Thèse

présentée à

L'Université Pierre & Marie Curie - Paris VI

pour obtenir le titre de

Docteur

Spécialité

Informatique

par

Paulo Jorge Pires Ferreira

Sujet de la thèse:

Larchant: ramasse-miettes dans une mémoire partagée répartie avec persistance par atteignabilité

Soutenue, le 10 Mai 1996, devant le jury composé de :

MM.	Jean-Jacques Levy	Président
	Christian Queinnec	Rapporteurs
	Michel Raynal	
	Claude Girault	Examinateurs
	José Marques	
	Marc Shapiro	
	Willy Zwaenepoel	

A Yashmin.

Remerciements

Je remercie Claude Girault, Professeur à l'Université Pierre et Marie Curie, responsable de l'enseignement de troisième cycle en informatique, pour m'avoir accepté comme étudiant de doctorat, et pour avoir participé à ce jury.

Je remercie Jean-Jacques Levy, Professeur à l'École Polytechnique et Directeur de Recherche à l'INRIA, pour avoir accepté de présider le jury de cette thèse.

Je remercie Christian Queinnec et Michel Raynal, respectivement Professeurs à l'École Polytechnique et à l'Université de Rennes I, pour l'intérêt qu'ils ont manifesté pour mon travail en acceptant d'être les rapporteurs de cette thèse.

Je remercie José Marques et Willy Zwaenepoel, respectivement Professeurs à l'Université Technique de Lisbonne et à *Rice University*, pour l'intérêt qu'ils ont porté à mon travail en acceptant de participer à ce jury.

Je remercie très chaleureusement Marc Shapiro, Responsable Scientifique du Projet SOR, qui m'a encadré, pour son exigence, ses conseils et sa rigueur. Ses commentaires et critiques ont directement influencé mes travaux. Je lui suis très reconnaissant d'avoir stimulé et encouragé ce travail.

Je remercie tous les membres du Projet SOR à l'INRIA: Yann pour son travail sur le banc d'essais OO7, Mokrane pour avoir trouvé et corrigé des erreurs dans Larchant, Mesaac pour sa disponibilité et ses conseils avisés, Hervé pour m'avoir aidé à bien écrire en Français, David pour nos discussions sur les ramasse-miettes, George pour son expertise de LaTeX et d'Emacs, Povl Koch pour nos discussions sur les systèmes basés sur un espace d'adressage plat, Ian et Aline pour leur aide concernant respectivement l'Anglais et le Français, Julien pour son aide quand je suis arrivé au Projet SOR, Guillaume pour son expertise WWW, et Nelly pour sa disponibilité.

Tout au long de mon séjour à l'INRIA, j'ai profité des discussions avec des personnes extérieures au Projet SOR. Je remercie: Laurent Amsaleg pour nos discussions sur les ramasse-miettes, Hank Levy pour nos discussions concernant la mémoire partagée répartie, Marcin et Marie-Hélène pour m'avoir aidé avec les automates, Fabio pour m'avoir montré la puissance des *state-charts*.

Je remercie la JNICT (programmmes *Ciência* et *PRAXIS XXI*) pour la bourse qui m'a été attribuée et qui m'a permis de mener ma recherche, et l'INESC pour m'a avoir encouragé à faire mon doctorat hors du Portugal.

Les conditions de travail offertes à l'INRIA sont exceptionnelles. Les moyens mis à ma disposition durant cette thèse m'ont permis de mener librement mon

travail.

Je remercie toute ma famille pour son soutien extraordinaire malgré la distance qui nous sépare. En particulier, je remercie mes parents qui m'ont toujours encouragé et montré le plaisir de la recherche.

Finalement, je remercie très spécialement mon épouse pour son amour, son encouragement, sa patience et sa compréhension.

Paulo Ferreira

Acknowledgments

I thank Claude Girault, Professor at *Université Pierre et Marie Curie*, for having accepted me as a Ph.D. student and for being a member of the jury.

I thank Jean-Jacques Levy, Professor at *École Polytechnique* and Director of Research at INRIA, for being the President of the jury.

I thank Christian Queinnec and Michel Raynal, Professors at *École Polytechnique* and at *Université de Rennes I* respectively, for having accepted to be the *rapporteurs* of this thesis.

I thank Professors José Marques from the Technical University of Lisbon, and Willy Zwaenepoel from Rice University, for having accepted to be members of the jury.

I warmly thank Marc Shapiro, leader of Project SOR at INRIA and my research advisor, for his care, exigence, and guidance. His comments and criticism helped me to improve my ideas and the description of my work.

I thank all the elements of the Project SOR at INRIA: Yann for his work on the OO7 benchmark, Mokrane for his help to discover and correct some bugs of Larchant, Mesaac for his constant availability and criticism, Hervé for his valuable help with French writing, David for our discussions concerning garbage collection, George for his expertise with LaTeX and Emacs, Povl Koch for our discussions concerning single address space systems, Ian and Aline were of considerable value concerning English and French writing respectively, Julien for his helping hand when I arrived at INRIA, Guillaume for his aid with the WWW, and Nelly for her valuable logistic support.

I have also benefited from discussions with people outside Project SOR. I thank them all: Laurent Amsaleg for the debates we had concerning garbage collection, Hank Levy for our discussions concerning distributed shared memory, Marcin and Marie-Hélène for helping me with state-machines, and Fabio for having showed me the usefulness of state-charts.

I'm grateful to JNICT (programs *Ciência* and *PRAXIS XXI*) that provided the scholarship which allowed me to undertake this research, and INESC for having encouraged me to pursuit a Ph.D. program outside Portugal.

I'm grateful to INRIA that provided me a wonderful environment for doing my work. The resources available allowed me to develop my research with great freedom and efficiency.

I thank all my family for the extraordinary support they always gave me in spite of the distance separating us. In particular, I thank my parents for helping me to discover, since I was a young boy, the pleasures of learning and researching; they provided me a home where such pursuits were encouraged.

Finally, thanks are not adequate for the love, patience, encouragement, and understanding that my wife has given to me.

Paulo Ferreira

Avant-propos

Cette thèse est le résultat d'un effort de recherche effectué à l'INRIA. Elle a deux parties: la première partie est un long résumé en Français de la thèse; la deuxième partie est la version complète de la thèse en Anglais.

Quelques aspects de cette thèse sont presentés dans les articles suivants (par ordre chronologique):

Paulo Ferreira and Marc Shapiro. Distribution and Persistence in Multiple and Heterogeneous Address Spaces. *Proceedings of the International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, Asheville NC (USA), December 1993.

Paulo Ferreira and Marc Shapiro. Garbage Collection of Persistent Objects in Distributed Shared Memory. *Proceedings of the International Workshop on Persistent Object Systems (POS)*, Tarascon (France), September 1994.

Paulo Ferreira and Marc Shapiro. Garbage Collection and DSM Consistency. *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey CA (USA), November 1994.

Paulo Ferreira and Marc Shapiro. Garbage Collection in the Larchant Persistent Distributed Shared Store. *Proceedings of the Workshop on Future Trends in Distributed Computing Systems (FTDCS)*, Cheju Island (Republic of Korea), August 1995.

Marc Shapiro and Paulo Ferreira. Larchant-RDOSS: a Distributed Shared Persistent Memory and its Garbage Collector. *Proceedings of the International Workshop on Distributed Algorithms (WDAG)*, Le Mont St. Michel (France), September 1995.

Paulo Ferreira and Marc Shapiro. Larchant: Persistence by Reachability in Distributed Shared Memory through Garbage Collection. *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, Hong Kong, May 1996.

Ces articles peuvent être obtenus via World Wide Web à l'address http://www-sor.inria.fr/SOR/projects/larchant.

Foreword

This thesis documents the result of a research effort done at INRIA. It has two parts: the first is a long abstract, in French, of the whole thesis; the second part is the thesis itself, in English.

Some results presented in this thesis appear in the following articles (chronological order):

Paulo Ferreira and Marc Shapiro. Distribution and Persistence in Multiple and Heterogeneous Address Spaces. In *Proceedings of the International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, Asheville NC (USA), December 1993.

Paulo Ferreira and Marc Shapiro. Garbage Collection of Persistent Objects in Distributed Shared Memory. In *Proceedings of the International Workshop on Persistent Object Systems (POS)*, Tarascon (France), September 1994.

Paulo Ferreira and Marc Shapiro. Garbage Collection and DSM Consistency. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey CA (USA), November 1994.

Paulo Ferreira and Marc Shapiro. Garbage Collection in the Larchant Persistent Distributed Shared Store. In *Proceedings of the Workshop on Future Trends in Distributed Computing Systems (FTDCS)*, Cheju Island (Republic of Korea), August 1995.

Marc Shapiro and Paulo Ferreira. Larchant-RDOSS: a Distributed Shared Persistent Memory and its Garbage Collector. In *Proceedings of the International Workshop on Distributed Algorithms (WDAG)*, Le Mont St. Michel (France), September 1995.

Paulo Ferreira and Marc Shapiro. Larchant: Persistence by Reachability in Distributed Shared Memory through Garbage Collection. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, Hong Kong, May 1996.

These articles can be freely obtained on the World Wide Web at the URL http://www-sor.inria.fr/SOR/projects/larchant.

Résumé

Le modèle de Larchant est celui d'un *Espace d'Adressage Partagé* (comprenant tous les sites du réseau ainsi que la mémoire secondaire) basé sur la *Persistance par Atteignabilité*. Pour donner l'illusion d'un espace d'adressage partagé par tous les sites du réseau, Larchant met en œuvre un mécanisme de *Mémoire Partagée Répartie*. La persistance par atteignabilité est apportée par un *Glaneur de Cellules* (GC, ou encore ramasse-miettes) qui parcourt le graphe d'objets à partir de la racine de persistance. Les objets non atteignables sont appelés *miettes*. Le GC nettoie la mémoire en recyclant l'espace occupé par les miettes.

Le glanage dans les systèmes répartis à grande échelle a été considéré jusqu'à peu comme infaisable. Les principaux problèmes rencontrés sont dus à l'incohérence des données et à des considérations portant sur l'échelle et les performances. Cette thèse présente les solutions apportées par Larchant pour résoudre ces problèmes.

Le GC de Larchant combine traçage et listage de références. Le GC fait du traçage sur les partitions en mémoire. Le GC fait du listage sur les références qui, si elles étaient parcourues, impliqueraient des entrées-sorties. Le ramassage de miettes est totalement dissocié de la cohérence, *i.e.*, le glanage est effectué même s'il y a des répliques incohérents sur le site où le GC s'exécute. Le glaneur fonctionne de façon concurrente et asynchrone par rapport aux applications. Les frontières de listage des références changent dynamiquement et indépendamment sur chaque site de manière à nettoyer les cycles de miettes.

L'algorithme du GC à été prouvé correct (innocuité et vivacité) par induction. Les résultats de performance du GC de Larchant montrent que nos objectifs d'orthogonalité par rapport à l'incohérence et à l'échelle ont été atteints.

Mots-clés: systèmes répartis, systèmes à objets, persistance par atteignabilité, mémoire partagée répartie, glaneur de cellules réparti.

Abstract

The model of Larchant is that of a *Shared Address Space* (spanning every site in a network including secondary storage) with *Persistence By Reachability*. To provide the illusion of a shared address space across the network, despite the fact that site memories are disjoint, Larchant implements a *distributed shared memory* mechanism. Reachability is accessed by tracing the pointer graph, starting from the persistent root, and reclaiming unreachable objects. This is the task of *Garbage Collection* (GC).

GC was until recently thought to be intractable in a large-scale system, due to problems of scale, incoherence, asynchrony, and performance. This thesis presents the solutions that Larchant proposes to these problems.

The GC algorithm in Larchant combines tracing and reference-listing. It traces whenever economically feasible, *i.e.*, as long as the memory subset being collected remains local to a site, and counts references that would cost I/O traffic to trace. GC is orthogonal to coherence, *i.e.*, makes progress even if only incoherent replicas are locally available. The garbage collector runs concurrently and asynchronously to applications. The reference-listing boundary changes dynamically and seamlessly, and independently at each site, in order to collect cycles of unreachable objects.

We prove formally that our GC algorithm is correct, *i.e.*, it is safe and live. The performance results from our Larchant prototype show that our design goals (scalability, coherence orthogonality, and good performance) are fulfilled.

Keywords: distributed systems, object-oriented systems, persistence by reachability, distributed shared memory, distributed garbage collection.

Table des Matières / Contents

Index des Figures / List of Figures

Index des Tableaux / List of Tables

Partie/Part I

Larchant: ramasse-miettes dans une mémoire partagée répartie avec persistance par atteignabilité

(Long résumé/Long abstract)

Chapitre 1

Introduction

L'objectif global de *Larchant* est d'offrir un *stockage persistant réparti* (PDS, *Persistent Distributed Store*) qui facilite le partage des données. Larchant garantit que l'accès aux données à partir de différents sites et/ou à des moments différents reste simple et efficace. De plus, la persistance, les entrées/sorties (E/S), la gestion de la mémoire et la répartition sont transparents et automatiques. Avec Larchant, les programmeurs peuvent se concentrer sur les problèmes de leurs applications sans se laisser perturber par les bogues mémoire, le cache, la cohérence, les accès distants, les E/S, etc. Les applications type destinées à Larchant sont les systèmes de CAO-FAO, le multimédia, les bases de données financières, etc.

Le modèle de Larchant est celui d'un *espace d'adressage partagé* [29, 93, 105] (s'étendant sur chaque site du réseau, incluant les zones de stockage secondaire) avec la *persistance par atteignabilité* (PBR, *Persistance By Reachability*) [8]. Ce modèle est très attirant par la simplicité qu'il apporte au programmeur.

Pour fournir l'illusion d'un espace d'adressage partagé recouvrant le réseau, bien que les mémoires des sites soient disjointes, Larchant met en œuvre un mécanisme de *mémoire partagée répartie* (DSM, *Distributed Shared Memory*) [28, 57, 80].

L'accès à une donnée nécessite de trouver sa référence par navigation en partant d'une *racine persistante* bien connue (*par ex.*, un serveur de noms). Dans un système utilisant la PBR [34, 48, 84, 112], un objet atteignable depuis la racine persistante est lui-même *persistant*, il doit ainsi être stocké. Un objet atteignable par transitivité à partir de la racine persistante est aussi persistant. Les objets non atteignables ne sont plus nécessaires, ils peuvent donc être ramassés (et la mémoire compactée). De tels objets sont appelés *miettes*.

L'atteignabilité est confirmée en construisant le graphe des pointeurs, en partant de la racine persistante, et en ramassant les objets non atteignables. Ce processus peut être effectué soit par une gestion manuelle de la mémoire, soit automatiquement par un *Glaneur de Cellules* (GC, ou encore ramasse-miettes) [127].[1]

[1] *Garbage collector* en Anglais.

La gestion manuelle de la mémoire est source d'erreurs (*par ex.*, les pointeurs invalides et les fuites de mémoire) qui conduit fréquemment à la violation de la condition fondamentale d'*intégrité référentielle* : suivre un pointeur doit toujours être valide, *i.e.*, si un objet persistant `x` pointe vers un objet `y`, alors `y` doit aussi être persistant.

Le GC dans les systèmes répartis à grande échelle a été considéré jusqu'à peu comme infaisable. Les principaux problèmes rencontrés sont dus à l'incohérence des données et à des considérations portant sur l'échelle et les performances. Cette thèse présente les solutions apportées par Larchant pour resoudre ces problèmes.

L'algorithme du GC dans Larchant associe le traçage et le comptage de références. Il trace tant que c'est économiquement faisable, *i.e.*, tant que le sous-ensemble mémoire reste local à un site, et il compte les références qui demanderaient un trafic d'E/S coûteux pour un traçage. Les limites du comptage de références changent dynamiquement et de façon indépendante sur chaque site, de façon à collecter (ou ramasser) les cycles d'objets non atteignables. Le plus important est que les sous-ensembles mémoire répliqués sont collectés indépendamment et sans interférence avec les besoins en cohérence des applications.

Ce travail est le premier à proposer un algorithme de GC pour une PDS qui soit *(i)* extensible en terme d'échelle, *(ii)* orthogonal à la cohérence, *i.e.*, qu'il progresse même si seules des répliques incohérentes sont disponibles localement, *(iii)* totalement asynchrone par rapport aux applications, et *(iv)* collecte les cycles de miettes en changeant dynamiquement les frontières du mécanisme réparti de comptage de références.

Cette thèse est organisé de la façon suivante. Le prochain chapitre dresse un panorama des techniques traditionnelles de GC. Le Chapitre 3 présente notre modèle des programmes applicatifs, les protocoles de cohérence, et le modèle du GC. Le Chapitre 4 décrit les algorithmes de traçage et de comptage de références dans Larchant. Dans le Chapitre 5 nous présentons brièvement l'implémentation du GC dans Larchant. Le Chapitre 6 expose une étude des performances de nos algorithmes de GC. Le Chapitre 7 conclut la thèse par un résumé des idées les plus importantes.

Chapitre 2

Panorama des algorithmes de GC existants

Les techniques fondamentales de GC sont traditionnellement regroupées en deux grandes familles : les approches fondées sur le comptage de références et celles s'appuyant sur un traçage du graphe d'objets.

Dans les GC fondés sur le comptage de références (*reference counting* en Anglais), chaque objet compte le nombre de références qui le désignent [31, 33]. Ce compte est traditionnellement représenté par un entier stocké dans l'en-tête de chaque objet. Le compteur d'un objet est incrémenté à chaque fois que sa référence est dupliquée. Inversement, lorsqu'une référence vers un objet est détruite, le compteur associé est décrémenté. L'espace mémoire occupé par un objet est ramassé dès que son compteur de références passe à zéro.

Les GC fondés sur un traçage effectuent un parcours du graphe d'objets depuis la racine de persistance et traversent les objets en suivant successivement leurs références. Les deux techniques élémentaires fondées sur le traçage sont : le marquage-balayage (*mark-and-sweep* en Anglais) [87] et la copie des objets (*copy* en Anglais) [50, 88].

L'algorithme de marquage-balayage fonctionne de la manière suivante. Partant de la racine de persistance, le GC distingue les objets atteignables des miettes en marquant les objets visités lors de sa traversée du graphe. Ensuite, le GC balaye la totalité de la mémoire pour trouver puis ramasser tous les objets non marqués.

L'algorithme de GC par copie fonctionne de la manière suivante. La mémoire est divisée en deux demi-espaces. Durant l'exécution normale des programmes, seul le demi-espace appelé espace d'origine (*from-space* en Anglais) est utilisé alors que l'autre demi-espace, appelé espace destination (*to-space* en Anglais), est conservé vierge. Le GC copie les objets atteignables via la racine de persistance, depuis l'espace d'origine vers l'espace destination. Lorsque la copie s'achève, le rôle des deux demi-espaces est échangé atomiquement. L'espace destination devient l'espace d'origine, et vice versa.

Il existe une littérature abondante dans le domaine du GC que ce soit pour les

multiprocesseurs [7, 20, 43, 63, 92], ou pour des systèmes client-serveur répartis (voir l'étude de Plainfossé[97]). En revanche, peu de travaux ont été réalisés sur un GC dans un réseau faiblement couplé avec une DSM faiblement cohérente.

La différence fondamentale entre un GC dans Larchant et un GC dans un multiprocesseur est l'échelle et le surcoût de synchronisation : si nous appliquons un algorithme de GC conçu pour des multiprocesseurs (*par ex.*, Appel[7]) à notre cas, le surcoût serait inacceptable du fait des coûts de communication et de synchronisation. Ces coûts sont dus au fait que les algorithmes de GC multiprocesseurs actuels supposent implicitement l'existence d'objets cohérents.

La plupart des travaux précédents sur les GC répartis [85, 97] considèrent que les processus communiquent par messages (sans mémoire partagée), en utilisant un mélange de traçage et de comptage. Chaque processus trace ses pointeurs internes ; les références hors des frontières du processus sont comptées lorsqu'elles sont envoyées dans des messages. Cependant, aucun des ces travaux n'accepte un mécanisme de mémoire répartie partagée dans laquelle des applications s'exécutant sur différents sites peuvent accéder à de multiples répliques du même objet de façon concurrente, comme Larchant le permet.

Finalement, les travaux précédents sur le ramassage de miettes dans un DSM sont rares et ne résolvent pas nos problèmes puisqu'ils supposent des objets cohérents [72, 78].

Chapitre 3

Modèle de Larchant

Nous utilisons un vocabulaire standard de la littérature sur les ramasse-miettes [40]. Le *mutateur* est le programme applicatif qui modifie dynamiquement le graphe des pointeurs ; il crée des objets, utilise des pointeurs, et affecte des pointeurs. Un effet de bord de l'affectation de pointeurs est que certains objets deviennent non atteignables.

Dans un système réparti, le mutateur est en fait composé de multiples threads indépendantes s'exécutant sur différents sites ; par extension, nous appelons chacune de ces threads un mutateur.

Le *collecteur* est le composant système qui identifie et ramasse les miettes créées par le mutateur. Notre collecteur est composé d'un nombre de threads s'exécutant sur différents sites ; nous appelons chacune d'elles un collecteur.

Notre algorithme de GC est basé sur un modèle très simplifié. Ainsi, plusieurs opérations traditionnellement associées à la gestion de la cohérence ne sont pas présentes dans notre modèle ; certaines parce qu'elles ne sont pas liées au GC (par exemple, la lecture et l'écriture sans utiliser de pointeurs) et d'autres parce que notre algorithme leur est indépendant (par exemple, les verrous de la DSM). En particulier, l'algorithme de GC tolère des écritures arbitraires et des données incohérentes. La simplicité du modèle garantit que l'algorithme de GC s'applique à une large variété de systèmes de stockage partagés.

3.1 Modèle de mémoire

La mémoire est décomposée en deux niveaux de granularité. *(i)* L'*objet* est l'unité d'allocation, de désallocation et d'identification. Il est aussi l'unité de cohérence et de propagation des mises à jour (*i.e.*, la dissémination de la réplique la plus à jour d'un objet). Un objet peut contenir plusieurs pointeurs. Soit un pointeur est nul soit il pointe vers un objet. *(ii)* Un *bunch* est l'unité de cache et de rassemblement. Il contient un nombre quelconque d'objets.

Désormais, les objets seront notés x, y, z, etc. L'adresse de l'objet x sera notée @x. Les bunches seront notés en majuscules B, C, etc. Puisque toute structure

peut être répliquée (être en cache) sur de multiples sites, nous distinguons les répliques à l'aide d'un indice de site, *par ex.,* x_i, x_j, pour les répliques de x observées respectivement sur les sites i et j.

Une variable pointeur ptr dans un objet x sera notée x.ptr. Afin de simplifier la notation nous allons supposer que les objets n'ont qu'un seul pointeur et identifier x.ptr par x.

3.2 Modèle du mutateur

Un mutateur s'exécutant sur le site i voit l'objet x au travers de la réplique présente dans le cache du site, x_i. Tout mutateur peut écrire ou lire un objet x vers lequel il possède un pointeur dans son cache.

Pour des questions de GC, l'opération pertinente exécutée par les mutateurs est l'*affectation* d'une variable pointeur au sein d'un objet. Une opération d'affectation effectuée sur le site i, réalisant une lecture sur l'objet y et une écriture sur x, est notée $<x := y>_i$. Cette opération est atomique en local uniquement, *i.e.*, la lecture et l'écriture des répliques locales sont indivisibles. Il n'est pas nécessaire que cette opération soit atomique pour les répliques distantes.

Notez que l'opération d'affectation peut conduire à la destruction d'un pointeur et à la création d'un nouveau, ce qui est effectué de façon imprévisible par les mutateurs, modifiant ainsi l'atteignabilité des objets.

3.3 Modèle de cohérence

Cette section identifie les opérations de cohérence ayant un rapport avec le GC. De façon à accepter différents modèles de cohérence, nous n'imposons pas le moment auquel de telles opérations ont lieu. (En pratique, les opérations de cohérence sont provoquées par l'activité du mutateur.)

Notre supposition minime est que, à tout moment, chaque objet ne possède qu'un seul *propriétaire*. Le propriétaire d'un objet est le seul site autorisé à disséminer sa valeur aux autres sites. Cette dissémination est effectuée par un message de *propagation*.

La propagation de x à partir de son site propriétaire i au site j fait que x_j devient égal au x_i propagé. Notez que cette opération n'exclut pas les écritures concurrentes (ou les lectures) effectuées sur les autres répliques de x.

Le propriétaire d'un objet peut changer. Ce changement s'effectue par un message de *propriété*: le transfert de propriété de x du site i vers le site j fait que le site j est le nouveau propriétaire de x.

D'autre opérations de cohérence peuvent être construites comme des cas particuliers des précédentes. Par exemple, l'invalidation du cache se fait en recevant un message de propagation contenant la valeur nulle.

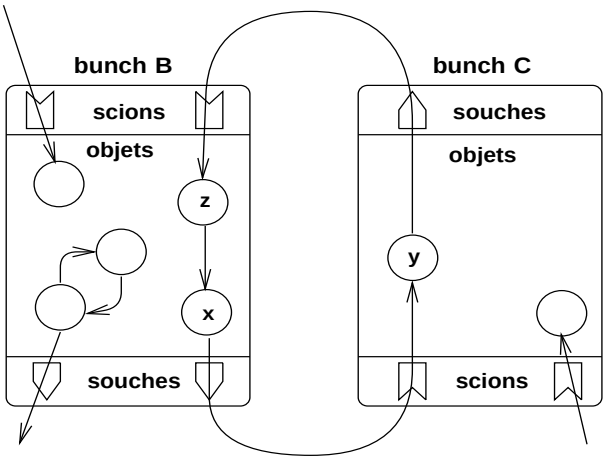


Figure 3.1: *Souches et scions.*

3.4 Modèle du GC

Un bunch conserve la trace des pointeurs inter-bunch, de façon à permettre son ramassage indépendant. Un pointeur sortant est décrit par une *souche* et un pointeur entrant par un *scion*[1] (voir la Figure 3.1). Une souche identifie le scion correspondant et vice versa. Chaque réplique d'un bunch possède son propre ensemble de souches et de scions.

Nous donnons ici, sans le justifier, la liste des opérations liées à notre algorithme de GC (pour les techniques de marquage-balayage et de copie [127]) :

- create.scion(Bx, Cy) : créer un scion dans C qui décrit le pointeur inter-bunch de x dans B vers y dans C.
- delete.scion(Bx, Cy) : détruire le scion dans C qui décrit le pointeur inter-bunch de x dans B vers y dans C.
- $\mathsf{scan}_i(x)$: parcourir x_i afin de trouver les objets qu'il pointe directement.
- move(y, @y') : déplacer l'objet y à sa nouvelle adresse @y'.
- $<\mathsf{patch}(x, @y')>_i$: corriger x_i avec la nouvelle adresse @y' de y.

Le déclenchement et l'innocuité de ces opérations seront étudiés dans le Chapitre 4.

Quelques temps après qu'un mutateur ait créé un nouveau pointeur inter-bunch, le collecteur alloue une souche dans le bunch source et *crée* un scion dans la cible. De façon similaire, après la disparition d'un pointeur inter-bunch, le

[1] Les structures similaires trouvées dans le système à passage de messages des chaînes de SSP (*Stub-Scion Pair*) [108], ou chaîne de paires souches-scions, ont inspiré les noms de souche (*stub*) et de scion. Contrairement aux chaînes de SSP, les souches et les scions de Larchant ne sont pas des indirections participant au processus du mutateur, mais simplement des structures de données auxiliaires décrivant les pointeurs inter-bunch.

collecteur peut finir par désallouer la souche correspondante et *détruire* le scion cible. Le nombre de pointeurs inter-bunch sur un objet est approximativement égal au nombre de scions pour cet objet.

La boucle principale d'un GC par traçage utilise l'opération de *parcours*. Pour une certaine réplique x_i, cette opération détermine les autres objets pointés par x_i. Un collecteur par copie *déplace* les objets (pour compacter le mémoire et réduire la fragmentation) et *corrige* les pointeurs avec les nouvelles adresses.

Notez que les opérations de parcours et de correction sont locales à un site, *i.e.*, elles sont appliquées à une réplique locale et elles ne nécessitent pas une opération de propagation en dépit des lectures et des écritures implicites sur l'objet concerné (ainsi, les répliques locales peuvent être incohérentes). D'un autre côté, l'opération de déplacement s'applique à chaque réplique d'un objet. Comme nous le décrivons dans le Chapitre 4, toutes les opérations du GC ne nécessitent pas des données cohérentes.

Dans la terminologie associée aux GCs, un *flip* désigne le moment pendant lequel le mutateur est interrompu ; pendant le flip, le collecteur (pour l'algorithme de marquage-balayage ou de copie) effectue certaines opérations ayant pour but de terminer la collecte. Par exemple, pendant le flip, un collecteur marquage-balayage reparcourt les objets qui, après avoir été parcourus, ont été modifiés du fait d'une activité concurrente du mutateur.

Enfin, un objet est *sale* après avoir été modifié par une opération d'affectation (du mutateur) ; il reste sale tant qu'il n'a pas été parcouru. Un bunch est sale s'il contient un objet sale.

Chapitre 4

GC dans Larchant

Dans l'idéal, un algorithme de GC devrait être *complet*, *i.e.*, devrait finir par collecter *toutes* les données non atteignables. Les seuls algorithmes que l'on peut prouver complets sont basés sur le traçage et utilisent une synchronisation globale. Ce principe n'est pas extensible, génère un grande quantité d'E/S et perturbe les applications. D'un autre côté, les algorithmes de comptage de références sont extensibles mais ils sont incomplets puisqu'ils ne collectent pas les cycles de miettes. Apparemment c'est sans espoir. Mais, comme nous allons le voir, il est possible d'éviter le traçage global et ses inconvénients.

Nous affirmons que la complétude parfaite n'est pas faisable dans un système de grande échelle. Nous proposons donc une solution approximative qui ne peut être prouvée comme complète, mais que nous croyons appropriée pour les situations réelles. Elle fonctionne en associant le traçage et le comptage de références.

Le GC dans Larchant s'approche d'un traçage global avec une série de traçages locaux sans synchronisation et partitionnés. Chaque bunch est collecté sur le site où il est mis en cache, avec un algorithme de traçage, indépendamment du reste de la mémoire. De plus, si un bunch est répliqué, chacune des ses répliques est aussi collectée de façon indépendante avec le même algorithme.

En considérant les scions d'un bunch comme des racines intérimaires, un bunch peut être collecté indépendamment des autres. La collecte d'une réplique d'un bunch (*collecteur intra-bunch*) s'effectue comme suit. Tout objet pointé directement à partir d'un scion est considéré comme atteignable et on recherche ses pointeurs. Si un objet atteignable pointe vers un autre objet du même bunch, le collecteur intra-bunch considère de façon transitive l'objet pointé atteignable. S'il pointe en dehors du bunch englobant, le collecteur alloue une souche. Ainsi, le résultat de la collecte d'un bunch est un ensemble d'objets atteignables et un nouvel ensemble de souches. Les objets non compris dans l'ensemble sont des miettes.

Lorsqu'une collecte intra-bunch est terminée, le *collecteur inter-bunch* (algorithme de comptage de références) compare le nouvel ensemble de souches avec l'ancien (résultant d'un précédent GC intra-bunch). Les souches qui n'existaient

pas précédemment indiquent qu'un nouveau pointeur inter-bunch sortant a été créé par le mutateur. Les souches qui ont disparu (*i.e.*, qui ne sont plus dans le nouvel ensemble de souches) indiquent un pointeur inter-bunch sortant qui n'existe plus.

Pour chaque nouvelle souche allouée (par le collecteur intra-bunch), le collecteur inter-bunch procède à la création du scion correspondant. Pour chaque souche qui n'appartient plus à l'ensemble, le collecteur inter-bunch demande la destruction du scion cible correspondant.

Ainsi, le collecteur inter-bunch détruit les scions, autorisant les futures collectes intra-bunch à ramasser les objets qui n'étaient atteignables que parce qu'ils étaient pointés par les scions détruits. (Comme nous le verrons dans le Chapitre 4, les opérations de création et de destruction sont asynchrones par rapport aux mutateurs.)

