandidatesBuffer
end
```

Figure 5.7: Cycle collector: Mark-Candidates Procedure

- **Process-Buffers stage:** iterates over the new and middle buffers, while filtering non-relevant candidates. In addition, it performs a cyclic swapping of buffer roles.

**Procedure Mark-Candidates and Procedure Mark** (Figures 5.7-5.8) perform the mark stage. This stage is performed on the *oldCandidatesBuffer*'s objects that have survived all filters of the previous collections. Procedure Mark-Candidates first filters more candidates: those that were released during this collection[7] and those that were re-added to the candidate buffer in this collection (and thus are *newly-buffered*). Note that it also pops out of the buffer (and clears buffer statuses of) gray objects: those objects are reachable from other candidates that have already been traced during this stage, and thus they could only belong to a garbage cycle rooted from an already traced candidate. Procedure Mark is applied to all the other candidates.

Procedure Mark performs a depth-first traversal over the candidates' sliding-view subgraphs. An object reached for the first time is colored gray, its $CRC$ is initialized and if it is not identified as alive (was not modified, is not local and is not buffered in a younger candidate buffer), its sliding-view descendants (which are not acyclic) are traced using the Read-Sliding-View procedure. A parameter to the function is a flag signifying whether the current object is scanned due to a reference found in the heap (and therefore, an inner reference is found and the $CRC$ should be decremented) or it is scanned because it is a

[7]The deletion of the last pointer to a shared cell will recycle it immediately, regardless of whether there is a reference to it in a candidate buffer.

```
Procedure Mark (obj: object, isTopLevel: Boolean)
begin
1.   if obj.color != gray then      // FIRST TIME REACHED IN THIS MARK STAGE
2.     obj.color := gray
3.     obj.CRC := RC
4.     if !isTopLevel then      // REACHED AS A DESCENDANT
5.       obj.CRC--
6.     if obj.LogPointer != NULL ∨ obj ∈ Roots
                  // CHECK WHETHER OBJECT WAS WITNESSED LIVING
7.         ∨ is-Buffered-Not-Old(obj) then
8.       push obj into LiveStack      // IT WAS IDENTIFIED ALIVE
9.     else
10.      replica := Read-Sliding-View(obj)
11.      for each o in replica do
12.        if !is-Acyclic(o) then
13.          Mark(o,false)
14.  else      // PREVIOUSLY MARKED
15.    obj.CRC--
end
```

Figure 5.8: Cycle collector: Mark Procedure

candidate in the buffer (and therefore, its $CRC$ value should not be decremented). If the object has been reached before, its $CRC$ is not initialized (as it was initialized before), but is decremented. Objects that are alive are pushed onto the *LiveStack*, and their descendants are later colored black (these objects are not traced at this stage).

**Procedure Read-Sliding-View** (Figure 5.9) serves for getting the sliding-view values of a given object. If the object has not been modified since the sliding view was taken, its current values are also its sliding-view values. Otherwise, its pointer slots at the recent sliding view can be found by looking at the log entry that is pointed at by the *LogPointer*[8]. Note that an object may be modified by mutators while the replica is taken (lines 3-5).

[8]Procedure getOldObject actually returns only the non-null pointers saved by Procedure Update. This information suffices for tracing an object correctly.

```
Procedure Read-Sliding-View(obj: object)
begin
1.   if obj.LogPointer = NULL then      // CHECK IF OBJECT HAS BEEN MODIFIED
2.      replica := copy(obj)      // READ ITS DESCENDANTS FROM HEAP
3.      if obj.LogPointer != NULL then      // CHECK AGAIN IF COPIED REPLICA IS VALID
4.            // OBJECT HAS BEEN MODIFIED WHILE BEING READ.
5.         replica := getOldObject(obj.LogPointer)      // GET REPLICA FROM BUFFERS
6.   else      // OBJECT HAS BEEN MODIFIED; USE BUFFERS TO OBTAIN REPLICA
7.      replica := getOldObject(obj.LogPointer)
8.   return replica
end
```

Figure 5.9: Cycle collector: Read-Sliding-View Procedure

```
Procedure Mark-Live-Black
begin
1.   while LiveStack is not empty
2.      obj := pop(LiveStack)
3.      Mark-Black(obj)
end
```

Figure 5.10: Cycle collector: Mark-Live-Black Procedure

**Procedure Mark-Live-Black** (Figure 5.10) colors black the non-black objects in *LiveStack* and their non-black sliding-view descendants. The objects in *LiveStack* were all pushed during the mark stage.

**Procedure Mark-Black** (Figure 5.11) is the actual procedure that colors the sliding-view subgraph of an object as black.

**Procedure Scan-Candidates and Procedure Scan** (Figures 5.12-5.13) perform the scan stage. Each gray candidate in the *oldCandidatesBuffer* with a non-zero *CRC* is considered live (and so are all its sliding-view descendants), and thus the object and its descendants are colored black. Else, it is colored white, and Procedure Scan is invoked on its children. Note that although we use Procedure Mark-Live-Black as the second stage of the algorithm, still an object may be colored white and then re-colored black.

```
Procedure Mark-Black(obj: object)
begin
1.    if obj.color != black then
2.       obj.color = black
3.       replica := Read-Sliding-View(obj)
4.       for each o in replica do
5.          Mark-Black(o)
end
```

Figure 5.11: Cycle collector: Mark-Black Procedure

```
Procedure Scan-Candidates
begin
1.    for each cand in oldCandidatesBuffer do
2.       Scan(cand)
end
```

Figure 5.12: Cycle collector: Scan-Candidates Procedure

**Procedure Collect-White and Procedure Reclaim** (Figures 5.14-5.15) perform the collect stage. Each white candidate is a root of a garbage cycle, and thus these cycles' objects (this candidate and all white objects reachable from it) are colored black and reclaimed. Since we are dealing with a garbage cycle whose objects are reclaimed one by one, one object may still reference another object in the cycle that was just released. Thus, when iterating over the object's descendants, one should check if the descendant is already released (line 4 in Procedure Reclaim). For a similar reason, we also mark any cycle object as released (line 1 in Procedure Reclaim), before iterating over its descendant and actually freeing it.

While reclaiming a garbage cycle, the reference counts of objects referenced by this cycle are decremented. At first, it seems that the reference count of such an object could not reach zero since if it does, then its *CRC* should have reached zero (during the cycle collection), and it would have been colored white (and reclaimed as part of the cycle). However, there are objects that our algorithm does not trace, such as inherently acyclic objects and objects buffered in newer candidate buffers. Such objects could be solely referenced by garbage cycles, and thus when releasing a garbage cycle, their reference count reaches zero. Hence, such objects are released using the reference-counting recursive deletion (line 11 in Procedure

```
Procedure Scan (obj: object)
begin
1.    if obj.color = gray then
2.       if obj.CRC = 0 then
                    // CURRENTLY NO EVIDENCE OF obj BEING EXTERNALLY REACHABLE
3.          obj.color := white
4.          replica := Read-Sliding-View(obj)
5.          for each o in replica do
6.             Scan(o)
7.       else
8.          Mark-Black(obj)      // MARK ITS RELEVANT SUBGRAPH AS ALIVE
end
```

Figure 5.13: Cycle collector: Scan Procedure

Reclaim)[9]. Such recursive deletions can end-up reclaiming black candidate buffered in the *oldCandidatesBuffer* (which motivates line 3 in Procedure Collect-White).

**Procedure Process-Buffers** (Figure 5.16) prepares the next invocation of the cycle collection algorithm by filtering non-relevant candidates and preparing the buffers for the next collection. It first iterates over all the middle buffers, while rejecting candidates which have either died during current collection or were newly buffered during it. In addition, since the oldest buffer of this buffer set would be the oldest buffer in the next collection, the status of its candidates is modified (from *mid-buffered*) to *old-buffered*. Next, it traverses the new buffer, while rejecting candidates that have died in the last (current) collection and changing the status of the remaining candidates to *mid-buffered* (as *newCandidatesBuffer* would be considered as a middle buffer in the next collection). Finally, it performs a cyclic swapping between the buffers' roles.

[9]Note that the recursive deletion, i.e., Procedure RC-Free, modifies the released object status to *non-buffered*.

```
Procedure Collect-White
begin
1.    for each cand in oldCandidatesBuffer do
2.       remove cand from oldCandidatesBuffer
3.       if !is-Released(cand) then
                    // IF NOT RELEASED DURING PREVIOUS CYCLES RELEASES
4.          cand.status := non-buffered
5.          if cand.color = white then      // GARBAGE CYCLE ROOT
6.             Reclaim(cand)
end
```

Figure 5.14: Cycle collector: Collect-White Procedure

## 5.5 An Implementation for Java

We implemented the proposed collector on the Jikes RVM (research virtual machine) [1]. The entire system, including the collector itself is written in Java (extended with unsafe primitives available only to the Java Virtual Machine implementation in order to access raw memory). In this section, we would like to point out some of our implementation choices.

### 5.5.1 Memory Allocator

Our implementation employs the non-copying non-coalescing allocator of Jikes, which is based on the allocator of Boehm, Demers, and Shenker [17]. This allocator is well suited for collectors that do not move objects. Small objects are allocated from per-processor segregated free-lists built from 16KB pages divided into fixed-size blocks. Large objects are allocated out of 4KB blocks with a first-fit strategy. This allocator keeps the fragmentation low and allows efficient reclamation of objects.

### 5.5.2 Object Headers

The object layout of our memory manager within the Jikes RVM version that we used is displayed in Figure 5.17. Jikes' basic object header contains two words[10]. One word of

[10] An array object's header includes an additional length field.

```
Procedure Reclaim (obj: object)
begin
1.   mark obj as released      // SO THAT THE is-Released PROCEDURE DESCRIBED IN
                               // SECTION 5.4.2 WOULD IDENTIFY IT AS RECLAIMED
2.   // NO NEED TO CHECK LOGPOINTER: obj IS A CYCLIC GARBAGE OBJECT
3.   for each child child of obj do
4.     if !is-Released(child) then
5.       if child.color = white then
6.         Reclaim(child)
7.       else
8.         child.RC--
9.         if child.RC = 0 then      // child IS NOT PART OF THE CYCLE
10.              // THE RC COLLECTOR SHOULD FREE IT
11.          RC-Free(child)      // RECURSIVE DELETION
12.  obj.color := black
13.  return obj to the general purpose allocator.
end
```

Figure 5.15: Cycle collector: Reclaim Procedure

the header is a *status* word supporting memory management, synchronization, and hashing. The second word of the header holds a reference to the *Type Information Block (TIB)* for the object's class (which serves as Jikes' virtual method table).

In order to support reference counting and cycle collection, our implementation employs an additional two words in an object's header. The first word is used to hold the *LogPointer* of an object (detailed in Section 5.4.1). Similarly to the mechanism of Bacon et al. [6], the second word holds the reference counts, the color, and the buffer status. As we use three colors, two bits are necessary for color representation. Two bits are also needed to represent the four buffered states (discussed in Section 5.4). The rest is devoted to representing the RC and *CRC*. Each count includes an overflow bit. When the overflow bit is set, the excess count is stored in a hash table. In practice, this hash table never contains more than a few entries.