4.1 Traçage

La principale difficulté des algorithmes de collecte intra-bunch est de ramasser un bunch (concurremment avec les mutateurs) sans nécessiter d'opérations de cohérence, afin d'éviter de perturber les applications. Par exemple, le collecteur intra-bunch doit être capable de progresser sans imposer qu'une réplique d'un objet soit à jour pour être parcourue. Un raisonnement similaire s'applique aux opérations de déplacement et de correction.

Chaque site trace ses bunches (dans le cache local) indépendamment des autres sites, même si ses bunches peuvent être répliqués ailleurs. Cela pose les questions suivantes :

- Les collecteurs doivent-ils être synchronisés entre eux ?
- Les parcours demandent-ils des données cohérentes ?
- Si on utilise un algorithme de GC par copie :
  - Est-il nécessaire de synchroniser les collecteurs pour décider où déplacer un objet ?
  - Est-il nécessaire d'effectuer certaines opérations de cohérence avant (ou après) avoir déplacé un objet ou avoir corrigé ses pointeurs internes ?
  - Est-il nécessaire de synchroniser le flip ?

Une réponse "oui" à n'importe laquelle de ces questions aurait un impact sur l'extensibilité et l'efficacité. Dans le reste de cette section nous montrerons que, aussi surprenant que cela puisse paraître, les réponses sont toutes "non", sous les bonnes conditions. Cela a pour conséquence que le collecteur intra-bunch n'est pas en compétition avec les applications pour la cohérence des données, qu'il n'y pas de synchronisation entre les collecteurs et les mutateurs ou entre les

différents collecteurs, et que les messages du GC sont asynchrones et s'échangent en tâche de fond. Le prix à payer est un certain degré de conservation, et certains messages devant être délivrés en ordre causal [100].

4.1.1 Parcours

Le parcours d'une réplique incohérente ne prend bien évidemment pas en compte les écritures de pointeurs survenant sur un autre site. Ce n'est cependant pas un problème. En fait, le parcours d'une réplique non à jour résulte simplement en une décision plus conservative concernant l'atteignabilité des objets pointés.

D'un autre côté, le parcours des seules répliques les plus à jour d'un objet n'assure pas l'innocuité. Un objet z peut ne plus être référencé par la réplique la plus à jour d'un objet y mais être quand même atteignable à partir d'une réplique incohérente de y (y et z étant tous deux dans le même bunch). Ainsi, un objet ne peut être collecté qu'après être devenu non atteignable à partir de l'union de toutes les répliques de ses objets source. Ce que nous appelons la *règle de l'union*. Nous étudierons ce problème plus en détail dans la Section 4.2, quand nous décrirons le collecteur inter-bunch.

4.1.2 Déplacement et correction

Dans cette section, nous étudions les opérations de déplacement et de correction par rapport à la cohérence. Le premier problème est d'éviter que deux sites ne déplacent de façon concurrente leurs répliques locales du même objet vers deux endroits différents. Une solution évidente à ce problème serait que le site qui veut déplacer un objet demande la propriété de l'objet avant de le déplacer. Il est clair que cette solution n'est pas satisfaisante puisqu'elle interfère avec les besoins de cohérence des applications.

Une solution simple n'interférant pas avec les besoins de cohérence des applications serait la suivante : le site propriétaire de x décide de l'endroit où le déplacer ; les sites non propriétaires déplaceraient leurs répliques de x à la même adresse après réception d'un message de déplacement émis par le propriétaire. Un objet atteignable serait alors déplacé de la façon suivante. *(i)* Le site i, le propriétaire de x, envoie un message move(x, @x') aux sites possédant dans leur cache un pointeur vers x, lui-même compris. *(ii)* Un site j recevant ce message déplace sa réplique x_j à la nouvelle position et corrige les pointeurs en conséquence. Notez que les messages de déplacement sont délivrés en tâche de fond. Ainsi, ils ne perturbent pas les applications.

Le second problème est le suivant : le collecteur doit-il acquérir la propriété d'un objet afin de corriger un de ses pointeurs avec la nouvelle position de la cible ? La réponse est non, parce que cette opération n'est visible que sur le site où elle a lieu. Notez que, même avec des protocoles de cohérence qui demandent la propriété d'un objet dans lequel on souhaite écrire (*par ex.*, la *entry-consistency* [14]) l'opération de correction peut être effectuée sans détenir la propriété d'un objet.

Enfin, le collecteur d'une réplique d'un bunch peut réaliser le flip de façon indépendante (sans synchronisation) des collecteurs des autres répliques du même bunch. Le collecteur intra-bunch peut même procéder au flip avant d'avoir reçu tous les messages de déplacement concernant les objets atteignables qui ne sont pas détenus localement. (Le déplacement de tels objets peut être retardé jusqu'à la prochaine exécution du GC.)

4.1.3 Ramassage des cycles de miettes inter-bunch

L'algorithme de GC intra-bunch est complet par rapport au bunch collecté, *i.e.*, il ramasse toutes les miettes qui sont entièrement dans ce bunch. Cependant, il est incomplet par rapport aux autres bunches puisqu'il ne ramasse pas un cycle de miettes qui traverse les frontières du bunch (*par ex.*, le cycle inter-bunch de la Figure 3.1 : les objets x, y et z).

Le même algorithme que celui qui ramasse un seul bunch peut être utilisé pour ramasser n'importe quel groupe de bunches. La seule différence par rapport au ramassage d'un seul bunch réside dans le traçage d'un groupe : *(i)* les scions des pointeurs inter-bunch internes au groupe ne sont pas considérés comme des racines, et *(ii)* le traçage traverse les frontières des bunches internes au groupe. Cet algorithme ramasse un cycle inter-bunch non atteignable à partir de bunches extérieurs au groupe. Par exemple, dans la Figure 3.1, une collecte du groupe comprenant les bunches B et C ne considérerait pas les scions pointant vers y et z comme des membres de la racine. Le cycle de miettes inter-bunch constitué par les objets x, y et z serait alors ramassé. Ainsi, une collecte de groupe est complète par rapport au groupe ramassé.

La collecte de groupe signifie qu'un ensemble arbitraire de sous-ensembles de la mémoire peut être ramassé, sur un seul site, indépendamment du reste de la mémoire. Le choix du groupe à collecter est heuristique, et doit maximiser la quantité de miettes ramassées et minimiser le coût.

Pour former de tels groupes, Larchant utilise une heuristique basée sur la localité. Un groupe contient tous les bunches actuellement dans le cache du site. Cette heuristique évite les surcoûts d'E/S. Cependant, elle n'autorise pas le ramassage des cycles inter-bunch passant par des bunches qui résident en partie sur le disque. Ces miettes peuvent être ramassées par une heuristique de groupement plus agressive. Le surcoût en E/S doit être équilibré par le gain espéré. Nous souhaitons expérimenter la heuristique basée sur la localité avec un grand nombre d'applications et réaliser des études de simulation. Ensuite, si nécessaire, des heuristiques plus complexes seront le sujet de recherches futures.

4.2 Comptage de références

Nous observons tout d'abord que le parcours de la réplique du propriétaire d'un objet y seul n'assure pas l'innocuité : par exemple, un certain objet z pourrait être atteignable d'une réplique incohérente y_j et non atteignable à partir de la

réplique la plus à jour y_i. Par conséquent, une destruction pourra être lancée que lorsque l'objet cible z deviendra non atteignable à partir de *toutes* les répliques de l'objet source y.

Comme nous l'avons déjà mentionné dans la section 4.1.1, c'est ce que nous appelons la règle de l'union: la destruction n'assure pas l'innocuité qu'avec l'union des souches de toutes les répliques. Un site non propriétaire de l'objet y informe le propriétaire de y de l'existence des souches par un message d'*union*. Après qu'une souche (du fait d'un pointeur sortant à partir de y) ait disparu de tous les sites ayant dans leur cache une réplique de y, le propriétaire envoie une destruction qui concerne le scion correspondant.

Plusieurs protocoles de cohérence imposent que seul le propriétaire d'un objet y peut l'écrire (*par ex.*, la *entry-consistency* [14]). Dans ce cas, un objet non atteignable à partir d'une réplique non propriétaire y_j ne peut pas devenir atteignable sous la simple volonté de y_j, parce que cela signifie une écriture sur y sur le site j. Ainsi, l'ensemble des souches d'un site non propriétaire est monotone décroissant, et par conséquent les messages d'union peuvent être délivrés de façon asynchrone, dans l'ordre FIFO. En résumé, la règle de l'union peut être implémentée à moindre coût lorsque seul le propriétaire d'un objet a le droit d'y écrire.

4.2.1 Asynchronisme

Le collecteur inter-bunch doit tenir compte des affectations de pointeurs (réalisées par des mutateurs concurrents) parce qu'elles peuvent conduire à la création de nouveaux pointeurs inter-bunch. De tels pointeurs doivent être traqués afin d'allouer les souches correspondantes et créer le scion cible. Sinon, le collecteur intra-bunch pourrait ramasser un objet atteignable.

Une observation importante est que ces créations n'ont pas besoin d'être effectuées dès que le pointeur inter-bunch correspondant apparaît (résultant d'une opération d'affectation). Les créations peuvent donc se faire de manière asynchrone. Ceci présente un avantage de performance et de portabilité non négligeable puisque le mutateur n'est pas stoppé, cela peut éviter certaines tâches, et autoriser le regroupement des messages pour la création de scions.

Sans perdre en généralité, considérons l'exemple suivant. Des objets x, y et z sont alloués respectivement dans des bunches B, C et D. L'objet x est détenu par le site i, y par le site j et z par le site k. Supposons qu'initialement x_i, y_i et y_j soient atteignables, que les deux répliques de y soient égales (y a été propagée du site j vers le site i), et que x_i soit nul. La situation initiale est donnée dans la Figure 4.1.

Maintenant, considérons la séquence suivante (voir Figure 4.2). Le mutateur exécute $<x := y>_i$, créant ainsi un pointeur inter-bunch de x vers z. Ensuite, les deux répliques de y sont modifiées de façon à ce qu'elles ne pointent plus vers z (*par ex.*, $<y := 0>_j$ et propagation de y de j vers i). Puis, les bunches B et C sont ramassés sur les sites i et j.

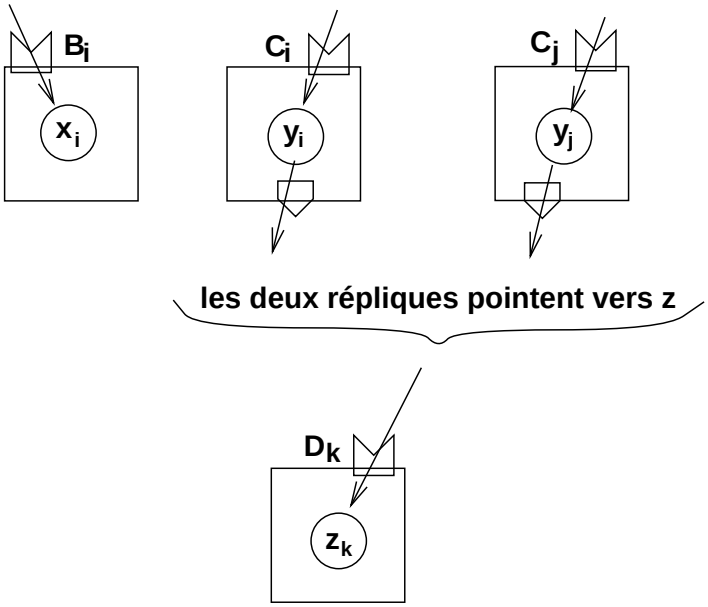


Figure 4.1: *Situation de départ.*

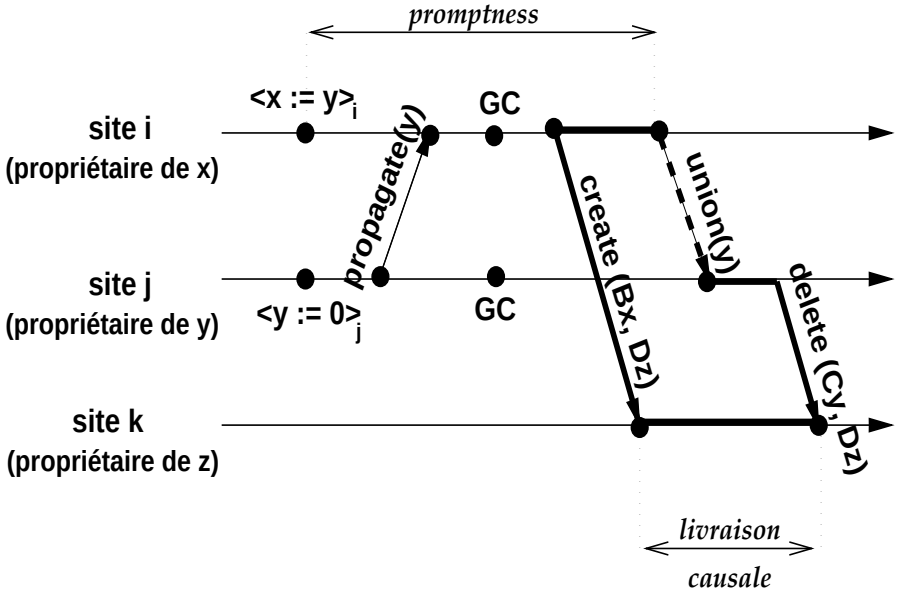


Figure 4.2: *Le message d'union transporte la dépendance causale (montrée par les lignes épaisses).*

Il est clair que la innocuité dépend de l'ordre de livraison relative des créations et des destructions sur le site k: *(i)* le message de création doit être envoyé du site i avant que le message de destruction ne soit envoyé du site j, et *(ii)* le message de création doit être délivré au site k avant la destruction. Pour garantir la première condition, tous les messages de création d'une même séquence d'un collecteur intra-bunch doivent être envoyés avant tout message d'union ou de destruction de la même séquence. La seconde condition est garantie en délivrant les messages de création et de destruction suivant un ordre causal. Notez que s'il n'y a pas de réplication, la innocuité demande que tous les messages de création d'une séquence d'un collecteur intra-bunch soient envoyés avant tout message de destruction de la même séquence.

Pour conclure, l'opération de création peut être effectué pendant toute la période de temps indiquée par *promptness* dans la Figure 4.2 (au maximum avant le message d'union). Un protocole de communication asynchrone avec une livraison causale est nécessaire.

Chapitre 5

Implémentation du GC dans Larchant

Le prototype de Larchant implémente le protocole de *entry-consistency* [14]. Ce protocole offre le modèle traditionnel de lecteurs multiples et d'écrivain unique: à chaque objet il ne peut être associé que, soit plusieurs jetons de lecture, soit un seul jeton d'écriture. Les sites détenant un jeton de lecture sont assurés de lire une réplique cohérente de l'objet correspondant. Chaque objet possède un propriétaire, qui est soit le site détenant le jeton d'écriture de l'objet, soit le site qui a détenu en dernier le jeton d'écriture. Un jeton d'écriture ne peut être obtenu qu'à partir du propriétaire de l'objet, tandis qu'un jeton de lecture peut être obtenu à partir de tout site qui en détient déjà un. Un jeton est obtenu en effectuant une opération d'*acquisition* (*acquire* en Anglais) de lecture ou d'écriture; le jeton est libéré par une *libération* (*release* en Anglais) correspondante. L'acquisition d'un jeton d'écriture pour un objet x implique l'*invalidation* de toutes les répliques de x pouvant être lues. (Voir Bershad[14] pour plus de détails.)

Nous avons implémenté deux algorithmes de GC intra-bunch: marquage-balayage et copie. Tous deux s'exécutent de façon concurrente par rapport aux mutateurs. Ils sont basés sur la technique à base de réplication de Nettles et O'Toole [91]. Évidemment, les deux collecteurs peuvent aussi s'exécuter en mode non concurrent, *i.e.*, le mutateur est stoppé pendant que les collecteurs s'exécutent.

Dans ce chapitre nous nous concentrerons sur l'asynchronisme du collecteur inter-bunch par rapport aux mutateurs.

5.1 Asynchronisme

Nous observons tout d'abord qu'un pointeur inter-bunch ne peut apparaître ou disparaître qu'en écrivant dans un objet x (opération d'affectation). Ceci demande que le site où s'exécute le mutateur détienne le jeton d'écriture de x. Ainsi, lorsque le mutateur acquiert le jeton d'écriture de x, Larchant enregistre

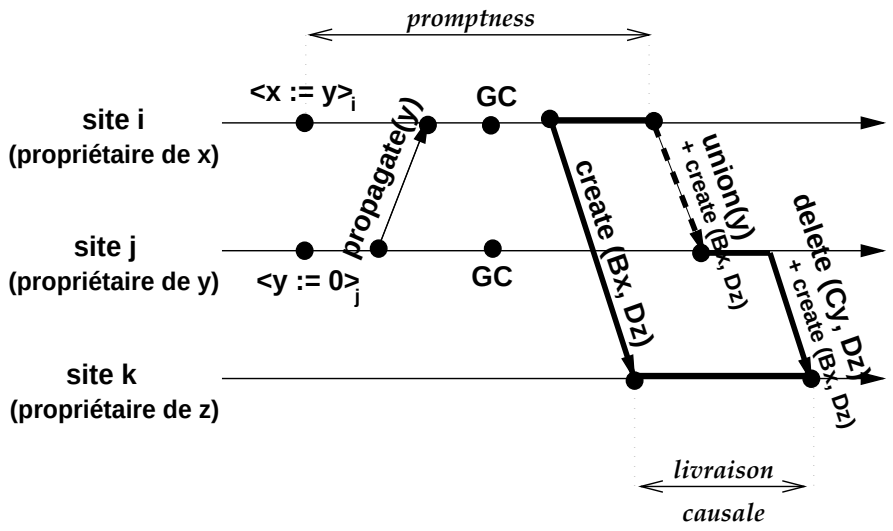


Figure 5.1: *La superposition assure la causalité.*

l'adresse de x (dans un journal appelé *GC-log*). Le GC-log contient les objets sales.

Plus tard, lorsque le collecteur intra-bunch s'exécute, il parcourt chaque objet sale dans le cache local (présent dans le GC-log). Pour chaque nouveau pointeur inter-bunch trouvé, le collecteur alloue la souche correspondante.

Ensuite, lorsque le collecteur inter-bunch s'exécute : pour chaque nouvelle souche, il demande la création du scion correspondant ; pour chaque souche disparue, il lance soit un message d'union (le site local n'est pas le propriétaire de l'objet correspondant) soit un message de destruction après avoir appliqué la règle d'union (le site local est le propriétaire de l'objet correspondant). Ceci garantit que les créations sont toujours faites avant les messages d'union ou de destruction. Cependant, la innocuité impose aussi un ordre causal. Pour cela, les messages de création sont superposés aux messages d'union et de destruction.

Considérons le cas illustré dans la Figure 4.2, restreint au protocole de *entry-consistency*. Notez que l'opération <y := 0>$_j$ peut être effectuée sur le site j parce qu'il est le propriétaire de y. La propagation de y de j vers i se fait via une acquisition de lecture (opération de *entry-consistency*) effectuée par le site i. Dans notre implémentation (voir Figure 5.1) pour assurer la causalité, create.scion(Bx, Cy) est superposé au message d'union (de i vers j) et à delete.scion(Cy, Dz) (de j vers k).

Chapitre 6

Performance des algorithmes de GC

Les collecteurs intra-bunch et inter-bunch s'exécutent de façon concurrente aux mutateurs ; ainsi le temps de pause du GC n'est dû qu'aux flips dans la collecte intra-bunch.

Pour obtenir des résultats de performance concernant le temps de pause du GC, nous avons exécuté le banc d'essais suivant. Sur chaque site où le bunch en cours de collecte est dans le cache (un maximum de 6 sites dans notre expérimentation), le mutateur crée des objets et en insère certains dans une liste ; les objets dans la liste sont atteignables, les autres sont des miettes.[1] La taille des bunches va varier de 1 Mo à 64 Mo. Nous montrons dans la Figure 6.1 les résultats obtenus pour le collecteur par copie dans le mode concurrent.

La taille du bunch et le nombre d'objets atteignables n'ont pas d'impact sur le temps de pause du GC parce que la majeure partie du travail du collecteur est effectuée de façon concurrente au mutateur. Lors du flip, seuls les objets modifiés par le mutateur après le parcours du collecteur sont reparcourus (et déplacés dans le cas d'un collecteur par copie).

Le nombre de sites possédant le bunch répliqué dans leur cache n'a pas d'impact sur le temps de pause du GC parce que chaque site effectue son ramassage de façon indépendante et asynchrone par rapport aux autres sites. En particulier, lors du flip, il n'y a pas de communication entre les sites possédant une réplique du même bunch dans leur cache.

Pour comparaison, nous montrons dans la Figure 6.2 les résultats obtenus avec le même banc d'essais pour le collecteur par copie dans le mode non concurrent, *i.e.*, quand le mutateur est stoppé pendant l'exécution du collecteur. Le temps de pause du GC est toujours indépendant du nombre de sites ayant dans leur cache une réplique du bunch en cours de ramassage. Cependant, pour des grands bunches, le temps de pause du GC est très perturbant.

[1] Nous avons exécuté nos bancs d'essais sur un réseau de stations DEC Alpha. Chacune possède 64 Mo de mémoire principale, 1 Go de disque, un cache de données de 8 Ko, un cache secondaire de 512 Ko, et une fréquence de 150 MHz. Elles sont interconnectées via FDDI.

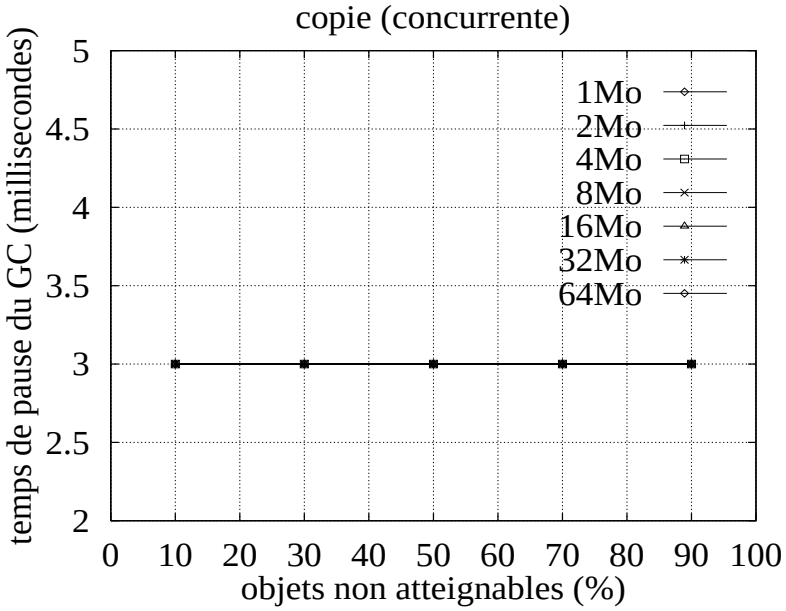


Figure 6.1: *Temps de pause du GC par copie concurrente.*

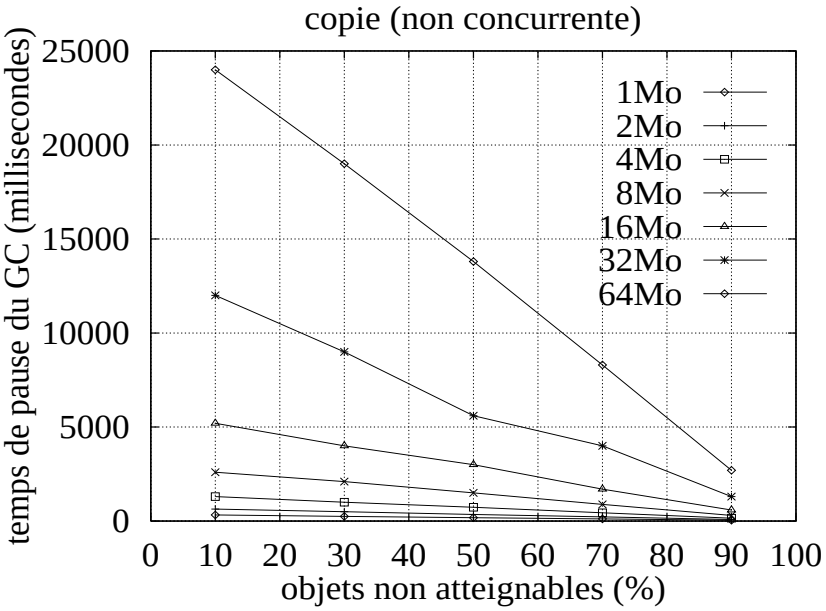


Figure 6.2: *Temps de pause du GC par copie non concurrente.*

Pour obtenir plus de résultats, nous avons aussi implémenté un simulateur d'édition de texte coopérative, appelé TX1. Un document TX1 est une structure hiérarchique contenant des objets pour les sections, sous-sections, paragraphes, lignes, etc. Le simulateur n'effectue rien d'autre que des allocations d'objets et des affectations de pointeurs. Nous avons mesuré les temps de pause du GC pour des collectes intra-bunch concurrentes avec TX1 s'exécutant sur un, deux et trois sites avec un bunch de 4 Mo et différentes quantités de miettes. Les temps de pause du GC ne sont jamais supérieurs à 5 millisecondes pour l'algorithme de copie. Pour l'algorithme de GC marquage-balayage les temps de pause du GC ne sont jamais supérieurs à 40 millisecondes.

Nous avons également porté le très connu banc d'essais OO7 [26], largement utilisé pour mesurer les performances des bases de données orientées objet. Notre objectif principal était de tester la fiabilité du prototype de Larchant. Le banc d'essais OO7 standard s'exécute sur un seul site et ne génère pas une grande quantité de miettes. Nous l'avons modifié de façon à lui faire générer plus de miettes en effectuant plus de changements dans le graphe des pointeurs. Lorsque nous lançons OO7 sur un seul site les temps de pause du GC par copie sont toujours inférieurs à 5 millisecondes.

Chapitre 7

Conclusion

Nous venons de décrire le modèle de Larchant, qui offre un partage et une répartition transparents, la persistance par atteignabilité et la gestion automatique de la mémoire. La persistance par atteignabilité impose de tracer le graphe des pointeurs à partir d'une racine persistante et de ramasser les données non atteignables. C'est le travail du GC.

Le collecteur dans Larchant ramasse de façon opportuniste des groupes locaux de bunches : il trace tant que la trace reste locale au site, et compte les références qui demanderaient des surcoûts d'E/S dans le cas d'un traçage. Il n'y a ni coordination ni de synchronisation entre les applications et les collecteurs. Tous les messages du collecteur sont asynchrones ; cependant, pour assurer l'innocuité, une livraison causale doit être assurée.

Étant donnée la diversité des modèles de cohérence utiles, nous avons choisi de ne faire que des suppositions minimales sur la cohérence. Chaque site collecte ses répliques de bunch indépendamment des autres sites sans imposer la disponibilité locale d'une mémoire cohérente. C'est pourquoi nos résultats sont appliquables de façon générale.

Les résultats de performance du GC de Larchant montrent que nos objectifs d'orthogonalité par rapport à l'incohérence et à l'échelle ont été atteints.

Partie/Part II

Larchant: garbage collection in a cached distributed shared store with persistence by reachability

Chapter 1

Introduction

An essential part of tomorrow's computing will be workers and organizations carrying out cooperative tasks interacting via shared information. The need for sharing data is well known in applications such as interactive computer aided design (for financial or engineering purposes) or cooperative tools, for example. Information is shared either concurrently or at different times, thus it must be available at different locations and persist beyond completion of a particular application.

The overall goal of the *Larchant* project is to provide a *Cached Distributed Shared Store* that makes data sharing simple and efficient. For this purpose, Larchant supports automatic and transparent persistence, memory-management, input-output, and distribution. With Larchant, programmers may concentrate on application development without being distracted by system issues such as memory bugs, caching, coherence, remote access, input-output, etc.

The model of Larchant is that of a *Shared Address Space* [93] with *Persistence By Reachability* (PBR) [8]. The address space spans every participating site in a network including permanent storage. This model is very attractive given its simplicity for application programmers.

A shared address space [29, 105] enables programs to share data by using ordinary pointers. With this model, distributed programming is very simple, since it provides the same memory abstraction as in the centralized case. To provide the illusion of a shared address space across the network, although site memories are disjoint, Larchant implements a *distributed shared memory* (DSM) mechanism [28, 57, 80]. Data is replicated in multiple sites for performance and availability.

Accessing data entails finding its reference by navigation from a well-known *persistent root* (*e.g.*, a name server). In systems with PBR [34, 48, 84, 112] a datum reachable from the persistent root is *persistent*, thus it should persist on permanent storage. A datum transitively reachable from the persistent root is persistent also. Unreachable data is not needed and can be reclaimed (and memory compacted). Such data is said to be *garbage*.

Reachability is accessed by tracing the pointer graph, starting from the persis-

tent root, and reclaiming unreachable data. This can be done either via manual memory-management, or automatically via *Garbage Collection* (GC) [127].

Manual memory-management is extremely error-prone (*e.g.*, dangling pointers and storage leaks) frequently resulting in the violation of the fundamental requirement of *Referential Integrity*: following a pointer should always work, *i.e.*, if persistent datum `x` points to datum `y`, then `y` must be persistent also.

If in a centralized short-lived program GC might be considered as an expensive luxury, in contrast, in a cached distributed shared store such as Larchant, the nightmare of manual memory-management increases as the number of objects, pointers and users scales up. Then GC becomes indispensable.

GC automatically ensures referential integrity, therefore improving program reliability and programming productivity. In addition, it may compact memory thus reducing fragmentation and improving locality.

Our goal for a garbage collector in Larchant can be summarized as follows: to support persistence by reachability in a distributed shared address space, transparently and efficiently. In the next section we summarize the main problems we must solve in order to achieve this goal.

1.1 Thesis Main Issues

In this document we address the issue of GC in Larchant. The main problems we tackle can be summarized as follows:

- *Scalability.*

  The GC algorithm must be capable of collecting a huge pointer graph without incurring into high performance costs due to computation, input-output, and synchronization. In particular, the collection of the whole graph in a single phase is clearly not scalable.

- *Coherence interference.*

  The GC algorithm must not compete with applications for holding coherent data replicas. Such competition would interfere with applications coherence needs. For example, if the collector on some site requires to access a coherent object, that would prevent an application to write into another replica of that same object at the same time.

- *Performance.*

  Applications performance overhead due to GC must be minimized. In particular, interactive applications should not be disrupted by the collector in such a way that its user would be annoyed.

The GC algorithm must solve the problems mentioned above, *i.e.*, it must be scalable, capable of making progress even if the locally cached data is incoherent, and efficient. In addition, it must be correct. This means that it must be *safe*

(persistent data must not be reclaimed), *live* (at least some garbage must be reclaimed), and, as much as possible, *complete* (all garbage must be eventually reclaimed). These are the specific goals for our GC algorithm: correctness, scalability, orthogonality to coherence, and good performance.

GC algorithms found in the literature are not adequate for a cached distributed shared store such as Larchant (see Chapter 2 for more details). This is due to the fact that Larchant is different from most other systems with the same overall goal (support for sharing) [2, 66, 96, 110, 114] because it extends caching with PBR. In particular, the coherence interference problem, and the issue of safety in presence of replicated data, possibly incoherent, were never addressed before.

1.2 Thesis Main Contributions

This thesis contains three main contributions. The first one is a general model of a cached distributed shared store with PBR, which we call the Larchant model. The second and most important contribution is a GC algorithm for the Larchant model, which is proved to be correct, is scalable and efficient. We believe the GC algorithm is widely applicable to other shared stores with PBR given the minimal assumptions we make regarding the interactions between applications, coherence protocols, and GC needs. In particular, the GC algorithm is orthogonal to coherence and executes asynchronously w.r.t. applications. The third contribution is a running prototype on top of which distributed applications with persistent objects can be executed.

1.3 Thesis Roadmap

In Chapter 2 we describe the basic uniprocessor GC algorithms found in the literature. We focus on their fundamental characteristics and present their extensions to RPC-based distributed systems, transactional systems, log-based file systems, multiprocessors, and DSM systems. We also provide some basic insights about their unsuitability to a cached distributed shared store such as Larchant.

In Chapter 3 we motivate the Larchant model by making a brief analysis of current systems with support for distributed data sharing (*e.g.*, file systems, databases, etc.). Then, we describe the Larchant model with a focus on characterizing in a general and widely applicable manner, the interactions among applications and coherence of data. The GC algorithm (presented in Chapter 4) is designed for the Larchant model.

In Chapter 4 we start by stating the specific problems of GC in Larchant along with an outline of the corresponding solutions. Then, we describe our GC algorithm: first intuitively, then formally in order to provide sufficient theoretical arguments concerning its correctness (safety and liveness).

Chapter 5 describes our implementation of the Larchant model (its architecture) and the GC algorithm in particular. We focus on the interaction between GC and a specific coherence protocol (entry-consistency [14]).

Chapter 6 studies, discuss, and analysis the performance of Larchant's GC.

Chapter 7 summarizes the most important aspects of this work and presents some directions for future research.

Chapter 2

Overview of GC Algorithms

Garbage collection has been a research topic for more than 30 years [33, 87]. It was first investigated in the domain of artificial intelligence. The applications then developed (mainly with symbolic programming languages, *e.g.*, Lisp) had very complex data graphs. In such applications, memory management (allocation and deallocation of memory) is a very intricate problem. When left to manual programming, it is the cause of serious program errors: violation of referential integrity due to dangling pointers, and memory exhaustion due to storage leaks.

Some years later, object-oriented languages and systems (*e.g.*, Smalltalk [67]) faced the same problem.

With the advent of distributed systems and PBR (persistence by reachability) the need for GC has increased even more. In such systems, manual memory management becomes a nightmare as the number of objects, pointers and users scales up.

In this chapter we study and discuss the most important GC algorithms found in the literature. First, we introduce the terminology that will be used in the rest of this document. Second, we present the basic uniprocessor GC algorithms. Then, we describe their extensions to RPC-based distributed systems, transactional systems, log-based file systems, multiprocessors, and DSM systems. We conclude this chapter with a discussion concerning the GC algorithms here described; we give some insights regarding their unsuitability for a distributed shared store based on caching and PBR, such as Larchant.

We address only the basic aspects of the GC algorithms. For more details on a particular algorithm or its variants we refer the reader to the existing literature [32, 97, 127].

We describe the GC algorithms characterizing them according to the following three aspects:

- *Type of Algorithm.* There are two basic types: reference counting, and reference tracing (tracing for short). The general term of tracing was taken from Lang [76].

- *Functioning Mode.* The functioning mode characterizes a GC algorithm in terms of the memory on which the collector works on each run (*e.g.*, a subset of all the memory) and the moments in which the collection takes place with respect to applications.
- *System Requirements.* A GC algorithm is characterized w.r.t. its needs in terms of knowledge of references location, and the possibility of moving reachable objects to different addresses in order to compact memory.

2.1 Terminology

An *object* is a contiguous set of bytes.

A *reference* identifies an object. In its simplest form it is a pointer. We will use equivalently the term reference and pointer along this document.

The *heap* designates a portion of virtual memory (possibly covering the whole memory) where objects are allocated. Hereafter, we will use the terms heap and memory equivalently.

We follow the standard vocabulary of the GC literature [40]. The *GC-root* is the set of references that forms the starting point for the GC graph tracing. The *mutator* is the application program that dynamically modifies the pointer graph: it creates objects, dereferences pointers, and assigns pointers. The *collector* is the system component that identifies and reclaims unreachable objects. In a distributed system, a collector is composed of a number of threads executing in different processes; we still call each one a collector. Similarly, the mutator is actually composed of multiple independent threads running in different processes; by extension, we call each of these threads a mutator.

A pointer assignment can result in: *(i)* creation of a new reference (to an object just created), *(ii)* duplication of an already existing reference, and *(iii)* discarding a reference to an object. Note that cases *(ii)* and *(iii)* may happen as the result of a single assignment operation. As a side-effect of pointer assignment, some reachable objects become unreachable. Unreachable objects, said to be *garbage*, are not needed and can be reclaimed.

A GC algorithm is *safe* when it does not reclaim reachable objects, and is *live* if it eventually reclaims some garbage. *Completeness* means that a GC algorithm eventually reclaims *all* garbage. Obviously, every GC algorithm aims at being safe and complete. However, as will become clear, this goal is not easy to achieve as scalability is at odds with completeness.

2.2 Basic Uniprocessor GC Algorithms

In this section we describe the two fundamental uniprocessor GC algorithms: *Reference Counting* and *Tracing*. We present their most relevant advantages and inconvenients which are related to completeness, performance, and memory

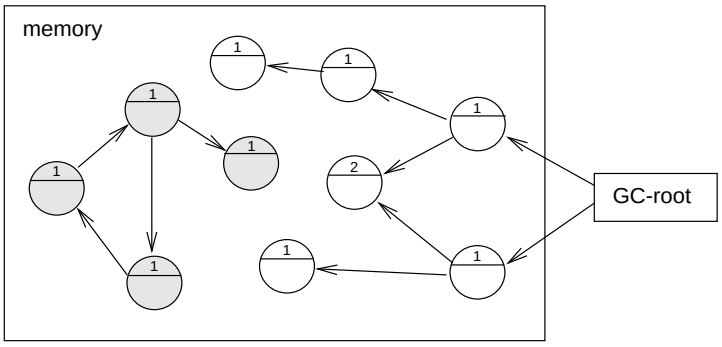


Figure 2.1: *Uniprocessor reference counting. The counter associated with an object is indicated at its top. The shaded objects are unreachable, however they are not reclaimed.*

fragmentation. Then, we describe several variants of these two algorithms which are based on the functioning modes and system requirements.

2.2.1 Reference Counting

The basic idea of the reference counting algorithm [31, 33] is the following. A *counter* is associated to each object, denoting the number of references to it. When an object is created, a single reference points to it, and its counter is initially one. Each time a reference is duplicated the object's counter is incremented. Each time a reference to an object is discarded, its counter is decremented. Therefore, reference counting preserves the invariant that the value of an object's counter is always equal to the number of references to it. Figure 2.1 illustrates this algorithm.

When a counter drops to zero, the corresponding object is no longer reachable and can be safely reclaimed. (Garbage objects are therefore reclaimed immediately after becoming unreachable.) However, the converse is not true, *i.e.*, an object may be unreachable but its counter greater than zero. This happens when an object is part of a cycle of unreachable objects (see Figure 2.1).

When an object is reclaimed (its counter has become zero) its pointers are discarded. Thus, reclaiming one object may lead to the transitive decrementing of reference counts and reclaiming many other objects.

This algorithm is simple to implement. However, it has three main problems. First, it is inefficient: its cost is proportional to the amount of work done by the mutator because counters must be updated whenever references are assigned. (This problem is addressed by variants of this algorithm such as Deferred Reference Counting [38].)

The second problem is fragmentation: the free space recovered from reclaimed objects is interspersed with reachable objects. This reduces locality of reference because as new objects are allocated in the free space recovered from reclaimed objects, unrelated objects are interleaved in memory. This may lead to a situa-

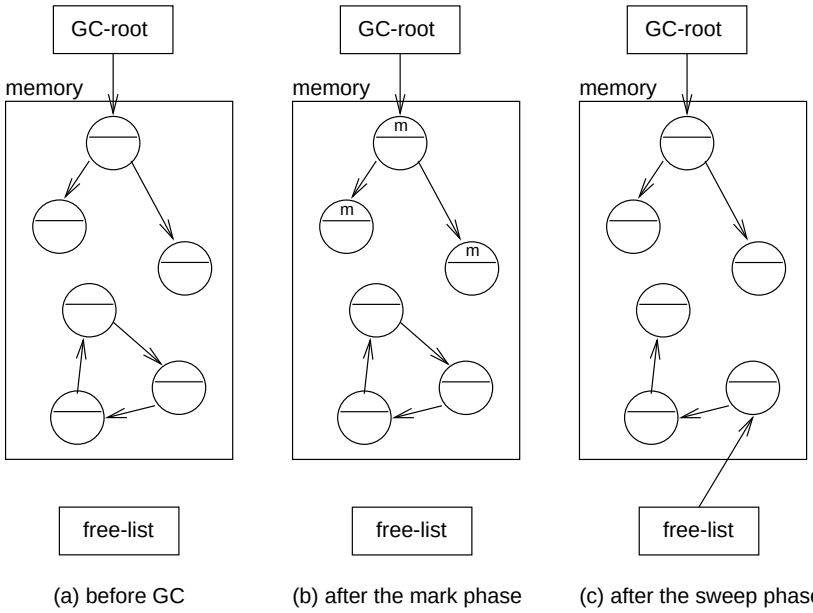


Figure 2.2: *Uniprocessor mark-and-sweep algorithm; marked objects are represented with a "m" in the header.*

tion in which the working set of an application is scattered across many virtual memory pages, so that those pages are frequently swapped in and out of main memory.

The third and major problem is that cycles are not reclaimed; thus, this algorithm is not complete. This may lead to memory exhaustion.

2.2.2 Tracing

There are two algorithms of the tracing type: *Mark-and-Sweep* and *Copy*. Tracing algorithms traverse the pointer graph, from the GC-root, to determine which objects are reachable. Unreachable objects are reclaimed.

The main advantage of tracing algorithms is their ability to reclaim cycles of unreachable objects.

2.2.2.1 Mark-and-Sweep

A mark-and-sweep collector [87] has two phases: *(i)* trace the pointer graph starting from the GC-root and mark every object found, and *(ii)* sweep (*i.e.*, examine) all the heap reclaiming unmarked objects. Figure 2.2 illustrates this algorithm.

During the mark phase every reachable object is marked (setting a bit in the object's header, for example) and scanned for pointers. This phase ends when there are no more reachable objects to mark.

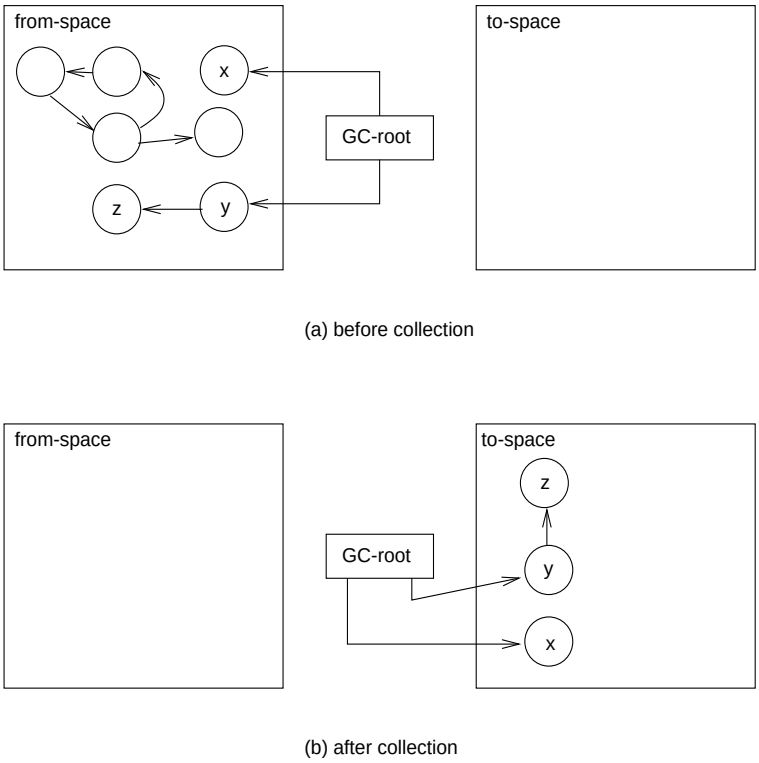


Figure 2.3: *Uniprocessor copy algorithm.*

During the sweep phase the collector detects which objects are not marked and inserts their memory space in the free-list. When the collector finds a marked object it unmarks it in order to make it ready for the next collection. This phase ends when there is no more memory to be swept.

This algorithm has two main problems. The first one is fragmentation. Just as with reference counting, free space recovered from reclaimed objects is interspersed with reachable objects. The second problem is that the cost of the sweep phase is proportional to the size of the heap.

2.2.2.2 Copy

In the copy algorithm [50, 88], the collector traces the pointer graph from the GC-root, and moves each object reached to another location. In this way, memory is compacted, therefore eliminating fragmentation, and allocation of objects is done linearly. This is an advantage of this algorithm.

Figure 2.3 illustrates the algorithm. The collector divides the heap in two disjoint *semispaces* called *from-space* and *to-space*. During normal mutator execution objects are allocated linearly in from-space. Once the collection starts, the collector moves reachable objects to to-space. Unreachable objects are left in from-space. When every reachable object has been moved, the roles of the semispaces is exchanged: the to-space becomes the from-space and vice-versa. This transition is called *flipping*. Flipping is atomic w.r.t. the mutator.

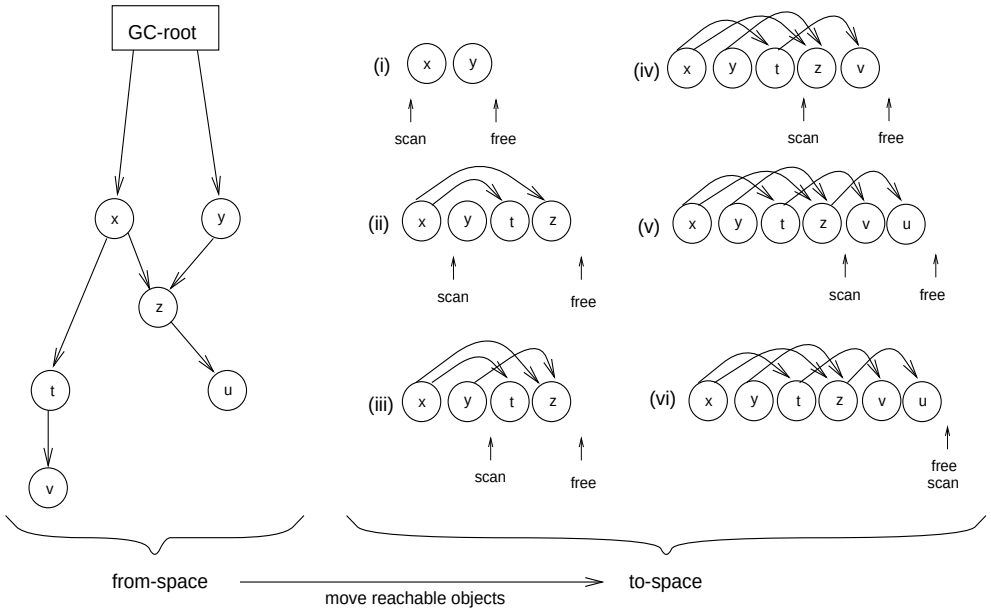


Figure 2.4: *Cheney algorithm.*

The Cheney algorithm [30] is a well known technique to move reachable objects from from-space to to-space. We describe it with the help of an example (see Figure 2.4). Objects immediately accessible from the GC-root (x and y) are the first to be moved to to-space. These objects form the initial set for a breadth-first traversal that is implemented with the help of two pointers: free which points to the first free address in to-space, and scan which points to the first object in to-space not yet scanned. Then, the object pointed by the scan pointer (x in this case) is scanned for pointers into from-space. Each object reached (t and z) is moved to to-space and the free pointer updated. In addition, the pointers in the scanned object are patched according to the new locations of the moved objects, and a forwarding pointer (pointing to to-space) is left in their old location. When y is scanned, z will be reached and given the forwarding pointer there left, z is not moved again; the pointer in y is simply patched. This algorithm ends when every object in to-space has been scanned.

An inconvenient of the copy algorithm is that only half of the memory space available is used at any point in time: the to-space is a wasted resource between collections.

2.2.3 Functioning Modes

In this section we address the various functioning modes that can be applied to the fundamental algorithms previously described. The functioning modes are essentially orthogonal to these algorithms.

These modes characterize a GC algorithm in terms of the memory on which the collector works on each run (*e.g.*, a subset of the all memory) and the moments in which the collection takes place w.r.t. mutators:

- *Partitioned* - Only a sub-set of the whole memory, called a partition, is garbage collected (independently from the rest of the memory).

  The existence of multiple partitions is useful in systems where is necessary to have a garbage collected heap co-existing with a manually-managed heap [25, 46], for example.

  Partitioned GC is also useful in large persistent and/or distributed systems because different partitions may be collected in parallel and independently from each other (see Section 2.3).

- *GC-only* - The mutator is halted while the collector runs. The time interval during which a mutator is halted due to GC is called *GC pause time*. This time may be unacceptable in some cases, for example, in interactive applications where the user would be annoyed by such pauses.

- *Incremental* - Small units of GC are interleaved with small units of mutator execution. Each GC pause time is smaller than in the GC-only mode.

- *Concurrent* - The mutator and the collector run concurrently. The usefulness of this mode is that collection adds no pauses on top of time-slicing.

  In the rest of this section, and when the difference is not relevant, we will use the term incremental to designate both incremental and concurrent functioning modes.

Note that reference counting algorithms are inherently incremental because the collector and mutator execution is interleaved.

In the rest of this section we focus on the incremental and partitioned variants of tracing collectors.

2.2.3.1 Incremental Tracing

The main issue of incremental tracing is how to ensure the correct execution of the collector when it competes with the mutator for the same data. Concerning this issue, there is no significant difference between mark-and-sweep and copy algorithms.

Before going into more details concerning this fundamental aspect it is useful to see how both mark-and-sweep and copy (incremental collectors) can be described as variants of an abstract *tricolor marking* algorithm [40].

The tricolor marking algorithm works as follows. Objects subjected to collection are colored *white*, and when the collection is finished, reachable objects must be colored *black*. A collection is finished when there are no more reachable objects to blacken. In a mark-and-sweep collector this coloring can be implemented by setting mark bits (objects whose bit is set are black). In a copy collector, the coloring is the process of moving reachable objects from from-space to to-space. Objects in the from-space are white; objects in the to-space are black.

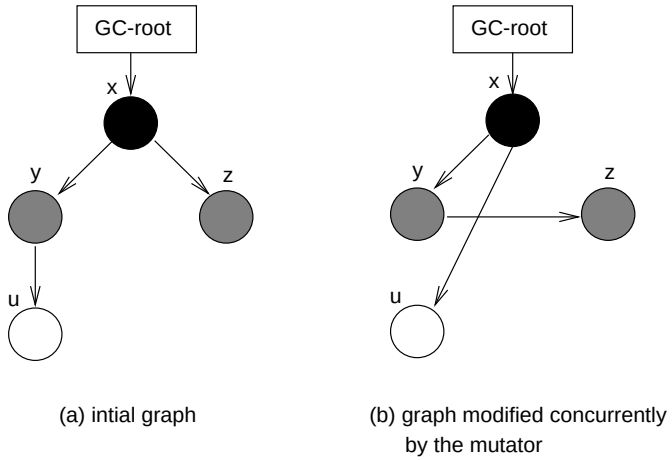


Figure 2.5: *Erroneous GC tracing due to concurrent mutator activity.*

The main difficulty with incremental tracing GC is that while the collector is tracing the pointer graph, as a result of mutator activity, the graph may change while the collector "isn't looking". If this happens, the collector may not find some reachable objects. Thus, the mutator cannot be allowed to change the pointers graph "behind the collector's back" in such a way that the collector fails to find all reachable objects.

For this purpose, there is a third color: *gray*. A gray object is one that has been reached by the collector tracing but its descendents may not have been. Once a gray object has been scanned it becomes black and its descendents are colored gray. (In a copy collector, the gray objects are those that have already been moved to to-space but have not yet been scanned.)

Black objects may not point to white objects. This invariant allows the collector to assume that it is "finished with" black objects, and can continue to traverse gray objects. If the mutator creates a pointer from a black object to a white one, it must somehow notify the collector that its assumption has been violated. Therefore, the collector must be capable of keeping track of graph changes resulting from mutator activity, and re-trace parts of the graph adequately. This ensures that the collector is aware of every change concerning the pointers graph.

Figure 2.5 illustrates this problem by showing the need for coordination between the mutator and the collector. Suppose that object x has been completely scanned (and therefore blackened); its descendents (y and z) have been reached and grayed. Now, suppose that the mutator swaps the pointer from x to z with the pointer from y to u. The only pointer to u is now in object x, which has already been scanned by the collector (this violates the invariant: black object x pointing to white object u). If the tracing continues without any coordination, y will be blackened, z will be reached again (from y) and u will never be reached at all, and hence will be unsafely considered garbage.

This problem is solved by coordinating the mutator with the collector. This

can be done with the following two techniques: *read-barrier* or *write-barrier*.

Read-Barrier A read-barrier is used to detect when the mutator attempts to read a pointer to a white object; immediately, the collector scans the object and colors it gray; since the mutator cannot read pointers to white objects, it cannot write them into black objects.

There are many incremental collectors using a read-barrier [7, 10, 23, 63, 99, 133]. We describe here a representative example using a copy algorithm. The best-known incremental copy collector is Baker's [9]. A collection starts with a flip that conceptually invalidates (for mutator accesses) all objects in from-space, and moves to to-space all objects directly reachable from the GC-root. Then, the mutator is allowed to resume. Any object in from-space that is accessed by the mutator must first be moved to to-space; this is enforced by the read-barrier. Thus, the mutator is never allowed to see pointers into from-space; if so, the referent is immediately moved to to-space.

In terms of the tricolor marking algorithm, objects in to-space that were already scanned are black; objects in to-space but not yet scanned are gray. Objects still in from-space are white. New objects created by the mutator while the collection is taking place, are allocated in the to-space and are colored black.

Write-Barrier A write-barrier detects when the mutator tries to write a pointer into a black object; when that happens, the write is trapped or recorded [7, 20, 35, 40, 115, 131].

We describe here an interesting copy collector using a write-barrier; it is called *replication-based GC* [91, 94]. The basic idea is the following. While the collector moves objects to to-space, the mutator continues to access the from-space versions of objects, rather than the "replicas" in to-space. When every reachable object has been moved to to-space, a flip is performed and the mutator then starts seeing the to-space replicas.

This technique eliminates the need for a read-barrier because all the reachable objects are conceptually moved to to-space at once (when the flip occurs). However, it is necessary to have a write-barrier because the mutator sees the old version of objects in from-space; if an object has already been moved to to-space and then the mutator writes into it, the replica in to-space is now out-of-date w.r.t. the version seen by the mutator. Thus, the write-barrier catches all mutator writes (and stores them in a log) in order to allow the collector to propagate those modifications to the to-space replicas when the flip takes place. In other words, all the modifications to objects in from-space must be applied to the corresponding replicas in to-space, so that the mutator sees the correct values after the flip.

2.2.3.2 Partitioned Tracing

The application of the partitioned mode to the basic GC algorithms (reference counting and tracing) results in the following two variants: *(i)* hybrid collectors,

and *(ii)* generational collectors.

In the first variant each partition is traced independently from the others, and cross-partition references are handled with a reference counting algorithm [19, 90]. This variant is very interesting for large distributed and/or persistent systems and is described in more detail in Section 2.3.

The second variant, generational collectors, aims at reducing the GC pause time by decreasing the amount of memory that has to be collected. For this purpose, it takes advantage of the following empirical observation. In many applications most objects are reachable for a very short length of time; only a small portion remains reachable much longer [61, 81, 89, 120]. Thus, the idea is to separate the objects in (at least) two groups, those objects that are reachable only for a short length of time belong to the first group, the others belong to the second group, and to collect the first group more often.

Both mark-and-sweep and copy algorithms can be made generational [35]; in this section, we will focus on the copy algorithm.

Objects are segregated into multiple partitions by age. Each partition is called a generation. Younger generations are collected more often than older generations. The age of an object is approximated by the number of collections it has survived.

To avoid the cost of successive scans and moves (from from-space to-space) of those objects that remain reachable for a long time, partitions containing older objects are collected less often than the younger ones. Thus, once objects have survived a certain number of collections, they are moved to a less-frequently collected partition (instead of to-space).

To allow young generations to be collected without having to collect the older ones, the collector must be capable of finding pointers into the young generations. This requires either the use of a write-barrier similar to the one we found in the incremental functioning mode (see previous section) to keep track of such cross-partition pointers [6, 41, 89, 119, 129], or indirect pointers from older to younger generations [81].

The set of references pointing from older to younger generations is usually called remembered-set. When a younger generation is collected, the pointers in the corresponding remembered-set are part of the GC-root.

2.2.4 System Requirements

Tracing garbage collectors traverse the pointer graph. For this purpose, the collector must be capable of finding the pointers inside the GC-root and inside each reachable object.

In programming languages providing runtime type information, it is possible to differentiate pointers from raw data, with certainty. This is the case of Lisp and Smalltalk where there is enough type information that can be used to determine object layouts, including the locations of embedded pointers [5, 59, 116].

However, there are cases in which such runtime type information is not available, as is the case with the programming languages C [69] and C++ [47]. In such environments a possible solution consists of either a pre-processor [44] or the compiler [22, 51, 103] to statically generate type information for each data type. Another alternative is to take advantage of some specific language features (*e.g*, smart-pointers in C++) [36, 45] to generate the pointers locations.

Another solution, called *conservative* GC [11, 21], relies on a conservative pointer finding approach, *i.e.*, the collector treats anything that might be a pointer as a pointer. Thus, any properly aligned bit pattern that could be the address of an object is actually considered to be a pointer to that object.

There are a few problems concerning this approach. First, the conservative interpretation of ambiguous data (*e.g.*, considering an integer as a pointer) may lead to consider garbage objects as being reachable. This waste of memory can be a serious problem for memory intensive applications [102, 126]. Second, given that the collector does not differentiate raw data from pointers, it has to scan all the data inside reachable objects. This extra cost of scanning increases the GC pause time. Finally, a copy collector cannot move reachable objects and patch the corresponding pointers because, a non-pointer might be considered to be a pointer and would be mistakenly patched. However, in spite of these problems, there are cases for which a conservative approach is adequate, and sometimes the only that is feasible [124].

2.3 Distributed GC Algorithms

In this section we present the most interesting GC algorithms found in the literature for RPC-based distributed systems[1] (*i.e.*, with no support for persistence or DSM). These algorithms are extensions of the basic reference counting and tracing.

2.3.1 Reference Counting

Each process in the system holds one partition that is collected independently from the rest of the memory with a tracing algorithm. Cross-partition references are managed with a reference counting algorithm.

The extension of the uniprocessor reference counting algorithm to handle cross-partition references poses some problems. These problems, generally called *race conditions*, arise because objects' counters have to be incremented and decremented through messages exchanged between processes.

Such messages must be delivered reliably and in causal order [16, 74] to ensure safety and liveness: increment and decrement messages are not idempotent and must not be duplicated or lost.

[1] A distributed system is a set of processes communicating by messages, with no shared memory.

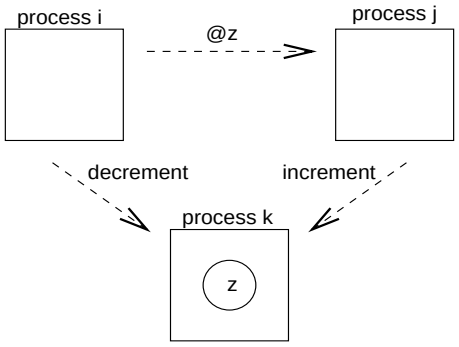


Figure 2.6: *Decrement/increment race condition with distributed reference counting.*

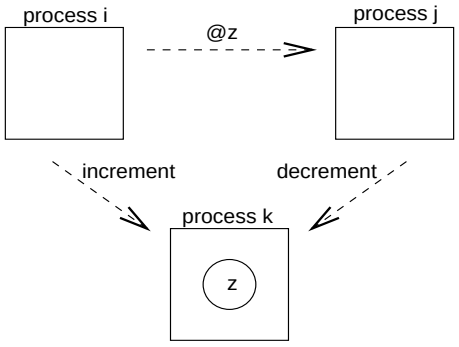


Figure 2.7: *Increment/decrement race condition with distributed reference counting.*

There are two types of race conditions, both possibly leading to the unsafe reclamation of a reachable object. We call them *decrement/increment* and *increment/decrement*.

Figure 2.6 illustrates the decrement/increment race condition. Suppose that process i holds a reference to object z in process k, and sends a message to process j containing @z, a reference to z. Process j receives this message and sends an increment message to k concerning object z; concurrently, process i deletes its reference to z and sends the corresponding decrement message to k. If the decrement message arrives first at k, then z is considered to be unreachable and is unsafely reclaimed.

At first glance, it seems that this race problem could be solved by simply making the sender process i conservatively emit the increment message before sending @z to j. However, as explained now, this does not solve the problem and leads us to the scenario for the second race problem.

The increment/decrement race is illustrated in Figure 2.7: the sender process issues the increment message, not the receiver as in the previous case. Suppose that process i holds a reference to object z in process k, and sends a message to process j containing @z. Now, j receives this message and immediately discards it. Therefore, it sends a decrement message to k concerning z. Concurrently,

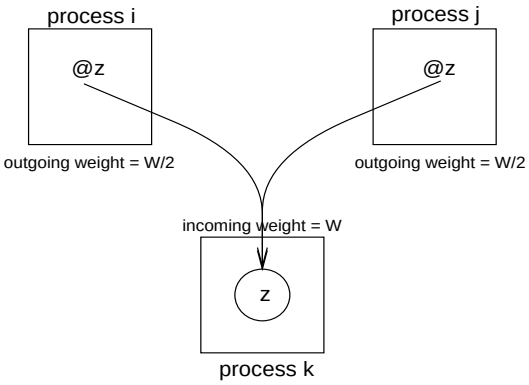


Figure 2.8: *Weighted reference counting*

process i sends the corresponding increment message to k. Once again, if the decrement message arrives first at k, z is unsafely reclaimed.

An obvious solution for these two race problems is to acknowledge the increment message before sending a decrement or a reference to a remote process, respectively. However, this introduces communication overhead and the GC algorithm is still not resilient to failures. For these reasons there are many variants of the reference counting algorithm that can be grouped in the following main categories: weighted reference counting [15, 123], indirect reference counting [58, 95], and reference listing [18, 85, 106, 108]. Each one of these variants avoids the transmission of increment messages, therefore solving the two race problems previously described. We describe these variants now.

2.3.1.1 Weighted Reference Counting

Each cross-partition reference has two associated weights: a weight at the source process (outgoing weight); a weight at the target process (incoming weight). For any object z, the incoming weight must be equal to the sum of the outgoing weights associated to the cross-partition references pointing to z (see Figure 2.8).

When a cross-partition reference is first created, both incoming and outgoing weights are equal (a even positive value, usually a power of two).

When the holder of a reference passes it (through a message) to another process, it divides the current outgoing weight in two parts (normally equal), retains one, and sends the other along with the message. When the receiver process receives the message, it associates to the new outgoing cross-partition reference the weight just received; if a cross-partition reference to the same object already exists, it simply adds the received weight to the current one. Thus, the sum of outgoing weights is always kept equal to the incoming weight at the target process.

When a process deletes an outgoing cross-partition reference, it sends a decrement message with the associated weight to the target process. The target process receives that message and subtracts the received weight from its incom-

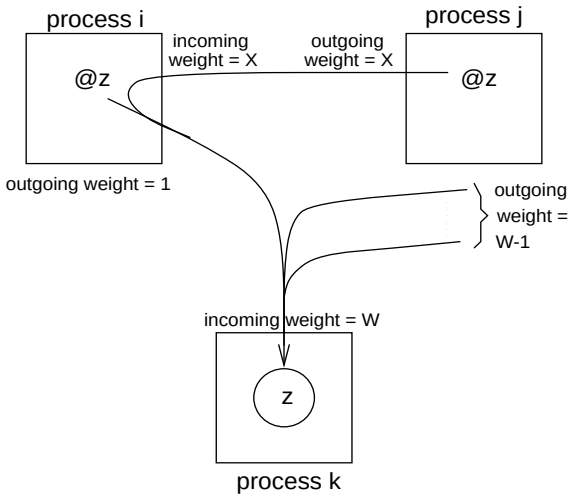


Figure 2.9: *Problem with weight reference counting.*

ing weight (corresponding to the cross-partition reference). When the incoming weight becomes zero, then the corresponding object may be reclaimed.

Note that this algorithm solves the two race problems previously described but still needs reliable communication (no loss or duplication) for ensuring safety and liveness. In addition, this algorithm has the following problem: the limit imposed on the number of times a reference may be sent to another process is limited by the initial associated weight. This problem is solved by the use of an extra indirection as explained now.