```
Procedure Process-Buffers
begin
1.   // FILTER CANDIDATES AND CHANGE STATUSES
2.   for each buff which is midCandidatesBuffer do
3.     for each cand in buff do
4.       if is-Released(cand) ∨ cand.status = newly-buffered then
5.         remove cand from buff
6.       else if buff is the oldest midCandidatesBuffer buffer then
7.         cand.status := old-buffered
8.   for each cand in newCandidatesBuffer do
9.     if is-Released(cand) then
10.      remove cand from newCandidatesBuffer
11.    else
12.      cand.status := mid-buffered
13.  // SWAP-BUFFERS-ROLES
14.  tempBuffer := oldCandidatesBuffer
15.  oldCandidatesBuffer := oldest midCandidatesBuffer buffer
16.  make newCandidatesBuffer a midCandidatesBuffer
17.  newCandidatesBuffer := tempBuffer
end
```

Figure 5.16: Cycle collector: Process-Buffers Procedure

### 5.5.3 Triggering

In a stop-the-world garbage collector setting, mutators halt during a garbage collection. However, with concurrent collectors, mutators run during a collection and hence also consume memory. Therefore, when triggering a concurrent collection, our goal is, on one hand, to trigger a collection as late as possible so that we get as few collections as possible (to avoid garbage collection overheads). On the other hand, we would like to trigger collections early to ensure that they will complete their work before the mutators consume all available memory (otherwise, the mutators would have to halt, waiting for the collector thread to free memory). Our triggering mechanism keeps an estimation of the work the next collection will have to deal with and an estimation of the amount of free memory available. This work estimation

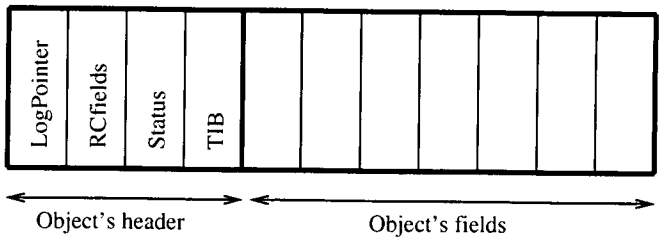



Figure 5.17: SVRC: The object model

is based on the number of objects created since the last collection and the number of (old) objects modified since the last collection. Whenever the ratio between the amount of work and the estimated available memory goes below a certain threshold, a collection is triggered.

The triggering of a reference-counting collection is similar when comparing the backup mark-and-sweep algorithm and the cycle collection algorithm. The backup mark-and-sweep collection is triggered instead of a reference-counting collection every fixed number of reference-counting collections whereas the cycle collector is run with each collection. The two alternatives differ in their floating garbage creation. The backup mark-and-sweep collector collects a garbage cycle in the first mark and sweep run after its formation. Hence, the average time required to collect a garbage cycle depends on (and is bounded by) the frequency of the mark-and-sweep activation. On the other hand, the cycle collection algorithm collects a garbage cycle when the corresponding candidate is processed. Since the cycle collector examines only the candidates that were accumulated $k$ collections ago, a garbage cycle is collected $k$ reference-counting collections after it was formed. Consequently, with both the cycle collector and the backup tracing collector, one can bound the number of reference-counting collections required for a garbage cycle's reclamation. The difference between the pace of collecting garbage cycles does not make a noticeable difference in the floating garbage accumulation for most benchmarks, because the fraction of garbage cycles is usually negligible; see Table 5.2 in Section 5.6.3.

### 5.5.4 Buffer implementation

The reference-counting collector and its cycle collection extension use several kinds of buffers, such as the *Roots* buffer, the *Updates* buffer, the *YoungObjects* buffer, the candidate buffers, and the *LiveStack* buffer. Although these buffers represent different kinds of information, all of them contain addresses of objects (and in case of the *Updates* buffer, also tagged addresses), and thus they all have the same basic buffer implementation. As common for garbage collectors implemented in Jikes, the memory of these buffers is set aside permanently.

These buffers are implemented by fixed-sized blocks in memory that contain addresses. We maintain a pool of such blocks that could be used for each one of the above buffers. If a block becomes full, another block is taken from the pool (in case the pool becomes empty, another block is allocated), and the newly taken block is connected to the previously used block (and vice versa). Whenever a block is no longer is use, it is returned to the block pool.

The *Roots* buffer, the *Updates* buffer and the *YoungObjects* buffer were all employed originally by the reference-counting sliding-view collector. The candidate buffers and the *LiveStack* buffer are buffers added in order to support cycle collection. The *LiveStack* buffer is maintained only during the cycle collection algorithm, but the candidate buffers are kept throughout the program. Each collection acquires a new candidate buffer at the beginning of the reference-counting collection, and releases the oldest candidate buffer at the end of the cycle collector collection.

### 5.5.5 Root set

The root set of the sliding-views reference-counting collector includes its global variables, its static variables, the contents of each thread's run-time stack and the snooped objects. The original implementation of the sliding-view reference-counting collector did not mark the roots explicitly. Instead, in the beginning of a collection, when determining the root set (and adding the addresses of objects referenced by the roots into the *Roots* buffer), the collector incremented the reference-count field of each root object, so that it would not be reclaimed in the current collection. At the end of a collection, the collector re-traversed the root set (i.e., objects referenced by the *Roots* buffer) decrementing the reference-count field of each root object.

During Procedure Mark (introduced in Figure 5.8), the cycle collection algorithm needs to know whether a traversed object belongs to the root set, in order to perform two of the optimizations described in Section 5.3.5 for the root set. To do this efficiently we mark all root objects (at the beginning of the collection) as root objects using a designated bitmap. Hence, checking whether an object belongs to the root set, involves only the relevant bit probation. When implementing these optimizations, one has to take this extra space overhead into account.

Note that the objects belonging to the root set, which have a zero reference count, are checked in the next collection to see whether they have become garbage. The objects belonging to this set, which have a non-zero reference count and are not referenced by the system roots in the next collection, are pushed into the candidate buffer of the next collection:

the number of pointers to these objects have been decreased between the two collections.

### 5.5.6 Candidate buffers

The candidate buffers are implemented using our standard buffer implementation described in Section 5.5.4. Inserting a candidate into this buffer (as described in Figure 5.5) requires only putting its address in the buffer, and modifying its status bit. Removing an object from the candidate buffers is done at various occasions by several procedures as presented in Figures 5.7, 5.14 and 5.16. Note that all of these instances occur while traversing a candidate buffer. Consequently, removing a candidate from the buffer never involves a sequential search for this candidate. While removing candidates we usually pack the valid entries in the buffer at the head of the buffer, thus, reducing overall buffer usage.

## 5.6 Measurements

**Platform and benchmarks**. We ran our measurements on a 4-way IBM Netfinity 8500R server with 550MHz Intel Pentium III Xeon processors and 2GB of physical memory. We used the SPECjvm98 benchmark suite and the SPECjbb2000 benchmark (both described in SPEC's web site [90]). We feel that the multithreaded SPECjbb2000 benchmark is more interesting, as the SPECjvm98 benchmarks are mostly single-threaded.

**Testing procedure.** We used the benchmark suite using the test harness, performing standard automated runs of all the benchmarks in the suite. Our standard automated run runs each benchmark five times for each of the JVM involved (each implementing a different collector). Finally, to understand the behavior of our collector better under tight (in Jikes) and relaxed conditions, we tested it on varying heap sizes. For the SPECjvm98 suite, we started with a 32MB heap size and extended it by 8MB increments until reaching a final large size of 96MB. For SPECjbb2000 we used larger heaps, starting from a 256MB heap size and extending it by 64MB increments until reaching a final large size of 704MB.

**The compared collectors.** We incorporated the cycle collection algorithm into two collectors: the Levanoni-Petrank reference-counting collector, and the more efficient age-oriented collector. Both collectors are also implemented in Jikes and are accompanied by a backup mark-and-sweep collector that is run infrequently to collect garbage cycles. For performance measurements, we ran both collectors accompanied by our cycle collection algorithm against both collectors when using the backup mark-and-sweep algorithm. In addition, we com-

pared characteristics of our cycle collection algorithm (with both collectors), against the characteristics of the previous on-the-fly cycle collector of Bacon and Rajan [6].

### 5.6.1 Performance

Our main benchmark is SPECjbb2000. SPECjbb2000 requires a multi-phased run with an increasing number of warehouses. The benchmark provides a measure of the throughput and we report the throughput ratio improvement when applied with the proposed cycle collection algorithm (compared to the same collector with a backup mark-and-sweep algorithm). Thus, the higher the ratio, the better our algorithm behaves, and in particular, any ratio larger than 1 implies that the cycle collector outperforms the tracing auxiliary collector.

The SPECjbb2000 benchmark reports the throughput achieved, and hence Figure 5.18 depicts the throughput ratio between using the cycle collector and a backup tracing collector when both are used with the Levanoni-Petrank collector on a varying number of warehouses and heap sizes. Note that with 1-3 warehouses the collector has a spare processor to run on, since the platform has four processors. In this case, throughput differences occur only when the collector is not efficient enough to free enough space for program threads on-going allocations. This is more noticeable with tight heaps. With 4-8 warehouses, the collector does not have a spare processor and its use of CPU directly affects the throughput. The tracing backup collector outperforms the cycle collector usually by 5%-10%.

The same measurements were run when the cycle collector and the backup tracing collector were used with the age-oriented collector; see Figure 5.19. Here, only old objects are collected with reference counting, and thus the cycle collector runs only on old candidates. In this case, there is not much difference between the two options for collecting cycles, except for tight heaps. Here cycle collection is preferable on backup tracing (as an add-on to reference counting) when the heap is tight.

When running the SPECjvm98 benchmarks on a multiprocessor, the collector runs concurrently with the program thread(s) on a spare processor. Figure 5.20 depicts the results both with the Levanoni-Petrank reference-counting collector as well as with the age-oriented collector[11]. The results do not point to a clear winner. Each application behaves somewhat differently and most of the differences are below 5%. The only clear noticeable difference is

[11]The SPECjvm98 benchmarks provide a measure of the elapsed running time. Here, the smaller the better. Thus, when reporting SPECjvm98 results we report the running time ratios. For clarity of presentation, we report the inverse ratio, so that higher ratios still show better performance of the cycle collector algorithm.

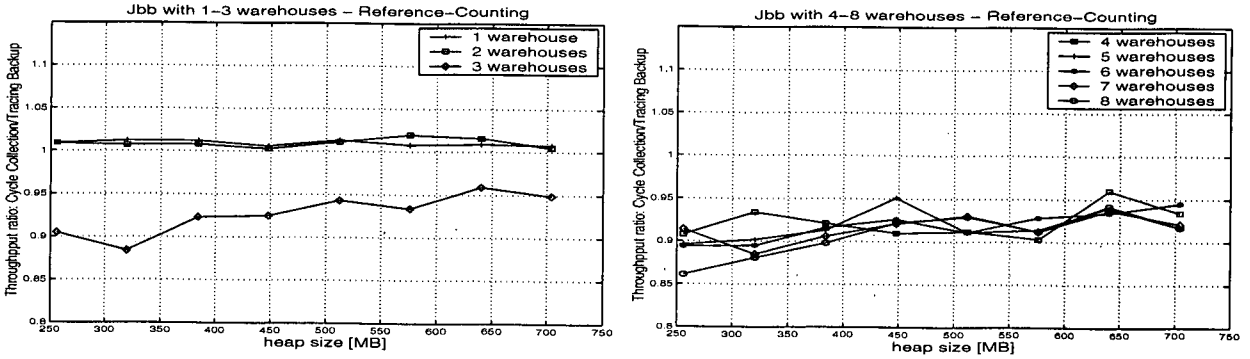



Figure 5.18: SPECjbb2000 on a multiprocessor: Cycle collection throughput ratio for the Levanoni-Petrank collector

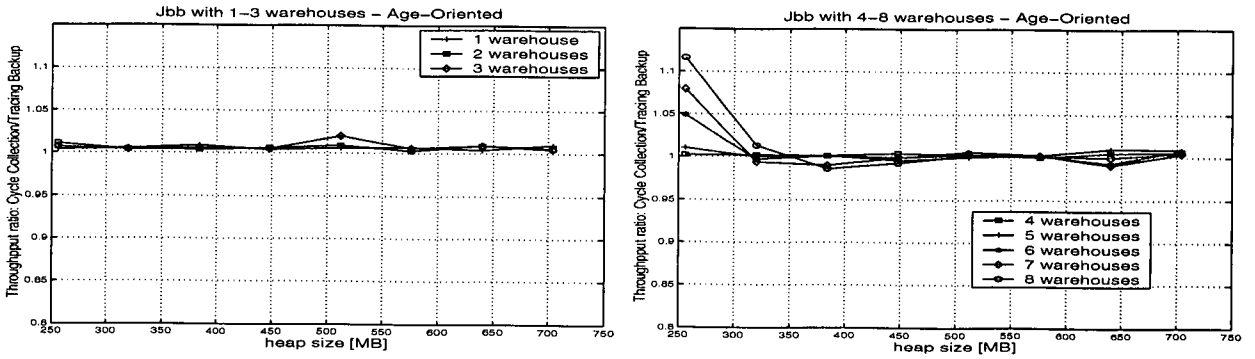



Figure 5.19: SPECjbb2000 on a multiprocessor: Cycle collection throughput ratio for the age-oriented collector

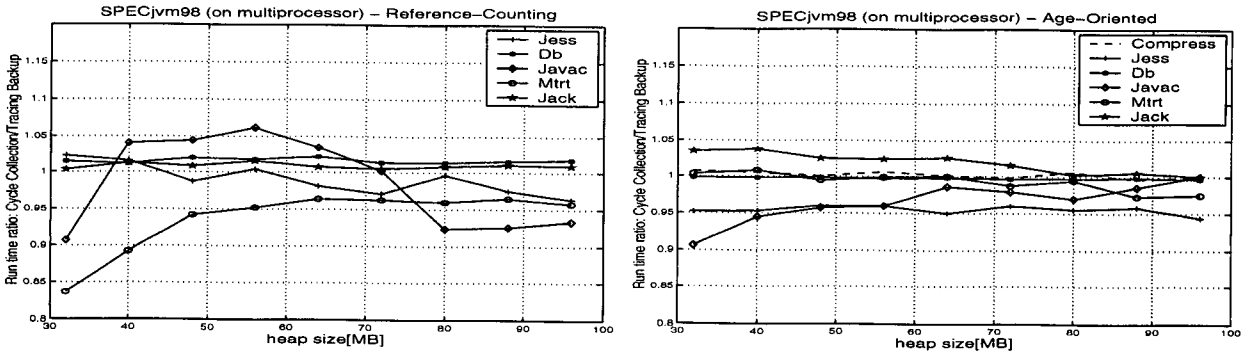



Figure 5.20: SPECjvm98 on a multiprocessor: Run-time ratio with cycle collection

| Benchmarks | compress | jess | db | javac | jack | mtrt | jbb-1 | jbb-2 | jbb-3 |
|---|---|---|---|---|---|---|---|---|---|
| Max. pause time | 1.0 | 1.3 | 0.7 | 1.7 | 1.0 | 0.9 | 0.8 | 0.6 | 1.1 |

Table 5.1: Cycle collection maximum pause time in milliseconds

with the _227_mtrt benchmark. The reason for this difference is that, for this benchmark, there exists an initial phase in which many objects are created and kept alive till the end of the run. These newly created objects induce a large amount of work on the cycle collector. During the (single) long collection, the mutators halt waiting for free space. The performance difference is noticeable only with the reference-counting collector, and not with the age-oriented collector. There, the cycle collector is not run on this pack of live young objects.

### 5.6.2 Pause times

We measured the maximum pause times of the Levanoni-Petrank reference-counting collector accompanied by our cycle collection algorithm. The maximum pause times for the runs of the SPECjvm98 benchmarks and the SPECjbb2000 benchmark are reported in Table 5.1. The SPECjvm98 benchmarks were run with a 64MB-sized heap and the SPECjbb2000 (with 1,2,3 warehouses) were run with a 256MB-sized heap size. In these measurements, the number of program threads is smaller than the number of CPU. Note that if the number of threads exceeds the number of processors, then large pause times appear because threads lose the CPU to other mutators or the collector. The length of such pauses depends on the operating system scheduler and is not relevant to the garbage or cycle collector. Hence we report only settings in which the collector runs on a separate spare processor.

The maximum pause time measured for all benchmarks was 1.7 ms. The maximum pause time of the Levanoni-Petrank reference-counting collector does not depend on whether it is accompanied by a tracing backup or by a cycle collector. The operation that determines the length of the pause time is the scanning of the roots of a single thread, which occurs in one of the handshakes.

### 5.6.3 Collector characteristics

**Amount of cyclic garbage.** Table 5.2 provides, for each benchmark, the number of garbage cycle objects reclaimed, the space they consume and the (space) fraction they consume

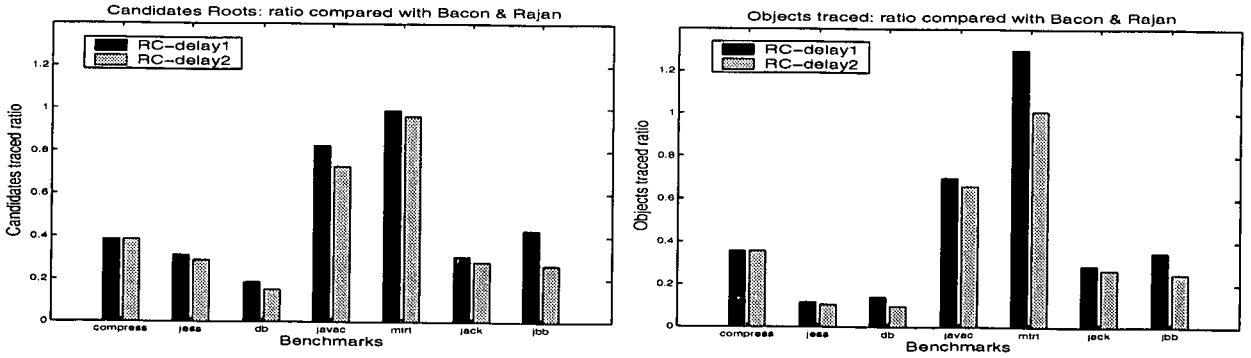



Figure 5.21: Cycle collection: Reduction in the tracing work and in the number of candidates compared to the collector of Bacon and Rajan

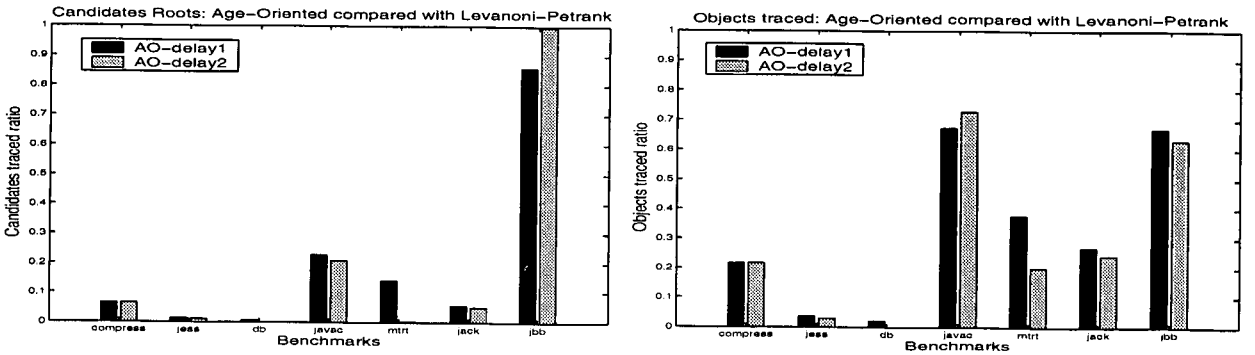



Figure 5.22: Cycle collection: Reduction in the tracing work and in the number of candidates when the age-oriented collector is used compared to the reference-counting collector

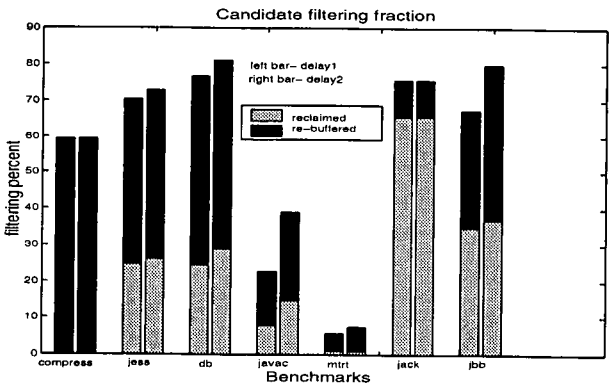



Figure 5.23: Cycle collection candidate filtering: Percentage of candidates filtered out

| Benchmarks | RC | | | AO | | |
|---|---|---|---|---|---|---|
| | cyclic objects reclaimed | cyclic bytes (in MB) | cyclic bytes fraction | cyclic objects reclaimed | cyclic bytes (in MB) | cyclic bytes fraction |
| compress | 108 | 84.08 | 40.4% | 0 | 0 | 0% |
| jess | 24 | 0.15 | 0% | 0 | 0 | 0% |
| db | 16 | 0.09 | 0% | 0 | 0 | 0% |
| javac | 1 M | 67.64 | 34.7% | 0.57 M | 37.02 | 19.0% |
| mtrt | 66052 | 5.78 | 1.5% | 66042 | 5.66 | 1.5% |
| jack | 8976 | 1.72 | 0.3% | 3360 | 0.62 | 0.1% |
| jbb | 146 | 0.88 | 0.1% | 0 | 0 | 0% |

Table 5.2: Cyclic garbage collected for each benchmark by our cycle collector, when incorporated with the reference counting and the age-oriented collectors

| Benchmarks | compress | jess | db | javac | mtrt | jack | jbb |
|---|---|---|---|---|---|---|---|
| Candidate buffers maximal size | 45.3 | 70.3 | 260.2 | 1136.7 | 250.6 | 46.1 | 9549 |

Table 5.3: Candidate buffers maximal size (in KB) with the reference-counting sliding-view collector (employing the cycle collector)

compared to the overall garbage size (i.e., including non-cyclic garbage). As the age-oriented collector only employs a cycle collection on old objects, it sometimes reclaims a smaller set of garbage cycles than the reference-counting collector.

The only benchmarks that produce a substantial amount of space in garbage cycles are _213_javac and _201_compress. In _201_compress, there are some dozens of garbage cycles composed of huge objects, and it, therefore, requires only a small amount of tracing. The _213_javac benchmark on the other part, contains thousands of garbage cycles, thus requiring a large amount of work of the cycle collector.

**Candidate buffers space overhead.** Table 5.3 provides, for each benchmark, the maximal space consumed by the candidate buffers while the benchmark was run with the reference-counting sliding-view collector. We measured the space consumed by the candidate buffer in each collection, and we report the maximal space recorded. These measurements were taken while SPECjvm98 benchmarks were run with a 64MB-sized heap and

SPECjbb2000 was run with a 256MB-sized heap.

**Amount of tracing.** To check that the proposed collector indeed traces much fewer objects than previous collectors, we compared it to the previous cycle collector of Bacon and Rajan [7]. To make the comparison fair, the new collector was measured only with the reference-counting collector. We did not compare the cycle collector of Bacon and Rajan incorporated into the age-oriented collector, as the original implementation of Bacon and Rajan does not include an age-oriented version. We report the ratios of the *candidates examined* and the ratio of *objects traced* when compared to those of [7]. To be extremely conservative we did not include the objects scanned during the additional verification phase of the algorithm in [7]. Thus, the actual advantage of the new collector is even higher than reported. The reason that we did not count the additional phase is that, in this additional phase, some of the objects were not actually *traced*. For these objects the actual operation only included work on their colors.

In the graphs presented in Figure 5.21, the lower the ratio, the better the behavior of the new algorithm, and any ratio smaller than 1 implies that it traced fewer candidates and objects. The collector characteristics were measured with two and three candidate buffers. In the first case, the implication is that candidates gathered until the current collection will be considered only at the next collection, i.e., there is a delay of one collection before handling the collected candidates. In the latter case, there is a delay of two collections. We thus denote these two cases *delay1* and *delay2*.

Both configurations traced fewer candidates compared to the previous cycle collector (of [7]) over all benchmarks. Also, when checking how many objects were traced, the new cycle collector traced substantially fewer objects except for one case: the _227_mtrt benchmark.

Comparing a delay of one collection (two buffers) and delay of two collections (three buffers), we see that the algorithm traces less with three buffers. This means that by delaying the handling of a candidate, its traversal may sometimes be spared. However, when measuring the benchmarks' throughput in these two cases, there was mostly no clear throughput superiority. This is due to the fact that the handling of candidate buffers (filtering candidates each collection) also consumes time. Hence, one should tune the number of candidate buffers used to its collector and its collection triggering policy. For our collectors, using two candidate buffers was usually enough, but for collectors that run frequent collections, we believe that using more buffers would improve efficiency.

In Figure 5.22, we report the additional saving when the cycle collector is used with the age-oriented collector (on the old generation only). The same cycle collector is used.

As can be seen, the further reduction in tracing is substantial for most benchmarks. This demonstrates the effectiveness of the age-oriented approach when cycle collection is used.

**Candidates filtering.** We measured the effect of each candidate filtering strategy we added, by measuring the fraction of candidates filtered out of all candidates identified. In Figure 5.23, the fraction of candidates that were filtered out is presented for delays of one and two cycles with the base reference-counting collector (of [63]). The blue portion represents the objects that were re-buffered during the delay (and were thus filtered out). The turquoise portion represents the objects that were reclaimed during the delay (and were thus filtered out). For each benchmark, the left bar represents a delay of one collection cycle, while the right bar represents a delay of two collection cycles. We distinguish between filtering by reclamation and filtering by re-buffering. The latter means that the object will be re-considered after $k$ cycles. We stress that these filtering techniques are executed on the candidates that have survived the filtering of acyclic objects used by Bacon and Rajan (in [7]), which is reported to be highly effective to start with. The fraction we present is the *additional* benefit obtained by the new filtering methods.

The new filtering techniques usually filter between 40%-80% of the surviving candidates. The only benchmark for which these techniques are not effective is the _227_mtrt benchmark, where only 7.26% of the candidates were eliminated. This may explain why _227_mtrt is the only benchmark for which the number of candidates traced is similar to that of [7]. It can be seen in Figure 5.23 that candidates are effectively eliminated by both methods. Finally, the first cycle delay is far more effective in delaying the check than the second cycle. Nevertheless, for some benchmarks (such as _213_javac and the SPECjbb2000 benchmarks) the filtering benefit obtained at the second cycle delay is non-negligible.

### 5.6.4 Wasted work

The cycle collector traces candidates. Some of the tracing ends up reclaiming cyclic structures, but others identify reachable structures. The latter may be thought as a "waste" of the collector's work. We measured the overall number objects that the cycle collector traced and the number of objects that were traced but not reclaimed. The fraction of wasted work is reported in Table 5.4.

In order to appreciate the amount of redundant work executed, one must also check the overall amount of tracing the collector has executed. For example, _201_compress, _202_jess and _209_db required above 90% percent of wasted work, but the cycle collector traced (sometimes substantially) fewer then 50,000 objects in each of these benchmarks. In fact, the

| Benchmarks | Levanoni-Petrank | | Age-Oriented | |
|---|---|---|---|---|
| | wasted work percent | number of traced objects | wasted work percent | number of traced objects |
| compress | 91.2% | 3674 | 100% | 792 |
| jess | 99.8% | 46830 | 100% | 2128 |
| db | 99.9% | 38548 | 100% | 1122 |
| javac | 43.9% | 5347924 | 55.1% | 3808987 |
| mtrt | 82.4% | 1127601 | 63.6% | 544145 |
| jack | 26.3% | 36554 | 5% | 10607 |
| jbb | 100% | 11917616 | 100% | 11127115 |

Table 5.4: Number of traced objects (by cycle collection), and the percent of futile tracing

amount of tracing is substantial only for the _213_javac, the _227_mtrt and the SPECjbb2000 benchmarks, out of which only _213_javac indeed contains many cycles. SPECjbb2000 is a long running program, requiring many garbage collections, which is the reason for the large amount of object tracing (although it hardly contains any cycles). Reasons for extra object tracings in _227_mtrt were discussed above.

## 5.7 Conclusions

We presented a new non-intrusive, complete, and efficient cycle collector adequate for use with a reference-counting garbage collector. The new cycle collector runs concurrently with the program threads, achieving negligibly short pauses of less than 2ms. It uses the sliding-views reference-counting collector of Levanoni and Petrank [62, 63] together with the synchronous cycle collector of Bacon and Rajan [7]. These algorithms do not naturally fit together since the original cycle collector requires a list of all reference-count decrements, whereas the original reference-counting collector is oblivious to most of these decrements. However, we provide a finer analysis of cycle collection, showing that the information gathered by the reference-counting collector is enough to guarantee reclamation of all unreachable cycles.

Building on the sliding-views mechanism yields a drastic improvement in efficiency. Much

of the work required to ensure concurrent correctness may be eliminated. We also add filtering techniques to further optimize the collector's performance. An additional theoretical contribution is the completeness of the collector. The resulting cycle collector is guaranteed to reclaim all garbage cycles, whereas the only available previously known concurrent collector [7] had a (rare) sequence of events that prevented it from collecting an unreachable cyclic structure forever.

We implemented the proposed cycle collector and provide the first direct comparison of running reference counting with a cycle collector against running reference counting with a backup tracing collector. Our results show that when reference counting is used to collect the entire heap, the backup tracing collector outperforms the cycle collector. However, when using the recommended setting in which reference counting is used for old objects and tracing is used for young objects, the cycle collector performed equally to the backup tracing collector, and even better on tight heaps.

# Chapter 6

# Improving the Memory Behavior of a Reference-Counting Garbage Collector via Prefetching

## 6.1 Introduction

The performance gap between memory latency and processors' speed is increasing, causing memory accesses to become a performance bottleneck. Cache hierarchies are used to reduce this gap, by keeping the currently used data close to the processor. Although the cache memory is faster to access than main memory, caches are of limited size and usually cannot hold the application's entire working set. This may cause a significant number of cache misses, yielding a considerable memory stall time (waiting for the data to be fetched onto the cache). Moreover, cache misses are expected to become even more significant in the future.

Data prefetching is a technique for reducing or hiding the memory stalls caused by cache misses. Using prefetching, data could be brought onto the cache in advance, thus hiding the latency of loads that miss the cache, and improving the overall program execution time (as prefetch is a non-blocking memory operation). On the negative side, prefetching suffers from several disadvantages such as an increase in memory traffic, cache pollution, and increase in the number of executed instructions. In addition, to achieve performance improvement, prefetch scheduling should be done with care. Data prefetched too early may be evicted

from the cache before used, while a late prefetch will not mask the system latency.

Compiler-inserted data prefetching have been proposed for predictable access patterns such as accesses to arrays, linked list and other pointer applications (e.g., [18, 67, 99]). Boehm [16] noticed that memory stalls, caused due to cache misses, consume a large fraction of a mark-sweep garbage collection running time, causing the collector to spend considerable time waiting for memory. In particular, tracing collectors traverse the application's live objects, where each live object is likely to be read exactly once, in a random order, during each garbage collection. As a considerable fraction of the heap is accessed during the collection, and since the heap is appreciably larger than the cache, most accesses yield a cache miss. However, since the addresses of the next objects to be traversed can usually be determined in advance, prefetch can be used to prevent these cache misses.

Previous work [16, 18, 22, 98] showed that prefetching is able to reduce the cost of a tracing garbage collection by improving memory performance. In this work we show that, similarly to a tracing collector, the data accessed by a reference-counting collector [26] can be anticipated. Thus, prefetching can be used to reduce the cache misses' overhead incurred by a reference-counting collector and improve the collector efficiency.

The representative reference-counting collector studied in this work employs the coalescing write-barrier introduced by [62, 63]. The coalescing write-barrier strategy eliminates a large fraction of the reference-count updates and drastically improves over the overall efficiency of the traditional reference-counting collector. This write barrier records the first pointer update of each object after a garbage collection cycle. A dirty flag indicates if an object has already been modified. If not, then before executing the pointer modification, all pointers in the object (i.e., all its current descendants) are written to a buffer. During the next collection, the reference counts of all these descendants are decremented and the reference counts of all new descendants of the object are incremented. Note that the count updates are not executed in the write barrier, but in the next collection.

We consider three main parts of the memory manager:

1. The reference-count increments stage,
2. the reference-count decrements and object deletion stage, and
3. the objects' allocation stage.

In accordance with these stages, we have identified five major opportunities where data accesses can be predicted in advance, and prefetch instructions may be inserted to improve

performance. These opportunities are spread across the three different reference-counting stages. We study each of these opportunities and measure the improved performance for each of the stages independently, and the overall garbage collection performance improvement as well.

**Implementation and measurements.** We have implemented the proposed prefetching insertions with a slightly modified version of the reference-counting collector supplied with the Jikes RVM [1]. We used the SPECjbb2000 benchmark, the SPECjvm98 benchmarks suite (both described in SPEC's web site [90]) and the DaCapo benchmarks suite [27]. Measurements were carried out on a dual Intel's Xeon 1.8GHz processors workstation. As reported in Section 6.5 below, it turns out that unlike tracing collectors, for which each live object is accessed a single time during the mark phase, most objects are accessed multiple times by the reference-counting collector. Moreover, repetitive accesses tend to be close in time. This reduces the potential benefits obtainable by prefetching. Nevertheless, our proposed prefetch insertions were able to reduce garbage collection overhead by as much as 14.9%, ultimately enabling an overall application speedup of 4.6%. On average, a 8.7% reduction in memory management overheads and a 2.2% overall application speedup were measured across all benchmarks.

**Chapter organization.** We start with reviewing the reference-counting collector in Section 6.2. The prefetch insertion opportunities of the reference-counting collector are presented in Section 6.3. Implementation and results are reported in Sections 6.4 and 6.5. Related work is discussed in Section 6.6 and we conclude in Section 6.7.

## 6.2 The reference-counting collector

In this work, we propose inserting prefetch instructions into a reference-counting collector in order to improve its efficiency. Section 6.3 describes the prefetch insertion opportunities. We start with a review of the reference-counting collector and a pseudo code that will be used to explain the possible insertions.

The basic idea underlying any reference-counting collector is to keep a reference count field for each object telling how many references exist to the object. Using the naive approach, whenever a pointer is updated the system invokes a *write barrier* that updates the reference counts. In particular, if the pointer is modified from pointing to $O_1$ into pointing to $O_2$, then the write barrier decrements the count of $O_1$ and increments the count of $O_2$. When the counter of an object is decremented to zero, it is reclaimed. The reference counts of all

its children are then decremented as well possibly causing more reclamations recursively.

Deutsch and Bobrow [31] eliminated most of the computational overhead required to adjust reference counters in their method of deferred reference counting. According to the method, local references are not counted; thus the need to track fetches, local pointer duplication and cancellation are deemed unnecessary. Only stores into the heap need be tracked. However, the immediacy of reference counting is lost to a certain extent, since garbage may be reclaimed only after the mutator state is scanned and accounted for.

Even with deferred reference counting, traditional use of reference counts requires frequent count updates. Whenever a heap reference is destroyed or overwritten, the reference count of the object it references is decremented, and whenever one is created or copied, the reference count of the object it references is incremented. The Levanoni-Petrank coalescing write barrier (presented in [63]) eliminates the vast majority of the reference-count updates by recording information on modified objects and using it to update the reference counts during garbage collection time. Consider a pointer slot that, between two garbage collections is assigned the values $o_0, o_1, o_2, \ldots, o_n$. Reference-counting collectors execute $2n$ reference-count updates for these assignments: RC($o_0$)$--$, RC($o_1$)$++$, RC($o_1$)$--$, RC($o_2$)$++$, $\ldots$, RC($o_n$)$++$. However, it is observed that only two are required: RC($o_0$)$--$ and RC($o_n$)$++$.

In order to be able to use this observation, the coalescing write barrier was proposed. Each object is associated with a dirty flag which is cleared during the collection. Then, during program run, whenever a pointer is modified, the dirty bit of the object holding this reference is probed. If the object is dirty (i.e., has been modified since the last collection), then the pointer assignment may proceed with no further action. Otherwise, the pointer slots of the object are copied to a thread-local buffer before the assignment is executed. That way, the "$o_0$" value of a modified slot (the value of the slot in the previous collection) is exactly the value recorded by the write barrier when the slot is modified. The "$o_n$" value of a modified slot is the value of the slot in the current garbage collection time. For a detailed description and motivation see [63]

To simplify this work and the interpretation of the results, we haven't used the full power of the collector described in [63]. In particular, the collector we have used works in a stop-the-world manner. An involved mechanism is developed in the original paper [63] to support collector concurrency and application parallelism.