Suppose that process i holds an outgoing cross-partition reference pointing to z in process k, and that the associated weight has dropped to 1 (*i.e.*, no more division of the weight is allowed). Now, if process i needs to send @z to process j, the collector creates a new incoming cross-partition reference coming from j that refers *indirectly* to z through the original outgoing cross-partition reference that points to k (see Figure 2.9).

2.3.1.2 Indirect Reference Counting

This algorithm avoids the use of increment messages by maintaining a distributed reference count for each remotely referenced object. Increments are always done locally, therefore with no communication. Remote references are counted with a counter associated to the corresponding pointer.

Associated to each remote pointer there is (see Figure 2.10): *(i)* the identification of the process from which that pointer came, and *(ii)* a variable that counts how many times that pointer was duplicated from the local process. The latter, is incremented each time the pointer is duplicated, therefore creating a new remote reference to the target object. Note that this increment is local, *i.e.*, there is no increment message sent. When the mentioned counter becomes zero, the process sends a decrement message to the process from which the pointer came.

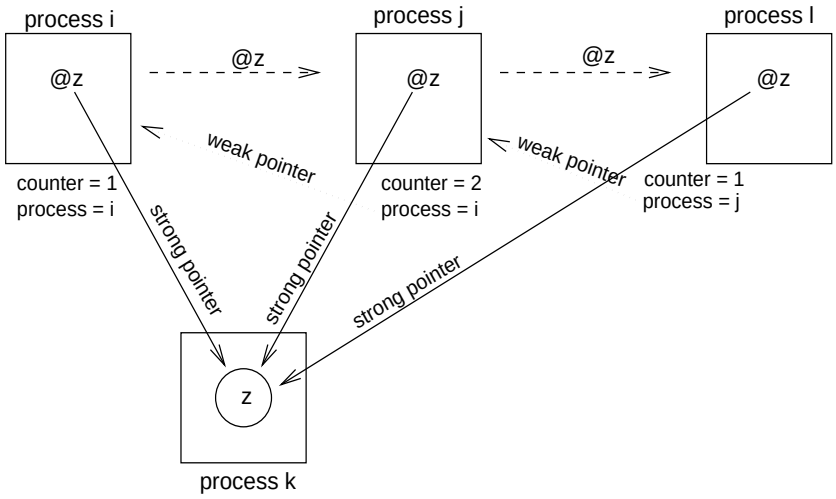


Figure 2.10: *Indirect reference counting.*

Sometimes in the literature [97], the reference to the remote object is called *strong pointer* (pointers to object z in Figure 2.10) in opposition to the notion of *weak pointer* which is implemented by the identification of the process from where the remote reference came (pointers to processes i and j in Figure 2.10). Weak pointers are used by applications when accessing the remote object while strong pointers are only for GC purposes (the process to which the decrement message is sent).

Note that this algorithm does not ensure safety in presence of duplicated messages and liveness is not preserved if messages are lost. In addition, the deletion of a remote reference may generate many decrement messages in cascade.

2.3.1.3 Reference Listing

In this algorithm an outgoing cross-partition reference goes indirectly through an auxiliary data structure called *stub*. Similarly, a incoming cross-partition reference goes indirectly through an auxiliary data structure called *scion* (instead of a reference counter). This is illustrated in Figure 2.11.

A stub identifies his matching scion and vice-versa (*i.e.*, the scion holds a back-pointer to the corresponding stub). In addition, a scion identifies the target object. A stub-scion pair fully identifies the corresponding cross-partition reference.

For safety, the invariant that must be kept is the following: if in some process there is a stub associated to a cross-partition reference, than the target object must be pointed by the corresponding scion. The increment and decrement messages of the basic reference counting algorithm are replaced by asynchronous *create* and *delete* messages (applied to scions), respectively. These messages are called *control messages*. A create (delete) message informs the target process that a scion must be created (deleted).

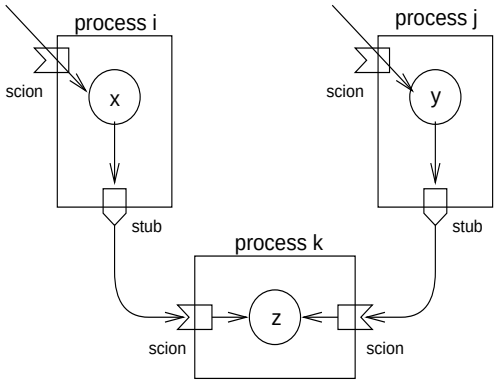


Figure 2.11: *Stubs and scions describing cross-partition references.*

The algorithm works as follows. Each process i collects its partition independently. (The GC-root includes the set of scions.) The collection generates a new set of reachable objects and a new set of stubs describing outgoing cross-partition references. By comparing the old set of stubs (before the local collection) with the new one, the collector discovers the stubs that no longer exist and sends a delete message concerning the corresponding scion. Note that, instead of sending delete messages, the collector can send the list of stubs; then, it is up to the receiver to find out which scions are no longer referenced.

The advantage of this algorithm resides on its fault-tolerance to message loss and duplication. For example, a message to delete a scion can be sent several times without violating safety; a repeated delete message is simply ignored. For more details, we forward the interested reader to the literature [108].

2.3.1.4 Cycles of Garbage

Reference counting algorithms do not collect cross-partition cycles of garbage. There are three solutions for this problem: an additional cross-partition tracing algorithm, object migration, and trial deletion. Cross-partition tracing algorithms are addressed in the next section.

Object Migration The basic idea of the migration technique [19, 111] is the following: migrating several objects to a single partition (in a process) in order to transform a distributed cycle into a local cycle that can be easily reclaimed with a tracing algorithm.

The problem with this technique is twofold: *(i)* some objects cannot be migrated due to systems constraints, and *(ii)* it is difficult to know which objects should be migrated (and when) in order to reclaim a maximum amount of cycles with a minimum cost.

Recent work [86] proposes a low-cost heuristic that helps to find which objects should be migrated. This heuristic is based on the "distance of objects" which is defined as follows. The distance of an object is the minimum number of

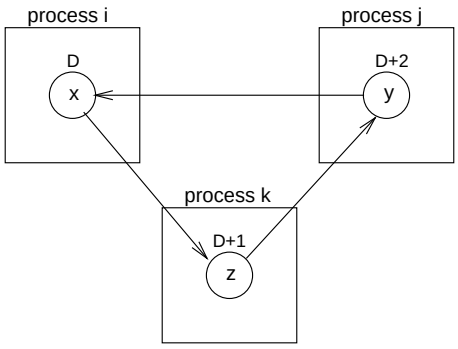


Figure 2.12: *Distances of objects in a cycle of garbage.*

cross-partition references in any path from the persistent root to that object; the distance of a unreachable object is infinity.

The distances are propagated inside a partition by the tracing collector and between partitions by sending the set of stubs to other processes (reference listing algorithm). With this mechanism, the distances of objects that belong to a cycle of garbage increase indefinitely, while the distances of others do not.

We provide here an example that illustrates how the distance of a cycle increases indefinitely. Initially, x is reachable from the persistent root and x's distance is D. Then, the pointer from the persistent root to x is discarded. Figure 2.12 shows each object distance at this instant. Now, imagine that process i collects its partition. Object x is reachable from y, thus x's distance will be that of y plus one. When the collector in k runs: z is reachable from x, thus the distance of z will be x's distance plus one. When the collector in j runs: y is reachable from z, thus the distance of y will be z's distance plus one. Consequently, the distances of x, y and z will increase indefinitely.

The basic idea of this algorithm is to migrate only those objects whose distance is above some threshold because those are likely to be garbage. The estimation of the threshold depends of the expected distance of reachable objects.

Trial Deletion The trial deletion technique [122] works as follows: an object suspected to belong to a cycle is tentatively considered to be unreachable by the distributed collector. This is done by pretending that its associated reference count is zero; then, if the associated reference count in fact drops to zero, the collector concludes that the object must be part of a cycle.

A difficulty with this solution is to decide which objects are suspected to belong to a cycle. In addition, the implementation of the temporary reference counts is complex given that multiple trial deletions may be occurring in parallel.

2.3.2 Tracing

The majority of the tracing algorithms in RPC-based distributed systems, are of the type mark-and-sweep. In this section we briefly present the most inter-

esting (non-partitioned) distributed mark-and-sweep algorithms found in the literature.

The basic distributed mark-and-sweep algorithm is as follows. In the mark phase each process marks objects reachable from the GC-root (set of scions). Each marked object is scanned and for each remote reference found a *marking message* is sent to the target process. The process receiving a marking message marks the corresponding object and continues the marking phase. When every process has already marked all the reachable objects and there are no marking messages in transit, the sweep phase starts. The sweep phase may be done by each process independently from any other process.

Note that this algorithm imposes a global trace, including every process in the network, before any garbage can be reclaimed. In addition, it is necessary to synchronize the mark and sweep phases. In other words, all processes must synchronize in order to agree on the termination of the marking phase. The difficulty with this global synchronization (besides its non-scalability) is that no marking messages may be in transit.

Another difficulty with distributed mark-and-sweep is to maintain the consistency of the description of cross-partition references (*i.e.*, stubs and scions). Alternatively, if some degree of inconsistency is allowed, then the GC algorithm must be made in such a way that such inconsistencies are safe.

2.3.2.1 Mark-and-Sweep with Timestamps

This technique [64] is like the basic distributed mark-and-sweep algorithm except that it uses timestamps instead of mark bits. The timestamp of a reachable object keeps increasing while the timestamp of a unreachable object eventually stabilizes. A global timestamp threshold is computed and every object marked with a lower timestamp is garbage. This algorithm avoids a global synchronization of every process needed to agree on the termination of the mark phase.

Each process has a clock that is used to record the time when the GC started locally (we call it *GC-start*). When the collector starts, each object directly referenced from the GC-root is timestamped with GC-start. Scions retain the timestamp last put into them. (When a scion is created, it is timestamped with the local process current timestamp.) Then, the collector propagates the timestamps associated with its local GC-root all along the reachable objects up to the reachable stubs. (Actually, only stubs and scions need stamps; reachable objects are simply marked.)

The collection in a process is expected to timestamp a stub with the largest local timestamp of any GC-root from which it is reachable. To ensure this, references in the GC-root are selected for tracing in the decreasing order of the corresponding timestamps.

At the end of a local collection stubs are sent to their target processes (with the corresponding timestamps). We call them *timestamping messages* (they are similar to the marking messages of the basic distributed mark-and-sweep

algorithm.) When a process receives a timestamping message it updates the timestamps of the corresponding scions to a value that is the maximum of their current value and that received in the message. In case a scion's timestamp is augmented, the local process records the fact that the new timestamp value has not yet been propagated. For this purpose, each process maintains a timestamp called *redo* whose value is equal to the greatest timestamp already locally propagated. Thus, when a scion's timestamp is increased, the redo is set to the scion's old timestamp if that is lower that its current value.

When all timestamping messages have been locally treated, the process sends an acknowledgment message to the sender. Once the sender process has received the acknowledgment of all timestamping messages sent, it can update its own redo value to the local GC-start (as long as it did not receive timestamping messages from other processes itself).

It can be shown that the threshold mentioned above (the one that allows the collector to detect unreachable objects) can be the global minimum of the redo's. Thus, any scion timestamped with a value below the minimum of the redo's is garbage.

The problems with this algorithm are the following. First, the computation of the timestamp threshold (minimum of redo's) is notoriously costly and unscalable as it depends on a global termination algorithm [118]. Second, it requires every process to cooperate (even though there is no global synchronization when the marking phase terminates). Third, if a process i crashes, the entire tracing eventually halts as the global minimum redo's algorithm will be stucked at i's redo value. Thus, the algorithm is not resilient to failures.

2.3.2.2 Logically Centralized Tracing

The idea of this variant [73] is to compute the global accessibility of objects on a single highly available centralized service. All the information manipulated by this service (possibly replicated for fault-tolerance) is stored in stable storage.[2] Each process performs asynchronous local collections and communicates with the central service providing it with accessibility information, *i.e.*, stubs and scions. From time to time, each process asks the central service about the accessibility of its scions. The answer may indicate that a scion is no longer reachable from any stub, thus it can be deleted.

The central service never has a consistent view of the reachability of every object given that processes collect asynchronously. Thus, the central service adopts a conservative approach in order to cope with such inconsistencies while being safe. For this purpose, it uses a timestamp protocol involving loosely synchronized clocks at each process and a bounded delay for messages in transit.

To reclaim cycles this variant may require a process to traverse its local pointer graph multiple times. This is not a good solution because of its cost in terms

[2] A *stable store*, as defined by Lampson [75], is a set of objects that move atomically from one consistent state to another.

of performance. Therefore, to reclaim distributed cycles a solution similar to the previous one (tracing with timestamps) must be used.

Finally, the central service can become a bottleneck and processes must transfer a fair amount of data.

2.3.2.3 Group Tracing

This algorithm [77] is designed to work in a partitioned memory. Each process holds a partition that is collected independently. The algorithm aims at reclaiming cross-partition cycles of garbage without requiring the cooperation of every process in the network. Only the processes containing objects that belong to the cycle being reclaimed must cooperate.

With this algorithm any set of processes may decide to form a group and perform a group-wide tracing independently from other processes. This tracing is able to reclaim every cross-partition cycle of garbage within the group. Objects pointed through incoming references from outside the group are considered to be reachable.

Groups can be formed and dismantled dynamically. Multiple collections on different groups can run in parallel and such groups may even overlap.

Given that groups are dynamic, if some process is down (or unreachable due to communication problems) the set of accessible processes can still form a group. Thus, GC is not blocked due to a crashed process.

One difficulty of this algorithm is finding the GC-root of a group, *i.e.*, the incoming references from processes external to the group. This is due to the fact that groups are dynamic, thus the GC-root of a group must be found afresh. In addition, the formation of a group imposes a non-negligible performance penalty.

Another problem resides on knowing which processes should be grouped (and when) in order to reclaim the maximum amount of cross-partition cycles of garbage.

2.4 GC in Transactional Systems

In this section we describe very briefly the most interesting GC algorithms found in the literature for transactional systems. These algorithms are extensions of the reference counting and tracing algorithms presented in the previous sections. Such extensions are needed to deal with the specific safety problems posed by transactional systems. We focus on these problems and corresponding solutions.

2.4.1 Transactional Reference Listing

Thor [82] is an object-oriented client-server database with multiple storage servers. Clients cache in main memory objects that are being accessed. The

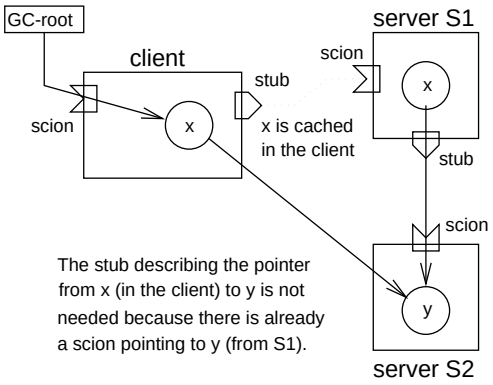


Figure 2.13: *Transactional reference listing.*

GC algorithm in Thor [85] is a fault-tolerant variant of the reference listing (recall Section 2.3.1.3). We call it transactional reference listing.

When a server recovers from a failure it must retrieve all the scions pointing to objects allocated in the server partitions. Scions whose stubs are held by other servers are easily recovered because such scions are kept in stable storage. On the other hand, scions whose stubs are held by clients are not kept in stable storage because that would be too expensive. Instead, a server keeps in stable storage a list of its clients. Thus, when a server recovers it knows who its clients are and sends them query messages asking for their stubs.

If a client process has not communicated with a server for a long time and does not respond to repeated query messages, the server assumes it has failed. However, the client may just be unable to communicate with that server because of network problems. It might happen that the client communicates with other servers.

Imagine that a server S1 assumes that a client has failed while a second server S2 does not (see Figure 2.13). Then, the first server discards the scions whose stubs were held by the client. This may cause the second server to unsafely reclaim an object y that is still reachable from the client. This erroneous behavior is due to the inconsistent views the servers have about the client.

To solve this problem, it is necessary that all servers get a consistent view of a client status. For this purpose there is an atomic shutdown protocol. Once this protocol is executed no server will honor requests from the client that has been shutdown. Thus, in the example above, S1 and S2 would agree that the client has crashed and y would be safely reclaimed because the client would not be allowed to follow the pointer from x to y.

A problem with this solution is that the shutdown protocol is not easily scalable.

2.4.2 Transactional Mark-and-Sweep

This algorithm [3] is a variant of the basic uniprocessor mark-and-sweep with the partitioned and incremental functioning modes, extended in order to cope

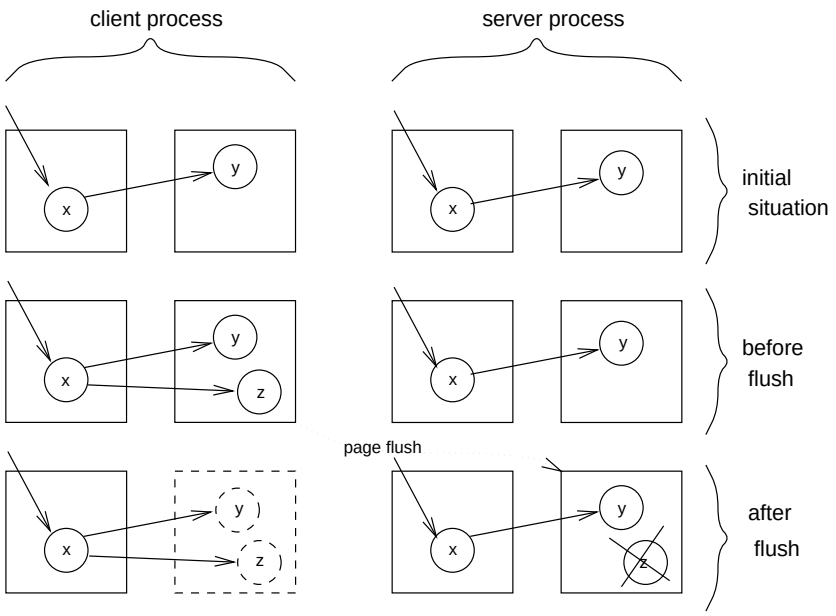


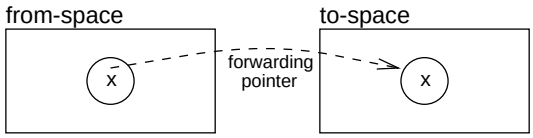
Figure 2.14: *Problem with transactional mark-and-sweep.*

with transactions in a client-server database. The collector runs on the server and was implemented in the EXODUS database [27]. We call it transactional mark-and-sweep.

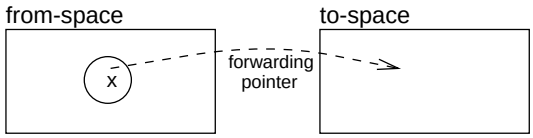
When a reference to some object is discarded inside a transaction, the pointed object is eligible for collection only after the commit of that transaction. This rule prevents the unsafe reclamation of an object that becomes unreachable during a transaction that will later abort. In fact, the abort of the transaction will make that object reachable again. Thus, any object that might become unreachable during a transaction will only be reclaimed if it remains unreachable after the commit of the transaction.

An object created during a transaction is eligible for reclamation only after the commit of the transaction that created it. This rule prevents the unsafe reclamation of reachable objects due to the flush of pages (from the client to the server) in an order that is not controlled by the collector. This is illustrated in Figure 2.14 (`x` is reachable from the GC-root). The application creates object `z` and makes it reachable from `x`. Then, the page containing objects `y` and `z` is flushed to the server. When the collector runs (in the server) only the page containing objects `y` and `z` has been flushed. Thus, `z` is unsafely reclaimed because it is not reachable from `x`.

The memory space reclaimed by the sweeping phase can be reused for allocating new objects only when the freeing of that space is reflected in stable storage. This ensures that during a recovery after a failure there will always be enough space for allocating objects. This is illustrated by the following example. Suppose that a client creates objects `x`, `y` and `z` in a page that had previously been garbage collected at the server. However, that page has not been written to stable storage yet; on disk it still contains a garbage object `t` (not yet swept).



(a) virtual memory before failure



(b) virtual memory after failure

Figure 2.15: *Problem with copy GC in presence of failures.*

Now, the client commits the transaction. As a result, the page and the associated log are sent to the server. The log is written to stable storage and the system crashes before the updated page reaches the disk. Then, recovery starts. The problem is that objects x, y and z must be created (according to the log) but there is not enough space as the page that is used for the recovery (the one on disk) still contains the garbage object t (not yet swept).

There are other mark-and-sweep collectors for databases [130] but they do not differ from the basic mark-and-sweep algorithm, as the one we have just described. Such algorithms are simply adapted to the particular transactional system in which they are integrated. They are very specific and not scalable.

2.4.3 Atomic Copy

This algorithm [37, 71] is based on Baker's copy GC (recall Section 2.2.3.1) and works on a single partition. Both functioning modes GC-only and incremental can be found in the literature.

Reachable objects are moved from from-space to to-space and patched accordingly. This may interfere with the stability of the store, as explained now. Imagine there is a failure while the collector is running. Then, during the failure recovery, the GC algorithm must find which objects have already been moved and which objects have already been patched. Otherwise, the pointer graph will be corrupted.

Figure 2.15 illustrates an example (for a more detailed study we refer the reader to the existing literature [71]). Suppose there is a failure after object x has been moved to to-space (a forwarding pointer was left in from-space) and only the from-space has been written to disk. Thus, after the failure the disk will not contain a valid version of x: the version in from-space has been owverwritten by the forwarding pointer, and the version in to-space is not available on disk.

Thus, if the collector were to be restarted after the failure, x would never be moved again to to-space and the pointer graph would be corrupted.

The solution to this problem consists of making the copy algorithm atomic. This requires that the recovery of a failure puts from-space and to-space in a state from which the collection can continue. This is done by applying the write-ahead log protocol [13] to the disk contents. For this purpose, both the move, and patch of each reachable object must be logged. This solution, even after being optimized, remains costly. We describe here the unoptimized version (incremental GC). The move of a reachable object proceeds as follows:

- Pin the from-space object that will be overwritten by the forwarding pointer.
- Move the object to to-space and insert the forwarding pointer in its from-space version.
- Create a log entry with the from-space and the to-space addresses of the object. Create another log entry with the to-space address of the object and its contents.
- Now, the move of the object is completed.
- After both log entries are written to disk, unpin the from-space object.

The scan and patch of a to-space object proceeds as follows:

- Pin the object to be scanned.
- Scan the object and patch its from-space pointers to to-space pointers, moving the pointed objects as needed.
- Create a log entry with the object contents.
- Unpin the object after the log entry is written to disk.

Concerning correctness, the GC algorithm must not consider to be unreachable an object x that became so due to a transaction that is still running. If the mentioned transaction aborts, object x is now reachable again (as it was before the transaction has started). Thus, the collector has to consider as part of the GC-root the log of running transactions.

Note that the collector could run as a single long user-level transaction or a series of short transactions. However, in the first case, this would lead to a long GC pause time which is highly disruptive. In the second case, it could easily lead to deadlock or to a situation in which a long user transaction could prevent the collector from making progress. For these reasons the collector runs at a level below the transactions support as sketched above.

2.4.4 Replicated Copy

The replication-based copy collector [94] (recall Section 2.2.3.1) is particularly well suited to mono-site transactional systems. This is due to the fact that while the collector moves objects to to-space, the mutator continues to access their from-space replicas, rather than the replicas in to-space.[3] When every reachable object has been moved to to-space, a flip is performed and the mutator then starts accessing the to-space replicas. If a process fails while the collector is running (before the flip) the collector simply restarts with the recovered from-space. All GC operations already done are lost.

2.5 GC in File Systems

Some file systems and databases have a mechanism to compact disk blocks. These are mostly of the time executed off-line and are explicitly invoked by the system manager. However, there is an interesting exception, the LFS disk storage manager [101]. LFS uses the concepts of log-structured file systems [56] and has an incremental copy garbage collector.

The disk is divided in large fixed-size pieces called segments. Each segment contains several blocks and a summary block. For each block in the segment, the summary block indicates the file number of the block's file and the position of the block within the file.

During normal operation, segments will become fragmented. The garbage collector task is to generate free segments, called clean segments, fom fragmented segments.

The algorithm has two phases and can be summarized as follows. In the first phase, the reachable blocks of fragmented segments are identified and read into main memory. The second phase combines the blocks into a new segment and writes them to disk.

The collector determines if a block is reachable using the summary block, the inode map, and inodes. In the first step, the segment's summary block is used to determine the file number and block offset of each block. Included in the summary block there is the file's version number from the inode map when the block was written. If the version number does not match the current version number of the file, the block is unreachable (it has been deleted or overwritten). If the previous step fails to determine the allocation status of the block, the inode and any indirect blocks that map the block of the file are examined to see if the block is still part of the file. The block is classified as being reachable if it still a member of the file.

For performance reasons it is desirable to collect the segments with the most free space, *i.e.*, with the smallest number of reachable blocks. This is due to the fact that the cost of the copy algorithm increases with the number of blocks that have to be moved to a new segment. For this purpose, LFS has a data structure

[3] The forwarding pointer does not overwrite the object in from-space.

that keeps an estimate of the number of reachable blocks in each segment. This data structure is updated when files are truncated or overwritten, and when segments are written or garbage collected.

2.6 GC in Multiprocessors

GC algorithms for multiprocessors [7, 20, 63, 92, 43] are mostly variants of the tracing type with the incremental/concurrent functioning mode

As the name indicates, multiprocessors have several processors. This capacity of processing can be used, for example, to execute the collector in a processor different from the ones where mutators run.

Multiprocessors frequently offer very efficient synchronization primitives that are used to ensure GC correctness in presence of concurrent access to the same data by mutators and the collector.

The incremental/concurrent functioning mode of tracing GC has already been addressed in the context of uniprocessor collection (recall Section 2.2.3.1). The important issue remains the same: how to ensure the correct execution of the mutator and the collector when both are competing for the same data. Therefore, we do not go into more details regarding GC in multiprocessors.

2.7 GC in DSM Systems

There is very few work on GC algorithms for DSM systems.

The most interesting problem regarding GC in these systems is coherence interference, as stated in Section 1.1. However, this issue is not even mentioned in the literature. Data is simply assumed to be coherent all the time.

Le Sergent[78] describes an extension of a copy collector first developed for a multiprocessor, to a DSM system. Object replicas are assumed to be coherent, the entire memory is collected at the same time, which is not scalable, and the garbage collector locks pages while scanning, which interferes with the coherence protocol and therefore disrupts applications.

Kordale's GC [72] is very complex and relies on a large amount of auxiliary information. This GC algorithm is based on the mark-and-sweep technique, and relies on the fact that objects must be coherent as well.

2.8 Discussion

In this section we provide some insights regarding the suitability of the GC algorithms described in this chapter, for several systems. These insights can be used as "rules of thumb" to decide which algorithm is most appropriate for a given environment. We also situate Larchant and its GC w.r.t. to the algorithms and systems presented in this chapter.

2.8.1 GC Considerations

The first observation is that no GC algorithm is universally good. In other words, depending on the system characteristics, a particular algorithm might be more appropriate than others. The best GC algorithm for a system may even change with time.

Reference counting algorithms seem to be most indicated for distributed systems because they are scalable. However, they fail to reclaim cycles.

Tracing algorithms reclaim both acyclic and cyclic garbage. However, they are not appropriate for distributed systems because they are not scalable.

Mark-and-sweep collectors must sweep the entire heap. This is a disadvantage for heaps where the majority of objects are unreachable. Then, for a uniprocessor, a copy algorithm is the most appropriate; on the contrary, if the majority of objects are reachable, a mark-and-sweep collector might be more efficient. Note that the proportion of reachable and unreachable objects may vary with time.

The basic GC algorithms must be adapted and modified in order to deal with the specific problems of persistence and distribution.

Concerning GC for DSM, there are only a few proposals and they all assume coherent objects, just as in a centralized system, therefore compromising scalability.

2.8.2 GC in Larchant

Larchant is a cached distributed store with PBR. It has a DSM mechanism that provides the illusion of a shared address space across the network, including secondary storage. Thus, at a first glance, we could envisage to chose a GC algorithm for Larchant similar to those used either in multiprocessor systems or in DSM systems.

If we apply a GC algorithm designed for multiprocessors (for instance, Appel's collector [7]) to Larchant, the overhead will be unacceptable due to synchronization and I/O costs. Similarly, the GC algorithms for DSM systems found in the literature are not appropriate for Larchant also. In both cases, the reason is that the collector needs every object to be coherent.

The GC algorithms for RPC-based distributed systems clearly indicate that a scalable solution should be based on reference counting. However, note that in contrast to Larchant, they do not consider the existence of multiple replicas of objects. In addition, while in RPC-based distributed systems there is one partition per process, in Larchant partitions and processes are orthogonal.

In conclusion, we can say that GC algorithms found in the literature are not adequate for Larchant because they do not handle replication, because they are not scalable, and because they require coherent data.

2.9 Summary

In this chapter we described the most relevant GC algorithms for uniprocessors, RPC-based distributed systems, transactional systems, log-based file systems, multiprocessors, and DSM systems.

There are two fundamental GC algorithms: reference counting and tracing. There are two tracing collectors: mark-and-sweep and copy. Reference counting algorithms do not reclaim cycles of garbage, are inherently incremental, and are scalable. Tracing algorithms reclaim cycles but are not scalable. The copy algorithm has the advantage of compacting memory, therefore reducing fragmentation, but needs to know the exact location of pointers.

When applied to distributed and/or persistent systems the two fundamental algorithms are modified in order to remain safe in presence of unreliable communication and process failures. They deal either with duplication or messages loss, or with process failures and transactional particularities such as uncontrolled page flush from the client to the server.

Existing GC algorithms do not provide a good solution for GC in Larchant because they do not address the problems raised by a cached distributed store with persistence by reachability: scalability, coherence interference, and safety in presence of replicated data possibly incoherent.

Chapter 3

Larchant Model

We start this chapter by motivating the Larchant model. We give an example that illustrates the simplicity of data sharing in Larchant, from the application programmer's point of view: sharing an object is just the matter of following a pointer.

Then, we briefly discuss the most relevant solutions currently used to support distributed data sharing: file systems, object-oriented databases, etc. We point out the disadvantages of each approach, regarding data sharing, for the applications we are interested in supporting (*e.g.*, interactive computer aided design and cooperative tools).

Then, we describe the Larchant model: a general model of a cached distributed shared store with PBR (persistence by reachability). Our focus is to characterize the interactions between applications and coherence protocols. This is the model for which the GC algorithm is designed (see Chapter 4). The model is general and we make minimal assumptions; therefore, it is widely applicable.

This chapter ends with an example of the mapping between the Larchant model and real systems.

3.1 Motivation

As stated in Chapter 1, the overall goal of Larchant is to provide a cached distributed shared store that makes data sharing simple and efficient. That is why Larchant enables programs to access and modify shared distributed persistent data without dealing with explicit I/O to a file or a database system. Applications navigate through the pointer graph by following pointers, from a persistent root, directly in virtual memory, with the system moving data between main memory and the secondary storage as needed. Larchant maintains coherent replicas of shared data, in each process participating in a cooperative task. Such local caching improves performance and data availability.

Consider the following example that illustrates the sharing behavior of the applications we are interested in: interactive computer aided design (CAD) and

cooperative tools. Suppose there is an object named "New EU Building" that constitutes a persistent root (for instance, the object name has been registered in the name server). This object contains a pointer to each of the building floors. Say that an architect, using a CAD application, wants to visualize the plan of the "4th Floor". Then, the CAD tool simply follows the pointer in the "New EU Building" that points to the "4th Floor" object. Transparently, this object is cached locally by reading it from disk. Later, if another application on some other site also needs to access the "4th Floor" object (*e.g.*, an engineer is checking the size of a heating duct) the system transparently gets a replica of the object according to some coherence protocol.

In these kind of applications, illustrated by the previous example, changes made by an application (running on some site) must be quickly integrated in the shared store, in order to be readable by other applications (running on different sites). For this sharing behavior, a mechanism supporting fine-grained client-to-client transfers, instead of a client-server-client transfer, is the most appropriate.

In addition, applications running on different sites may have to access the same data at the same time. To support this sharing efficiently, the sites where such applications are running need to cache replicas of the same data. This will become more evident as the physical memory on each site grows and data remains in main memory for longer periods.

Another characteristic of the applications we envisage to support is that they may require high performance data manipulation in main memory. This happens, for instance, when a CAD tool performs civil engineering computations.

3.2 Current Solutions for Distributed Data Sharing

In the past, distributed data sharing has been supported through file systems, object-oriented databases, RPC-based systems, and distributed shared memory (DSM). In general, such systems are inefficient, hard to use, or poorly adapted to the sharing behavior of our class of applications. With these systems, applications programmers must deal with three very different Application Programming Interface (API) sets, depending whether the data is in main memory, on secondary storage, or on a remote site. In contrast, applications above Larchant deal only with memory, using a single, simple and familiar API.

3.2.1 File Systems

Nowadays, distributed file systems [104, 125] are the dominant technology for the distributed sharing of persistent data.

A problem with file systems is that they do not support the complex data types needed by the applications we are interested in. Since files are unstructured, the representation of data in the file system does not match the type system used by the applications programming language. Therefore, applications programmers must worry about I/O format conversion (marshaling and unmarshaling

data). This is tedious, error-prone, and detracts programmers from their most important task: application design and development.

Finally, being client-server architectures, distributed file systems do not support fine-grained client-to-client transfers as desired for the sharing behavior we are considering (described in Section 3.1).

3.2.2 Object-Oriented Databases

One alternative for the distributed sharing of persistent data is the object-oriented database (OODB) technology [132]: O_2 [39], Thor [83], and GemStone [24] are only a few examples.

In contrast with file systems, OODBs provide excellent support for complex data types. In other words, they preserve the structure of data (no marshaling is needed). However, they are very heavyweight, and often come with their own specialized programming language.

OODBs are optimized towards searching for relevant data (through indexing, query languages, etc.) possibly with modest updates and minimal cooperative read/write interaction. Thus, they take the view that data resides mostly on secondary storage, with main memory being used as a temporary scratch buffer. In contrast, our applications need high performance data manipulation in main memory.

Like distributed file systems, existing distributed OODBs are limited by their client-server architectures: no fine-grained client-to-client transfers. In addition they do not support large-scale sharing.

3.2.3 RPC-based Systems

Remote Procedure Call (RPC) [17] is a basic communication mechanism that forms the basis for the client-server model [2, 66, 110, 114]. RPC solves the problems of identification and remote access. However, every remote data access is burdened with communication to the server, which becomes a performance and availability bottleneck. This makes the client-server architecture inadequate for interactive CAD and cooperative applications. In addition, RPC does not support coherence of data viewed by multiple clients. Finally, it imposes an interface definition language to program remote data access, separate from the programming language, and the client-server model is often unnatural.

3.2.4 DSM Systems

A distributed shared memory (DSM) [80] is a software device emulating a single shared memory over a network. DSM systems maintain the illusion of a distributed shared memory by synchronizing data access and moving data between processes when required, transparently to applications.

A DSM supports client-to-client transfers and replication, as needed by interactive CAD and cooperative applications. Data in memory is accessed via the type system of some programming language, and is simple, fast and intuitive. This makes application programming very easy since it provides the same memory abstraction as in the centralized case.

DSMs [14, 28, 68] have been mostly used to run parallel algorithms on networks of workstations. However, they have not been used for general programming, in part because of the lack of support for persistence by reachability.

3.2.5 Cached Distributed Shared Stores

We claim that the simplicity of programming provided by a DSM mechanism can be extended to distributed applications with persistent objects. The Larchant model is that of a cached distributed shared store resulting from the combined approach of DSM and PBR. It offers a shared memory spanning every site in a network, including secondary storage.

This model offers transparent distribution and persistence. Applications have uniform access to any object in the system independently of its location. The model hides both the distinction between local and remote data, and the distinction between short-term and long-term storage. Applications navigate through the shared store by following pointers in virtual memory. The system moves the necessary data between main memory and secondary storage or between the main memory of remote sites, according to application needs.

We have found very few systems extending DSM with persistence in the literature. One of them is Casper [121]. It takes advantage of the external pager mechanism [65] of the Mach operating system [1] to provide transparent access to data in secondary storage. Given that its GC specification is sketchy [70], it is not clear how PBR could be supported.

Feeley et. al [49] describe a transactional DSM that supports fine-grained distributed data sharing in cooperative applications. However, there is no support for PBR.

3.3 Larchant Model

The main goal of this description of the Larchant model is to establish the environment in which the GC algorithm execute (see Chapter 4). The model describes the interaction among applications and coherence of data. We only present those operations that are relevant w.r.t. GC.

The model is general in the sense that it makes minimal assumptions: *e.g.*, we tolerate arbitrary writes and non-coherent data. Many events traditionally associated with coherence management are not present; some because they are not relevant to PBR (*e.g.*, non-pointer writes); others because our GC algorithm is independent of them (*e.g.*, DSM tokens or locks) as will be made clear in Chapter 4. Given its minimal assumptions and consequent generality,

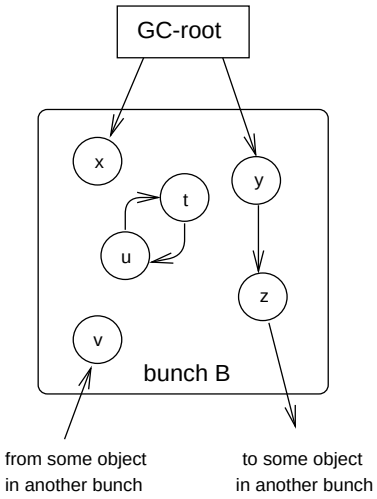


Figure 3.1: *A bunch with objects inside.*

we believe the Larchant model is capable of describing a wide range of cached distributed shared stores.

The Larchant model has the following components: memory model, network model, process model, mutator model, and coherence model. We describe them now.

3.3.1 Memory Model

In Larchant the memory is structured at two levels of granularity (see Figure 3.1). *(i)* By definition, the *object* is the unit of allocation, deallocation, identification, and coherence (see Section 3.3.5); an object may contain any number of references. *(ii)* The memory is divided in partitions called *bunches*; a bunch is the unit of caching and collection; it contains any number of objects. An object resides entirely within a single bunch.

A reference is either nil (value zero) or points to an object. We assume that a reference is a plain pointer; applications may arbitrarily assign a reference with a legal pointer value (see Section 3.3.4). This is a worst case from the GC point of view.

Hereafter, objects are noted x, y, z, etc. A pointer variable ptr inside object x is noted x.ptr. The address of object x is noted @x. Bunches are noted uppercase B, C, etc.

Objects are accessed locally via caching. They can be replicated on multiple sites. Their coherence is maintained with a coherence protocol (see its model in Section 3.3.5).

3.3.2 Network Model

A distributed system is a set of processes communicating by messages, with no shared memory. An atomic event E at some process i is noted $<E>_i$.

Events at some process occur in some serial order. We use the classical relation *happens before* [74].

For any message M, we note $<send.M>_i$ the sending event at process i, and $<deliver.M>_j$ the delivery and processing of M at its receiver process j. We assume causally-ordered communication [100].

3.3.3 Process Model

A process may contain any number of threads. It is composed of a mutator, a collector, and a *coherence engine* implementing the coherence protocol (see Section 3.3.5). Note that some processes may lack a mutator or a collector; for example, a system server does not have a mutator.

A mutator dynamically modifies the pointer graph: it creates objects and manipulates references (recall Section 2.1). The collector observes the pointer graph in order to identify and reclaim unreachable objects. The coherence engine implements the coherence protocol.

Mutators from different processes do not send messages directly to each other. They communicate by causing and observing updates on the shared store. Messages between processes flow only on behalf of collectors or coherence engines.[1]

In the following sections we describe the mutator model and the coherence model. The collector model is described in Chapter 4.

3.3.4 Mutator Model

Mutators modify the pointer graph: they create objects, dereference pointers, and assign pointers. Pointer assignments modify objects reachability.

As described in Section 2.1, a pointer assignment can result in: *(i)* creation of a new reference (to an object just created), *(ii)* duplication of an already existing reference, and *(iii)* discarding a reference to an object. Note that cases *(ii)* and *(iii)* may happen as the result of a single assignment operation.

A duplication of pointer y.ptr, pointing to object z, such that x.ptr will also point to z, executed by the mutator of a process i, is noted $<x.ptr := y.ptr>_i$. This operation is atomic at process i. This has the significance that the model does not allow hidden temporary pointer variables such as registers and stacks. This does not restrict the generality of the model, because it is always possible to expose temporary variables in the shared store.

[1] Note that this is not a restriction; in fact, our GC algorithms can be equally applied to RPC-based distributed systems. However, this is out of the scope of this document.

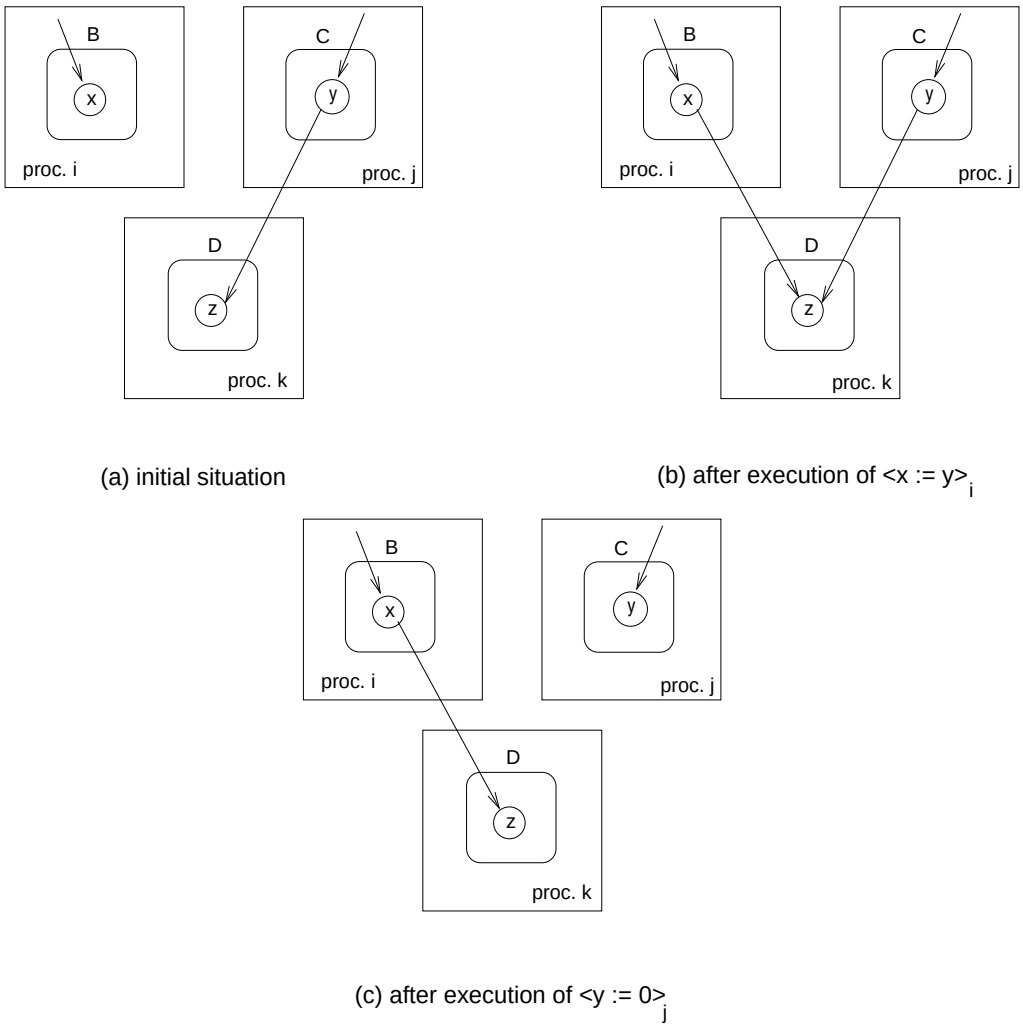


Figure 3.2: *Prototypical example of mutator execution.*

Object creation is a special case of pointer assignment. We assume that all objects always existed; an object creation consists of setting up a path from the persistent root to that object by means of an assignment operation.

When there is no ambiguity, we make the simplification that objects have only one pointer inside, and identify `x.ptr` with `x`. Note that this simplification does not restrict the validity of the model w.r.t. GC.

We say that an object `y` points to an object `z` when `y` contains `@z`. We say that an object `x` is nil when its pointer variable does not point to any object, *i.e.*, it contains the value zero (nil pointer).

An assignment operation performed by process `i`, such that `x` is made to point to the object pointed from `y`, is noted $\langle x := y \rangle_i$.

As will be shown in Chapter 4 (when proving the correctness of the distributed garbage collection algorithm) from the GC perspective, any mutator operation can be reduced to the following *prototypical example* (see Figure 3.2). Consider

two objects x and y located in bunches B and C, respectively. Initially x is nil and y points to object z located in bunch D. Now, mutators within processes i and j execute the following operations: $<x := y>_i$, $<y := 0>_j$, such that at the end x points to z and y is nil.

We say that object x is *GC-dirty* after being modified. We say a bunch is GC-dirty if it contains a GC-dirty object. An object remains GC-dirty until its contents are scanned looking for pointers to other objects (see Section 4.3.2).

3.3.5 Coherence Model

The coherence engine provides support for local data coherent access via caching, according to the coherence model.

As mentioned in Section 3.3.1, by definition, an object is the granule of coherence. However, the size of a coherence granule is not relevant for the present discussion; in fact, a coherence granule might contain several objects. In Section 5.6 we discuss how the GC algorithms are (not) affected when a coherence granule contains several objects.

Note that the granule of caching, a bunch, is different from the coherence granule, an object. The reason is that applications do not necessarily need coherent access to every object in a bunch. For some applications it is sufficient to access incoherent object replicas. This difference of caching and coherence granularity also contributes to avoid the problem of false sharing [12].

To simplify notation, we assume that all data is cached by every process. In order to model the case where an object is cached only in some processes, we represent an object not cached in some process i, as being cached with value zero (nil pointer).

A bunch B can be cached by multiple processes. The image of B in process i is called i's replica of B and is noted B_i. Similarly, the replicas of x and y observed by processes i and j are noted x_i and x_j, respectively.

The dissemination of an object x from a process i to process j is done by a *propagate* message; the corresponding events are noted $<\text{send.propagate}(x, i{\rightsquigarrow}j)>_i$ and $<\text{deliver.propagate}(x, i{\rightsquigarrow}j)>_j$. The former event puts the contents of x_i in the message, and the latter copies the object's contents from the message into x_j. (This makes x_j GC-dirty.) Each of these events is atomic w.r.t. other operations in the corresponding process.

At any point in time, for each object x there is a single process that is assumed to cache the most recent version of x. We call this process the *owner* of x. The owner of an object may change. The transfer of x's ownership from process i to process j is done by a message; the corresponding events are noted $<\text{send.owner}(x, i{\rightsquigarrow}j)>_i$ and $<\text{deliver.owner}(x, i{\rightsquigarrow}j)>_j$. The effect of the former operation is that sender process i relinquishes x ownership; the effect of the latter operation is that receiving process j becomes the new owner of x. Each of these events is atomic w.r.t. other operations in the corresponding process. As will be made clear in Chapters 4 and 5 these events (ownership transfer)

are not relevant for the GC algorithm. However, we take advantage of them for implementation purposes.

Other coherence operations can be modeled as special cases of the above ones. See Section 3.4 for more details on this.

In conclusion, our coherence model is based on a minimal set of operations that are common to all known coherence protocols: the owner process of an object x caches its most recent version and propagates x's updates to other processes. Non-owner processes caching coherent versions of x (equal to the owners's version) can also propagate x to other processes. Our model does not dictate how the owner is chosen or when propagates are sent. (In a practical system, coherence operations are caused by mutator activity.) It prescribes no particular coherence or concurrency control algorithm as long as the ordering is consistent with causality. In fact, we even allow conflicting writes at different processes; however, only the owner gets to propagate its updates.

3.4 Mapping of Real Systems to the Larchant Model

In this section we show how the Larchant model applies to a real system: DSM with the entry-consistency protocol [14]. (This is the coherence protocol that was implemented in Larchant; more details in Chapter 5.)

This protocol provides the traditional model of multiple readers and a single writer: at any point in time, there can either be several read tokens, or one exclusive write token associated with a shared object. Processes holding a read token are assured a coherent replica of the corresponding object.

Every shared object has a owner, which is either the process currently holding the object's write token, or the process that last held the write token. A write token can only be obtained from the object's owner. A read token can be obtained from any process already holding a read token.

A token is obtained by performing a read or write token *acquire* operation and is freed by the corresponding *release*. The acquisition of a write token for an object implies the *invalidation* of every readable replica of that object in other processes. Each process receiving an invalidation message acknowledges it via a reply.

It is clear that the acquisition of a read token in the entry-consistency protocol is modeled by the propagate operation. The acquisition of a write token is modeled by the operations propagate and ownership transfer.

The invalidation of an object in the entry-consistency protocol is modeled as a special propagate operation in which the propagated object is nil.

3.5 Summary

In this chapter we briefly discussed the most relevant solutions currently used to support distributed data sharing: file systems, object-oriented databases,

etc. In general, such systems are inefficient, hard to use, or poorly adapted to the sharing behavior of our class of applications.

Being client-server architectures, distributed file systems and existing distributed OODBs do not support fine-grained client-to-client transfers, as desired for the sharing behavior we are considering. RPC-based distributed systems impose an interface definition language to program remote data access, separate from the programming language, and the client-server model is often unnatural. Existing DSMs have not been used for general programming, in part because of the lack of support for persistence by reachability.

Then, we described the Larchant model. Applications above Larchant, deal only with memory, using a single, simple, and familiar API (Application Programming Interface); sharing an object is just the matter of following a pointer.

An important aspect to retain is the set of mutator and coherence operations: pointer assignment and object propagation. A mutator dynamically modifies the pointer graph by executing assignment operations. In the coherence model, the owner process of an object x caches its most recent version and propagates x's updates to other processes. (The owner of an object may change.) Non-owner processes caching coherent versions of x (equal to the owners's version) can also propagate x to other processes. Our model does not dictate how the owner is chosen or when propagates are sent.

These operations will be of great value in Chapter 4, when describing GC in Larchant, because they establish the environment for which the GC algorithm is designed.

Chapter 4

Garbage Collection in Larchant

In this chapter we describe the GC algorithm designed for Larchant. We start by presenting the fundamental problems that must be solved, and an outline of the corresponding solutions. Then, we give a general overview of the GC algorithm. A detailed description follows, which includes the GC model, the collection of replicated bunches, and the reclamation of cycles of garbage. This chapter ends with a formal proof of the correctness (safety and liveness) of the distributed GC algorithm.

The GC algorithm is designed for the Larchant model presented in the previous chapter. In Chapter 5 we show how the GC algorithms apply to a real system with a particular coherence protocol (entry-consistency [14]).

The main issues we tackle in this chapter are related to scalability, asynchrony, and orthogonality to coherence, of the GC algorithm. In addition to the basic requirement of good performance, these aspects raise interesting problems in terms of safety.

A fundamental requirement of any GC algorithm is to be correct, *i.e.*, safe and live, and ideally complete (recall Section 2.1 on terminology). If the GC algorithm fulfills this requirement then it ensures referential integrity, and reclaims garbage. In addition, safety, liveness, and completeness must be ensured without degrading significantly applications performance. In other words, not only the overall cost of GC must be as small as possible but also, interactive applications must not be disrupted by long continuous pauses that would be annoying to the user.

Ideally, a GC algorithm would be complete, thus it would eventually reclaim *all* unreachable data. Reference-counting algorithms, for example, are incomplete, since they do not reclaim cycles of garbage. The only known provably complete algorithms are based on global tracing, are synchronous, and require coherent data (recall Chapter 2). This does not scale, interferes with applications coherence needs, causes extra I/O, and does not have good performance. Apparently, the problem is hopeless. However, as we will show in this chapter, it is possible

to do some useful work without performing a global trace and therefore to avoid its drawbacks.

4.1 Problems and Outline of Solutions

In this section we describe the specific problems the GC algorithm has to solve. We also outline the corresponding solutions. They concern scalability, coherence interference, extra I/O, and performance.

A common thread to our solutions is that GC is opportunistic, *i.e.*, it does not cause events that mutators would eventually cause; the collector waits for them to happen and then takes advantage of it. This will become clear afterwards.

4.1.1 Scalability

We claim that perfect completeness is not feasible in a large-scale system. Thus, we propose an approximate solution that is not provably complete, but which we believe adequate for all real-life situations because it takes advantage of locality (as will be explained in Section 4.7).

Our solution combines partitioned tracing (within bunches) with reference-listing (across bunch boundaries). Each bunch replica is collected independently of other bunches and of other replicas of the same bunch; incoming cross-bunch pointers are members of the GC-root. A bunch replica is collected in the process where it is currently cached with a tracing algorithm. The *intra-bunch* collector is the system component that identifies and reclaims garbage inside a bunch. The *cross-bunch* collector is the system component that manages the cross-bunch reference-lists; it does not reclaim objects, it simply updates the mentioned lists.

This algorithm is not complete, since it does not reclaim cross-bunch cycles of garbage. Apparently, the only way of reclaiming such cycles is to perform a global trace, which is not scalable, and would generate a huge amount of extra I/O. The problem of reclaiming cross-bunch cycles of garbage is addressed below, in Section 4.1.3.

4.1.2 Coherence Interference

The problem of coherence interference is illustrated by the following examples.

Imagine the intra-bunch collector scanning an object replica to find its internal pointers, and follow them to continue the traversal of the pointer graph. Then, the question is: does the collector need a coherent replica of the object being scanned? If the answer is yes, as it happens with existing GC algorithms (recall Chapter 2) it is clear that the collector will be interfering with applications coherence needs.

Now, suppose that the intra-bunch collector implements a copy algorithm. Then, when patching a pointer inside an object replica (to reflect the new

location of a reachable descendent that has already been moved) should the collector require exclusive write-access to modify the object replica? If exclusive write-access is needed, read-access to all other replicas of the object being patched would have to be invalidated, therefore interfering with the coherence needs of applications.

Another example of the coherence interference problem is related to safety. Imagine an object z in bunch D that is pointed from several replicas of an object y (in bunch C) which are not necessarily coherent. Then, if the collector, at some point in time, forces every replica of y to be coherent, in order to conclude about z's reachability, once again, it will be interfering with applications coherence needs.

As stated in Chapter 2, GC algorithms found in the literature do not handle the coherence interference problem. They implicitly assume that either there are no replicas or all replicas are kept coherent at all times. The adoption of such solutions in a cached distributed shared store, such as Larchant, would incur into unacceptable costs due to communication and synchronization overhead.

To avoid the coherence interference problem, the collection of a bunch replica must make progress even if the objects it contains are not coherent (in the process where the intra-bunch collection is taking place). We achieve this goal by making our GC algorithm orthogonal to coherence. Thus, the garbage collector can work with objects that are incoherent for applications' purposes, while being safe and live.

The fundamental observation that guided our design is that GC coherence needs are less strict than applications' [55]. This enables GC to be performed without interfering with applications' coherence needs and requiring very little synchronization or communication. More details on this in Section 4.4.2.

4.1.3 Extra I/O

GC based on global memory tracing incur into extra I/O. This is due to the fact that such collectors traverse the pointer graph even when the corresponding objects are not cached in main memory.

In contrast, in Larchant (as mentioned in Section 4.1.1) a bunch replica is collected in a process where it is currently cached by application request. Therefore, the collection of a bunch does not cause extra I/O.

However, a difficulty arises when reclaiming cross-bunch cycles of garbage because the only known provably complete algorithms are based on global tracing and employ a global synchronization. This causes a large amount of extra I/O.

Our solution to reclaim cross-bunch cycles of garbage, without incurring into extra I/O, consists of extending the tracing performed by the intra-bunch collector to include several bunches at once. Bunches are grouped and collected as if they were a single bunch. Thus, memory partitions change dynamically and seamlessly, and independently in each process, in order to reclaim cycles of unreachable objects.

The significance of group collection is that any arbitrary subset of the store can be collected, in a single process, independently of the rest of the memory. The choice of a group to be collected is heuristic, and should maximize the amount of garbage reclaimed and minimize the cost. In Section 4.7 we describe the heuristic used in Larchant. It takes advantage of locality, therefore avoiding extra I/O.

4.1.4 Performance

There are two fundamental difficulties concerning GC performance. First, ensuring that GC pause times remain sufficiently small, in order to be unnoticed bu the user, independently of bunch size, proportion of reachable objects, and number of bunch replicas. Second, keeping track of cross-bunch references without instrumenting every pointer assignment with a write-barrier, to find out if a new cross-bunch pointer has been created. (Recall that when a bunch replica is collected, the incoming cross-bunch pointers are members of the GC-root.)

To solve the first problem, the intra-bunch collector runs concurrently with mutators, and is orthogonal to coherence. To solve the second problem, keeping track of cross-bunch pointers without instrumenting every pointer assignment, the cross-bunch collector runs asynchronously w.r.t. applications and certain GC operations are safely delayed.

Hence, in Larchant both the intra-bunch and the cross-bunch collector execute concurrently and asynchronously w.r.t. applications. This functioning mode raises important issues in terms of safety that will be addressed intuitively in Sections 4.4 and 4.5, and formally in Section 4.10.

4.1.5 GC Algorithms

The intra-bunch collector uses a tracing algorithm. Now, the question is what type of tracing: mark-and-sweep or copy. As mentioned in Chapter 2 both algorithms have their pros and cons. In particular, collection time depends of the heap size and the proportion of reachable objects, respectively.

We believe that some applications will benefit from a copy collector, while for others a mark-and-sweep algorithm is the most appropriate. Such a choice depends on the application programming language, memory usage, specific application requirements in terms of GC pause time, etc. Note that even for a single application, the best GC algorithm may vary with time: we could imagine an application in which the collector would be a mark-and-sweep until memory gets too fragmented, and then changing automatically to a copy algorithm. The bottom line is that there is no universal solution.

For this reason, we decided to support both mark-and-sweep and copy algorithms. It is up to the application programmer to chose the most appropriate GC algorithm.

4.2 General Overview

The main aspects of GC in Larchant can be summarized as follows:

- Each process collects independently and asynchronously from other processes.
- A bunch is collected independently and asynchronously from the rest of the store, with a tracing algorithm (intra-bunch collector) that runs concurrently with mutators; GC between bunches (cross-bunch collector) is based on reference-listing.
- To reclaim cycles of garbage, bunches are dynamically grouped and collected at the same time in some process; groups are formed with an heuristic that avoids extra I/O.
- The GC algorithm is orthogonal to coherence, *i.e.*, each bunch replica is collected independently and no coherence operation is needed for GC purposes.
- The reference-listing algorithm is asynchronous w.r.t. applications, and GC specific messages are exchanged asynchronously in the background (we assume causal delivery).

To enable a bunch to be collected independently from the rest of the memory, a bunch keeps track of cross-bunch pointers. An outgoing pointer is described by a *stub* and a incoming pointer by a *scion* (see Figure 4.1).[1] Each bunch replica has its own replica of stubs and scions. The number of cross-bunch pointers to some object is safely approximated by the number of scions to that object.

By considering a bunch's scions as its GC-root, a bunch can be collected independently of others. An object y in bunch C pointed by a scion will not be reclaimed; y is said *protected*. The result of collecting a bunch is a set of reachable objects and a new set of stubs. Objects not in the reachable set are reclaimed.

By comparing the sets of stubs before and after an intra-bunch collection, the cross-bunch collector discovers which cross-bunch pointers were created and which stubs no longer exist. In other words, stubs that were not in the stub set before the collection but are in the (new) stub set after the collection correspond to cross-bunch pointers that were created by the mutator. Thus, the corresponding scions must be created. Stubs that were in the stub set before the collection but are not in the (new) stub set after the collection correspond to cross-bunch pointers that no longer exist. Thus, the corresponding scions may be deleted. In Section 4.5 we describe when and how scions are created and deleted.

[1] The names stub and scion are inspired by the similar structures found in the SSP (stub-scion pair) Chain message-passing system [108]. In contrast to SSP Chains, Larchant's stubs and scions are not indirections participating in the mutator computation, but simply auxiliary data structures.

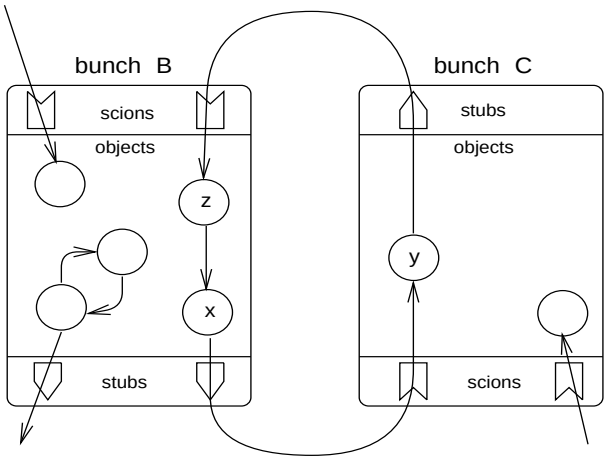


Figure 4.1: *Two bunches containing objects, stubs, and scions.*

Safety is ensured by ordering constructive events (creation of scions) always before destructive events (deletion of scions or garbage reclamation) that may depend on them, and the use of causal delivery. This a fundamental aspect of our distributed GC algorithm.

Finally, note that in Larchant the root of persistence is simply a special scion that is never deleted. Every reachable object can be reached (directly or transitively) from this special scion.

4.3 GC Model

In this section we present the GC model, *i.e.*, we describe the operations that manipulate the stubs and scions, and the operations that characterize a tracing intra-bunch collection. Then, based on this set of operations, in the rest of this chapter we describe in detail the intra-bunch and cross-bunch algorithms, and prove their correctness.

4.3.1 Stubs and Scions

An outgoing cross-bunch pointer is described by a stub, and an incoming cross-bunch pointer by a scion (see Figure 4.1). Thus, for each cross-bunch pointer there is a stub in the source bunch and a scion in the target bunch. Each bunch replica has its own replica of stubs and scions. Note that stubs and scions are simply auxiliary data structures; they are not seen by applications code.

A cross-bunch pointer from object $\mathsf{x} \in \mathsf{B}$ to object $\mathsf{y} \in \mathsf{C}$, is noted $\mathsf{Bx}\rightarrow\mathsf{Cy}$. A pointer $\mathsf{Bx}\rightarrow\mathsf{Cy}$ observed by process i is noted $(\mathsf{Bx})_\mathsf{i} \rightarrow \mathsf{Cy}$.

A stub describing $\mathsf{Bx}\rightarrow\mathsf{Cy}$ is noted $\mathsf{stub}(\mathsf{Bx}, \mathsf{Cy})$. The scion for the same cross-bunch pointer is noted $\mathsf{scion}(\mathsf{Bx}, \mathsf{Cy})$. The cross-bunch collector manipulates stubs and scions according to the reference-listing algorithm (see Section 4.5).

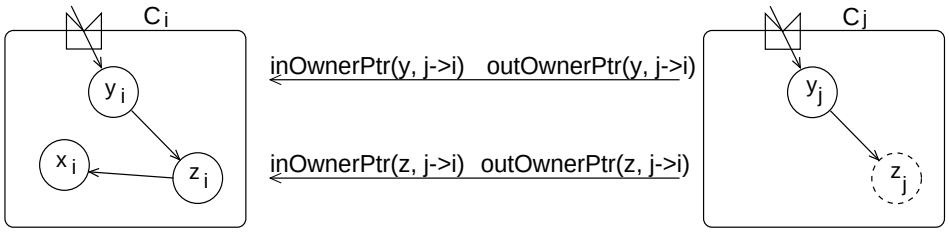


Figure 4.2: *Bunch replicas with outOwnerPtrs and inOwnerPtrs. The dashed line indicates that* z *is not cached in* j.

The creation of stub(Bx, Cy), executed in process i, is noted <create.stub(Bx, Cy)>$_i$. Note that there is no operation for deleting a stub. As mentioned in the previous section, a stub is implicitly deleted by replacing a set of stubs with a new one resulting from an intra-bunch collection.