```
Procedure Update(o: object, offset: int , new: object)
begin
1.   if not o.dirty then      // OBJECT NOT DIRTY
2.      add o to ModBuffer
3.      o.dirty :=true     // SET DIRTY
4.      foreach field ptr of o which is not NULL
5.         add ptr to DecBuffer
6.   write( o, offset ,new)
end
```

Figure 6.1: Reference counting: Update Operation

### 6.2.1 Pseudo code

Next, we present a general pseudo code of a reference-counting collector which employs the coalescing write barrier. In the below pseudo code we assume the existence of two buffers:

- *ModBuffer:* contains the addresses of the objects which were created or modified since the previous collection.
- *DecBuffer:* contains the addresses of the objects which were referenced in the previous collection by objects in the *ModBuffer*, i.e., referenced by objects that were modified since the previous collection.

#### Mutator cooperation

The mutators need to execute garbage-collection related code on two occasions: when updating an object and when allocating a new object. This is accomplished by the Update (Figure 6.1) Procedure and the New (Figure 6.2) Procedure, respectively.

**Procedure Update** (Figure 6.1) describes the write barrier which is activated at each (heap's) pointer assignment (ignoring synchronization details). During the first modification of an object after a collection, the write barrier records the modified object in *ModBuffer* and it sets its dirty bit. Next, the modified object's pointers are recorded in the *DecBuffer*. After the logging has occurred, the actual pointer modification happens.

**Procedure New** (Figure 6.2) is used when allocating an object. Upon the creation of an object, its address is logged onto *ModBuffer*, and the *dirty* bit of the new object is set. There is no need to record its children slot values as they are all null at creation time.

```
Procedure New(size: Integer, obj: Object)
begin
1.  Obtain an object obj of size size from the allocator.
2.  insert the address of obj into ModBuffer
3.  obj.dirty := true
4.  return obj
end
```

Figure 6.2: Reference counting: Allocation Operation

```
Procedure Collection-Cycle
begin
1.  accumulate all object directly reference by the program roots onto Roots
2.  Process-ModBuffer
3.  Process-DecBuffer-and-Release
4.  Prepare-Next-Collection
end
```

Figure 6.3: Reference counting- Collection Cycle

### Phases of the collection

The collector's algorithm runs in phases as follows.

- **Mark roots:** the objects directly reachable from the program roots are marked.
- **Process ModBuffer:** The collector increments the reference count of the current descendants of objects logged in *ModBuffer*, while clearing the dirty marks of these objects.
- **Reclaim garbage:** the collector decrements the reference counts of the objects logged in *DecBuffer* while reclaiming objects (and their descendants using recursive deletion) which have a zero reference count and which are not referenced by the system roots.
- **Prepare next collection:** un-marks the objects referenced from the program roots and prepares the buffers for the next collection.

### Collector code

```
Procedure Process-ModBuffer
begin
1.    for each object obj whose address is in ModBuffer do
2.       obj.dirty := false
3.       // INCREMENT CURRENT REFERENT OF THE OBJECT obj
4.       for each pointer ptr of obj do
5.          increment rc of object referenced by ptr
end
```

Figure 6.4: Reference counting- Process-ModBuffer

```
Procedure Process-DecBuffer-and-Release
begin
1.    for each object obj whose address is in DecBuffer do
2.       obj.rc--
3.       if obj.rc = 0 ∧ obj ∉ Roots then
4.          for each pointer ptr of obj do
5.             push ptr onto DecBuffer
6.          return obj to the general purpose allocator.
end
```

Figure 6.5: Reference counting- Process-DecBuffer-and-Release

The reference-counting collector's code for collection cycle $k$ is presented in **Procedure Collection-Cycle** (Figure 6.3). The collector's procedures follow.

**Procedure Process-ModBuffer** (Figure 6.4) handles the objects logged in *ModBuffer*. These are all the objects that were modified or created since the previous collection cycle. This procedure first clears the dirty bit of an object logged in *ModBuffer*, and then increments the reference count of the objects it references.

**Procedure Process-DecBuffer** (Figure 6.5) decrements the reference counts of objects logged in *DecBuffer*, and performs the recursive deletion if necessary.

**Procedure Prepare-Next-Collection** (Figure 6.6) cleans the *Roots*, *ModBuffer* and *DecBuffer* buffers.

```
Procedure Prepare-Next-Collection
begin
1.   Roots := ⊘
2.   ModBuffer := ⊘
3.   DecBuffer := ⊘
end
```

Figure 6.6: Reference counting- Prepare-Next-Collection

### 6.2.2 Allocation using segregated free lists

A garbage collector is accompanied by a memory allocator that serves the application's allocations requests and the collector's reclamations requests. The collector's job is to find unreachable objects and tell the allocator that these objects may be reclaimed and subsequently reallocated. A standard allocator that is used with our reference-counting collector (and other non-moving collectors) is the segregated free lists allocator. A couple of prefetch insertion opportunities that we propose relate to this allocator. We review this allocator below.

Conceptually, a segregated free lists allocator [102] employs multiple linked lists of available memory. Each free list holds chunks of a particular size. Upon an allocation request, a chunk is taken from the free list of the appropriate size. When a chunk of memory is freed, it is returned to the appropriate free list according to its size.

The implementation we use (the one delivered with Jikes RVM) uses a block-oriented segregated free lists [17] that works as follows. The heap is divided into blocks, and each block only holds objects of a single size. Thus, a block with an associated size is partitioned to chunks of that size. The free list of any given size consists of a chain of blocks. Each block has an associated bit-per-chunk mark array (bitmap), which records the occupancy status of each chunk. When a chunk is allocated the relevant bit is marked. The bit is un-marked when the object held in this chuck is reclaimed.

To avoid an excessive number of lists, the allocator does not maintain a separate free list for each possible object's size. An allocation request of a certain size is rounded up, so that the chuck returned is the first available chunk of the free list of the closest larger or equal size. If the relevant free list is empty, a block is taken from the blocks pool, the block is divided to the appropriate size, and the first chunk of this block is returned.

When an object is reclaimed, the bit corresponding to the corresponding chunk is un-

marked. An empty block (whose all chunks are free) may be returned to the blocks pool, and used later with a different size. In order to return empty blocks to the blocks pool, at the end of a collection cycle the blocks' bitmaps are scanned.

## 6.3 Prefetching for Reference Counting

We now proceed to describing the prefetch opportunities existing for a reference-counting collector (accompanied by a segregated free lists allocator), and the prefetch insertions that we have applied. Recall that the work of the reference-counting based memory manager is divided into three stages.

1. The reference-count increments stage (done during the Process-ModBuffer Procedure presented in Figure 6.4).
2. The reference-count deletions stage (done during the Process-DecBuffer-and-Release Procedure presented in Figure 6.5).
3. The objects' allocation stage.

We propose five prefetch opportunities for the three stages.

### 6.3.1 Process-ModBuffer stage

Consider the pseudo code of the Process-ModBuffer Procedure presented in Figure 6.4. In this procedure, the collector clears the dirty marks of the objects logged in *ModBuffer*, while incrementing the reference count of their descendants. For this phase we propose two prefetch opportunities. The modified Process-ModBuffer Procedure, including the prefetch instructions, is presented in Figure 6.7. An explanation follows.

The first prefetch opportunity appears during the traversal of *ModBuffer*. The scan of each object referenced by *ModBuffer* imposes a potential cache miss. Since *ModBuffer* is traversed sequentially, this cache miss can be anticipated and avoided. A prefetching of the object that should be scanned in the next iteration is inserted just before scanning the current object. One can imagine prefetching further ahead and one can easily modify the presented mechanism to achieve that, but we have obtained the most significant improvements by prefetching a single address ahead. This follows a standard prefetch strategy used in loops handling predictable array referencing patterns (e.g., [99]). The general strategy is

```
Procedure Process-ModBuffer
begin
1.    prefetch the first object whose address is in ModBuffer
2.    previous := dummyObject
3.    for each object obj whose address is in ModBuffer do
4.       prefetch the next object whose address is in ModBuffer
5.       obj.dirty := false
6.       // INCREMENT CURRENT REFERENT OF THE OBJECT obj
7.       for each pointer ptr of obj do
8.          prefetch the rc field of the object referenced by ptr
9.          increment rc of object referenced by previous
10.         previous := ptr
11.   increment rc of object referenced by previous
end
```

Figure 6.7: Reference counting- Process-ModBuffer with prefetch

to place fetch instructions inside the loop body so that data for the future loop iteration(s) is prefetched during the current iteration. Lines 1 and 4 in Figure 6.7 execute the proposed prefetch. We will later refer to this strategy as the *ModBuffer-traversal* strategy.

The second prefetch opportunity exploited during this stage appears during the reference-count increments. Each such increment incurs a cache miss if corresponding reference-count field is not present in the cache. To handle this potential cache miss, we slightly delay the increment of an object's reference count. When an increment to a count is required, the count of the object is prefetched. In the implementation we use, the reference count of each object is located in the object header; however, the same technique applies also when the count is stored in an auxiliary table. The location of the count is stored in a temporary variable named *previous*. Next, the procedure handles the next object before returning to the (hopefully cached) count and incrementing it. To avoid a special treatment to the first iteration and the implied 'if' statement, we use a dummy object whose reference count is incremented when the first count is prefetched. This delaying strategy is presented in lines 2 and 8-11 of Figure 6.7. We will later refer to this strategy as the *delay-increment* strategy.