The creation and deletion of a scion involves two bunches: the source and the target of the corresponding cross-bunch pointer. The operation of creating scion(Bx, Cy) is done with a message, thus it is constituted of two events: <send.create.scion(Bx, Cy)>$_i$ issued at the source process i (caching the source object x) and <deliver.create.scion(Bx, Cy)>$_j$ at the receiving process j (caching the target object y).[2] These two operations are atomic in the process where they are executed. Similarly, the operation of deleting scion(Bx, Cy) is constituted by <send.delete.scion(Bx, Cy)>$_i$ and <deliver.delete.scion(Bx, Cy)>$_j$.

There is a special type of stubs and scions called, *outOwnerPtr* and *inOwnerPtr*, respectively. These special stubs and scions are used only between replicas of the same bunch. Each bunch replica has its own set of outOwnerPtrs and inOwnerPtrs.

There is a pair outOwnerPtr-inOwnerPtr for each object that is replicated or that is directly accessible from a replicated object, as illustrated by Figure 4.2: objects y_j (replicated) and z_j (directly accessible from a replicated object); object x is not accessible by following a pointer in process j, thus there is no outOwnerPtr-inOwnerPtr associated to x. Note that a pair outOwnerPtr-inOwnerPtr points always to the owner of the concerned object (possibly through some indirection).

The inOwnerPtr for object y, owned by process i, replicated in process j, is noted inOwnerPtr(y, j→i). The corresponding outOwnerPtr is noted outOwnerPtr(y, j→i).

These special stubs and scions are needed to ensure the safety of the intra-bunch collector, as will be explained in Sections 4.3.3 and 4.4. Basically, inOwnerPtrs are part of the GC-root when collecting a bunch replica.

[2] If the process owning the source and target objects is the same, the message becomes a simple procedure call; we assume instantaneous delivery.

4.3.2 Tracing a Bunch

An intra-bunch collection of bunch replica B_i executed at process i is noted $GC_i(B)$. We do not assume that tracing a bunch is atomic.

An intra-bunch tracing collector uses the *scan* operation. For some object x, scan determines what other objects y, z, ..., are pointed to by x. A scan operation executed in process i on object replica x_i is noted $scan_i(x)$. When an object is scanned, for each one of its outgoing cross-bunch pointers, the corresponding stub is created.

Note that an object can be scanned even if its enclosing bunch is not being collected. (The circumstances in which this happens will become clear afterwards.)

A mark-and-sweep collector marks reachable objects. The operation of marking object replica x_i in process i is noted $<mark(x)>_i$.

A copy collector moves objects (to compact memory and reduce fragmentation) and patches pointers with the new addresses. The operation of moving a reachable object x to a new address @x', is done with a message, thus it is constituted by two events: $<send.move(x, @x')>_i$ and $<deliver.move(x, @x')>_j$, where i is the sending process and j the receiver. (If i equals j, the message becomes a simple procedure call.) A patch operation executed in process i, concerning y_i pointing to x that has been moved to @x', is noted $<patch(y, @x')>_i$. (In Section 4.4.2 we explain how every replica of a same object is moved to the same address.)

Unreachable objects are reclaimed. A reclaim operation executed in process i concerning object replica x_i is noted $<reclaim(x)>_i$. In a mark-and-sweep collector this is implemented by adding the unreachable object's memory space to a free list (during the sweep phase). In a copy collector, all unreachable objects are reclaimed at once when the flip is performed.

The n^{th} intra-bunch collection of a bunch replica B_i, scans reachable objects in B_i, therefore creates stubs for each outgoing pointer. These stubs are inserted in a new set of stubs noted $stubs^n(B_i)$. The set of stubs before the n^{th} $GC_i(B)$ is noted $stubs^{n-1}(B_i)$.

Finally, a GC-dirty object (defined in Section 3.3.4) remains GC-dirty until scanned. Thus, when an object is GC-cleaned, the stubs corresponding to its outgoing cross-bunch pointers are created. A GC-dirty bunch remains GC-dirty as long as it contains a GC-dirty object.

4.3.3 Union Rule

The union rule is common to the intra-bunch and cross-bunch collection algorithms. It ensures safety in presence of replication. The cross-bunch GC algorithm respects this rule through the use of stubs and scions. The intra-bunch GC algorithm respects this rule through the use of inOwnerPtrs and outOwnerPtrs.

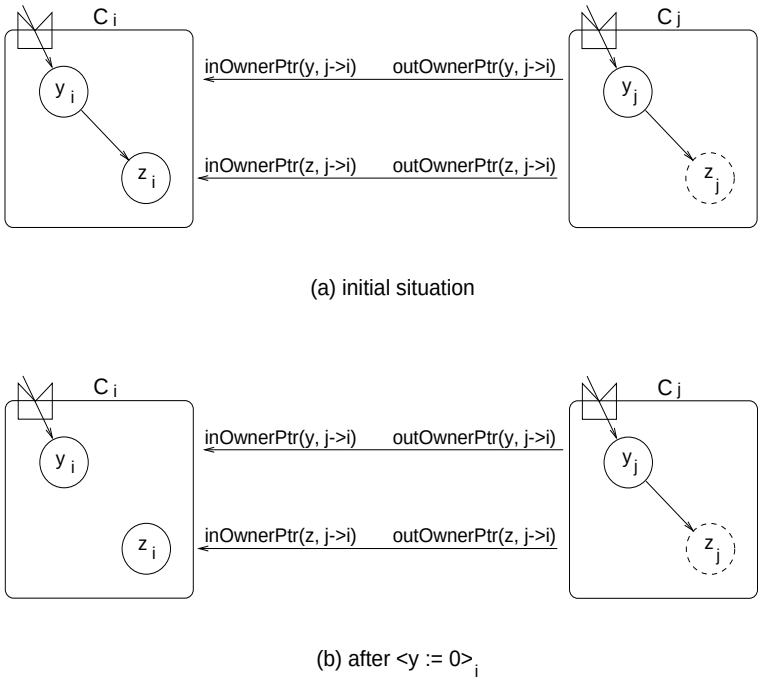


Figure 4.3: *Example illustrating the need of union rule (single bunch). The dashed line indicates that* z *is not cached in* j.

We motivate the need for the union rule with two examples. The first one considers a single replicated bunch. The second example considers two different bunches with cross-bunch pointers.

Suppose a bunch C containing objects y and z. Bunch C is cached in processes i and j. Initially y_i points to z_i; then, y is propagated to j (Figure 4.3-(a) illustrates this situation). From now on, the mutator in j can access z by following the pointer in y_j. Now, imagine that due to mutator activity in i, z_i becomes unreachable (*e.g.*, mutator executes $<y := 0>_i$) as shown in Figure 4.3-(b). Then, if $GC_i(C)$ runs without considering the set of inOwnerPtrs as members of the GC-root, z_i is unsafely reclaimed. This shows that z_i may be reclaimed only after it becomes unreachable in *all* processes caching a replica of bunch C.

The second example is similar to the previous one with the following difference: z is allocated in bunch D cached in process k (see Figure 4.4). Initially both replicas of y point to z. Now, imagine that due to mutator activity in i, z becomes unreachable from y_i (*e.g.*, mutator executes $<y := 0>_i$). In this case, object z is still accessible from y_j. Obviously, object z may be reclaimed only after it becomes unreachable from all replicas of the source object y.

In short, a target object can be reclaimed only if it is not reachable from the union of *all* replicas of its source objects (independently of their location, either in the same or in a different bunch). We call this the *union rule*.

The intra-bunch GC algorithm enforces the union rule by considering every inOwnerPtr as member of the GC-root. An object will be reclaimed only when

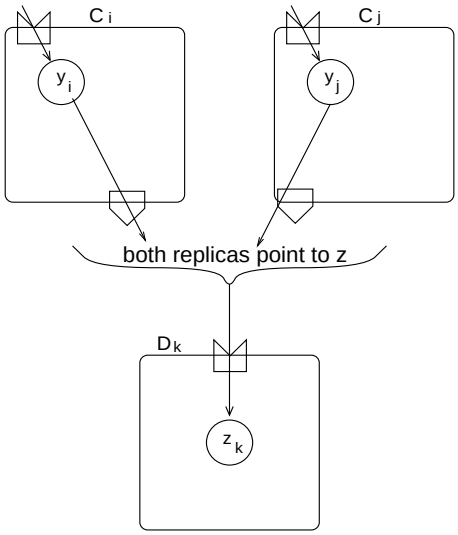


Figure 4.4: *Example illustrating the need of union rule (two bunches).*

it becomes unreachable, not only from any object locally cached, but also no longer protected by any inOwnerPtr.

The cross-bunch GC algorithm enforces the union rule by considering every stub from all replicas as members of the GC-root. Thus, the last scion pointing to an object can be safely deleted only after the corresponding outgoing pointers have disappeared in all replicas of the source object.

In Sections 4.4 and 4.5 we will explain with more detail how the union rule is enforced by the intra-bunch and cross-bunch algorithms, respectively.

4.4 Intra-Bunch GC

In this section we describe the intra-bunch GC algorithm. Both mark-and-sweep and copy collectors are addressed. We first consider there is no replication, *i.e.*, we describe the collection of a single bunch cached only in a single process. Then, we present the collection of a bunch replicated in multiple processes.

4.4.1 GC without Replication

An intra-bunch collection for a non-replicated bunch proceeds as follows. The GC-root is the bunch's set of scions. Any object pointed at directly from a scion is reachable, therefore it is marked (or moved in the case of copy GC).

Every marked (or moved) object is scanned for pointers. If such an object points to another object inside the same bunch, the intra-bunch collector transitively marks (or moves) the pointed-to object. If it points outside the enclosing bunch, the collector creates a stub in the stub set with the highest index. Thus, the result of collecting a bunch is a set of reachable objects (those that were marked or moved) and a new set of stubs. Objects not in the reachable set are garbage.

The intra-bunch collector (both mark-and-sweep and copy algorithms are considered) is based on the algorithm from Nettles and O'Toole (replication-based technique, described in Section 2.2.3.1) and runs concurrently with the mutator. Once the intra-bunch collector starts running, any object modification done by the mutator is detected (see how in Section 5.4.3) and the object's address is registered in a log, called *GC-log*. When the collector flips, every object in the GC-log is scanned again (and moved again, in the case of a copy collector).[3]

The GC finishes when every reachable object has been marked (or moved and patched) and the collector has flipped. (Note that, with the mark-and-sweep collector, the sweep phase is also done concurrently with the mutator, after the flip.)

Any tracing algorithm could be used for the intra-bunch GC. However, when using a concurrent copy algorithm, the replication-based technique helps to avoid disruptiveness as will be made clear in Section 4.4.2.3.

4.4.2 GC with Replication

The general algorithm described in the previous section applies equally to the intra-bunch collection of a replicated bunch. The main additional issues in this case are: *(i)* to ensure safety in presence of replication (via the union rule), and *(ii)* to avoid coherence interference, *i.e.*, the collector must be able to collect a bunch replica containing incoherent objects; in other words, no coherence operations should be issued on behalf of GC.

Consider a bunch replicated in multiple processes. Each process runs concurrently to the mutator an intra-bunch collector thread for the bunch. This raises the following questions:

1. Must these collectors synchronize with each other?
2. Does scanning need coherent data?
3. Is it necessary to synchronize the flip?
4. When using a copy GC algorithm:
   (a) Is it necessary to synchronize the collectors to decide where to move an object?
   (b) Is it necessary to perform some coherence operation before or after patching an object's internal pointers?

A "yes" answer to any of these questions would impact efficiency. In the rest of this section we will show that, conditions exist where the answers are all "no". As a consequence, the intra-bunch GC algorithm does not compete with

[3] In the case of a mark-and-sweep collector, the flip is the instant when the mark phase is finished and the sweep can start. With a copy collector, the flip consists of exchanging the roles of from-space and to-space. In both cases, mutators are halted for the duration of the flip.

applications for coherent data, there is no synchronization between collectors and mutators or between different collectors, and GC specific messages are asynchronous and exchanged in the background. The price to pay for these features is some degree of conservativeness, and the need for causal delivery [16, 100] of some messages.

4.4.2.1 Union Rule

As mentioned in Section 4.3.3, the intra-bunch GC algorithm enforces the union rule by considering every inOwnerPtr as member of the GC-root. (Recall that a pair outOwnerPtr-inOwnerPtr always point to the owner of the corresponding object.)

A pair outOwnerPtr-inOwnerPtr is created when there is a propagate operation that will enable the access (by a remote process) to a locally owned object, for the first time (otherwise the pair already exists). A pair outOwnerPtr-inOwnerPtr is updated whenever the owner of an object changes in order to point always to the corresponding object's owner. A pair outOwnerPtr-inOwnerPtr is deleted as the result of a garbage collection of the bunch replica holding the outOwnerPtr. These are the only operations on inOwnerPtrs and outOwnerPtrs. In the rest of this section we explain their creation and deletion with more detail.

Before a $<$send.propagate(y, i$\rightsquigarrow$j)$>_i$ is issued, the following operations are executed in process i: *(i)* create inOwnerPtr(y, j$\rightarrow$i) (if it does not exist yet), and *(ii)* for each object directly reachable from y_i, create the corresponding inOwnerPtr (if it does not exist yet). Before the corresponding $<$deliver.propagate(x, i$\rightsquigarrow$j)$>_j$ is executed, perform the following operations in process j: *(i)* create outOwnerPtr(y, j$\rightarrow$i) (if it does not exist yet), and *(ii)* for each object directly reachable from y_j, create the corresponding outOwnerPtr (if it does not exist yet). (Note that if the process i from which an object y is propagated is not the owner of an object x that is directly reachable from y, there will be a chain of pairs outOwnerPtr-inOwnerPtr for x passing through i in direction of x's owner. However, this aspect is not relevant for this discussion.)

When a bunch replica B_j is collected, a new set of outOwnerPtrs is generated (just like with stubs for cross-bunch pointers). By comparing the old set with the new set of outOwnerPtrs, the cross-bunch collector discovers which ones have disappeared. (A disappearing outOwnerPtr results from the fact that the corresponding object replica is no longer reachable locally.) For each disappeared outOwnerPtr, the collector sends a delete message to the process holding the corresponding inOwnerPtr.

When a locally owned object y in bunch C is no longer protected by any inOwnerPtr, this means that y is no longer reachable in any replica of C cached in processes which do not own y. Thus, y can be reclaimed if it becomes unreachable locally (in the owner process). In other words, the union rule is evaluated at the owner process and consists of ensuring that an object will not be reclaimed at its owner process while there is at least one inOwnerPtr protecting

that object.

4.4.2.2 Scan

The scanning of an incoherent object replica y_j evidently does not take into account pointer writes occurring at some other process (*e.g.*, at the object's owner). However, this is not a problem. In fact, scanning an out-of-date replica simply results in making a more conservative decision about the pointed objects reachability. Thus, objects reachable from an incoherent replica y_j will not be reclaimed; this is ensured by the union rule previously described.

The worse that can happen is that the intra-bunch collector in process j fails to reclaim some objects (pointed from y_j) that are in fact unreachable because they are no longer pointed from the most up-to-date replica y_i (in y's owner). These objects will be reclaimed later (according to the union rule) either after a coherent replica of y becomes available in process j, or after y_j has become unreachable.

4.4.2.3 Move and Patch

In this section, we study the move and patch operations of an intra-bunch copy collection with respect to coherence. The first issue is to avoid two processes moving (their local replicas of) the same object to two different locations concurrently. One obvious solution to this problem would be: the process that wants to move an object would acquire the object's ownership before moving it. As at any moment there is a single owner for each object, the situation in which two processes move their local replicas to different addresses would not arise. However, this solution is clearly undesirable, since it interferes with applications coherence needs.

A simple solution that does not interfere with applications coherence needs is as follows. The owner process of x decides where to move it; non-owner processes will move their replicas of x to the same address after receiving a move message from the owner. Therefore a reachable object x in bunch B is moved as follows. *(i)* Process i, the owner of x, issues a $\langle$send.move(x, @x')$\rangle_i$ to processes caching a pointer to x (these processes are those for which a inOwnerPtr exists in i) including to itself; *(ii)* A process j executing $\langle$deliver.move(x, @x')$\rangle_j$, moves its replica x_j to the new location and patches pointers accordingly.

Note that move messages are exchanged in the background, *i.e.*, mutators do not depend on such messages, because they do not access to-space objects. In fact, mutators can freely access a from-space object for which a move is being executed. Thus, move messages do not disrupt mutators.

It's worthy to note that if the intra-bunch collector uses some other copy algorithm instead, such as Baker's (recall Section 2.2.3.1) for instance, the mutator could be disrupted. In Baker's algorithm, any object in from-space that is accessed by the mutator must first be moved to to-space. Thus, if a mutator in process j, tries to access its replica of object x, owned by process i, the mutator

would have to block while the collector in process j waits for the new address of x in to-space.

The second important issue of the copy algorithm, is the following: must the intra-bunch collector acquire the ownership (or some other lock for coherence purposes) of an object in order to patch one of its pointers with the target's new location?[4] Surprisingly, the answer is no, because this operation is visible only at the process where it occurred. Each process patches independently, thus no coherence operation is required.

4.4.2.4 Flip

The intra-bunch collector may flip independently, with no synchronization, from the collectors of other replicas of the same bunch.

In the case of a copy collector, a flip may occur before all the move messages concerning reachable objects not locally owned have already been delivered. Thus, collectors in different processes may flip at distinct times. Therefore, the GC pause time experienced by a mutator does not depend on any synchronization with other processes (see Chapter 6 for performance results).

When a collector flips before having received all the move messages for not locally owned objects, those objects remain in from-space. They will be moved later when the corresponding move messages are delivered. (In fact, such moves can be delayed until the next local garbage collection.)

The price to pay for this lack of synchronization when flipping is the following. First, the semi-space from where non-locally owned objects were not moved can not be immediately freed; this can happen only after every reachable object has been moved. Second, the same object may be referenced via two distinct addresses in different processes: the old address in from-space (processes that have not yet moved the object) and its new address in to-space (processes that have already moved it). This implies that pointers contained in a propagate or a ownership message might need to be swizzled.

In this case, a flip differs slightly from the definition given in Section 2.2.2.2. In fact, instead of changing the roles of from-space and to-space, the flip extends the from-space with the to-space. This results in delaying to free the original from-space.

When flipping, those objects that are reachable but were not moved to to-space (because the corresponding move message has not been delivered yet) are scanned and patched.

Given that collectors may flip independently, move messages must be delivered in causal order w.r.t. to propagate and ownership messages. Causality prevents a process from receiving a propagate (or a ownership) message with a to-space

[4] It's worthy to note that the patch operation implies writing into an object and on some DSM coherence protocols (*e.g.*, entry-consistency [14]) this requires the previous acquisition of an exclusive write-lock.

address (from a process where the collector has already flipped) before the corresponding move message has been delivered.

4.5 Cross-Bunch GC

In Larchant, the cross-bunch collector is based on the reference-listing algorithm (recall Section 2.3.1.3). The fundamental issues are: keeping track of cross-bunch pointers, in order to update the reference-lists, without slowing down applications significantly, and ensuring safety in presence of replication.

As mentioned in Section 4.1.4, instrumenting every pointer assignment with a write-barrier to synchronously create the corresponding stub-scion pair, would have poor performance.

The advantage of making the cross-bunch collector asynchronous to mutators, comes from the substantial performance gains since the mutator is not halted, it might avoid work, and enables message batching.

In this section we explain how cross-bunch reference-listing can be made asynchronous to mutators. We first consider there are no replicated bunches. Then, we extend this result to consider replication.

4.5.1 GC without Replication

The basic idea behind the reference-listing algorithm is the following. The collection of a bunch generates a new set of stubs. By comparing the old and new sets of stubs (before and after an intra-bunch collection) the cross-bunch collector discovers which cross-bunch pointers no longer exist and therefore which scions should be deleted.

Without loss of generality, consider the prototypical example illustrated by Figure 4.5 (special case of Figure 3.2 since there is no replication). Objects x and y are located in bunches B and C, respectively. Bunches B and C are cached in process i; bunch D is cached in process k. Initially x is nil and y points to object z located in bunch D. Now, the mutator within process i, running concurrently to the cross-bunch collector, executes $<x := y>_i$, then $<y := 0>_i$, such that at the end x points to z and y is nil.

With a cross-bunch collector asynchronous to mutators (pointer assignments are not instrumented) we must answer the following question: for how long can the creation of scion(Bx, Dz) be safely delayed? (Note that with a synchronous distributed reference-listing algorithm the cross-bunch collector creates the stub-scion pair corresponding to Bx→Dz at the instant the mutator in i executes $<x := y>_i$.)

Consider Figure 4.6 which shows the execution of the prototypical example of Figure 4.5. Event $<$send.create.scion(Bx, Dz)$>_i$ is caused by $<x := y>_i$ and occurs some time after the latter. Event $<$send.delete.scion(Cy, Dz)$>_i$ is caused by $scan_i(y)$ that follows $<y := 0>_i$, and occurs after $GC_i(C)$. Clearly,

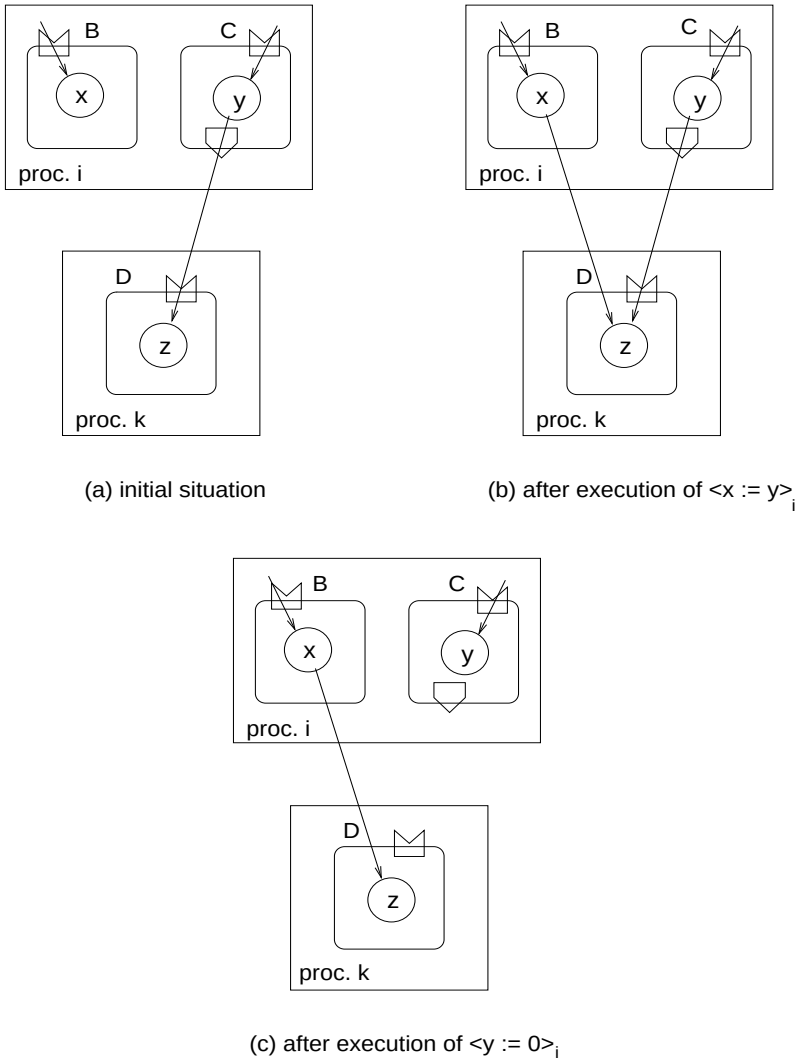


Figure 4.5: *Prototypical example of mutator execution (non-replicated case) illustrating the creation and destruction of cross-bunch pointers.*

<send.create.scion(Bx, Dz)>$_i$ must not be delayed so much that <deliver.delete-.scion(Cy, Dz)>$_k$ happens before <deliver.create.scion(Bx, Dz)>$_k$. Otherwise, z could be unsafely reclaimed by $GC_k(D)$. Thus, assuming FIFO communication, <send.create.scion(Bx, Dz)>$_i$ must be issued before <send.delete.scion(Cy, Dz)>$_i$. This is explained in more detail now.

We examine how late a create message may safely be delayed; we call *promptness* the corresponding safety condition. By definition, at the time of the assignment <x := y>$_i$, y is reachable and z is reachable via y. As long as D is not collected, it is not necessary to execute <deliver.create.scion(Bx, Dz)>$_k$. Therefore, scion(Bx, Dz) does not have to be created at the instant of <x := y>$_i$. We can state the promptness condition thus:

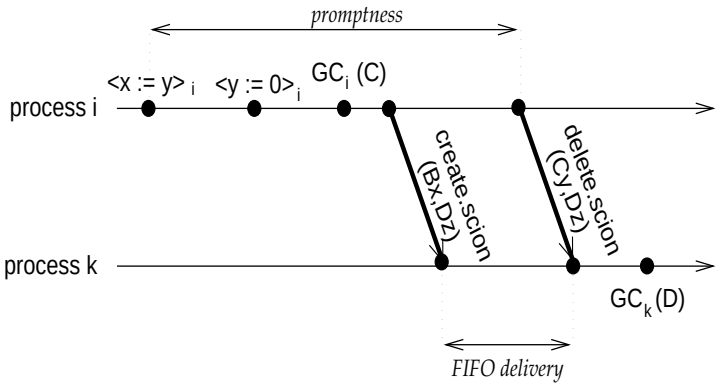


Figure 4.6: *Timeline of example illustrated by Figure 4.5.*

$$\text{<deliver.create.scion(Bx, Dz)>}_k \textit{ before } z \textit{ could otherwise be reclaimed} \quad (4.1)$$

This condition can be implemented with the following (unrealistic) algorithm: before $GC_k(D)$, scan every object in all other bunches, looking for references into D, and create any new stub-scion pairs as needed. This algorithm can be improved by scanning only the GC-dirty objects, since a new reference can only appear by assignment. Even this improved version cannot realistically be implemented, because it would mean synchronous communication with every process that might be caching a GC-dirty object. However this algorithm does contain a very interesting idea: taking advantage of the scan operations done by the intra-bunch collection to detect new outgoing cross-bunch pointers.

Now, observe that the only pointer that is positively known to point to z is the one from y. So, assuming messages are delivered in the order sent, the promptness condition becomes:

$$\text{<send.create.scion(Bx, Dz)>}_i \textit{ before } \text{<send.delete.scion(Cy, Dz)>}_i \quad (4.2)$$

Recall that we are currently assuming that there is no replication. Thus, <send.delete.scion(Cy, Dz)>$_i$ might be sent either because y is assigned to, or because it became unreachable (no longer pointed from another bunch); but it will not be sent until the enclosing bunch C is collected. On the other hand, to detect the cross-bunch pointer Bx→Dz, object x (which is GC-dirty) must be scanned. This will happen when bunch B is collected. In short, this suggests that an algorithm based only on the GC-dirty information available at process i can be safe.

Thus, promptness is ensured as follows. When the intra-bunch collector runs, it scans at least all (locally cached) GC-dirty objects, therefore creating the corresponding stubs. Then, all create messages resulting from the appearing

stubs generated by an intra-bunch collector run, are sent by the cross-bunch collector before any deletes (for the disappeared stubs) resulting from the same intra-bunch collection run. (Note that this takes into account the condition 4.2.) We prove the correctness of this algorithm in Section 4.10.

4.5.2 GC with Replication

In this section we extend the reference-listing algorithm to cope with replication. The main issues are the safety and asynchrony of the cross-bunch collector in presence of multiple replicas, which are not necessarily coherent.

4.5.2.1 Union Rule

We now explain how the cross-bunch collector is ensured to be safe in presence of replication. The safety problem of the reference-listing algorithm is illustrated by the following example.

Suppose that object y is replicated in multiple processes such that pointer Cy→Dz has many images: $(Cy)_{p^1} \rightarrow Dz$, ..., $(Cy)_{p^n} \rightarrow Dz$ observed by processes $p^1, \ldots, p^n$, respectively. Then, scanning only the owner's replica of an object y alone, is not safe. Thus, the question is in what circumstances can the cross-bunch collector delete scion(Cy, Dz), therefore allowing z to be reclaimed.

Given that there are multiple replicas of y, not necessarily all coherent, the answer is: scion(Cy, Dz) can be deleted only after z has become unreachable from *all* replicas of y. As already mentioned (recall Section 4.3.3) we call this the union rule: a scion pointing to a target object z can be safely deleted only after the corresponding stubs have disappeared in all replicas of the source object.

This rule is implemented by the owner of the source object. The owner evaluates the union of the stubs of all replicas (of the source object); then, if the result is an empty set, it may issue a <send.delete.scion> for the corresponding scion. In other words, a process caching a replica y_i informs the owner of y, process j, of y_i's stubs by a union message: events <send.union(y, i⇝j)>$_i$ and <deliver.union(y, i⇝j)>$_j$. Then, the owner process j performs the union of all stubs for all y's replicas. If the resulting set is empty, it performs <send.delete.scion(Cy, Dz)>$_j$.

Many DSM coherence protocols impose that only the owner of an object y can write it (*e.g.*, entry-consistency [14]). In this case, a non-owner replica y_i cannot cause an object unreachable from y_i to become reachable, because to do so requires writing y in process i. Thus, the set of stubs at a non-owner process is monotonically decreasing. Therefore, the union rule can be implemented cheaply, using asynchronous FIFO communication.

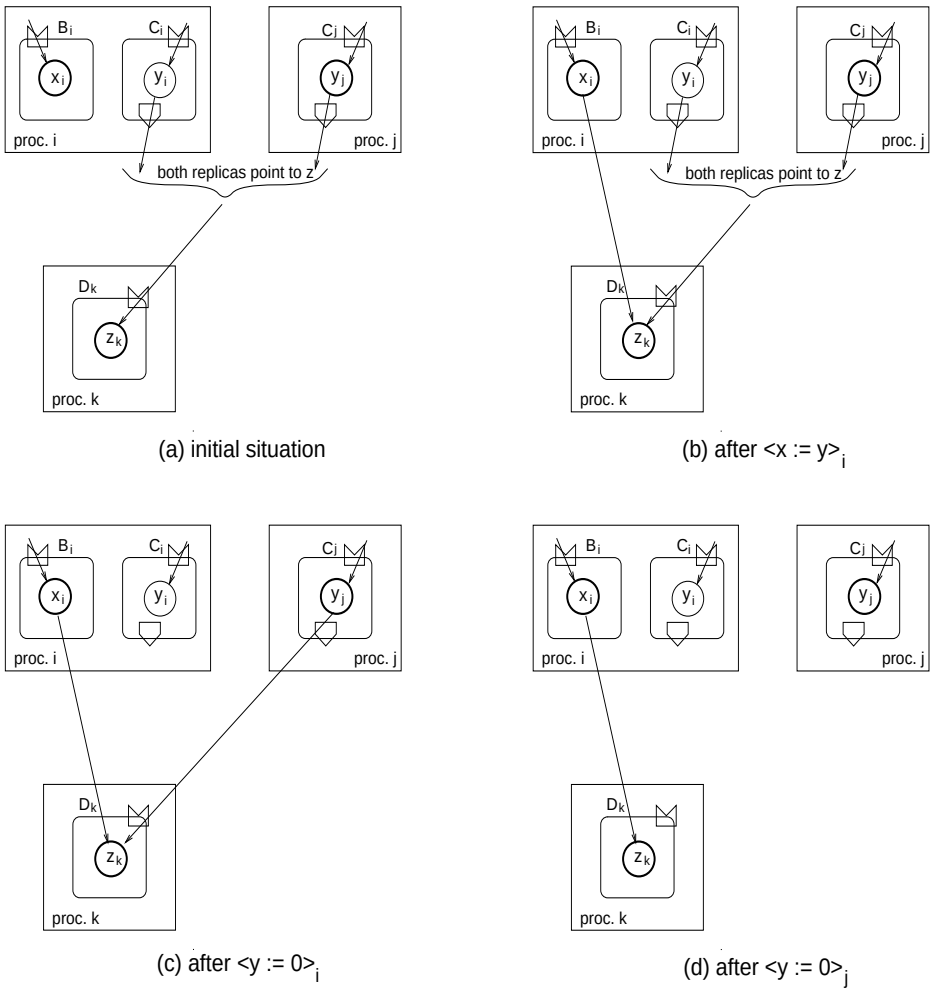


Figure 4.7: *Prototypical example of mutator execution (replicated case) illustrating the creation and destruction of cross-bunch pointers. Objects are shown in bold, on their owner process.*

4.5.2.2 Promptness

In this section we study how long the creation of a stub can be safely delayed, extending the promptness condition to the replicated case.

Without loss of generality, consider the prototypical example illustrated in Figure 4.7 (replicated case of Figure 3.2). Objects x and y are located in bunches B and C, respectively. Bunch C is cached by processes i and j. Object x is owned by i, y is owned by j, and z is owned by k. Initially x_i is null and both replicas of y point to object z located in bunch D. Then, the mutators within processes i and j execute the operations $\langle x := y \rangle_i$, $\langle y := 0 \rangle_i$, and $\langle y := 0 \rangle_j$, such that in the end x points to z and y is null. (Note that instead of $\langle y := 0 \rangle_i$, a propagate of y from j to i, would have the same effect: both replicas of y no longer pointing to z.)

As in the non-replicated case we want to render the cross-bunch collector asyn-

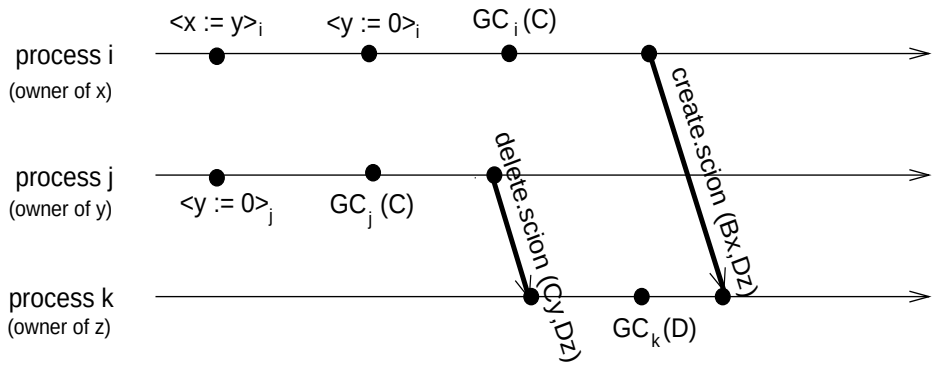


Figure 4.8: *Timeline showing a possible problem with the asynchronous creation of scions: the stub-scion pair describing cross-bunch pointer* Bx→Dz *is created too late (for safety purposes).*

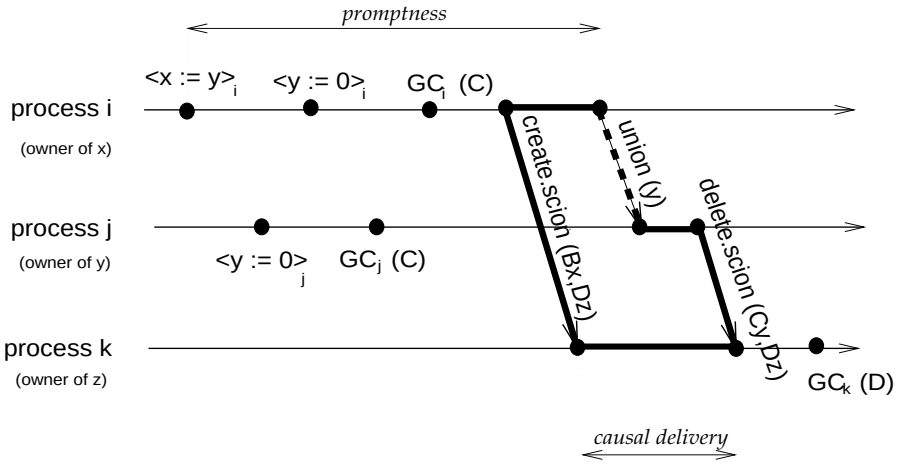


Figure 4.9: *Timeline showing the union rule to ensure promptness. Message* <send.create.scion(Bx, Dz)>i *can be delayed at most until* <send.union(y, i⇝j)>i. *Note that the union message carries the causal dependency (shown in bold lines) between the create and delete messages.*

chronous to mutators. So, once again, we must answer the question: for how long can the creation of scion(Bx, Dz) be safely delayed?

This issue is illustrated in Figure 4.8: suppose that the creation of scion(Bx, Dz) (due to <x := y>$_i$) is delayed in such a way that <deliver.delete.scion(Cy, Dz)>$_k$ is executed first. Then, z could be unsafely reclaimed by GC$_k$(D), because there is no scion protecting z (note that z is still referenced from x).

We can use the same unrealistic algorithm described for the non-replicated case (recall Section 4.5.1): before GC$_k$(D), scan every object in all other bunches, looking for references into D, and create any new scions as needed. However, this algorithm is even more difficult to implement than in the non-replicated case because it implies synchronizing every process that might be caching a GC-dirty object replica.

Safety clearly depends on the relative delivery order of create and delete events at process k. The promptness condition is therefore: *(i)* the create message must be sent from process i before the delete message is sent from process j, *i.e.*, create and delete messages must be sent in a safe order, and *(ii)* the create message must be delivered at process k before the delete message, *i.e.*, create and delete messages must be delivered in a safe order.

Now, consider the situation illustrated by Figure 4.9, which extends Figure 4.8 with the union rule. To ensure the promptness condition, all create messages resulting from an intra-bunch collector run must be sent before any union message in the same run, and create and delete messages are delivered in causal order [16].

To conclude, the create operation may be safely issued during all the *promptness* time period indicated in Figure 4.9 (at the latest before the union message is issued). Furthermore, causal delivery is necessary to ensure safety.

4.6 Interaction of Intra-Bunch and Cross-Bunch GC

In this section we describe the interaction between the intra-bunch and cross-bunch collectors. The basic aspect is that between any two intra-bunch collections in a process (not necessarily of the same bunch) the cross-bunch collector must run. The cross-bunch collector issues the messages for creating the scions corresponding to new stubs generated by an intra-bunch collection. These are vital for safety.

Let us give an example to illustrate this. Suppose that bunches B, C and D are all cached in process i. Objects x and y are allocated in B; t in C; z in D. Now, the mutator modifies x and y by creating pointers By→Dz and Bx→Ct. Thus, x and y are GC-dirty. This situation is illustrated in Figure 4.10 (no scions protecting t and z for the moment). Then, when GC$_i$(D) runs, the intra-bunch collector must scan all GC-dirty objects in i because they may contain pointers to objects in D (this is the case of y). As x and y are scanned, z is reachable, and the corresponding stubs are created: stub(By, Dz) and stub(Bx, Ct). Note that we are collecting only bunch D, thus t reachability is not considered. Now,

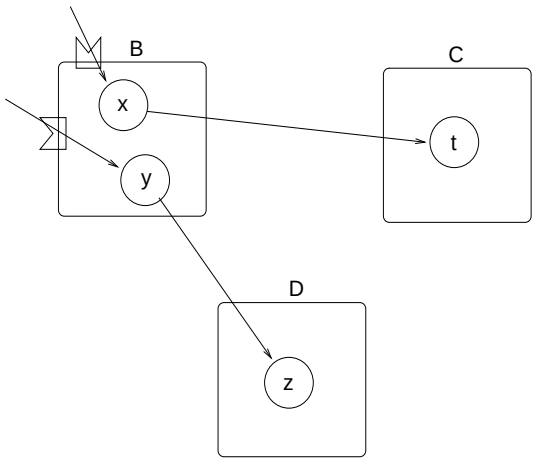


Figure 4.10: *Example showing the need of cross-bunch collection between each pair of intra-bunch collections.*

suppose that $\mathsf{GC_i(C)}$ runs after $\mathsf{GC_i(D)}$ has finished, and before the cross-bunch collector runs in i. Objects x and y are no longer GC-dirty; thus they will not be scanned. Because $\mathsf{scion(Bx, Ct)}$ does not exist yet, t is unsafely reclaimed.

The execution of the cross-bunch collector after $\mathsf{GC_i(D)}$ and before $\mathsf{GC_i(C)}$ prevents this problem because it creates the scions corresponding to $\mathsf{stub(By, Dz)}$ and $\mathsf{stub(Bx, Ct)}$. It compares the stub sets before and after $\mathsf{GC_i(D)}$ and creates $\mathsf{scion(By, Dz)}$ and $\mathsf{scion(Bx, Ct)}$.

Figure 4.11 illustrates the concurrent execution of: intra-bunch collections of a bunch replica $\mathsf{B_i}$, cross-bunch collections, and mutator, in process i. Intra-bunch collections are noted intra; cross-bunch collections are noted cross; the mutator execution is noted mutator; $\mathsf{stubs}^i(\mathsf{B_i})$ is noted $\mathsf{stubs(i)}$.

By computing the set difference $\mathsf{stubs}^i(\mathsf{B_i})$-$\mathsf{stubs}^{i-1}(\mathsf{B_i})$, the cross-bunch collector discovers the new stubs, corresponding to new outgoing cross-bunch pointers. The cross-bunch collector then sends the messages to create the corresponding scions. By computing the set difference $\mathsf{stubs}^{i-1}(\mathsf{B_i})$-$\mathsf{stubs}^i(\mathsf{B_i})$, the cross-bunch collector discovers the disappeared stubs. Then, the cross-bunch collector sends the messages to delete the corresponding scions.

When a stub is created, it is inserted in the most recent set of stubs. For example, when i^{th} intra-bunch collection runs, the most recent set of stubs is $\mathsf{stubs}^i(\mathsf{B_i})$. Thus, new stubs are inserted in this set.

A new empty set of stubs is created at the beginning of each cross-bunch collection. Stubs created either concurrently with or after the cross-bunch collector are inserted in this new set. The reason for this is the following. Imagine that a stub is created while the cross-bunch collector is comparing $\mathsf{stubs}^i(\mathsf{B_i})$ and $\mathsf{stubs}^{i-1}(\mathsf{B_i})$. If the new stub is inserted in $\mathsf{stubs}^i(\mathsf{B_i})$, the corresponding scion will never be created because the next cross-bunch collector would not consider it as a new stub when comparing $\mathsf{stubs}^i(\mathsf{B_i})$ and $\mathsf{stubs}^{i+1}(\mathsf{B_i})$. Thus, the new stub, created while the cross-bunch collector runs, is inserted in $\mathsf{stubs}^{i+1}(\mathsf{B_i})$.

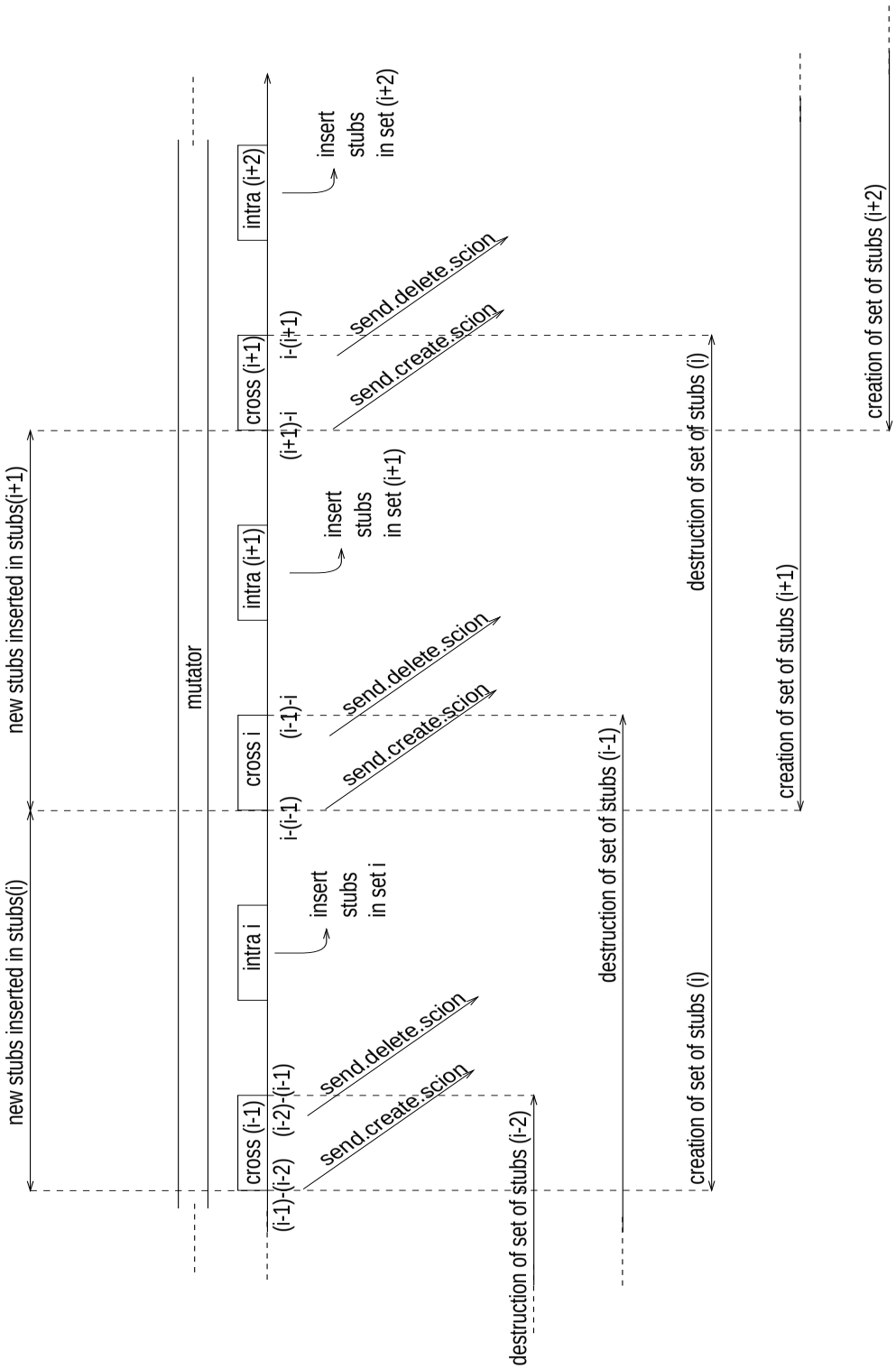


Figure 4.11: *Timeline showing concurrent execution of mutator, intra-bunch and cross-bunch collectors. (The flip times are not represented.)*

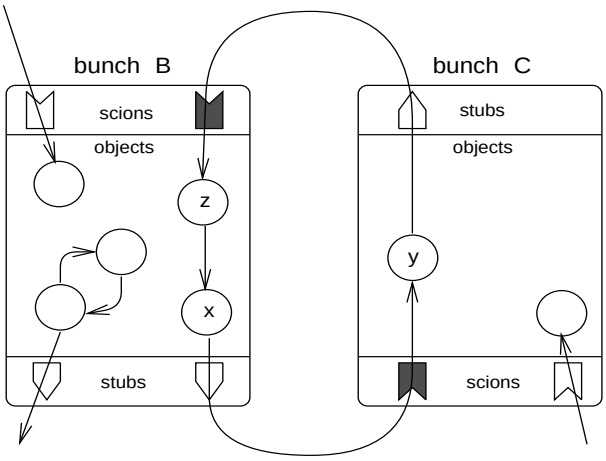


Figure 4.12: *Reclaiming a cross-bunch cycle of garbage. When collecting the group of bunches* B *and* C*, shaded scions are not members of the GC-root. Thus, objects* x*,* y*, and* z *are reclaimed.*

Finally, observe that a set of stubs can be destroyed after the cross-bunch collector has compared it with its successor.

4.7 Reclaiming Cross-Bunch Cycles of Garbage

The intra-bunch GC algorithm is complete w.r.t. the collected bunch, *i.e.*, it reclaims all garbage that is entirely within the bunch. However, it is incomplete w.r.t. other bunches, since it does not reclaim a cycle of garbage that crosses the bunch boundary (*e.g.*, cross-bunch cycle in Figure 4.12: objects x, y, and z). In this section we describe our GC algorithm to reclaim such cycles and discuss its completeness.

4.7.1 GC Algorithm

Note that the same tracing algorithm that collects a single bunch (recall Section 4.4) can collect any group of bunches. The only difference is that to trace a group: *(i)* scions for cross-bunch pointers internal to the group are not considered as members of the GC-root, and *(ii)* tracing continues across bunch boundaries internal to the group.

This algorithm reclaims a cross-bunch cycle unreachable from bunches outside the group, *i.e.*, it is complete w.r.t. to the group being collected. Thus, in Figure 4.12, a group collection including bunches B and C would not consider the shaded scions as members of the GC-root. The cross-bunch garbage cycle constituted by objects x, y and z would therefore be reclaimed and the corresponding scions deleted.

4.7.2 Discussion

The significance of group collection is that any arbitrary subset of the store can be collected, in a single process, independently of the rest of the memory. The choice of the group to be collected is heuristic, and aims at maximizing the amount of garbage reclaimed while minimizing the cost.

Larchant uses a locality-based heuristic. A group contains all the bunches currently cached in the process. This heuristic avoids extra I/O costs. (Recall that this was one of the problems presented in Section 4.1.) However, it does not enable the reclamation of cross-bunch cycles enclosed in bunches that reside partially on disk. Also, cycles of garbage where some bunches, but not all, are replicated in multiple processes, are not reclaimed.

This garbage might be reclaimed with a more aggressive grouping heuristic. For example, the intra-bunch collector could force bunches to be cached in a process at the same time, even if applications do not do so naturally. This extra I/O cost needs to be balanced against the expected gain. We intend to experiment with the locality-based heuristic over a wide number of applications, and to do some simulation studies. Then, if experimental results mandate it, more complex heuristics will be the topic of future research. A full study of the most appropriate heuristic and its cost is out of the scope of this document.

4.8 Summary of GC Operations

In this section we summarize all the operations of the Larchant model and their implications in terms of GC. This description (see Table 4.1) consolidates the presentation of the GC algorithms done so far, and facilitates the understanding of the next sections in which we formally address their correctness.

4.9 Correctness of Intra-Bunch GC

Given that the algorithms used by the intra-bunch collector are based on the well known mark-and-sweep and copy techniques, their correctness can be formally proved with an approach similar to that found in the literature [40, 42]. Thus, from the theoretical point of view, there is no particular difficulty or novelty in proving the safety and liveness of the intra-bunch collector. We simply present the invariants that must hold.

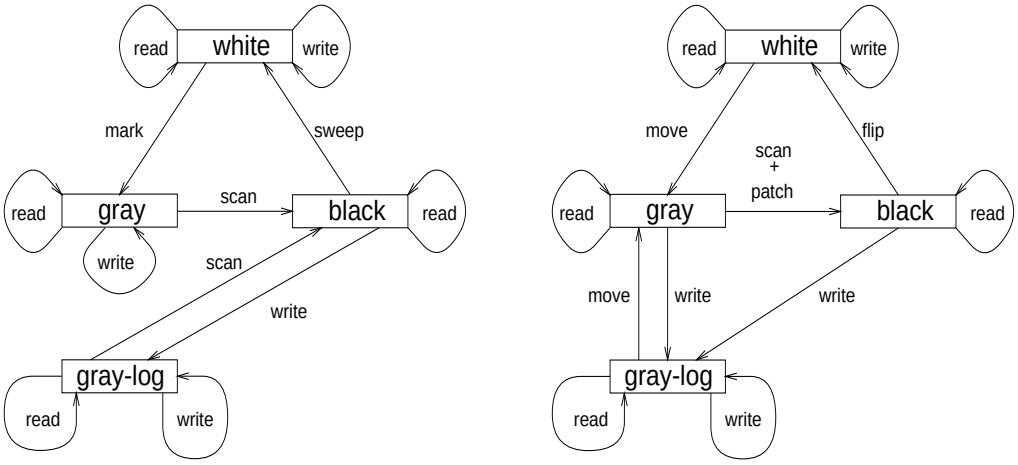
On the contrary, there is no previous work regarding the proof of safety and liveness of an asynchronous cross-bunch collector in presence of replication. Thus, its correctness is formally proved in Section 4.10.

4.9.1 Safety and Liveness

Conceptually, both the mark-and-sweep and copy intra-bunch collectors can be described in terms of the tricolor algorithm introduced in Section 2.2.3.1, as

model	operations	description	GC implications
mutator	$<x := y>_i$	assignment operation performed by process i, such that x is made to point to the object pointed from y	modifies the pointer graph.
coherence	$<send.propagate(x, i \rightsquigarrow j)>_i$	put the contents of x_i in the message	create the corresponding inOwnerPtrs; GC-clean x_i and send messages to create the corresponding scions (see Section 4.10.2.2).
	$<deliver.propagate(x, i \rightsquigarrow j)>_j$	copy the object's contents from the message into x_j	create the corresponding outOwnerPtrs.
intra GC	$<mark(x)>_i$	mark object x_i in process i	color x_i gray.
	$scan_i(x)$	scan x_i in process i	color x_i black and create corresponding stubs.
	$<send.move(x, @x')>_i$	i sends new address of x to processes caching a pointer to x	process i moves x_i to to-space and communicates x's new address to other processes.
	$<deliver.move(x, @x')>_j$	j receives new address of x	process j moves x_j to to-space.
	$<patch(y, @x')>_i$	on i patch y with new address of x	on i reconstruct the pointer graph in to-space.
cross GC	$<send.create.scion(Bx, Cy)>_i$	i sends message to create scion(Bx, Cy)	results from GC-cleaning x_i.
	$<deliver.create.scion(Bx, Cy)>_j$	j receives message to create scion(Bx, Cy)	create scion(Bx, Cy) in j.
	$<send.delete.scion(Bx, Cy)>_i$	i sends message to delete scion(Bx, Cy)	results from $GC_i(B)$.
	$<deliver.delete.scion(Bx, Cy)>_j$	j receives message to delete scion(Bx, Cy)	delete scion(Bx, Cy) in j.
	$<send.union(y, i \rightsquigarrow j)>_i$	i sends union message to j	results from $GC_i(C)$.
	$<deliver.union(y, i \rightsquigarrow j)>_j$	j receives union message from i	apply union rule in j.

Table 4.1: *Summary of operations in the Larchant model.*



(a) mark-and-sweep (b) copy

Figure 4.13: *Intra-bunch GC represented as a tricolor algorithm. The "read" transition represents an assignment operation in which the object appears in the right-hand side; the "write" transition represents an assignment operation in which the object appears in the left-hand side.*

illustrated in Figure 4.13. The fourth color, *gray-log*, models the fact that a mutator may modify an object that has already been scanned.[5]

The difference between schemes (a) and (b) in Figure 4.13 is due to the fact that with the copy algorithm, when an object that has already been moved is written by the mutator, it is necessary to move it again to the same address in to-space (recall the replication-based technique in Section 2.2.3.1). This difference is irrelevant from the point of view of correctness. In fact, scheme (b) also describes the mark-and-sweep algorithm if we accept to "re-mark" an object (the move operation between colors gray-log and gray would be replaced by a "re-mark") and to (unnecessarily) change an object's color from gray to gray-log when it is written even if not yet scanned.

Given the replication technique of the copy collector, there are two replicas that must be considered: the object in from-space and its replica in to-space. Thus, each color identifies the merged state of both from-space and to-space replicas of an object as explained now.

The colors have the following meaning:

- mark-and-sweep algorithm:
  - white: object not yet marked;
  - gray: object already marked but not yet scanned;

[5]Recall that the object's address is inserted in the GC-log if the mutator modifies it after having been scanned; this is the rational for the name gray-log.

  - black: object already marked and scanned;
  - gray-log: object written after being marked and scanned.

- copy algorithm:
  - white: object in from-space, not yet moved;
  - gray: object in from-space already moved, and object in to-space not yet scanned;
  - black: object in from-space already moved, and object in to-space already scanned;
  - gray-log: object in from-space written after it has been moved and after object in to-space has been scanned.

Before an intra-bunch collection starts, every object is white. When an object is reached by the intra-bunch collector while traversing the pointer graph, the collector colors it gray. (This coloring consists of setting a mark bit associated to the object in case of a mark-and-sweep collector, or of moving the object to to-space in case of a copy collector.) A gray object becomes black after having been scanned (and patched in case of a copy collector). If a black object is written by the mutator, it becomes gray-log. With the mark-and-sweep algorithm, a gray-log object becomes black after having been scanned. With the copy algorithm, a gray-log object becomes gray after having been moved.

The invariant that the intra-bunch collector must ensure is the following: *black objects may not point to white objects.* In the case of a copy collector, another invariant must be ensured: *the mutator only accesses from-space objects.* These invariants are typical of the corresponding basic GC algorithms with the incremental/concurrent functioning mode (recall Section 2.2.3.1).

It's worthy to note that the tricolor algorithm applies not only to the collection of a non-replicated bunch but also to the collection of a replicated one. The only difference resides in the move operation (when using a copy collector) as explained now. When the process where the intra-bunch collector is running owns the object to be moved (being "grayed") the collector proceeds as in the non-replicated case, *i.e.*, sends a move message, including to itself; thus it moves its object replica to to-space. If the process does not own the object, then it will receive the object's address in to-space when the corresponding move message is delivered (sent by the owner). Then, it moves the object to to-space. (Note that the collector does not have to block while waiting for the move message as will be explained in Section 5.4.2; in addition, it can flip even without having received such messages, as explained in Section 4.4.2.4.)

4.10 Correctness of the Cross-Bunch GC

In this section we prove the correctness of the cross-bunch GC algorithm. We first use a simplified model of the cached distributed shared store, without replication. Then, we extend the proof to the case where objects are replicated.

4.10.1 Cross-Bunch GC without Replication

In this section, we describe rigorously the cross-bunch GC algorithm without replication. We present the invariants that must hold, and prove the algorithm is safe and live.

Given that for the time being we consider there is no replication, no coherence events appear. Thus, we introduce the concept of *holder*. The holder is the equivalent of owner in the replicated case: at any point in time, an object is cached by a single process which we call its holder. The model does not specify how or when the holder process may change.

We define the propagate operation in the non-replicated case as follows. When object x held by process i is propagated to j by operations $<$send.propagate(x, i$\rightsquigarrow$j)$>_i$ and $<$deliver.propagate(x, i$\rightsquigarrow$j)$>_j$, it is thereafter held by j and not by i; we say that x *migrates* from i to j. (Note that when an object x migrates, all the stubs corresponding to its outgoing pointers, and the scions pointing to x, are also migrated.)

4.10.1.1 Safety

The cross-bunch GC algorithm must satisfy the following obvious safety invariant:

Safety Invariant 1 *No reachable object is reclaimed.*

Since we are considering only cross-bunch pointers, the above invariant is equivalent to:

Safety Invariant 2 $(\forall\, B, \forall\, x \in B\!:\, \exists\, Bx{\rightarrow}Dz) \Rightarrow (\exists\, E, \exists\, t \in E\!:\, \exists\, \mathsf{scion}(Et, Dz))$

This reads "if any object points to z, then some scion protects z." Note that this is weaker than the more intuitive "if x points to z, then scion(Bx, Dz) exists"; indeed, the scion that protects z does not need to match the pointer.

Given that an object may be reclaimed only when its enclosing bunch is collected, scion(Et, Dz) is only needed at the time of $\mathsf{GC}_k(D)$, *i.e.*, when D is collected in some process k. In other words, Safety Invariant 2 is only evaluated at the time of $\mathsf{GC}_k(D)$. Under this assumption, Safety Invariant 2 is equivalent to:

Safety Invariant 3 $(\forall\, B, \forall\, x \in B\!:\, \exists\, Bx{\rightarrow}Dz) \Rightarrow (\exists\, E, \exists\, t \in E\!:\, \exists\, \mathsf{scion}(Et, Dz)$ *at the time of* $\mathsf{GC}_k(D)$ *for the process* k *holding* D)

This reads "at the time D is collected in process k, if any object points to z, then some scion protects z." The scion is needed only at the time when the target bunch D is collected.

4.10.1.2 Algorithm

Let us return to the prototypical example illustrated in Figure 4.5, and observe that the initial situation satisfies Safety Invariant 2. At the time of <x := y>$_i$, object z is reachable (from y) and is protected by some scion, say scion(Et, Dz) (presumably, but not necessarily, E=C and t=y). As long as that scion continues to exist, it is safe to delay the creation of scion(Bx, Dz).

The following rules and algorithm ensure the correctness of the cross-bunch collector (intuitively presented in Section 4.5.1) as will be formally proved in the next sections:

Comprehensive Scanning Rule

When an intra-bunch collection takes place at process i, every GC-dirty object y held by i must be GC-cleaned.[6]

GC-Clean Migration Rule

An object can be migrated from process i only if every object held by i is GC-clean, and the messages to create the corresponding scions have been sent.

Cross-Bunch Collection Algorithm

At process i, between each pair of intra-bunch collections, execute these steps in the following order:

1. For all new cross-bunch stubs, perform the corresponding <send-.create.scion>$_i$.
2. For all disappearing cross-bunch stubs, perform the corresponding <send.delete.scion>$_i$.

The Comprehensive Scanning Rule means that when an intra-bunch collection takes place in process i every GC-dirty object (locally held) must be scanned, therefore creating the corresponding stubs.

The GC-Clean Migration Rule means that an object cannot migrate from process i without having scanned all GC-dirty objects in i, thus after having created the corresponding stubs, and sent the messages to create their scions.

The cross-bunch collector runs between each pair of intra-bunch collections and compares the set of stubs before and after collection (recall Section 4.6). The Cross-Bunch Collection Algorithm ensures that for each intra-bunch collector run, all messages to create scions are sent before any message to delete a scion.

To understand the role of the Comprehensive Scanning Rule, consider Figure 4.5 after the mutator has executed <x := y>$_i$ and <y := 0>$_i$. Then, suppose that GC$_i$(C) runs without fulfilling the Comprehensive Scanning Rule, *i.e.*, without GC-cleaning x. According to the Cross-Bunch Collection Algorithm, <send.delete.scion(Cy, Dz)>$_i$ will be sent. Thus, scion(Cy, Dz) is deleted.

[6] Recall that when an object is GC-cleaned the corresponding stubs are created (as described in Section 4.3.2).

Then, if D is collected, object z is unsafely reclaimed. The Comprehensive Scanning Rule prevents the above scenario as it forces x to be GC-cleaned, thus stub(Bx, Dz) to be created, when $GC_i(C)$ runs; then, according to the Cross-Bunch Collection Algorithm, <send.create.scion(Bx, Dz)>$_i$ is performed before <send.delete.scion(Cy, Dz)>$_i$. Since we assumed causal delivery, scion(Bx, Dz) is created before scion(Cy, Dz) is deleted. Consequently, z is not reclaimed by $GC_k(D)$.

It is also interesting to understand the role of the GC-Clean Migration Rule. Consider Figure 4.5 after the mutator has executed <x := y>$_i$ and before <y := 0>$_i$. Now, suppose that object y migrates to some other process j without fulfilling the GC-Clean Migration Rule. At this moment, the only scion that is known to protect z is scion(Cy, Dz). Suppose that the mutator at j executes <y := 0>$_j$, $GC_j(C)$ runs, and <send.delete.scion(Cy, Dz)>$_j$ is performed. Then, scion(Cy, Dz) no longer exists and z may be unsafely reclaimed by $GC_k(D)$. The GC-Clean Migration Rule prevents the above scenario as it forces x to be GC-cleaned, thus stub(Bx, Dz) to be created, and <send.create.scion(Bx, Dz)>$_i$ to be performed, all before object y gets propagated to j. Therefore, z is not unsafely reclaimed because scion(Bx, Dz) is created before deleting scion(Cy, Dz) (assuming causal delivery).

4.10.1.3 Proof of Safety

To prove the safety of the cross-bunch collection algorithm we start by proving the following lemma.

Lemma 1 *Let* $x^1, \ldots, x^n$ *located respectively in bunches* $B^1, \ldots, B^n$ *all held by process* p*, be the set of all protected objects pointing to* $z \in D$ *held by process* k*. Then,* $\forall i, 1 \leq i \leq n$*:*

- x^i *is GC-clean* $\wedge \exists$ scion($B^i x^i$, Dz)*, or*
- $\exists j, 1 \leq j \leq n$, x^i *is GC-dirty* $\wedge \exists$ scion($B^j x^j$, Dz)

This lemma says that as long as there is an object pointing to z, held by some process p, there exists at least a scion protecting z.

Proof of Lemma 1:

We prove this lemma by induction over the size of the set containing objects pointing to z, for a single process.