```
Procedure Process-DecBuffer-and-Release
begin
1.   prefetch the rc field of the first object whose address is in DecBuffer
2.   for each object obj whose address is in DecBuffer do
3.     prefetch the rc field of the next object whose address is in DecBuffer
4.     obj.rc--
5.     if obj.rc = 0 ∧ obj ∉ Roots then
6.       prefetch the word containing the mark-bit relevant to obj
7.       for each pointer ptr of obj do
8.         push ptr onto DecBuffer
9.       return obj to the general purpose allocator.
10.          //UNMARK THE MARK-BIT RELEVANT TO obj
end
```

Figure 6.8: Reference counting- Process-DecBuffer-and-Release with prefetch

### 6.3.2 Process-DecBuffer-and-Release stage

Figure 6.5 describes the Process-DecBuffer-and-Release Procedure, in which the collector decrements the reference counts of the objects logged in *DecBuffer*, and recursively reclaims the dead objects. This phase also includes two prefetch opportunities. The modified Process-DecBuffer-and-Release Procedure, which includes these prefetch modifications, is presented in Figure 6.8. A description follows.

Similarly to the Process-ModBuffer stage, the first prefetch opportunity for the Process-DecBuffer-and-Release stage occurs with the traversal of *DecBuffer*. The reference-count decrement of an object referenced by *DecBuffer* can lead to a cache miss if the reference-count field is not present in the cache. We exploit the loop prefetch strategy described in the previous stage and prefetch the reference count field of the next object in the buffer before handling the current one. Lines 1 and 3 of Figure 6.8 present this prefetch strategy. We will later refer to this strategy as the *DecBuffer-traversal* strategy.

The second prefetch opportunity of this stage occurs during the reclamation of an object. Once an unreachable object is discovered (line 3 of Figure 6.5), the object is first scanned and all its descendants are recorded in the *DecBuffer*; only then the object is reclaimed. As described in Section 6.2.2, the reclamation of an object sums up to un-marking the mark-bit

```
Procedure Build-Block-Free-List
begin
1.   markWordAddress := address of the first word in the block's bitmap
2.   markWordEnd := address of the last word in the block's bitmap
3.   previousFree := cursor address
4.   while markWordAddress ≤ markWordEnd
5.     markWord := word referenced by markWordAddress
6.     foreach bit in markWord
7.       if bit is not set then
8.         objectRef := address of chunk relevant to bit
9.         write objectRef into previousFree
10.        previousFree := objectRef
11.    markWordAddress += size of word
12.  write null into previousFree
end
```

Figure 6.9: Reference counting- Build-Block-Free-List

corresponding to this object. As the word containing this mark-bit may not be present in cache, we get another potential cache miss. This potential miss is handled by prefetching the relevant mark-bit word as soon as we realize that the object should be reclaimed. Namely, the prefetch is performed right after line 3 of Figure 6.5. This way, the miss penalty for unsetting the relevant bit later is reduced or even eliminated. Line 6 in Figure 6.8 present this prefetch modification. We will later refer to this strategy as the *object-release* strategy.

### 6.3.3 Build-Block-Free-List stage

The fifth prefetch opportunity occurs with the segregated free lists allocator. Each free list employs a cursor pointing to the next available chunk for allocation in the corresponding size. After allocating using the chunk referenced by the cursor, the cursor is advanced to the next available chunk. To save scanning the bitmap during each allocation, a linked list of free chunks is created for each block after each collection. In fact, this list is created lazily. When a block is exhausted and a new block is selected, all its free chunks are linked for future allocations. The pseudo-code of the Procedure Build-Block-Free-List which builds the

```
Procedure Build-Block-Free-List
begin
1.   markWordAddress := address of the first word in the block's bitmap
2.   markWordEnd := address of the last word in the block's bitmap
3.   previousFree := cursor address
4.   while markWordAddress ≤ markWordEnd
5.     markWord := word referenced by markWordAddress
6.     markWordAddress += size of word
7.     prefetch markWordAddress
8.     foreach bit in markWord
9.       if bit is not set then
10.        objectRef := address of chunk relevant to bit
11.        write objectRef into previousFree
12.        previousFree := objectRef
13.  write null into previousFree
end
```

Figure 6.10: Reference counting- Build-Block-Free-List with prefetch

described free list for a given block is presented in Figure 6.9.

Interestingly, it turned out that it is useful to prefetch the next mark-bit word while processing the current one. Initially, we expected the hardware to do this prefetching automatically, since the address space of the block's bitmap is traversed sequentially; but our experiments reveal that this is not the case. Maybe the reason is that the bitmap traversal is interrupted by writing data into the free chunks (line 9 of Figure 6.9). To sum up, the Build-Block-Free-List Procedure was modified to exploit this loop prefetching strategy as presented in lines 6-7 of Figure 6.10.

## 6.4 An Implementation for Java

We have implemented the proposed collector in the Jikes RVM (research virtual machine) [1]. The entire system, including the collector itself is written in Java (extended with unsafe primitives available only to the Java Virtual Machine implementation to access raw memory).

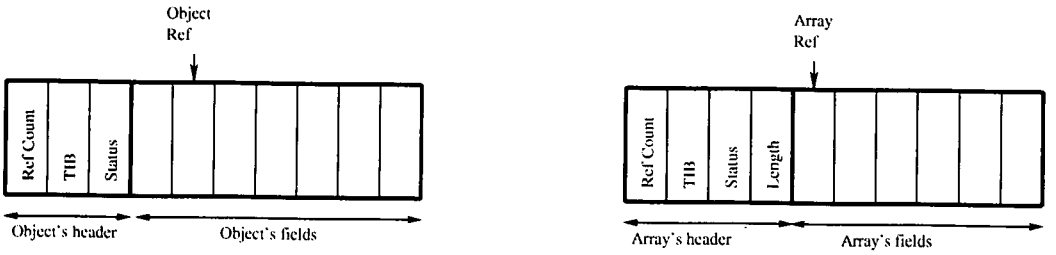



Figure 6.11: Jikes object model for reference counting

### 6.4.1 Object layout

The object layout of our memory manager within the Jikes RVM version we used is displayed in Figure 6.11. Jikes RVM' basic object model uses a two word header. One word holds a TIB *Type Information Block* pointer for the object's class (which serves as Jikes' virtual method table). The other word ("status word") contains a thin lock, and a few unallocated bits that can be used for other purposes. An array object's header includes an additional length field. In order to support reference-counting collection, Jikes employs an additional third word in the object's header to keep the reference count of an object (and other related reference-counting information). Note that for both arrays and scalar objects all elements of the header are available at the same offset from the reference to the object/array.

### 6.4.2 Object prefetching

As presented in section 6.3, an object is prefetched either in order to manipulate its reference-count field or in order to scan it (i.e., determine its descendants). According to the object layout presented in Figure 6.11, when prefetching the reference count of an object we prefetch 16 bytes a head of the object reference, and when prefetching the object's fields we prefetch 12 bytes a head of the object reference (as the scan should look at the object's TIB).

Note that when prefetching an object for scanning purposes, we have only prefetched one word of the object (the one starting from the TIB). A large object (or a large references array) may obviously contain many reference slots which are not included in this word, and should arrive to the cache during the object scan. Moreover, the reference slots may not even be presented in the beginning of the object (as the first fields of the object may include non-reference fields). However, probing each object's TIB reference, in order to check which exact object's parts should be prefetched, is costly: it involves more overhead and it could only be done after the object header is available in the cache. In addition, if references are present in adjacent object's words (such as in a large references array), hardware prefetch is

the best prefetching solution anyway. Due to these reasons and since most objects are small, we assume that most of the references are contained in the objects' beginning, and hence we chose only prefetching the object's first word. An alternative approach, to prefetch the first two words before scan, was less successful.

## 6.5 Measurements

**Platform and benchmarks.** We have run our measurements on a dual Intel's Xeon 1.8GHz processors workstation. This processors have a 16KB sized L1 cache and a 512KB sized L2 cache. We have used the SPECjvm98 benchmark suite, the SPECjbb2000 benchmark[1] (both described in SPEC's web site [90]), and the DaCapo benchmarks [27].

**The collector.** We have inserted the prefetch instructions suggested in Section 6.3 into the reference-counting collector of Jikes [1]. Next, we have compared the original reference-counting collector of Jikes, against the collector modified to include these prefetch instructions. The reference-counting collector is accompanied by a dedicated cycle collection algorithm. Since we are interested solely in the effects of prefetch over the reference-counting collector, and since the cycle collection algorithm has a characteristic behavior that resembles tracing collectors, we have disabled this cycle collection algorithm. For most applications this means a negligible increase in the heap size. We stress that both collectors (i.e., the one that employs prefetching and the one that doesn't) run without cycle collection.

**Testing procedure.** We have performed standard automated runs of the detailed benchmarks. Our standard automated runs run each benchmark ten times for both the original reference-counting collector and the modified reference-counting collector. We report the average of these runs. To guaranty a fair comparison of the garbage collection characteristics, we included only runs in which each benchmark performs the same amount of garbage collections on both collectors.

### 6.5.1 Prefetch improvements

Table 6.1 presents the improvements achieved by using prefetching. A negative percentage represents a performance improvement, while a positive percentage represents a deteriora-

[1] We have slightly modified SPECjbb2000, to run a fixed number of transactions instead of running during a fixed time period.

| Benchmarks | overhead reduction | | | | overall benchmark improvement |
|---|---|---|---|---|---|
| | Process ModBuffer | Process DecBuffer and Release | Sweep Blocks | overall gc | |
| jess | -0.1% | -0.9% | -11.9% | -1.8% | -0.9% (45.5%) |
| db | -11.2% | -6.7% | -8.0% | -8.5% | -0.9% (10.0%) |
| javac | -12.2% | -8.4% | -18.4% | -12.3% | -3.4% (27.3%) |
| mtrt | -12.3% | -3.3% | -12.3% | -8.0% | -1.5% (19.2%) |
| jack | -16.3% | -5.9% | -23.2% | -10.8% | -3.1% (28.3%) |
| jbb | -8.3% | -6.5% | -26.5% | -14.9% | -4.6% (31.0%) |
| fop | -12.5% | -8.1% | -11.3% | -10.5% | -2.1% (19.7%) |
| antlr | -16.1% | -8.4% | -24.3% | -14.6% | -1.3% ( 8.4%) |
| pmd | -8.7% | -8.4% | -12.3% | -9.6% | -3.3% (34.7%) |
| ps | 3.0% | -3.7% | -13.0% | -1.7% | -0.6% (37.5%) |
| hsqldb | -18.8% | -11.0% | -17.6% | -14.9% | -4.6% (30.4%) |
| jython | -9.4% | -0.5% | -6.6% | -4.4% | -1.7% (38.1%) |
| xalan | 2.4% | -1.2% | -24.4% | -0.6% | -0.6% (91.6%) |
| **average** | **-9.3%** | **-5.6%** | **-16.1%** | **-8.7%** | **-2.2%** |

Table 6.1: Reduction in reference-counting overheads obtained by prefetching

tion in performance. Columns 2-4 present the improvements achieved for each one of the reference-counting collector stages implemented by the Process-ModBuffer Procedure (presented in Figure 6.4), by the Process-DecBuffer-and-Release Procedure (presented in Figure 6.5), and by the Build-Block-Free-List Procedure (presented in Figure 6.9). These presented improvements are calculated relatively to the corresponding reference-count stages. Hence, for example, a -10.0% appearing on the second column indicates a 10.0% performance improvement of the **Process-ModBuffer** stage. The fifth column presents the overall reference-counting's performance improvement achieved. The sixth column introduces the overall throughput improvement achieved for the benchmark (i.e., reduction in the overall benchmark runtime). As this improvement depends on the garbage collection fraction out of the entire benchmark, we have also included this fraction in parenthesis. Note that this fraction contains, in addition to the collection overhead, also allocation overhead introduced by the Build-Block-Free-List Procedure.

| Benchmarks | Process ModBuffer fraction | Process DecBuffer and Release fraction | Sweep Blocks fraction |
|---|---|---|---|
| jess | 36.6% | 51.5% | 11.0% |
| db | 39.2% | 50.5% | 8.9% |
| javac | 31.6% | 38.6% | 28.4% |
| mtrt | 26.8% | 46.4% | 25.2% |
| jack | 31.3% | 54.5% | 11.6% |
| jbb | 24.9% | 34.0% | 40.1% |
| fop | 38.7% | 39.5% | 20.8% |
| antlr | 30.6% | 42.4% | 25.6% |
| pmd | 30.8% | 43.8% | 25.3% |
| ps | 38.2% | 52.5% | 7.3% |
| hsqldb | 30.1% | 41.0% | 26.9% |
| jython | 32.4% | 50.8% | 15.7% |
| xalan | 43.6% | 51.8% | 4.4% |

Table 6.2: Reference-counting profiling

Normally, Jikes runs the Build-Block-Free-List Procedure lazily when a new block is selected for allocations. Therefore, while the Process-DecBuffer-and-Release and the Process-DecBuffer-and-Release Procedures are activated once per a garbage collection cycle, the Build-Block-Free-List Procedure is activated numerous times during the benchmark run (i.e., between the collections). In order to accurately measure the time overhead of this procedure, we have slightly modified Jikes reference-counting collector to activate the Build-Block-Free-List Procedure continuously (non-lazily), for all non-empty blocks, once at the end of each collection.

To make the picture complete, Table 6.2 presents the distribution of the reference-counting collector overhead within the three different stages. These three stages together impose almost a 100% of the reference-counting collector overhead (as stages such as scanning threads' stack are not included). The presented distribution combined with the prefetch improvement achieved in each stage (appearing in columns 2-4 in Table 6.1), creates the overall garbage collection improvement introduced in column five of Table 6.1.

One can see that the employed prefetch strategies reduce the overall overhead of reference-counting for all benchmarks. This is emphasized by the last line of Table 6.1, which presents the average improvement of each column. For most benchmarks, prefetching is able to reduce the overhead imposed by each one of the three stages. Note, however, that the improvements are not steady among the different benchmarks and among the different stages. We study this issue in Section 6.5.2 below.

### 6.5.2 Reference-counting objects' access behavior

In this section we study the memory access patterns of a reference-counting collector in order to understand the potential of prefetch instruction insertions. Recall that tracing collectors traverse the application's live objects in an arbitrary order (depending on the objects graph). If a mark table is used by a tracing collector, each live object is likely to be read exactly once during a collection, since if it was already traversed, its corresponding mark bit in the mark table would indicate that it should not be traversed again.

To analyze the way reference counting accesses objects, we ran the following measurement. We consider each object scan and each reference-count update as a single memory access[2], as each such operation may cause a cache miss. We started by recording the address of each such access into gc-log files, one log file per collection. Next, each gc-log file was analyzed in the following manner. For a given window size $w$, we have checked for each access, if the same address was accessed during the last $w$ (distinct) accesses[3]. For each benchmark, we outputted the fraction of hits, in which an access has repeated itself within the window size, as a function of the window size.

The results appear in Table 6.3. We ran the above measurements with five window sizes: 100, 1000, 10000, 100000, and 1000000. The different window sizes demonstrate access behavior for various cache sizes. The smaller windows are more representative of L1 cache-miss behavior, whereas the larger window sizes represent behavior with L2 cache sizes. In addition, we have also separated the measurements into the **Process-ModBuffer** phase and the **Process-DecBuffer-and-Release** phase to see if the patterns are different for the various stages. These measurements appear in Tables 6.4 and 6.5.

The measurement should be read as follows. If a 40% percentage appears under the 100 column of the slot referring to a certain benchmark, it means that 40% of the accesses

[2]In this approximate measure, we count an object's scan as a single access, although it may be involve multiple accesses, e.g., because and object may be large and may contain several slots.

[3]We always consider a first access of an object in a collection as a miss.

| Benchmarks | window size | | | | |
|---|---|---|---|---|---|
| | 100 | 1000 | 10000 | 100000 | 1000000 |
| jess | 58.2% | 64.1% | 66.9% | 68.0% | 72.7% |
| db | 15.1% | 15.8% | 16.8% | 68.3% | 80.2% |
| javac | 31.3% | 37.5% | 40.5% | 42.8% | 50.9% |
| mtrt | 13.6% | 17.0% | 18.5% | 19.8% | 22.1% |
| jack | 25.9% | 27.9% | 29.0% | 30.5% | 36.3% |
| jbb | 23.5% | 30.5% | 34.7% | 38.2% | 46.2% |
| fop | 28.5% | 33.4% | 35.1% | 37.0% | 42.0% |
| antlr | 23.2% | 26.2% | 28.1% | 29.1% | 33.3% |
| pmd | 28.9% | 32.0% | 35.4% | 39.9% | 47.3% |
| ps | 78.2% | 79.5% | 79.8% | 80.2% | 90.7% |
| hsqldb | 26.0% | 27.9% | 29.5% | 31.7% | 40.0% |
| jython | 54.5% | 55.4% | 56.1% | 56.5% | 57.1% |
| xalan | 0.4% | 0.6% | 2.8% | 99.0% | 99.5% |
| average | **31.3%** | **34.4%** | **36.4%** | **49.3%** | **55.3%** |

Table 6.3: Percentage of repeated object accesses (hit ratios) for the entire collection

were to memory locations that have been previously accessed during the last 100 accessed (distinct) addresses. A higher percentage is highly correlated to low cache-miss ratio and to a reduced potential for effective prefetching.

It turns out that unlike tracing collectors, the repeated access with reference counting is quite high and the repeated accesses have temporal proximity. Hence, many memory accesses hit the L1 cache, making the prefetch a burden, or hit the L2 cache, making the prefetch less effective. Another interesting property is that the probability of a hit is (sometimes much) higher during the **Process-ModBuffer** stage than during the **Process-DecBuffer-and-Release** stage. To better understand this phenomenon, we have also measured for each of these stages the fraction of hits, in which an access has repeated itself within an infinite window size (i.e., the fraction of accesses which deal with an object which was already accessed in this stage). This measurement, introduced in the seventh column of Tables 6.4 and 6.5, exhibits two reasons for the hit ratio difference of the two stages.

| Benchmarks | window size | | | | | |
|---|---|---|---|---|---|---|
| | 100 | 1000 | 10000 | 100000 | 1000000 | $\infty$ |
| jess | 62.4% | 69.4% | 73.3% | 74.0% | 74.5% | 74.6% |
| db | 17.7% | 18.5% | 19.7% | 70.1% | 70.6% | 70.6% |
| javac | 42.6% | 50.0% | 53.7% | 55.6% | 58.8% | 60.0% |
| mtrt | 19.5% | 24.7% | 26.4% | 27.6% | 27.9% | 27.9% |
| jack | 37.2% | 39.9% | 41.2% | 42.3% | 42.7% | 42.7% |
| jbb | 33.6% | 42.0% | 48.2% | 51.1% | 55.1% | 55.4% |
| fop | 38.5% | 45.2% | 47.5% | 49.7% | 51.8% | 51.9% |
| antlr | 36.0% | 39.8% | 42.5% | 43.8% | 44.1% | 44.1% |
| pmd | 42.7% | 46.8% | 51.3% | 56.7% | 58.4% | 58.4% |
| ps | 82.1% | 83.8% | 84.2% | 84.4% | 84.4% | 84.4% |
| hsqldb | 41.5% | 43.9% | 45.8% | 47.6% | 48.2% | 48.2% |
| jython | 55.5% | 56.9% | 57.8% | 58.3% | 58.4% | 58.4% |
| xalan | 0.7% | 0.9% | 3.1% | 99.1% | 99.1% | 99.1% |
| average | **39.3%** | **43.2%** | **45.8%** | **58.5%** | **59.5 %** | **59.7%** |

Table 6.4: Percentage of repeated object accesses (hit ratios) for the Procedure Process-ModBuffer

First, the infinity hit ratio of both stages demonstrates that there is an inherent better hit ratio for the **Process-ModBuffer** stage. This means that the per-object number of accesses during the **Process-ModBuffer** stage is higher than the per-object number of accesses during the **Process-DecBuffer-and-Release** stage. The reason is that during the **Process-DecBuffer-and-Release** stage, an object is accessed only when its reference count is decremented. We do not count the possible following object's scan (if it should be reclaimed) as an access, as this access is tied to its previous access. However, during the **Process-ModBuffer** stage an object is considered accessed when it is scanned (if it was logged in the *ModBuffer*) and when its reference-count field is incremented. Thus, consider for example a newly created object which is referenced by a single object. This object would be accessed twice during the **Process-ModBuffer** stage: when it is reached during the *ModBuffer* traversal and when its reference count is incremented. However, if its reference count is later decremented during the **Process-DecBuffer-and-Release** stage, it would

| Benchmarks | window size | | | | | |
|---|---|---|---|---|---|---|
| | 100 | 1000 | 10000 | 100000 | 1000000 | $\infty$ |
| jess | 53.8% | 58.9% | 60.3% | 61.2% | 66.6% | 67.8% |
| db | 12.4% | 13.0% | 13.9% | 66.1% | 67.5% | 67.5% |
| javac | 17.2% | 21.9% | 24.1% | 26.4% | 40.4% | 50.2% |
| mtrt | 7.0% | 8.2% | 9.6% | 10.5% | 15.0% | 19.0% |
| jack | 13.9% | 15.1% | 16.1% | 17.7% | 29.1% | 39.0% |
| jbb | 12.5% | 18.0% | 20.1% | 24.3% | 36.6% | 51.8% |
| fop | 15.4% | 18.0% | 18.8% | 20.4% | 29.0% | 38.6% |
| antlr | 7.6% | 9.7% | 10.5% | 11.3% | 20.2% | 32.0% |
| pmd | 12.7% | 14.6% | 16.6% | 20.1% | 34.2% | 51.1% |
| ps | 74.2% | 75.2% | 75.3% | 75.9% | 83.7% | 84.0% |
| hsqldb | 7.5% | 8.7% | 10.1% | 12.6% | 25.9% | 37.7% |
| jython | 53.4% | 54.0% | 54.3% | 54.7% | 55.7% | 58.8% |
| xalan | 0.2% | 0.3% | 2.6% | 98.8% | 99.1% | 99.1% |
| average | **22.1%** | **24.3%** | **25.6%** | **38.5%** | **46.4%** | **53.4%** |

Table 6.5: Percentage of repeated object accesses (hit ratios) for the Procedure Process-DecBuffer-and-Release

only be accessed once.

The second reason for the higher hit ratio for the **Process-ModBuffer** stage is highlighted by the difference between the 1000000 column and the infinity column of Tables 6.4 and 6.5. It could be seen that for the **Process-ModBuffer** stage there is hardly any difference between these two columns, while there is some gap (for some of the benchmarks) for the **Process-DecBuffer-and-Release** stage. The reason is that adjacent objects logged in the *ModBuffer* were, very probably, created or modified also adjacently in time, sometimes by the same method. Thus, such objects tend to reference similar objects or even each other. For example, newly created objects are often being referenced by other objects, immediately after being created. Hence, the newly created object's log would be adjacent to the log of the object which was modified to reference it (and hence was also logged). Therefore, program dictates high locality, which is seen during the **Process-ModBuffer** stage which orderly

process the *ModBuffer*. An interesting future work could examine several alternatives of processing the *DecBuffer* during the **Process-DecBuffer-and-Release** stage, to see which one yields the best locality.

### 6.5.3 Prefetch strategy profiling

Table 6.1 introduced in Section 6.5.1 presents the prefetch improvements achieved for the different reference-counting stages. However, two different prefetch strategies were implemented in both the **Process-ModBuffer** stage and the **Process-DecBuffer-and-Release** stage. Tables 6.6-6.7 break the overall improvement into the shares of each particular strategy.

Table 6.6 relates to the **Process-ModBuffer** stage, displaying the effect of the *ModBuffer-traversal* strategy and the effect *delay-increment* strategy presented in Section 6.3.1. As can be seen, the *ModBuffer-traversal* strategy is the major cause for the prefetch improvement of the **Process-ModBuffer** stage.

To understand why, we have further analyzed the objects' access behavior of the **Process-ModBuffer** stage. This stage includes two access types:

- The scan of objects logged in the *ModBuffer*. These objects were either created or modified since the last garbage collection. These objects are prefetched by the *ModBuffer-traversal* strategy.
- The reference-count increment of objects referenced by the above objects. These objects are prefetched by the *delay-increment* strategy.

Table 6.8 presents the fraction of accessed objects of each type. As can be seen, if there is any advantage to one of these stages, it is to the one using the *delay-increment* strategy, failing to explain why it is less beneficial. However, we have also compared the repeated access patterns for various sized windows as above. These results are presented in Tables 6.9 and 6.10. It can be clearly seen that the percentage of hits for the traversal of the *ModBuffer* was substantially lower. This is what makes it more susceptible to prefetch insertion improvements.

The pattern of accesses to objects whose reference-count is incremented shows that most of the increments are done after the object was lately accessed (and hence are probably available either in the L1 cache or in the L2 cache). On the other hand, most of the objects logged in *ModBuffer* are objects created since the last collection (i.e., not old objects which were logged because they were modified since the last collection). According to Table 6.9,

| Benchmarks | ModBuffer traversal share | delay increment share | Process ModBuffer improvement |
|---|---|---|---|
| jess | -5.5% | 5.3% | -0.1% |
| db | -7.4% | -3.8% | -11.2% |
| javac | -9.2% | -3.0% | -12.2% |
| mtrt | -9.5% | -2.9% | -12.3% |
| jack | -13.8% | -2.5% | -16.3% |
| jbb | -7.9% | -0.3% | -8.2% |
| fop | -10.7% | -1.8% | -12.5% |
| antlr | -13.2% | -2.8% | -16.1% |
| pmd | -6.6% | -2.1% | -8.7% |
| ps | -4.0% | 7.1% | 3.0% |
| hsqldb | -12.5% | -6.3% | -18.8% |
| jython | -10.8% | 1.4% | -9.4% |
| xalan | 0.2% | 2.1% | 2.4% |
| **average** | **-8.5%** | **-0.7%** | |

Table 6.6: A break of the prefetching improvement due to the two strategies involved in the Process-ModBuffer stage

many of these newly created objects are encountered for the first time during the collection when they are encountered during the traversal of the *ModBuffer*. Therefore, the *ModBuffer-traversal* strategy has more potential of hiding (or reducing) cache miss' stalls, which explains the different contribution of each of the two strategies.

Another possible reason for the *delay-increment* strategy being less beneficial is that it does not only add the prefetch instruction overhead. It also stores the object whose reference-count should be incremented into a temporary variable for each reference-count increment (line 10 of Figure 6.7), thus increasing the register pressure, causing an additional overhead.

Table 6.7 relates to the **Process-DecBuffer-and-Release** stage, displaying the effect of the *DecBuffer-traversal* strategy and the effect of the *object-release* strategy (presented in Section 6.3.2). Here, the strategy responsible for most of the benefit is the *DecBuffer-*

| Benchmarks | DecBuffer traversal share | object release share | Process DecBuffer and Release improvement |
|---|---|---|---|
| jess | -0.9% | -0.1% | -0.9% |
| db | -6.4% | -0.2% | -6.7% |
| javac | -6.8% | -1.6% | -8.4% |
| mtrt | -2.0% | -1.4% | -3.3% |
| jack | -4.3% | -1.7% | -5.9% |
| jbb | -5.6% | -0.9% | -6.5% |
| fop | -6.7% | -1.3% | -8.1% |
| antlr | -6.9% | -1.4% | -8.4% |
| pmd | -6.6% | -1.8% | -8.4% |
| ps | -3.4% | -0.1% | -3.7% |
| hsqldb | -10.0% | -1.0% | -11.0% |
| jython | -0.1% | -0.4% | -0.5% |
| xalan | -0.8% | -0.4% | -1.2% |
| **average** | **-4.7%** | **-0.9%** | |

Table 6.7: A break of the prefetching improvement due to the two strategies involved in the Process-DecBuffer-and-Release stage

*traversal* strategy. We have not further analyzed the objects' access behavior of the **Process-DecBuffer-and-Release** stage, since as opposed to the **Process-ModBuffer** stage, it does not access two types of objects (it accesses objects and bitmap's words).

### 6.5.4 Hardware counters measurements

To understand better the effect of the inserted prefetch instructions, we have measured several relevant hardware counters using PAPI (the Performance API [77]). These counters were measured during the garbage collection work of both versions of the reference-counting collector: with and without prefetching. Table 6.11 presents the difference of these counters between the versions for the entire garbage collection work. Columns 2-4 present the differ-

| Benchmarks | scanned object | rc-update object |
|---|---|---|
| jess | 25.7% | 74.3% |
| db | 28.4% | 71.6% |
| javac | 39.8% | 60.2% |
| mtrt | 71.8% | 28.2% |
| jack | 57.1% | 42.9% |
| jbb | 43.1% | 58.9% |
| fop | 48.1% | 51.9%% |
| antlr | 55.8% | 44.2% |
| pmd | 41.0% | 59.0% |
| ps | 15.6% | 84.4% |
| hsqldb | 51.6% | 48.4% |
| jython | 41.5% | 58.5% |
| xalan | 0.8% | 99.8% |
| **average** | **40.0%** | **60.0%** |

Table 6.8: Profile of the objects accessed during the Process-ModBuffer stage

ence in the number of cycles stalled on any resource, the L2 load misses difference and the data translation look aside buffer (TLB) misses difference. Column 5 presents the overall garbage collection improvement (which has been presented in Table 6.1 and is repeated here for easier evaluation).

It can be seen that the number of L2 cache misses does not vary much, although it does decrease on average. However, there is usually a noticeable decrease in the number of TLB misses and in the number of cycle stalls. Our belief is that the prefetch was issued a bit too late to completely eliminate the L2 cache miss. But since the data has started moving towards the cache, we see a decrease of the stalls and the TLB misses. The option of running the prefetch instructions earlier has a cost in temporary variables and was not beneficial in practice.

| Benchmarks | window size | | | | |
|---|---|---|---|---|---|
| | 100 | 1000 | 10000 | 100000 | 1000000 |
| jess | 2.3% | 2.5% | 2.7% | 2.9% | 4.8% |
| db | 2.9% | 3.2% | 3.6% | 4.1% | 4.4% |
| javac | 24.3% | 26.2% | 26.9% | 27.5% | 29.3% |
| mtrt | 3.9% | 4.2% | 4.4% | 4.6% | 4.6% |
| jack | 10.7% | 11.1% | 11.3% | 11.6% | 11.8% |
| jbb | 22.7% | 29.5% | 29.9% | 30.3% | 30.4% |
| fop | 23.3% | 24.1% | 24.6% | 25.0% | 25.7% |
| antlr | 14.4% | 15.0% | 15.5% | 15.8% | 15.8% |
| pmd | 19.0% | 19.7% | 21.0% | 23.5% | 24.3% |
| ps | 5.4% | 5.6% | 5.8% | 6.0% | 6.0% |
| hsqldb | 25.8% | 26.4% | 26.7% | 27.1% | 27.2% |
| jython | 3.1% | 3.3% | 3.5% | 3.6% | 3.6% |
| xalan | 28.0% | 29.4% | 30.9% | 33.5% | 34.9% |
| average | **14.3%** | **15.4%** | **15.9%** | **16.6%** | **17.1%** |

Table 6.9: Already accessed objects' percentages for the modified objects logged in Mod-Buffer

## 6.6 Related work

VanderWiel and Lilja [99] provide a detailed survey examining diverse prefetching strategies, such as hardware prefetching, array prefetching and other software prefetching.

Similarly to our approach, several previous studies proposed adding, by hand, prefetch instructions to specific locations in garbage collectors algorithm. However, they all studied tracing collectors. Boehm [16] proposed prefetching nodes that are pushed to the mark stack during the mark phase of a mark-sweep collector, in order to make it later available when popped from stack to be scanned. This prefetching strategy yields improvements in execution time, although suffering from prefetch timing problems: too early prefetches and too late prefetches. These timing problems were addresses by [22, 98]. Both suggested improved prefetching strategies to the mark phase by imposing some sort of FIFO processing over the mark stack, in order to control the time between the data prefetch and its actual access. In

| Benchmarks | window size | | | | |
|---|---|---|---|---|---|
| | 100 | 1000 | 10000 | 100000 | 1000000 |
| jess | 83.2% | 92.5% | 97.8% | 98.6% | 98.7% |
| db | 23.5% | 24.6% | 26.0% | 96.2% | 96.8% |
| javac | 54.7% | 65.8% | 71.4% | 74.1% | 78.4% |
| mtrt | 59.5% | 77.2% | 82.5% | 86.4% | 87.3% |
| jack | 72.5% | 78.2% | 81.1% | 83.2% | 84.0% |
| jbb | 42.0% | 51.5% | 62.2% | 66.9% | 73.8% |
| fop | 52.5% | 64.7% | 68.8% | 72.5% | 76.0% |
| antlr | 63.4% | 71.2% | 76.6% | 79.1% | 79.8% |
| pmd | 59.2% | 65.6% | 72.4% | 79.8% | 82.2% |
| ps | 96.3% | 98.3% | 98.7% | 98.9% | 98.9% |
| hsqldb | 58.2% | 62.6% | 66.0% | 69.5% | 70.5% |
| jython | 92.8% | 94.9% | 96.4% | 97.1% | 97.2% |
| xalan | 0.5% | 0.7% | 2.9% | 99.6% | 99.6% |
| average | **58.3%** | **65.2%** | **69.4%** | **84.8%** | **86.4%** |

Table 6.10: Already accessed objects' percentages for the objects whose reference count was incremented during the Procedure Process-ModBuffer

another related work, Cahoon [18] employs prefetching to improve the memory performance of a generational copying garbage collector.

Another prefetch strategy to improve the memory management sub-system locality was suggested by Appel [3]. Appel emulates a write-allocate policy on a no-write-allocate machine by prefetching garbage before it is written (during its space allocation). Hence, this relevant cache line will be allocated and the write (occurring during the object allocation) will hit the cache.

Our approach tries to hide the latency caused by data cache misses by prefetching data ahead of reference. Several other studies try to improve the locality of programs by moving objects wisely in the heap. Chilimbi and Larus [23] place objects with high temporal affinity next to each other, so they are likely to reside in the same cache block. Calder et .al. [19] employ data placement algorithms to find a placement that decreases inter-objects conflict.

| Benchmarks | Cycles stalled | L2 cache misses | TLB misses | overall gc |
|---|---|---|---|---|
| jess | -8.8% | -1.1% | -17.3% | -1.8% |
| db | -4.3% | -0.7% | 10.1% | -8.5% |
| javac | -17.4% | -1.8% | -6.6% | -12.3% |
| mtrt | -15.8% | -0.6% | -20.1% | -8.0% |
| jack | -20.0% | -3.2% | -21.1% | -10.8% |
| jbb | -14.8% | 2.5% | -9.4% | -14.9% |
| fop | -6.6% | 0.1% | -14.0% | -10.5% |
| antlr | -6.3% | 0.1% | -15.1% | -14.6% |
| pmd | -9.6% | 0.2% | -11.2% | -9.6% |
| ps | -2.0% | 0% | -21.6% | -1.7% |
| hsqldb | -14.1% | -0.8% | -21.8% | -14.9% |
| jython | -4.8% | 0.4% | -1.6% | -4.4% |
| xalan | -0.7% | 0.2% | 37.6% | -0.6% |
| **average** | **-9.6%** | **-0.4%** | **-8.6%** | |

Table 6.11: Hardware counters measurements

## 6.7 Conclusions

We have studied prefetch opportunities for a modern reference-counting garbage collector. It turns out that several such opportunities typically exist for reference counting. We have implemented such prefetch insertions on the Jikes Research JVM and it turns out that prefetching is effective in reducing stall times and improving garbage collection efficiency. In particular, the average garbage collection times were reduced by 8.7%.

We have also measured the memory access patterns of the collector and found out that, unlike tracing collectors, objects are accessed repeatedly, reducing the potential benefit due to prefetching. These measurements were able to explain the effectiveness of the various strategies at the various stages. Wherever objects are repeatedly accessed and hits are expected, prefetch insertions are less effective.

# Chapter 7

# Patterns for the Efficient Use of Managed Memory

## 7.1 Introduction

An especially intriguing issue in Programming Languages is the tension between the desire to use a high level object oriented language such as Java or C#, which provides software engineering benefits allowing fast construction of reliable and maintainable software, and the popular choice of a lower level language that may yield more efficient programs. We project this issue onto the memory management subsystem and try to deal with the issue of using garbage collection versus a possibly more efficient manual allocation and de-allocation technique.

Automatic memory management is an important feature of Java and C#, relieving programmers of the worry of a timely de-allocation of allocated memory objects. It is well acknowledged for solving two most notorious bugs: memory leaks and dangling pointers. In addition, the fact that developers can "forget about memory" imparts a more relaxed development environment, which may yield faster development of better code. Although garbage collection is a powerful software engineering construct and programmers are encouraged to use it, it incurs an overhead on the runtime system and it may significantly slow down the execution of a program. Consequently, researchers have spent a vast amount of time on trying to improve garbage collector performance [55].

However, and in spite of modern garbage collector efficiency, a naive use of the memory

manager abstraction in a user code may impose a high cost on performance. One extremely inefficient use that often occurs in typical benchmarks is a user method that is invoked a huge number of times during the run of the program and allocates an object each time it is called, referencing it with another object's instance variables. If each such allocation renders the previously allocated object unreachable, then to ease the load, it is enough to allocate this object once and use it repeatedly. Such a solution reduces the number (and overall overhead) of allocations and it reduces the number (and overall overhead) of garbage collections. It also improves program locality, as the same memory location is used repeatedly throughout the run.

In many programs, excessive allocation is executed in a small number of specific allocation sites (and for a small number of classes). In this work we[1] first propose a profiler that can identify allocation sites that allocate excessively. Second, we propose two patterns to modify these allocations when the modification can be easily carried out in a local manner: *compound object pooling* and *SOfMA* (single object for multiple allocations). We study these patterns in comparison to the standard *object pooling* pattern (see for example [95]). These patterns, described in Section 7.2, drastically reduce the number of objects that are allocated; instead, objects are reused, with three benefits. First, the time spent on allocation significantly decreases. Second, the number of garbage collections is reduced, implying an overall reduction in garbage collection overhead. Finally, a space reuse for a frequently touched object improves program locality.

This work provides patterns that deal with the tension between the abstraction and the efficiency of the memory management subsystem. We stress that we do not recommend employing wide-scale explicit memory management. We strongly believe that automatic memory management is important and should be used throughout the development process. Automatic memory management cannot be beaten by our patterns; rather, they should accompany one another. A good way to enjoy the best of both worlds is to develop applications using the garbage collector in order to speed up the development process and obtain more reliable and maintainable code. When development is completed, the proposed mechanisms should be invoked to help improve the efficiency of the application via small local modifications of a few (or even a single) allocation site. The rest of the allocations still use automatic memory management.

We have explored the SPECjvm98 benchmark suite [90], the SPECjbb2000 benchmark [90], and the DaCapo benchmark suite [27]. The profiler identified many of these benchmarks as candidates for improvement and we have upgraded the performance of these benchmarks us-

[1]This project is a joined work with Yoav Ossia, from IBM Haifa Research Lab.

ing simple local modifications via the two proposed patterns. The modifications were applied to a small number of allocation sites (typically to a single allocation site in each benchmark). Since small local modifications made a significant efficiency improvement for many standard benchmarks, we expect a similar phenomenon with actual commercial applications. The patterns proposed in this work may also be used with legacy code previously developed.

Our profiler extends the Merlin trace generator [46]. The profiler collects information during the program run and its output highlights the candidate allocation sites. It also indicates which pattern may be used to improve performance. Making these profiler-driven modifications fully automate is an interesting future work. Full automatization can add the safety guaranty. In this work we concentrate on very simple and local modifications, for which safety can be easily checked.

**Measurements.** We ran measurements comparing the performance of the original versions of the applications and the modified versions. These were all run on different JVMs and various memory managers to verify the universal effectiveness of the patterns. The results show substantial improvements in performance, depending on the JVM, the garbage collector, and the benchmark. For example, with compound pooling SPECjbb2000 can be improved by 3.0–9.3%. The SOfMA pattern improves the _227_mtrt benchmark of the SPECjvm98 suite by 6.9–14.3%. Breaking down the improvement measured between the reduction in garbage collection work and the reduction in application overheads, we see that almost always both improve substantially. One exception is the naive object pooling that, although it consistently reduces garbage collection work, also unfailingly impacts badly on the application throughput.

**Chapter organization.** The patterns used to reduce memory management overheads are introduced in Section 7.2. Section 7.3 introduces our profiler. Section 7.4 describes, for each benchmark, the profiler's output and the techniques chosen to be applied over it. Measurements are presented in Section 7.5. Related work is discussed in Section 7.6 and future work possibilities are presented in Section 7.7. We conclude in Section 7.8.

## 7.2 Explicit object management patterns

An allocation site is a line of code that allocates an object. When an allocation site is frequently called, but the allocated object becomes unreachable soon thereafter, we say that the allocation at that site is excessive. We propose three patterns dealing with the implied inefficiencies of excessive allocation sites: *object pooling*, *compound object pooling*

and *SOfMA*. We first review the (standard) object pooling pattern.

### 7.2.1 Object pooling

Object pooling is a mechanism that explicitly allocates and de-allocates objects within the program. A "pool" of objects is initially allocated and kept as a free list. Instead of using the memory manager, the program takes an object from the pool when it needs one, and returns it to the pool when it is no longer needed. When the required number of objects exceeds the number of pool objects, more objects are allocated to the pool on-the-fly. Object allocation and deallocation are replaced with pool operations (get-from-pool and return-to-pool). Traditionally, object pooling is recommended only for objects whose allocation is dependant on invoking a costly system resource (such as setting up a communication port), which can be eliminated by pooling.

Object pooling has other advantages. As it reuses the same objects instead of allocating new ones, it reduces garbage collection work and improves locality. In addition, the user's knowledge about when an object is no longer needed is likely to be more accurate than the object's un-reachability property that is used by the garbage collector [89]. Exploiting more accurate information can reduce the space usage and increase efficiency. Finally, reusing an object saves some of the system overhead on allocation of new objects, e.g., Java always resets all fields of a new object.

Nevertheless, object pooling has also drawbacks. First, the loss of automatic reclamation benefits for software engineering, security, and ease of debugging. Second, the memory manager usually allocates memory much faster than the pool yields an object. In addition, in a multi-threaded application the pool operations must either use synchronization (during allocation and de-allocation) or one pool per thread must be used, requiring more cycles to identify the adequate pool and more space overhead. Finally, pooled objects are reachable and thus create work for a tracing garbage collector on the lookout for them. This may become significant if large pools are used.

To sum up, we expect object pooling to reduce the overall garbage collection overhead, while (usually) decreasing the program threads' performance.

### 7.2.2 Compound object pooling

*Compound object pooling* is an improvement over traditional object pooling. In compound

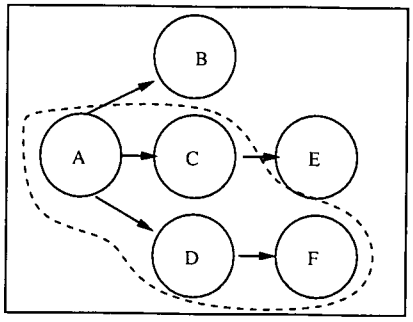



Figure 7.1: A compound pool sub-graph example

pooling, an object is implicitly pooled together with some (or all) of its descendants. We denote the object and its descendants that are pooled together with it *the object's sub-graph.* Compound pooling is applicable when the program allocates such a sub-graph frequently and consistently. Namely, the parent object is allocated, its descendants are allocated thereafter, and references are set in the parent to reference its descendants. A Compound pool keeps a parent object that also has its sub-graph descendant objects allocated and referenced by it. When the program needs to allocate a sub-graph, its root is taken from the pool by a get-from-pool operation, and a return-to-pool operation returns the root to the pool when the sub-graph is no longer needed.

Note the improvement over pooling all objects. The root obtained from the pool already references to allocated descendant objects. So further allocations and (reference) assignments are not required; only field initializations are executed. For a sub-graph that contains $n$ objects, we do not need to allocate $n$ objects from the system or obtain $n$ objects from a pool. We execute exactly one pool operation. Reclaiming of a subgraph has similar savings. This saving is even more effective if pool operations require synchronization by the program threads. However, compound pooling is not as widely applicable as traditional pooling. It can be effectively used only with sub-graphs, all of whose objects are allocated at the same time and become un-reachable at the same time.

Figure 7.1 depicts an example of an object graph with six objects. If $C$, $D$ and $F$ are always allocated together with $A$ and become unreachable whenever $A$ becomes unreachable, then $A$ can be pooled (implicitly) together with $C$, $D$ and $F$. The next time $A$, $C$, $D$ and $F$ are needed, the program will obtain them from the pool in single get-from-pool operation (and later return them in a single return-to-pool operation). Therefore, each such sub-graph reuse exchanges one pool operation for four object allocations and three assignments ($C$ to $A$, $D$ to $A$, $F$ to $D$). When the sub-graph becomes unreachable, a single pool operation replaces four naive pool operations or a portion of the garbage collection work needed to reclaim four objects. Note that objects $B$ and $E$ still require treatment by the program. They

are either allocated anew or just become referenced by $A$ and $C$, respectively (depending on the existing program code).

### 7.2.3 Single object for multiple allocations

The last pattern proposed is relevant when many instances of a class are allocated, but only one instance is reachable at each specific point in time. In such cases object pooling is not required, as even a single object can host all such allocations. More specifically, suppose a reference slot $p$ references an object of class $C$ and there is a frequently-called method that allocates an object of class $C$ to $p$. Further, suppose that an object that is allocated to $p$ is tightly dependant on $p$ in the sense that it is created with $p$ being the first reference to it and it becomes unreachable exactly when $p$ stops referencing it (upon assignment of a newly allocated object to $p$). In this case, we may allocate a single object to host all the transient objects to which $p$ references throughout the execution. Usually, we allocate this object as a referent of $p$ if $p$ is an instance variable, or we allocate it to an additional instance variable that is added to the object containing the method in which $p$ is local. The single fixed instance of this object is allocated once and is modified whenever an allocation to $p$ appears in the original code. We denote this approach *SOfMA*, *single object for multiple allocations.*

We found that, in several benchmarks, a large percentage of the overall allocated objects are allocated in this manner. In these extreme cases, SOfMA provided substantial improvement in performance. As with the object pooling patterns, SOfMA improves locality and reduces allocations and garbage collection overhead by employing object reuse. However, using SOfMA, the pool operations are no longer required. Thus, SOfMA does not reduce the program threads' performance as object pooling may do.

In the benchmarks that we explored we found two different types of locations in which such a reference $p$ appears. A description of these types follows.

**An instance variable.** In two of the benchmarks with which we worked (_201_compress and _209_db), newly allocated objects were repeatedly being assigned to a single instance variable during the program run (in a single allocation site). To deal with this case, we assigned a fixed single object to this instance variable at the beginning of the program, and modified it to have the relevant values whenever a new allocation of object is executed to it.

**A temporary variable.** The second location in which the reference $p$ appears is in a local method. In this case, we substitute the allocation to $p$ by using an object that is

allocated once when the object holding the method is created. Then, instead of allocating to $p$, we reuse the single allocated object. In this case, a multithreaded program requires more care. The modification can be applied if the object in which the method is a member is thread-local. Otherwise, two threads may execute the method at the same time and create a race condition. An alternative modification that avoids such race is to allocate the object on the stack. This requires compiler cooperation, or a placement of local variables representing the fields of the object on the stack and making the implied code modifications.

**Comparison with escape analysis and stack allocation.** Escape analysis [78, 24, 42] is a static analysis used to find references to objects that do not escape a method. These objects may be allocated on the stack to eliminate overhead of their allocation and reclamation. The first SOfMA type is not identified by escape analysis as the object allocated to $p$ escapes its allocating method. The second type is more related to escape analysis and may allow stack allocation. However, escape analysis will not necessarily identify the cases that we found in our benchmarks. This is because $p$ is passed to methods that assign it to a heap referent. While it is easy to see that this referent is removed, it is not clear whether the conservative analysis is smart enough to catch it. Refinements of escape analysis may be considered (as a future work) to identify such cases.

### 7.2.4 Example

Figure 7.2 introduces a simple example that demonstrates where applying the presented patterns is advised. The example's main class is the DATA class, which holds a variable sized array of POINTS. The SETDATA method creates a new array of POINTS and randomly initializes the POINT values. The method DOESEXISTS checks whether a certain POINT exists in the DATA's POINT array.

Note that each allocated POINT array is tightly dependant on the ARRAY instance variable of a DATA object: it is created (in SETDATA) with ARRAY being the first reference to it, and when a newly allocated POINT array is assigned to ARRAY, it becomes unreachable. Second, each such assignment to ARRAY makes all POINT objects (which were until now referenced by ARRAY) unreachable. Later the SETDATA method allocates $n$ new POINTS, to be referenced by the newly allocated POINT array. Frequent invocations of SETDATA, or dealing with large $n$ parameters could create a heavy load for the memory manager. In this example, allocations of the POINT array can be reduced by using the SOfMA pattern, and allocations of POINTS can be reduced by using the pooling pattern. A POINT array must be allocated only if the current POINT array is smaller than the required allocated array size. If

```
import java.util.Random;

public class Data {
    class Point {
      private int x;
      private int y;
      Point(int _x, int _y) {
        x = _x;
        y = _y;
      }
      public boolean equals(Point p) {
        return ( x == p.x && y == p.y );
      }
    }

    private Point[] array;
    private int size;
    static private Random generator =
          new Random();

      public void setData(int n) {
        array = new Point[n];
        size = n;
        for ( int i=0 ; i<n ; i++ ) {
          int x = generator.nextInt();
          int y = generator.nextInt();
          array[i] = new Point(x,y);
        }
      }

      public boolean doesExists(int x,
                    int y) {
        Point search = new Point(x,y);
        for ( int i=0 ; i<size ; i++ )
          if ( search.equals(array[i]) )
            return true;
        return false;;
      }
  }
```

Figure 7.2: Example of where the patterns are necessary

the current POINT array suffices for the allocation, then it can be re-used: one should only be able to distinguish between the array's real size and its size as viewed by the application (which may be smaller). In addition, one can reduce the number of POINT allocations by re-using the existing ones. If the "new" array is smaller or equal in size to the previous one, one should only set the values of the relevant POINTS, while moving the extra POINTS into a POINT pool. If the "new" array is larger than the previous one, POINTS should be taken from the POINT pool (note that before a POINT array replaces an existing one, the POINTS of the "old" array should be pushed into the pool).

The DOESEXISTS method includes another case of a referent tightly dependant on its reference slot: the SEARCH variable, which is assigned a newly allocated POINT object upon

each invocation of DOESEXISTS. As before, if DOESEXISTS is frequently executed, it abuses the memory manager. However, unlike the above SOfMA case, here the tightly dependant referenced object is method local. Hence, the SOfMA pattern for an object that is local to a method may be used. One adds a POINT instance variable to the DATA object, and allocates this POINT when the DATA object is created. Each DOESEXISTS invocation should set the value of this POINT (instead of allocating a new POINT).

## 7.3 The profiler

In order to effectively apply the above three patterns, we propose a profiler that identifies candidate allocation sites. A main criterion for being a candidate is being an allocation site that heavily allocates objects that die fast. In addition, classes whose objects are extremely large should also be considered as candidates, since even saving relatively few allocations of large objects may have a large impact on garbage collector work.

Our profiler outputs information regarding the behavior of each allocation site's objects during the examined application run. For each allocation site, the profiler outputs the class of allocation, the number of objects allocated, the number of objects that became unreachable, the amount of space its objects have consumed, and the death rate of objects allocated in this site. We chose to measure the death rate of a site by calculating the percentage of the site's objects that die in each collection. A death rate of 100% means that none of the site's allocated objects has ever survived a collection, while a death rate of 0 means that all of them survived. For each allocation site and in each collection, our profiler counts the number of objects (of that site) that are allocated in the heap before and after the collection. The death rate is computed per collection, and for the entire run of the application as a geometric mean of the collections' death rate. We usually search for sites with high death rates, as pooling such objects is expected to be the most beneficial (an exception are classes whose objects are large).

Profiling by site yields several advantages. First, the code modifications that we propose are relevant per allocation site. Second, even for frequently allocated instances of a class, various benchmarks usually employ only a few major allocation sites (sometimes only one) to allocate most of the objects. To yield a substantial benefit, we focus on the major allocation sites. Third, different allocation sites may create objects (of the same type) that behave differently (for example, different death rates for different allocation sites). Hence, profiling per site filters out other sites' "noise".

| Objects allocated | Dead objects | Death rate | Bytes allocated | Avg. size | Object type |
|---|---|---|---|---|---|
| 39098 | 38045 | 86.44 | 3753408 | 96.00 | Lspec/jbb/Order |

| Correlation | Dead | Null | Field |
|---|---|---|---|
| 0.0 | 0 | 0 | Lspec.jbb.infra.Factory.Container |
| 0.0 | 0 | 0 | Lspec.jbb.Company |
| 0.0 | 0 | 0 | Lspec.jbb.Customer |
| 100.0 | 38045 | 0 | [Ljava.lang.Object |
| 100.0 | 38045 | 0 | Ljava.util.Date |

Table 7.1: An example of the profiler output

Being interested in compound pooling, we would like to find classes whose objects could be the roots of a compound pool (i.e., could be pooled with its sub-graph). Thus, we search for classes whose objects have a perfect death correlation with one or several of their descendants, i.e., one or more of their descendants always become unreachable when they become unreachable. For that purpose, our profiler, implemented on Jikes [1], extends the Merlin trace generator [46] which determines the exact death time of an object. If an object and its descendant die at the same time, then a single event (such as reference modification) made the "top" object unreachable together with its descendants. Our profiler runs after a garbage collection cycle terminates, checking for each unreachable object whether its descendants are also unreachable and have the same death time. For each allocation site and for each descendant, the profiler details the percent of death correlation. If an object has a death correlation of, for example, 80% with its first descendant, it means that in 80% of the cases the parent object allocated in a specific site became unreachable, its first child became unreachable along with it. A 100% death correlation of an allocation site's objects with one or more of their descendants may imply that it would be beneficial to use compound pooling (if this allocation site is frequently called).

### 7.3.1 Profiler example- SPECjbb2000

Table 7.1 shows an example of the profiler output for a certain SPECjbb2000 allocation site. The class name is provided in the last column of the first sub-table. This first sub-table provides the number of objects allocated at that site, the number of objects that become

unreachable, the death rate, the total size of the allocated objects and the average size of an object. The second sub-table of Table 7.1 (the correlation table) is produced for any allocation site whose associate class has reference instance variables. For each reference instance variable (whose name is provided in the last column), the correlation table reveals the death correlation percent between the allocated object in this allocated site and the said reference instance variable (in column 1). Column 2 provides the number of times this referent became unreachable when the object became unreachable. Column 3 provides the number of times this reference was null when the object became unreachable. For the calculation of correlation percentage, we considered both numbers of unreachable and null reference fields. If there is a 100% correlation overall and the percentage of NULL references is small, then compound pooling could also be considered. Specifically in our example, Table 7.1 shows that ORDER has a high death rate: on average in each collection more than 86 out of 100 allocated instances of this object are reclaimed. The correlation table shows that while three reference instance variables have no correlation at all with the parent object ORDER, the other two reference instance variables [OBJECT and DATE both have 100% correlation to it (with no null values). Hence, ORDER is a candidate as the root of a compound pool's sub-graph with these two descendant objects. However, one should not stop here, but continue examining the death correlation of the two descendants. For example, if [OBJECT also has a 100% correlation with its children, then the sub-graph of ORDER, which should be put in the compound pool, should be larger (yielding more allocation reduction, and less GC overhead).

Overall, for SPECjbb2000 the profiler indicated that the ORDER object serves adequately as a sub-graph's root of a compound pool. It has a high death rate (86.4%), and a 100% correlation with both its OBJECT array and its DATE fields. Moreover, the OBJECT array field has a 100% death correlation with its ORDERLINE elements.

## 7.4 Profiler's output and benchmarks modifications

The effectiveness of the proposed techniques was measured on the SPECjvm98 benchmark suite, the SPECjbb2000 benchmark, and the DaCapo benchmark suite. The profiler was run on each one of these benchmarks to determine which allocation sites were adequate candidates for the patterns' application[2]. This section describes, for each benchmark, the

[2]Although Merlin is currently unable to produce exact death times for multithreaded applications, it was also used for profiling our multithreaded applications on a uniprocessor.

profiler's findings, and the patterns chosen to use as a consequence.

**_201_compress.** The profiler revealed that _201_compress allocates, in two adjacent allocation sites, some dozens of large byte array instances with a high death rate (93.0%). These two allocation sites allocate a substantial fraction of the overall allocation of the application. These byte arrays are tightly associated, as explained in Section 7.2.3, with two reference member fields of the HARNESS class. Thus, we chose to use the SOfMA technique and assign a single array to each of the two member fields at the beginning of the program, and modify this array to have the relevant values whenever a new allocation of a byte array is assigned to this reference. If, during the application run, a larger array than currently exists is allocated, then an array of the larger size is allocated and assigned to the appropriate member field. In order to compare the different implications of SOfMA and naive pooling, we extended _201_compress into two different versions one for each method. Naturally, the pool version required only two pooled arrays that were repeatedly assigned to the two reference member fields.

**_202_jess.** For _202_jess, the profiler indicated that the TOKEN objects are vastly allocated, and have a 100% death correlation with their VALUEVECTORS descendant array. The array of VALUEVECTORS is the most heavily allocated class (by number of objects and by overall consumed space). 99.9% of the TOKEN objects are allocated in a single site, which has a high death rate (87.9%). Hence, a compound pool of TOKEN objects would seem appropriate. The compound pool's sub-graph contains only two nodes (a TOKEN object and its VALUEVECTOR array), as the array has a 0% death correlation with its elements. Hence, two extensions were made for _202_jess: the first employs the above compound pool, and the second uses two naive pools: one of TOKEN and one of VALUEVECTOR arrays.

**_209_db.** The profiler revealed that _209_db allocates, in a single allocation site, a few hundreds instances of large arrays of class ENTRY, which require large memory and have a high death rate (95.7%). These ENTRY arrays are tightly associated with a DATABASE's class member field. Thus, two extensions were made: the SOfMA extension, which is the most appropriate technique in this case (assigning a single array to the member field at the beginning of the program), and the naive pool extension.

**_227_mtrt.** The profiler showed that _227_mtrt allocates over 3.5 million VECTOR objects in three allocation sites, which had high death rates (99.99%). These objects did not escape their allocating methods and were tightly associated with a method local reference. In addition, the object holding the reference was thread-local. Thus, it was possible to apply the SOfMA technique, by replacing the allocation in these methods' local variable with a VECTOR object allocated once when the object holding the method is created. The relevant

methods' allocations are replaced to reuse this single allocated VECTOR. In addition to the SOfMA extension, a naive pool extension was also written. To avoid synchronizing on the pools, the multithreaded _227_mtrt employs two local pools, one for each thread.

**_213_javac, _222_mpegaudio and _228_jack.** We did not apply the proposed techniques on _222_mpegaudio, _213_javac, and _228_jack. _222_mpegaudio's allocation activity is very small, and thus it contains no class that is appropriate for allocation improvements. We could not apply any technique on _213_javac and _228_jack since our techniques require modifications of the code, which was not available to us. However, for both _213_javac and _228_jack the profiler indicated that there are some potential allocation sites for improvement: a single allocation site of the INSTRUCTION class in _213_javac, and a single allocation site of the LJAVA.UTIL.VECTOR class in _228_jack could be modified to host compound pooling with their descendant object arrays.

**SPECjbb2000.** The profiler's output for SPECjbb2000 indicated that SPECjbb2000 is appropriate for compound pooling of the ORDER class (as detailed in Section 7.3.1). Hence, two extensions were made for SPECjbb2000: the first employs the above compound pool, and the second uses four naive pools: ORDERS pool, DATES pool, OBJECT array pool and ORDERLINES pool. As with _227_mtrt, each SPECjbb2000 warehouse (thread) employs its own local pool.

Since the size of the ORDERLINE array is not fixed (its size randomly varies between 5 and 15 elements), the compound pooling version allocated 15 ORDERLINES per ORDER, which is the maximal number of ORDERLINES used per ORDER. Hence, the compound pool version of SPECjbb2000 deals with an 18 node sub-graph. In contrast, the naive pool version allocates the exact number of ORDERLINES.

**ps.** According to the profiler, the ps benchmark contains a two node sub-graph that may be used with compound pooling. This sub-graph's root is LJAVA.UTIL.STACK, which has a 99.97% death rate and is vastly allocated in a single allocation site. LJAVA.UTIL.STACK has a 100% death correlation with its LJAVA.LANG.OBJECT array field. Hence, a compound pool extension was made for ps. Since LJAVA.UTIL.STACK is a built-in Java class, making a naive pool extension for ps requires modifying Java's LJAVA.UTIL.STACK, and so we did not create a naive pool version for ps. Any pooling of the LJAVA.UTIL.STACK (that does not modify LJAVA.UTIL.STACK) is an implicit compound pool.

In the single allocation site in ps of LJAVA.UTIL.STACK with which we dealt, LJAVA.UTIL.STACK did not escape the method in which it was allocated. However, since it is a recursive method, more than one LJAVA.UTIL.STACK can be reachable at a time, and thus a SOfMA technique is not appropriate. Upon the method termination, the LJAVA.UTIL.STACK is empty, and

thus no code that empties the LJAVA.UTIL.STACK had to be inserted.

**Other DaCapo benchmarks.** We have also profiled the other DaCapo benchmarks which are available for run on Jikes. Two benchmarks in this suite (antlr, fop) do not have a major allocation site, and so modifying a couple of allocation sites cannot improve performance significantly. The other four benchmarks in the suite (bloat, hsqldb, jython and xalan) do contain one or two major allocation sites, but they cannot be modified using the proposed patterns. None of them fitted the SOfMA pattern. For compound pooling, we needed a 100% correlation between the object and at least one of its children. There was no fit excessive allocation site.

## 7.5 Measurements

**Implementation.** We have implemented a standard object pool using Java's ARRAYLIST class, an unsynchronized resizable-array implementation of Java's LIST interface. We've employed ARRAYLIST in a stack-like manner, i.e., the last object inserted into the pool, would be the first to be drawn.

**Platform.** Measurements were run on a 4-way IBM Netfinity 8500R server with a 550MHz Intel Pentium III Xeon processors and 2GB of physical memory.

**The modifications.** For each benchmark, we selected (according to the profiler's output) one class whose objects were found to be the most appropriate for applying our techniques. Next, the benchmarks were modified to use a pool of these classes' objects. For benchmarks where the profiler's output indicated that the chosen class is also appropriate for compound pooling, we also modified the benchmark to use a compound pool. Hence, for these benchmarks a pool version and a compound pool version are both available[3]. While the compound pooling version employs a pool (or pools) only for the "root" object, the (naive) pooling version also contains a pool (or pools) for each descendant class that dies together with the "root" object. In addition, for relevant benchmarks, a SOfMA version of the benchmark was also created.

**The comparison.** The overall running time (or throughput) of the original benchmark was compared with each one of its available modifications (pooling, compound pooling, or SOfMA).

**JVMs.** In order to examine the effects of the three techniques, we ran our measurements

[3]Except for ps, as detailed in 7.4.

on different JVMs employing different memory managers. The different constellations used were:

- Jikes RVM [1] with the parallel mark and sweep collector.
- Jikes RVM [1] with the parallel Appel-style ([2]) collector employing a flexible sized young generation which consumes all the usable heap space. This Appel-style collector employs a copying collector in minor collections and a mark and sweep collector during full heap collections.
- IBM's 1.4.1 jdk with the default parallel mark and sweep collector, which infrequently employs compaction.
- IBM's 1.4.1 jdk with the concurrent collector (reported in [9]). This is an incremental mark and sweep collector, as the program threads execute some of the collection work in each allocation. It is also a concurrent collector as most of the collection is done while mutators are running.

The heap size for the measurements was set to enable a standard maximal heap occupancy of 60%, i.e., the live objects did not occupy more than 60% of the heap right after a full collection.

### 7.5.1 Allocation activity

Table 7.2 describes the allocation activity of each benchmark and the difference between the original and modified versions. The number of allocated bytes and objects are shown in columns 2 and 3, respectively. Columns 4 and 5 provide the fraction of reduced allocation (of bytes and objects) in the (naive) pool version of these benchmarks. Column 6 presents the number of allocation sites that were modified in the benchmark to obtain the improvement. The measurements were taken with naive pooling (except for the ps benchmark), but the other methods yield similar numbers as the reuse of objects is almost the same.

As one can see, there is substantial reduced allocation activity in most benchmarks, even though we have focused on only a couple of allocation sites. This demonstrates that even focusing on only one popular allocation site can significantly affect the application behavior. Usually there is a correlation between the reduction in space allocated and the reduction in the number of objects allocated. This is not the case for _201_compress and _209_db, because for these benchmarks extremely large objects were pooled. Hence the reduction

| Benchmarks | original allocation | | reduced allocation | | sites |
|---|---|---|---|---|---|
| | bytes | objects | bytes | objects | |
| compress | 105.4MB | 10230 | 79.4% | 4.8% | 2 |
| jess | 283.5MB | 7.94M | 58.2% | 47.4% | 1 |
| db | 74.7MB | 3.21M | 26.8% | - | 1 |
| mtrt | 159.4MB | 6.64M | 54.4% | 57.0% | 3 |
| jbb [4] | 1995MB | 59.37M | 18.6% | 9.0% | 1 |
| ps | 440.3MB | 7.27M | 8.2% | 13.0% | 1 |

Table 7.2: Allocation activity: How many bytes and how many objects were allocated by the original benchmark and the percent of this allocation activity reduced with naive pooling.

in space allocated is much larger than the reduction in the number of objects allocated (especially in _209_db, where the reduction in the number of objects allocated is negligible).

### 7.5.2 Results

The overall improvement in the application throughput was compared for each modified benchmark and for each of the JVMs described above. The separate effects on the program threads (mutators) and on the collector were also measured. The results appear in Table 7.3. Table 7.3 also details, for each (original version of the) benchmark at each configuration, the fraction of time used by garbage collection. The only exception was the concurrent collector, which does not produce this data. Note that all the numbers included in the Table represent percentages. In the second column, the letter 'J' stands for Jikes and the letter 'I' stands for IBM.

As expected since applying the patterns heavily reduces the benchmarks' allocations on the heap, all three approaches always reduced the amount of work needed by the garbage collector. However, while compound pooling and SOfMA also improve the mutators' throughput, object pooling decreases their performance. Our measurements of the mutators' improvement (for the naive object pooling) accounts for two pool operations in the modified version for each allocation in the original version. Hence, the results indicate that creating an object is usually faster than applying the two pool operations on it (even though pooling should improve locality) [5]. On the other hand, SOfMA eliminates many allocations, and

[5]Our measurements did not include objects that require heavy resource initialization, such as setting up

compound pooling replaces a full sub-graph allocation (rather than a single object allocation) and their assignments with two pool operations.

**compound pooling vs. object pooling.**

Compound pooling was used for benchmarks _202_jess, SPECjbb2000 and ps. The sub-graphs of _202_jess and ps contain only two nodes, while that of SPECjbb2000 contains 8-18 nodes (as the OBJECT array includes 5-15 ORDERLINE elements).

In all pooling and compound pooling versions of these benchmarks, a reduction in garbage collection work was obtained. Nevertheless, compound pooling performs much better when the efficiency of the mutators is compared. Naive pooling actually degrades the performance of _202_jess and SPECjbb2000, since it executes many more pooling operations than compound pooling. The results of _202_jess and ps show that even compound pooling a sub-graph of only two nodes is beneficial in most cases. However, since the sub-graph is much larger in SPECjbb2000, the efficiency improvement (of the mutators) there is higher.

To conclude, compound pooling improves the overall performance of all three benchmarks. It outperforms naive pooling, which sometimes slightly degrades performance.

**SOfMA vs. object pooling**

**_227_mtrt.** _227_mtrt allocates millions of objects, which do not escape their allocating method, in three allocation sites. The garbage collection work was reduced in both the SOfMA and object pooling versions. However, while the SOfMA version also improves the mutators' work, the pool version usually degrades their performance. The reason is that the SOfMA version really saves millions of allocations, while the pool version replaces the allocations with pool operations. As a result, the SOfMA version improves the overall application run, while the (naive) pool version degrades the overall performance. [6]

**_201_compress and _209_db.** In both benchmarks, large arrays tightly dependant on a class instance variable were present. Although in both benchmarks, at most a few hundred object allocations are saved using the patterns, saving these allocations introduces a significant reduction on garbage collection work, since these objects are large ones. Note that since not many instances of these "problematic" objects were allocated, there were not many allocations to save or to trade with pool operation. Thus, our measurement showed that both traditional pooling and SOfMA usually had minor impact on the mutators' performance

a communication port.

[6]Note that escape analysis identified the stack allocation opportunities in _227_mtrt (on both JVM's) only when we performed some method inlining. This demonstrates that SOfMA may improve application performance, even when escape analysis opportunities are available.

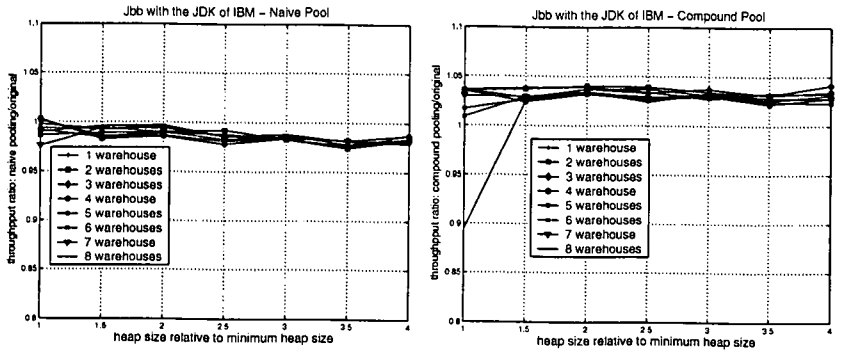



Figure 7.3: SPECjbb2000 on IBM's JDK- naive pooling throughput ratio (left) and compound pooling throughput ratio (right).

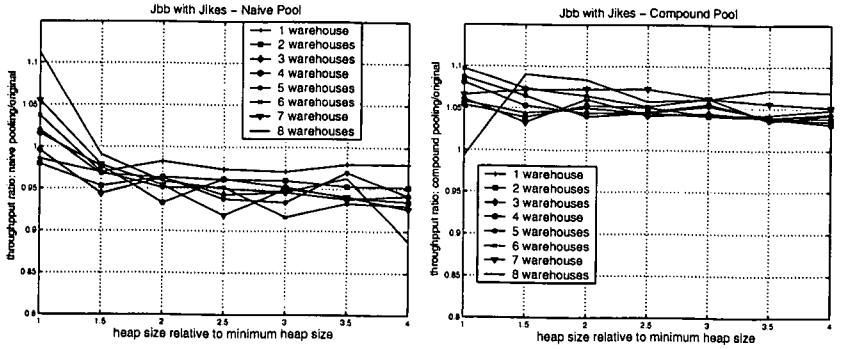



Figure 7.4: SPECjbb2000 on Jikes- naive pooling throughput ratio (left) and compound pooling throughput ratio (right).

(with a small advantage to the SOfMA approach). As a consequence, the overall performance of both benchmarks was usually improved.

### 7.5.3 SPECjbb2000 measurements

To show the effect of pooling over various heap sizes, we have run SPECjbb2000, the largest benchmark we've used, over different heap sizes (ranging from the minimal heap needed, to 4X the minimal heap needed). Measurements were taken using Jikes Appel-style parallel collector, and IBM's 1.4.1 jdk default collector. Figures 7.3-7.4 introduce the measured throughput ratios. These measurements show that compound object pooling consistently outperforms the original program, while naive pooling does usually worse. The relaxed heaps measurements show that compound object pooling remains superior even when the garbage collection fraction decreases, thus indicating that compound object pooling indeed improves the mutators throughput. Naive object pooling, however, gets less efficient as heap

grows, since the garbage collection impact (advantage) gets weaker, and the overhead it imposes over the mutators becomes the substantial factor.

### 7.5.4 Garbage collection impact

Table 7.3 shows a large reduction in the garbage collection work for all patterns and for all benchmarks. Table 7.4 shows the number of garbage collection cycles and their average amount of work (i.e., collection lengths) per collection after applying the three suggested patterns, compared to their original behavior. The cells referring to Jikes' Appel style collector contain two numbers (separated by a semicolon): the first represents the full collections, and the second represent the minor collections.

One should note that the Jikes mark and sweep collector consistently performed (sometimes much) fewer collections than IBM's mark and sweep collector. The reason for this is that Jikes has a larger heap space to allocate the objects. Jikes is self-hosted, therefore, it allocates objects in the same heap in which the application allocates. Thus, when sticking to the 60% occupancy, it also takes Jikes objects into account and seems to effectively provide a larger heap to the objects allocated by the benchmark. When running the application, Jikes' effectively larger heap translated into less garbage collection cycles.

As can be seen, the number of collections was reduced with all patterns and all JVMs. Another effect of our modifications is that when a collection takes place, the percentage of reachable objects grows. Since dealing here only with tracing collectors, we expect each collection to do more work. When analyzing the collections' relative amount of work, one can see that each collector was affected differently. The amount of work of Jikes' mark and sweep collections was hardly affected, while that of the IBM's collector usually increased. We suspect that Jikes' mark and sweep collector was less sensitive to the increased percent of reachable objects, since it already includes a larger set of reachable objects (being self hosted), and so the increased percentage of reachable objects was less significant in its case. With Jikes' Appel style collector, the amount of work of the minor collections increased (while major collections were usually not performed). Since the fraction of surviving objects in the nursery area is larger, more objects should be copied to the mature space, increasing the amount of work of the minor collection.

## 7.6 Related work

Berger et al. ([12]) examined applications, written in the C language, that use custom allocators to achieve performance improvements. They found that for most of these applications, and in particular for those using object pooling, a general purpose allocator performs as well as or better than the custom allocators. Our results, focusing on Java written applications, indicate that indeed traditional pooling usually degrades mutators' performance, but since it also reduces garbage collection work, it sometimes improves the overall performance. However, compound pooling was very effective in improving the overall performance, and usually also improved the mutators' performance.

Much research has been devoted to reducing the burden on the garbage collector. Escape analysis is discussed in Section 7.2.3. Region-based memory management ([96]) allocates each object into a specific region. Memory is reclaimed by reclaiming a region as a whole, hence freeing all the objects allocated in it efficiently. This reduces garbage collection work. Regions could be automatically inferred during compile time, or explicitly annotated by the programmer ([41]). Region inference could also be combined with garbage collection ([45]). Run-time maintained regions have also been proposed and explored in [20, 87].

Hirzel et al. [48, 47] have attempted reducing the garbage collection overhead by taking into account the connectivity of objects. Connectivity strongly correlates with object lifetimes and death times, and thus can be used to improve garbage collector's performance. Such garbage collectors use connectivity information to partition objects and to decide which objects to collect.

In [43], Gheorghioiu et al. have presented a static program analysis which finds all pairs of allocation sites, where an object allocated at one site may be live at the same time as any object allocated at the other site. Their algorithm infers from this result the unitary sites: the allocation sites from whom at most one object allocated is live at any given point in the execution of the program. It then statically preallocates (a fixed amount of) memory space for objects allocated in unitary sites. This approach resembles our SOfMA technique, however SOfMA also handles unitary sites whose objects escape into the heap (which is the case in _201_compress and _209_db), while the algorithm presented in [43], automatically assumes that such sites are non-unitary.

## 7.7 Future work

The interest in compound pooling stems from the ubiquitous use of objects whose reachability is determined by one single parent. To check how wide this phenomena is, we used the profiler to look for "popular" classes that had this property. To make the test realistic, we only measured *popular* classes, whose objects consume at least 100 kbytes of allocated space throughout the run. We looked for classes whose objects satisfy a full (100%) death correlation with at least one of their descendants. Only classes which contain references were considered *relevant*. Table 7.5 presents for each benchmark the number of relevant popular classes that satisfy the 100% death correlation property with at least one of their descendants (first row), and the number of relevant popular classes which do not satisfy this property (second row). Classes (or primitives) which do not contain references were not considered, but array of references were taken into account. It can be seen that the 100% death correlation property is ubiquitous among the different benchmarks. Note that these measurements are conservative in the sense that they count properties of classes rather than of allocation sites. For a class which does not have the 100% death correlation property, there could be allocation sites for which the property does exist. Furthermore, having more than one correlated descendant is useful in practice, but is not highlighted here.

Compound object pooling is one manner in which a 100% death correlation knowledge can be used. However, we used it via profiling and only to a single major allocation site. Higher effectiveness (and over many allocation sites) can be obtained if 100% death correlation could be statically determined. A static analysis applied on a certain class A could check whether a certain descendant B is always created with the A object being the first reference to it, and whether B is never assigned to another object. In such cases, static analysis can conservatively infer that when an A object dies, its B descendant dies together with it.

Such static analysis can be used, for example, to save allocations. Instead of allocating A and B separately by two different allocation requests, they could be both allocated sequentially by a single allocation request. Such joint allocation reduces the number of allocations and it also reduces fragmentation in allocators, such as allocation caches and segregated free lists. When A and B die together a larger sequential free space becomes available.

## 7.8 Conclusions

In this work we have proposed a profiler to identify and patterns to improve inefficient use of the memory management subsystem. Often, applications that are written in a garbage collected environment make naive and inefficient use of the garbage collector. Our solution started by proposing a profiler that identifies crucial allocation sites. These are (highly) prolific allocation sites whose allocated objects die young. We then proposed two patterns to modify these allocation sites (in a local manner) to improve efficiency: compound object pooling, and SOfMA (single object for multiple allocations). We studied these patterns and compared them to the standard object pooling pattern.

We used our profiler and applied the proposed modifications on various standard benchmarks. We then measured the improvement in efficiency for two JVMs and various garbage collection algorithms. In many of the benchmarks it was possible to make small modifications in a small number of allocation sites and obtain a substantial improvement in the overall application throughput. The effectiveness of the compound pooling pattern was demonstrated on the SPECjbb benchmark, for which naive object pooling deteriorates performance whereas compound object pooling obtains a 3.0–9.3% improvement depending on the specific JVM and garbage collector used. SOfMA was always at least as good as naive object pooling, and its effectiveness was demonstrated on the _227_mtrt benchmark for which object pooling deteriorated performance but SOfMA obtained an improvement of 6.9–14.3%.

| Bench marks | collector | overall improvement | | | % gc | reduction in mutator activity | | | reduction in gc activity | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | pool | comp' pool | SO-fMA | | pool | comp' pool | SO-fMA | pool | comp' pool | SO-fMA |
| comp-ress | J- ms | **11.7** | | **11.6** | 13.0 | 2.4 | | 2.6 | 73.1 | | 72.4 |
| | J- Appel | **11.3** | | **11.3** | 13.0 | 0.4 | | 0.5 | 84.4 | | 84.4 |
| | I- par' | **0.9** | | **1.6** | 1.7 | -0.6 | | 0.1 | 87.7 | | 90.5 |
| | I- conc' | **2.0** | | **2.1** | | | | | | | |
| jess | J- ms | **19.5** | **22.8** | | 38.0 | -1.0 | 4.2 | | 53.0 | 53.1 | |
| | J- Appel | **1.7** | **7.3** | | 7.2 | -1.8 | 3.9 | | 46.1 | 51.3 | |
| | I- par' | **-0.2** | **5.5** | | 9.8 | -5.9 | -0.1 | | 53.3 | 57.0 | |
| | I- conc' | **7.6** | **13.5** | | | | | | | | |
| db | J- ms | **2.9** | | **3.2** | 10.6 | -0.1 | | 0.1 | 28.4 | | 28.9 |
| | J- Appel | **2.2** | | **2.2** | 3.4 | -0.1 | | 0 | 66.9 | | 67.0 |
| | I- par' | **0.1** | | **0.3** | 1.8 | -0.3 | | 0 | 20.1 | | 19.3 |
| | I- conc' | **0.2** | | **0.7** | | | | | | | |
| mtrt | J- ms | **3.8** | | **14.3** | 21.4 | -8.2 | | 5.4 | 47.2 | | 47.2 |
| | J- Appel | **-5.0** | | **7.5** | 9.4 | -8.5 | | 4.8 | 28.7 | | 32.0 |
| | I- par' | **-5.2** | | **6.9** | 13.7 | -11.6 | | 3.2 | 35.1 | | 30.4 |
| | I- conc' | **-7.7** | | **11.9** | | | | | | | |
| jbb | J- ms | **-0.8** | **9.3** | | 7.9 | -2.8 | 7.4 | | 24.9 | 21.4 | |
| | J- Appel | **-1.0** | **8.2** | | 12.6 | -6.1 | 5.2 | | 37.7 | 20.2 | |
| | I- par' | **-0.8** | **3.0** | | 6.2 | -1.9 | 2.7 | | 15.7 | 4.5 | |
| | I- conc' | **-0.7** | **3.7** | | | | | | | | |
| ps | J- ms | | **4.7** | | 44.1 | | -0.6 | | | 11.5 | |
| | J- Appel | | **2.1** | | 13.4 | | 0.2 | | | 14.6 | |
| | I- par' | | **2.5** | | 4.4 | | 1.3 | | | 29.4 | |
| | I- conc' | | **5.5** | | | | | | | | |

Table 7.3: Speed-up improvement with different JVMs when applying the suggested patterns.

| Bench | collector | number of collections | | | | increase of work per collection | | |
|---|---|---|---|---|---|---|---|---|
| | | original | pool | comp' pool | SOfMA | pool (%) | comp' pool (%) | SOfMA (%) |
| comp- | J- ms | 7.6 | 2 | | 2 | 0.8 | | 0.9 |
| ress | J- Appel | 7;3.6 | 1;3.2 | | 1,3.2 | 0.2;0 | | 0;0 |
| | I- par' | 20 | 3 | | 3 | -16.9 | | -36.4 |
| jess | J- ms | 15 | 7 | 7 | | 0.7% | 0.5% | |
| | J- Appel | 0;92 | 0;39.8 | 0;39.8 | | n-a;24.8 | n-a;15.5 | |
| | I- par' | 108.6 | 46 | 43.6 | | 9.3 | 8.1 | |
| db | J- ms | 7 | 5 | | 5 | 0.1 | | 0.2 |
| | J- Appel | 1.2;32.4 | 0;12.2 | | 0;12.2 | n-a;101.4 | | n-a;96.7 |
| | I- par' | 13 | 10 | | 10 | 3.7 | | 5.0 |
| mtrt | J- ms | 11 | 6 | | 6 | 0.2 | | 0.2 |
| | J- Appel | 0;28 | 0;15.6 | | 0;15.6 | n-a;29.0 | | n-a;31.4 |
| | I- par' | 15 | 6 | | 7 | 62.0 | | 49.5 |
| jbb | J- ms | 78.6 | 60 | 58.4 | | -1.6 | 5.8 | |
| | J- Appel | 110;691 | 46.4;512 | 63.4;639 | | -0.1;0.1 | -0.5;2.4 | |
| | I- par' | 169.6 | 136.4 | 142.8 | | 4.8 | 13.4 | |
| ps | J- ms | 53 | | 48 | | | -2.3 | |
| | J- Appel | 1;511.6 | | 0.4;466.4 | | | 0.71;0.85 | |
| | I- par' | 119 | | 73 | | | 15.0 | |

Table 7.4: Garbage collection behavior when applying the suggested patterns

| Benchmarks | compress | jess | db | javac | mtrt | jack | jbb |
|---|---|---|---|---|---|---|---|
| correlated | 0 | 2 | 2 | 22 | 9 | 5 | 9 |
| non-correlated | 0 | 9 | 7 | 49 | 4 | 8 | 10 |

| Benchmarks | compress | antlr | bloat | fop | hsqldb | jython | ps | xalan |
|---|---|---|---|---|---|---|---|---|
| correlated | 9 | 16 | 21 | 16 | 3 | 1 | 9 | 5 |
| non-correlated | 10 | 12 | 23 | 13 | 10 | 10 | 8 | 8 |

Table 7.5: 100% death correlation per class

# Bibliography

[1] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeño in Java. In OOPSLA [73], pages 314–324.

[2] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[3] Andrew W. Appel. Emulating write-allocate on a no-write-allocate cache. Technical Report TR-459-94, Department of Computer Science, Princeton University, June 1994.

[4] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding view. In OOPSLA [74].

[5] Hezi Azatchi and Erez Petrank. Integrating generations with advanced reference counting garbage collectors. In *Proceedings of the Compiler Construction: 12th International Conference on Compiler Construction, CC 2003*, volume 2622 of *Lecture Notes in Computer Science*, pages 185–199, Warsaw, Poland, May 2003. Springer-Verlag Heidelberg.

[6] David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan, and Stephen Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In PLDI [84].

[7] David F. Bacon and V.T. Rajan. Concurrent cycle collection in reference counted systems. In Jørgen Lindskov Knudsen, editor, *Proceedings of 15th European Conference on Object-Oriented Programming, ECOOP 2001*, volume 2072 of *Lecture Notes in Computer Science*, Budapest, June 2001. Springer-Verlag.

[8] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[9] Katherine Barabash, Yoav Ossia, and Erez Petrank. Mostly concurrent garbage collection revisited. In OOPSLA [74].

[10] Mordechai Ben-Ari. On-the-fly garbage collection: New algorithms inspired by program proofs. In M. Nielsen and E. M. Schmidt, editors, *Automata, Languages and Programming. Ninth Colloquium*, pages 14–22, Aarhus, Denmark, July 12–16 1982. Springer-Verlag.

[11] Mordechai Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333–344, July 1984.

[12] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *OOPSLA'02 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Seattle, WA, November 2002. ACM Press.

[13] Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In PLDI [85], pages 153–164.

[14] Stephen M. Blackburn and Kathryn S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In OOPSLA [74].

[15] Daniel G. Bobrow. Managing re-entrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269–273, July 1980.

[16] Hans-Juergen Boehm. Reducing garbage collector cache misses. In Hosking [49].

[17] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

[18] Brendon Cahoon. *Effective Compile-Time Analysis for Data Prefetching in Java.* PhD thesis, 2002.

[19] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1998.

[20] Dante Cannarozzi, Michael Plezbert, and Ron Cytron. Contaminated garbage collection. In PLDI [83].

[21] *Proceedings of the 14th International Conference on Compiler Construction*, Edinburgh, April 2005. Springer-Verlag.

[22] Chen-Yong Cher, Anthony L. Hosking, and T.N. Vijaykumar. Software prefetching for mark-sweep garbage collection: Hardware analysis and software redesign. In *Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–210, Boston, MA, October 2004.

[23] Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. In Jones [54], pages 37–48.

[24] Jong-Deok Choi, M. Gupta, Maurice Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In OOPSLA [73], pages 1–19.

[25] T. W. Christopher. Reference count garbage collection. *Software Practice and Experience*, 14(6):503–507, June 1984.

[26] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[27] DaCapo benchmark suite. The dacapo benchmark suite - version beta051009. http://www-ali.cs.umass.edu/DaCapo/.

[28] Alan Demers, Mark Weiser, Barry Hayes, Hans-Juergen Boehm andDaniel G. Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 261–269, San Francisco, CA, January 1990. ACM Press.

[29] David Detlefs, editor. *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, Berlin, June 2002. ACM Press.

[30] John DeTreville. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, DEC Systems Research Center, Palo Alto, CA, August 1990.

[31] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[32] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.

[33] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, Portland, OR, January 1994. ACM Press.

[34] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 113–123. ACM Press, January 1993.

[35] Tamar Domani, Elliot Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In PLDI [83].

[36] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Elliot E. Salant, Katherine Barabash, Itai Lahan, Erez Petrank, Igor Yanover, and Yossi Levanoni. Implementing an on-the-fly garbage collector for Java. In Hosking [49].

[37] John R. Ellis, Kai Li, and Andrew W. Appel. Real-time concurrent collection on stock multiprocessors. Technical Report DEC–SRC–TR–25, DEC Systems Research Center, Palo Alto, CA, February 1988.

[38] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of High Performance Computing and Networking (SC'97)*, 1997.

[39] Christine Flood, Dave Detlefs, Nir Shavit, and Catherine Zhang. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, CA, April 2001.

[40] Shinichi Furusou, Satoshi Matsuoka, and Akinori Yonezawa. Parallel conservative garbage collection with fast allocation. In Paul R. Wilson and Barry Hayes, editors, *OOPSLA/ECOOP '91 Workshop on Garbage Collection in Object-Oriented Systems, Addendum to OOPSLA'91 Proceedings*, October 1991.

[41] David Gay and Alex Aiken. Memory management with explicit regions. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 313–323, Montreal, June 1998. ACM Press.

[42] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *International Conference on Compiler Construction (CC'2000)*, volume 1781 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[43] Ovidiu Gheorghioiu, Alexandru Salcianu, and Martin Rinard. Interprocedural compatibility analysis for static object preallocation. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, New Orleans, LA, January 2003. ACM Press.

[44] David Gries. An exercise in proving parallel programs correct. *Communications of the ACM*, 20(12):921–930, December 1977.

[45] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In PLDI [85], pages 141–152.

[46] Matthew Hertz, Steve M. Blackburn, K. S. McKinley, J. Eliot B. Moss, and Darko Stefanovic. Error-free garbage collection traces: How to cheat and not get caught. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems*, Marina Del Rey, CA, June 2002.

[47] Martin Hirzel, Amer Diwan, and Matthew Hertz. Connectivity-based garbage collection. In OOPSLA [74].

[48] Martin Hirzel, Johannes Henkel, Amer Diwan, and Michael Hind. Understanding the connectivity of heap objects. In Detlefs [29], pages 36–49.

[49] Tony Hosking, editor. *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*, Minneapolis, MN, October 2000. ACM Press.

[50] Paul R. Hudak and R. M. Keller. Garbage collection and task deletion in distributed applicative processing systems. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 168–178, Pittsburgh, PA, August 1982. ACM Press.

[51] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, University of Massachusetts, USA, 16–18 September 1992. Springer-Verlag.

[52] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying GC without stopping the world. In *Joint ACM Java Grande — ISCOPE 2001 Conference*, Stanford University, CA, June 2001.

[53] Lorenz Huelsbergen and Phil Winterbottom. Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. In Jones [54], pages 166–175.

[54] Richard Jones, editor. *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, Vancouver, October 1998. ACM Press.

[55] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[56] Haim Kermany and Erez Petrank. The compressor: Concurrent, incremental, and parallel compaction. In *Proceedings of SIGPLAN 2006 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Ottawa, Canada, June 2006. ACM Press.

[57] Elliot K. Kolodner and Erez Petrank. Parallel copying garbage collection using delayed allocation. Technical Report 88.384, IBM Haifa Research Lab., November 1999.

[58] Elliot K. Kolodner and Erez Petrank. Parallel copying garbage collection using delayed allocation. *Parallel Processing Letters*, 14(2), June 2004.

[59] H. T. Kung and S. W. Song. An efficient parallel garbage collection system and its correctness proof. In *IEEE Symposium on Foundations of Computer Science*, pages 120–131. IEEE Press, 1977.

[60] Leslie Lamport. Garbage collection with multiple processes: an exercise in parallelism. In *Proceedings of the 1976 International Conference on Parallel Processing*, pages 50–54, 1976.

[61] Yossi Levanoni and Erez Petrank. A scalable reference counting garbage collector. Technical Report CS–0967, Technion — Israel Institute of Technology, Haifa, Israel, November 1999.

[62] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. In *OOPSLA'01 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 36(10) of *ACM SIGPLAN Notices*, Tampa, FL, October 2001. ACM Press.

[63] Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for java. *ACM Transactions on Programming Languages and Systems*, 28(1), January 2006.

[64] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[65] Rafael D. Lins. Cyclic reference counting with lazy mark-scan. *Information Processing Letters*, 44(4):215–220, 1992. Also Computing Laboratory Technical Report 75, University of Kent, July 1990.

[66] Rafael D. Lins. An efficient algorithm for cyclic reference counting. *Information Processing Letters*, 83:145–150, 2002.

[67] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, September 1996. SIGLAN Notices 31(9).

[68] A. D. Martinez, R. Wachenchauzer, and Rafael D. Lins. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34:31–35, 1990.

[69] J. Harold McBeth. On the reference counter method. *Communications of the ACM*, 6(9):575, September 1963.

[70] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[71] David A. Moon. Garbage collection in a large LISP system. In Steele [93], pages 235–245.

[72] Kelvin D. Nilsen, Simanta Mitra, and Steven J. Lee. Method for efficient soft real-time execution of portable byte code computer programs. http://www.patentstorm.us/patents/6081665.html, 2000.

[73] *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, Denver, CO, October 1999. ACM Press.

[74] *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.

[75] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent GC for servers. In PLDI [85], pages 129–140.

[76] James W. O'Toole and Scott M. Nettles. Concurrent replicating garbage collection. Technical Report MIT–LCS–TR–570 and CMU–CS–93–138, MIT and CMU, 1993. Also LFP94 and OOPSLA93 Workshop on Memory Management and Garbage Collection.

[77] PAPI. The Performance API. http://icl.cs.utk.edu/papi/overview/.

[78] Young G. Park and Benjamin Goldberg. Escape analysis on lists. *ACM SIGPLAN Notices*, 27(7):116–127, June 1992.

[79] Harel Paz, David F. Bacon, Elliot K. Kolodner, Erez Petrank, and V. T. Rajan. Efficient on-the-fly cycle collection. Technical Report CS-2003-10, Technion, Israel Institute of Technology, November 2003. http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-info.cgi?2003/CS/CS-2003-10.

[80] Harel Paz, Erez Petrank, David F. Bacon, V.T. Rajan, and Elliot K. Kolodner. An efficient on-the-fly cycle collection. In CC [21].

[81] Harel Paz, Erez Petrank, and Stephen M. Blackburn. Age-oriented garbage collection. In CC [21].

[82] Manoj Plakal and Charles N. Fischer. Concurrent garbage collection using program slices on multithreaded processors. In Hosking [49].

[83] *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Vancouver, June 2000. ACM Press.

[84] *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Snowbird, Utah, June 2001. ACM Press.

[85] *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Berlin, June 2002. ACM Press.

[86] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In Hosking [49].

[87] Feng Qian and Laurie Hendren. An adaptive, region-based allocator for Java. In Detlefs [29], pages 127–138. Sable Technical Report 2002–1 provides a longer version.

[88] Ravenbrook. The memory management glossary. http://www.memorymanagement.org/glossary/.

[89] Ran Shaham, Elliot Kolodner, and Mooly Sagiv. Heap profiling for space-efficient Java. In PLDI [84].

[90] SPEC Benchmarks. Standard Performance Evaluation Corporation. http://www.spec.org/, 1998,2000.

[91] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[92] Guy L. Steele. Corrigendum: Multiprocessing compactifying garbage collection. *Communications of the ACM*, 19(6):354, June 1976.

[93] Guy L. Steele, editor. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, TX, August 1984. ACM Press.

[94] Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. Age-based garbage collection. In OOPSLA [73], pages 370–381.

[95] Bjarne Strousrup. *The C++ Programming Language, Third Edition.* Addison-Wesley, 2000.

[96] Mads Tofte. A brief introduction to Regions. In Jones [54], pages 186–195.

[97] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

[98] J. van Groningen. Faster garbage collection using prefetching. In C.Grelck and F. Huch, editors, *Proceedings of Sixteenth International WOrkshop on Implementation and Application of Functional Languages (IFL'04)*, pages 142–152, Lübeck, Germany, 2004.

[99] Steven P. VanderWiel and David J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.

[100] J. Weizenbaum. Symmetric list processor. *Communications of the ACM*, 6(9):524–544, September 1963.

[101] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.

[102] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, September 1995. Springer-Verlag.

[103] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Software and Systems*, 11(3):181–198, 1990.