Base case:
Initially a single protected object x^1 points to z: $x^1 \in B^1$ held by process p, and scion($B^1 x^1$, Dz) exists. Let $x^2 \in B^2$, also held by p, initially not pointing to z. Perform <$x^2 := x^1$>$_p$ (now x^2 also points to z) therefore x^2 is GC-dirty.

If x^1 is not assigned to thereafter, the lemma remains trivially true since x^2 is GC-dirty and scion($B^1 x^1$, Dz) exists.

Therefore consider that x^1 is changed by an assignment such as $<x^1 := 0>_p$. (The actual value of the right-hand side does not matter for the proof, except that when the right-hand side is a pointer to z it is as if the assignment did not take place.) Hence, x^1 is GC-dirty. Until an intra-bunch collection takes place at p, no <send.create.scion>$_p$ or <send.delete.scion>$_p$ is performed, and the lemma remains trivially true since x^2 is GC-dirty and scion(B^1x^1, Dz) exists.

When an intra-bunch collection does execute in process p, by the Comprehensive Scanning Rule, both x^2 (containing the new pointer) and x^1 (previously containing the pointer with a scion) are GC-cleaned because both are GC-dirty, therefore stub(B^1x^1, Dz) disappears (not in the new set of stubs) and stub(B^2x^2, Dz) is created.

According to the Cross-Bunch Collection Algorithm, <send.create.scion(B^2x^2,Dz)>$_p$ precedes <send.delete.scion(B^1x^1,Dz)>$_p$. We assumed that messages are delivered in causal order, hence <deliver.create.scion(B^2x^2,Dz)>$_k$ precedes <deliver.delete.scion(B^1x^1,Dz)>$_k$. Thus, at any moment, there exists at least a scion protecting z: either scion(B^1x^1, Dz), or scion(B^2x^2, Dz), or both.

Induction case:
Assume a set of objects $x^1, \ldots, x^j$ located respectively in bunches $B^1, \ldots, B^j$ all held by process p, pointing to z with the corresponding scions already created. Thus, object x^l, $1 \leq l \leq j$, points to z: $x^l \in B^l$ held by process p, and scion(B^lx^l, Dz) exists.

Let $x^{j+1} \in B^{j+1}$, also held by p, initially not pointing to z. Perform $<x^{j+1} := x^l>_p$ (now x^{j+1} also points to z) therefore x^{j+1} is GC-dirty.

If x^l is not assigned to thereafter, the lemma remains trivially true since x^{j+1} is GC-dirty and scion(B^lx^l, Dz) exists.

Therefore consider that x^l is changed by an assignment such as $<x^l := 0>_p$. (The actual value of the right-hand side does not matter for the proof, except that when the right-hand side is a pointer to z it is as if the assignment did not take place.) Hence, x^l is GC-dirty. Until an intra-bunch collection takes place at p, no <send.create.scion>$_p$ or <send.delete.scion>$_p$ is performed, and the lemma remains trivially true since x^{j+1} is GC-dirty and scion(B^lx^l, Dz) exists.

When an intra-bunch collection does execute in process p, by the Comprehensive Scanning Rule, both x^{j+1} (containing the new pointer) and x^l (previously containing the pointer with a scion) are GC-cleaned because both are GC-dirty, therefore stub(B^lx^l, Dz) disappears (not in the new set of stubs) and stub($B^{j+1}x^{j+1}$, Dz) is created.

According to the Cross-Bunch Collection Algorithm, event <send.create.scion($B^{j+1}x^{j+1}$,Dz)>$_p$ precedes <send.delete.scion(B^lx^l,Dz)>$_p$. We assumed that messages are delivered in causal order, hence <deliver.create.scion($B^{j+1}x^{j+1}$,Dz)>$_k$ precedes <deliver.delete.scion(B^lx^l,Dz)>$_k$. Thus, at any moment, there exists at least a scion protecting z: either scion(B^lx^l, Dz), or scion($B^{j+1}x^{j+1}$, Dz), or both.

This terminates the induction over the size of the set containing objects pointing to z, for a single process. □

We now prove the following theorem.

Theorem 1 *Let* $\mathsf{x}^1, \ldots, \mathsf{x}^n$*, located respectively in bunches* $\mathsf{B}^1, \ldots, \mathsf{B}^n$*, held by processes* $\mathsf{p}^1, \ldots, \mathsf{p}^n$*, be the set of all protected objects pointing to* $\mathsf{z} \in \mathsf{D}$ *held by process* k*. Then, for each process* $\mathsf{p} \in \{\mathsf{p}^1, \ldots, \mathsf{p}^n\}$*, there exists an object* $\mathsf{x}^m \in \mathsf{B}^m$ *held by* p *pointing to* z *and* $\mathsf{scion}(\mathsf{B}^m\mathsf{x}^m, \mathsf{Dz})$ *exists at* k*.*

This theorem implies Safety Invariant 2. The latter states that, whatever number of cross-bunch pointers point to $\mathsf{z} \in \mathsf{D}$, at least one scion protects z. Theorem 1 is stronger since it says that for each process containing a pointer to z, a scion from that process protects z. It does not say whether this scion effectively corresponds to an existing pointer to z.

Proof of Theorem 1:

We prove this theorem by induction over the size of the set containing processes with pointers to z.

Base case:
Let p^1, holding $\mathsf{x}^1 \in \mathsf{B}^1$, be the only process with a pointer to z. Let process p^2 initially not holding any object pointing to z.

The only way for process p^2 to gain a pointer to z is because process p^1 holding x^1 pointing to z, performs the operation $<\mathsf{send.propagate}(\mathsf{x}^1,\ \mathsf{p}^1 \rightsquigarrow \mathsf{p}^2)>_{\mathsf{p}^1}$. According to Lemma 1, either $\mathsf{scion}(\mathsf{B}^1\mathsf{x}^1, \mathsf{Dz})$ exists and x^1 is GC-clean, or some other scion protecting z exists ($\mathsf{scion}(\mathsf{B}^1\mathsf{x}^m, \mathsf{Dz})$, $\mathsf{x}^m \in \mathsf{B}^1$) and x^1 is GC-dirty.

In the first case, the theorem remains trivially true since $\mathsf{scion}(\mathsf{B}^1\mathsf{x}^1, \mathsf{Dz})$ exists at the instant $<\mathsf{send.propagate}(\mathsf{x}^1,\ \mathsf{p}^1 \rightsquigarrow \mathsf{p}^2)>_{\mathsf{p}^1}$ is performed.

In the second case, by the GC-Clean Migration Rule, process p^1 must GC-clean every GC-dirty object in p^1 and send the messages to create the corresponding scions. Thus, x^1 is scanned, $\mathsf{stub}(\mathsf{B}^1\mathsf{x}^1, \mathsf{Dz})$ is created, and $<\mathsf{send.create.scion}(\mathsf{B}^1\mathsf{x}^1,\mathsf{Dz})>_{\mathsf{p}^1}$ is performed. Therefore,$<\mathsf{send.create.scion}(\mathsf{B}^1\mathsf{x}^1,\mathsf{Dz})>_{\mathsf{p}^1}$ precedes $<\mathsf{send.propagate}(\mathsf{x}^1,\ \mathsf{p}^1 \rightsquigarrow \mathsf{p}^2)>_{\mathsf{p}^1}$.

Since we assumed that messages are delivered in causal order, $<\mathsf{deliver.create.scion}(\mathsf{B}^1\mathsf{x}^1,\mathsf{Dz})>_{\mathsf{k}}$ happens before any other message sent (after the propagate) by p^2 is delivered at k; in particular, a possible $<\mathsf{deliver.delete.scion}(\mathsf{B}^1\mathsf{x}^1,\mathsf{Dz})>_{\mathsf{k}}$ sent by p^2 will be delivered at k after $\mathsf{scion}(\mathsf{B}^1\mathsf{x}^1, \mathsf{Dz})$ has been created. Thus, at all times, there exist at least a scion protecting z: either $\mathsf{scion}(\mathsf{B}^1\mathsf{x}^1, \mathsf{Dz})$ or $\mathsf{scion}(\mathsf{B}^1\mathsf{x}^m, \mathsf{Dz})$.

Induction case:
Let $\mathsf{p}^1, \ldots, \mathsf{p}^h, 1 \leq h \leq n$ be the only processes pointing to z via the objects held $\mathsf{x}^1, \ldots, \mathsf{x}^h$ contained in bunches $\mathsf{B}^1, \ldots, \mathsf{B}^h$. Let process p^{h+1} initially not holding any object pointing to z.

The only way for process p^{h+1} to gain a pointer to z is because some process $\mathsf{p}^l \in \{\mathsf{p}^1, \ldots, \mathsf{p}^h\}$ holding an object $\mathsf{x}^l \in \mathsf{B}^l$, pointing to z, performs the operation $<\mathsf{send.propagate}(\mathsf{x}^l,\ \mathsf{p}^l \rightsquigarrow \mathsf{p}^{h+1})>_{\mathsf{p}^l}$. According to Lemma 1, either

$\mathsf{scion}(\mathsf{B}^l\mathsf{x}^l, \mathsf{Dz})$ exists and x^l is GC-clean, or some other scion protecting z exists ($\mathsf{scion}(\mathsf{B}^l\mathsf{x}^m, \mathsf{Dz})$, $\mathsf{x}^m \in \mathsf{B}^l$) and x^l is GC-dirty.

In the first case, the theorem remains trivially true as $\mathsf{scion}(\mathsf{B}^l\mathsf{x}^l, \mathsf{Dz})$ exists at the instant $<\mathsf{send.propagate}(\mathsf{x}^l,\ \mathsf{p}^l \rightsquigarrow \mathsf{p}^{h+1})>_{\mathsf{p}^l}$ is performed.

In the second case, by the GC-Clean Migration Rule, process p^l must GC-clean every GC-dirty object in p^l and send the messages to create the corresponding scions. Thus, x^l is scanned, $\mathsf{stub}(\mathsf{B}^l\mathsf{x}^l, \mathsf{Dz})$ is created, and $<\mathsf{send.create.scion}(\mathsf{B}^l\mathsf{x}^l,\mathsf{Dz})>_{\mathsf{p}^l}$ is performed. Therefore, $<\mathsf{send.create.scion}(\mathsf{B}^l\mathsf{x}^l,\mathsf{Dz})>_{\mathsf{p}^l}$ precedes $<\mathsf{send.propagate}(\mathsf{x}^l,\ \mathsf{p}^l \rightsquigarrow \mathsf{p}^{h+1})>_{\mathsf{p}^l}$.

Since we assumed that messages are delivered in causal order, $<\mathsf{deliver.create.scion}(\mathsf{B}^l\mathsf{x}^l,\mathsf{Dz})>_{\mathsf{k}}$ happens before any other message sent (after the propagate) by p^{h+1} is delivered at k; in particular, a possible $<\mathsf{deliver.delete.scion}(\mathsf{B}^l\mathsf{x}^l,\mathsf{Dz})>_{\mathsf{k}}$ sent by p^{h+1} will be delivered at k after $\mathsf{scion}(\mathsf{B}^l\mathsf{x}^l, \mathsf{Dz})$ has been created. Thus, at all times at least one scion from p protects z. □

4.10.1.4 Liveness

We assume that *(i)* messages for creation and deletion of scions are eventually delivered (in causal order), *(ii)* every bunch is eventually collected, and *(iii)* intra-bunch collection is complete w.r.t. the collected bunch.

Our GC algorithm is clearly not complete because it does not reclaim cross-bunch cycles of garbage. Thus, we only consider the existence of acyclic cross-bunch garbage.

We will not present here the full proof of liveness given its triviality and consequent lack of interest. We simply show the conditions that must hold and how they are ensured.

The obvious liveness condition is:

Liveness Condition 1 *An object not reachable is eventually reclaimed.*

Given that we are considering cross-bunch collection, Liveness Condition 1 is equivalent to:

Liveness Condition 2 *An object not protected by any scion is eventually reclaimed.*

This condition is obviously ensured by the assumptions that every bunch is eventually collected, and intra-bunch collection is complete w.r.t. the collected bunch.

Note that, for liveness we must ensure that if an object is no longer reachable from any incoming cross-bunch pointer, eventually no scion protects it. Thus, the following liveness condition must hold:

Liveness Condition 3 *An object no longer reachable is eventually not protected.*

This condition is obviously ensured because: *(i)* we assumed that every bunch is eventually collected and intra-bunch collection is complete w.r.t. the collected bunch, therefore eventually there will be no stubs for disappearing outgoing pointers, and *(ii)* we assumed that messages for deletion of scions are eventually delivered. Therefore, a scion representing a incoming cross-bunch pointer no longer existing, will be eventually deleted.

4.10.2 Cross-Bunch GC with Replication

In this section, we describe rigorously the cross-bunch GC algorithm with replication. We present the invariants that must hold, and prove the algorithm is safe and live. (Recall that in our coherence model, Section 3.3.5, we assume that all data is cached by every process.)

4.10.2.1 Safety

The safety invariants are the same as in the non-replicated case (see Section 4.10.1.1). Thus, we do not repeat them here.

4.10.2.2 Algorithm

Let us return to the prototypical example illustrated in Figure 4.7, and observe that the initial situation satisfies Safety Invariant 2. At the time of $<x := y>_i$, object z is reachable (from both replicas of y) and is protected by some scion, say scion(Et, Dz) (presumably, but not necessarily, $E{=}C_j$ and $t{=}y_j$). As long as that scion continues to exist, it is safe to delay the creation of scion(Bx, Dz).

The following rules and algorithm ensure the correctness of the cross-bunch collector (intuitively presented Section 4.5.2) as will be formally proved in the next sections:

Comprehensive Scanning Rule
When an intra-bunch collection takes place at process i, every GC-dirty object y_i cached by i must be GC-cleaned.

GC-Clean Propagation Rule
An object y can be propagated from its owner process i only if y_i is GC-clean, and the messages to create the scions corresponding to y_i's outgoing cross-bunch pointers have been sent.

Cross-Bunch Collection Algorithm
At process i, between each pair of intra-bunch collections, execute these steps in the following order:

1. For all new cross-bunch stubs, perform the corresponding $<$send-.create.scion$>_i$.

2. For all disappearing cross-bunch stubs: *(i)* process i, not owner of an object y, whose outgoing pointer no longer exists (corresponds to the disappeared stub) sends a union message to the owner of y, *(ii)* process i, owner of an object y, issues the corresponding <send.delete.scion>$_i$ if the union of all stubs (for all replicas of the source object pointing to y) is empty.

The Comprehensive Scanning Rule means that when an intra-bunch collection takes place in process i every GC-dirty object (locally cached) must be scanned, therefore creating the corresponding stubs.

The GC-Clean Propagation Rule means that an object y cannot be propagated from its owner process i without having GC-cleaned it, thus after having created the corresponding stubs, and sent the messages to create their scions.

The cross-bunch collector runs between each pair of intra-bunch collections and compares the set of stubs before and after collection (recall Section 4.6). The Cross-Bunch Collection Algorithm ensures that for each intra-bunch collector run, all messages to create scions are sent before any union or delete messages. It is always the owner process of an object y that applies the union rule (concerning y's stubs), *i.e.*, the owner checks if the union of all stubs for all replicas of y is empty.

To understand the role of the Comprehensive Scanning Rule, consider Figure 4.7 after mutators have executed <x := y>$_i$, <y := 0>$_i$, and <y := 0>$_j$. Then, suppose that GC$_i$(C) runs without fulfilling the Comprehensive Scanning Rule, followed by the cross-bunch collector that sends a union message to j (owner of y) indicating that stub(Cy, Dz) has disappeared in process i. According to the Cross-Bunch Collection Algorithm, <send.delete.scion(Cy, Dz)>$_j$ will be executed. Thus, scion(Cy, Dz) is deleted in k. Then, if D is collected, object z is unsafely reclaimed. The Comprehensive Scanning Rule prevents the above scenario as it forces x_i to be GC-cleaned when GC$_i$(C) runs, thus stub(Bx, Dz) to be created in process i; then, according to the Cross-Bunch Collection Algorithm, <send.create.scion(Bx, Dz)>$_i$ is performed before j applies the union rule and executes <send.delete.scion(Cy, Dz)>$_j$. Since we assumed causal delivery, scion(Bx, Dz) is created before scion(Cy, Dz) is deleted. Consequently, z is not reclaimed by GC$_k$(D).

It is also interesting to understand the role of the GC-Clean Propagation Rule. Consider Figure 4.7 after the mutator has executed <x := y>$_i$ and before <y := 0>$_i$. Now, suppose that object x_i is propagated to some process w (in the absence of the GC-Clean Propagation Rule) and then x_i is assigned in such a way that it no longer points to z. At this moment, the only scion that protects z is scion(Cy, Dz). Suppose that both replicas of y are modified by the mutators in processes i and j such that they no longer point to z. Hence, by the Cross-Bunch Collection Algorithm, scion(Cy, Dz) is deleted. Thus, z may be unsafely reclaimed by GC$_k$(D) (x_w still points to z). The GC-Clean Propagation Rule prevents the above scenario as it forces <send.create.scion(Bx, Dz)>$_i$ to be performed before x_i is propagated to w.

Note that the GC-Clean Propagation Rule is weaker than its non-replicated counterpart (GC-Clean Migration Rule). While the latter requires every locally held GC-dirty object to be GC-cleaned before the migration, the former requires only the object being propagated to be GC-cleaned. This is due to the fact that when an object is propagated (replicated case) a replica of the object and its stubs remain in the owner process. Thus, they are still "seen" by the cross-bunch collector in that process. On the contrary, in the non-replicated case, when an object is migrated its stubs go along. Thus, they will not be "seen" by the cross-bunch collector in the source process.

4.10.2.3 Proof of Safety

To prove the safety of the cross-bunch collection algorithm, we start by proving the following Lemma.

Lemma 2 *Let* $\mathsf{x}^1, \ldots, \mathsf{x}^n$ *be the set of all protected objects, located in bunch* B*, pointing to* $\mathsf{z} \in \mathsf{D}$ *owned by* k*. Let* $\mathsf{p}^1, \ldots, \mathsf{p}^m$ *be the set of all processes caching a replica of* x^i*. Then,* $\forall i, 1 \leq i \leq n$*:*

- x^i *is GC-clean* $\wedge\ \exists\ \mathsf{scion}(\mathsf{Bx}^i, \mathsf{Dz})$*, or*
- $\exists j, 1 \leq j \leq n$*,* x^i *is GC-dirty* $\wedge\ \exists\ \mathsf{scion}(\mathsf{Bx}^j, \mathsf{Dz})$

This lemma says that, as long as there is a replica of some object x located in B, pointing to z, cached in some process, there exists a scion protecting z.

Proof of Lemma 2:

We prove this lemma by induction *(i)* over the size of the set containing objects pointing to z, and *(ii)* over the size of the set containing processes caching replicas of the objects pointing to z.

Base case:
Initially there is a single object $\mathsf{x}^1 \in \mathsf{B}$ pointing to z, x^1 is owned by process p^1, $\mathsf{scion}(\mathsf{Bx}^1, \mathsf{Dz})$ exists, $\mathsf{x}^2 \in \mathsf{B}$ does not point to z and is also owned by p^1, and neither x^1 nor x^2 are cached in p^2. This initial situation obviously verifies the lemma as x^1 is GC-clean and $\mathsf{scion}(\mathsf{Bx}^1, \mathsf{Dz})$ exists.

Now, perform $<\mathsf{x}^2 := \mathsf{x}^1>_{\mathsf{p}^1}$; thus, x^2 points to z and is GC-dirty. If x^1 is not assigned to thereafter, the lemma remains trivially true as x^1 is GC-clean and $\mathsf{scion}(\mathsf{Bx}^1, \mathsf{Dz})$ exists.

Therefore, consider that x^1 is changed by an assignment such as $<\mathsf{x}^1 := 0>_{\mathsf{p}^1}$ (the actual value of the right-hand side does not matter for the proof, except that when the right-hand side is a pointer to z it is as if the assignment did not take place.) Hence, x^1 is GC-dirty. Until an intra-bunch collection or a propagate operation takes place at p^1, no $<\mathsf{send.create.scion}>_{\mathsf{p}^1}$ or $<\mathsf{send.delete.scion}>_{\mathsf{p}^1}$ is performed, and the lemma remains trivially true since x^2 is GC-dirty and $\mathsf{scion}(\mathsf{Bx}^1, \mathsf{Dz})$ exists. We examine these two cases now: intra-bunch collection and propagate operation in p^1.

When an intra-bunch collection does execute in process p^1, by the Comprehensive Scanning Rule, both x^2 (containing the new pointer) and x^1 (previously containing the pointer with a scion) are GC-cleaned because both are GC-dirty, therefore $\mathsf{stub}(\mathsf{Bx}^1, \mathsf{Dz})$ disappears (not in the new set of stubs) and $\mathsf{stub}(\mathsf{Bx}^2, \mathsf{Dz})$ is created.

According to the Cross-Bunch Collection Algorithm, event $<\mathsf{send.create.scion}(\mathsf{Bx}^2,\mathsf{Dz})>_{\mathsf{p}^1}$ precedes $<\mathsf{send.delete.scion}(\mathsf{Bx}^1,\mathsf{Dz})>_{\mathsf{p}^1}$. We assumed that messages are delivered in causal order, hence $<\mathsf{deliver.create.scion}(\mathsf{Bx}^2,\mathsf{Dz})>_{\mathsf{k}}$ precedes $<\mathsf{deliver.delete.scion}(\mathsf{Bx}^1,\mathsf{Dz})>_{\mathsf{k}}$. Thus, at any moment, there exists at least a scion protecting z: either $\mathsf{scion}(\mathsf{Bx}^1, \mathsf{Dz})$, or $\mathsf{scion}(\mathsf{Bx}^2, \mathsf{Dz})$, or both. Therefore, the lemma remains true when an intra-bunch collection occurs in p^1.

Now, we consider a propagate operation. Before a $<\mathsf{send.propagate}(\mathsf{x}^2,\ \mathsf{p}^1 \rightsquigarrow \mathsf{p}^2)>_{\mathsf{p}^1}$ takes place, according to the GC-Clean Propagation Rule, x^2 is GC-cleaned. Therefore x^2 is scanned, $\mathsf{stub}(\mathsf{Bx}^2, \mathsf{Dz})$ is created and $<\mathsf{send.create.scion}(\mathsf{Bx}^2,\mathsf{Dz})>_{\mathsf{p}^1}$ is performed. Therefore, the lemma remains true as x^2 is GC-clean and $\mathsf{scion}(\mathsf{Bx}^2, \mathsf{Dz})$ exists.

Induction case:
Let $\mathsf{p}^1, \ldots, \mathsf{p}^h, 1 \leq h \leq m$ be the only processes caching objects $\mathsf{x}^1, \ldots, \mathsf{x}^j, 1 \leq j \leq n$, all $\in \mathsf{B}$, all pointing to $\mathsf{z} \in \mathsf{D}$ owned by k. The scions for the pointers from $\mathsf{x}^1, \ldots, \mathsf{x}^j$ to z do exist. Object x^j is owned by p^w, $1 \leq w \leq m$.

Let object x^{j+1} owned by p^w, initially not pointing to z, and process p^{h+1} initially not caching any object pointing to z.

Now, perform $<\mathsf{x}^{j+1} := \mathsf{x}^j>_{\mathsf{p}^w}$; thus, x^{j+1} points to z and is GC-dirty. If x^j is not assigned to thereafter, the lemma remains trivially true as x^j is GC-clean and $\mathsf{scion}(\mathsf{Bx}^j, \mathsf{Dz})$ exists.

Therefore, consider that x^j is changed by an assignment such as $<\mathsf{x}^j := 0>_{\mathsf{p}^w}$ (the actual value of the right-hand side does not matter for the proof, except that when the right-hand side is a pointer to z it is as if the assignment did not take place.) Hence, x^j is GC-dirty. Until an intra-bunch collection or a propagate operation takes place at p^w, no $<\mathsf{send.create.scion}>_{\mathsf{p}^w}$ or $<\mathsf{send.delete.scion}>_{\mathsf{p}^w}$ is performed, and the lemma remains trivially true since x^{j+1} is GC-dirty and $\mathsf{scion}(\mathsf{Bx}^j, \mathsf{Dz})$ exists. We examine these two cases now: intra-bunch collection and propagate operation in p^w.

When an intra-bunch collection does execute in process p^w, by the Comprehensive Scanning Rule, both x^{j+1} (containing the new pointer) and x^j (previously containing the pointer with a scion) are GC-cleaned because both are GC-dirty, therefore $\mathsf{stub}(\mathsf{Bx}^j, \mathsf{Dz})$ disappears (not in the new set of stubs) and $\mathsf{stub}(\mathsf{Bx}^{j+1}, \mathsf{Dz})$ is created.

According to the Cross-Bunch Collection Algorithm, event $<\mathsf{send.create.scion}(\mathsf{Bx}^{j+1},\mathsf{Dz})>_{\mathsf{p}^1}$ precedes $<\mathsf{send.delete.scion}(\mathsf{Bx}^j,\mathsf{Dz})>_{\mathsf{p}^1}$. We assumed that messages are delivered in causal order, hence $<\mathsf{deliver.create.scion}(\mathsf{Bx}^{j+1},\mathsf{Dz})>_{\mathsf{k}}$ precedes $<\mathsf{deliver.delete.scion}(\mathsf{Bx}^j,\mathsf{Dz})>_{\mathsf{k}}$. Thus, at any moment, there exists at least a scion protecting z: either $\mathsf{scion}(\mathsf{Bx}^j, \mathsf{Dz})$, or $\mathsf{scion}(\mathsf{Bx}^{j+1}, \mathsf{Dz})$, or both.

Therefore, the lemma remains true when an intra-bunch collection occurs in p^w.

Now, we consider a propagate operation. Before a $<\mathsf{send.propagate}(\mathsf{x}^{j+1}, \mathsf{p}^w \rightsquigarrow \mathsf{p}^{h+1})>_{\mathsf{p}^w}$ takes place, according to the GC-Clean Propagation Rule, x^{j+1} is GC-cleaned. Therefore x^{j+1} is scanned, $\mathsf{stub}(\mathsf{Bx}^{j+1}, \mathsf{Dz})$ is created and $<\mathsf{send.create.scion}(\mathsf{Bx}^{j+1},\mathsf{Dz})>_{\mathsf{p}^w}$ is performed. Therefore, the lemma remains true as x^{j+1} is GC-clean and $\mathsf{scion}(\mathsf{Bx}^{j+1}, \mathsf{Dz})$ exists.

This terminates the induction over the size of the set containing objects pointing to z, and over the size of the set containing processes caching replicas of the objects pointing to z. □

We now prove the following Theorem.

Theorem 2 *Let* $\mathsf{x}^1, \ldots, \mathsf{x}^n$ *be the set of all protected objects pointing to* $\mathsf{z} \in \mathsf{D}$ *owned by process* k*, located respectively in bunches* $\mathsf{B}^1, \ldots, \mathsf{B}^n$*. Let* $\mathsf{p}^1, \ldots, \mathsf{p}^m$ *be the set of all processes caching a replica of all bunches mentioned above. Then,* $\forall i, 1 \leq i \leq n$*,* $\mathsf{scion}(\mathsf{B}^i\mathsf{x}^i, \mathsf{Dz})$ *exists at* k*.*

This Theorem implies Safety Invariant 2. Whereas the latter states that, whatever number of cross-bunch pointers point to $\mathsf{z} \in \mathsf{D}$, at least one scion protects z, Theorem 2 says that at least one scion per bunch containing a pointer to z, protects z. It does not say whether this scion effectively corresponds to an existing pointer to z.

Proof of Theorem 2:

We prove this theorem by induction *(i)* over the size of the set containing objects pointing to z in a single process, and *(ii)* over the size of the set of processes caching replicas of objects pointing to z.

Base case:
Assume that initially $\mathsf{x}^1 \in \mathsf{B}^1$ and $\mathsf{x}^2 \in \mathsf{B}^2$ are both owned by p^1, only x^1 points to $\mathsf{z} \in \mathsf{D}$ owned by k, $\mathsf{scion}(\mathsf{B}^1\mathsf{x}^1, \mathsf{Dz})$ exists, and process p^2 does not cache any object pointing to z. This initial situation obviously verifies the theorem as $\mathsf{scion}(\mathsf{B}^1\mathsf{x}^1, \mathsf{Dz})$ exists.

Perform $<\mathsf{x}^2 := \mathsf{x}^1>_{\mathsf{p}^1}$ (now x^2 also points to z); therefore x^2 is GC-dirty. If x^1 is not assigned to thereafter, the theorem remains trivially true as $\mathsf{scion}(\mathsf{B}^1\mathsf{x}^1, \mathsf{Dz})$ exists.

Therefore, consider an assignment such as $<\mathsf{x}^1 := 0>_{\mathsf{p}^1}$ (the actual value of the right-hand side does not matter for the proof, except that when the right-hand side is a pointer to z it is as if the assignment did not take place.) Until an intra-bunch collection or a propagate operation takes place at p^1, no $<\mathsf{send.create.scion}>_{\mathsf{p}^1}$ or $<\mathsf{send.delete.scion}>_{\mathsf{p}^1}$ is performed, and the theorem remains trivially true.

In the first case, *i.e.*, when an intra-bunch collection takes place at p^1, by Lemma 1 we have that at any moment, there exists at least a scion protecting z: either $\mathsf{scion}(\mathsf{B}^1\mathsf{x}^1, \mathsf{Dz})$, or $\mathsf{scion}(\mathsf{B}^2\mathsf{x}^2, \mathsf{Dz})$, or both.

In the second case, *i.e.*, before $<\mathsf{send.propagate}(\mathsf{x}^2,\ \mathsf{p}^1 \rightsquigarrow \mathsf{p}^2)>_{\mathsf{p}^1}$ is performed, by Lemma 2 we have that $\mathsf{scion}(\mathsf{B}^2\mathsf{x}^2, \mathsf{Dz})$ exists.

Induction case:
Let objects $\mathsf{x}^1, \ldots, \mathsf{x}^j$, located in bunches $\mathsf{B}^1, \ldots, \mathsf{B}^j$, all pointing to z, all owned by $\mathsf{p}^h, 1 \leq h \leq m$.

Initially, $\mathsf{x}^{j+1} \in \mathsf{B}^{j+1}$, owned by p^h does not point to z, $\mathsf{scion}(\mathsf{B}^j\mathsf{x}^j, \mathsf{Dz})$ exists, and process p^{h+1} does not cache any object pointing to z. This initial situation obviously verifies the theorem as $\mathsf{scion}(\mathsf{B}^j\mathsf{x}^j, \mathsf{Dz})$ exists.

Perform $<\mathsf{x}^{j+1} := \mathsf{x}^j>_{\mathsf{p}^h}$ (now x^{j+1} also points to z), therefore x^{j+1} is GC-dirty. If x^j is not assigned to thereafter, the theorem remains trivially true as $\mathsf{scion}(\mathsf{B}^j\mathsf{x}^j, \mathsf{Dz})$ exists.

Therefore, consider an assignment such as $<\mathsf{x}^j := 0>_{\mathsf{p}^h}$ (the actual value of the right-hand side does not matter for the proof, except that when the right-hand side is a pointer to z it is as if the assignment did not take place.) Until an intra-bunch collection or a propagate operation takes place at p^h, no $<$send-.create.scion$>_{\mathsf{p}^h}$ or $<\mathsf{send.delete.scion}>_{\mathsf{p}^h}$ is performed, and the theorem remains trivially true.

In the first case, *i.e.*, when an intra-bunch collection takes place at p^h, by Lemma 1 we have that at any moment, there exists at least a scion protecting z: either $\mathsf{scion}(\mathsf{B}^j\mathsf{x}^j, \mathsf{Dz})$, or $\mathsf{scion}(\mathsf{B}^{j+1}\mathsf{x}^{j+1}, \mathsf{Dz})$, or both.

In the second case, *i.e.*, when event $<\mathsf{send.propagate}(\mathsf{x}^{j+1},\ \mathsf{p}^h \rightsquigarrow \mathsf{p}^{h+1})>_{\mathsf{p}^h}$ happens, by Lemma 2 we have that $\mathsf{scion}(\mathsf{B}^{j+1}\mathsf{x}^{j+1}, \mathsf{Dz})$ exists. □

4.10.2.4 Liveness

Just as for the non-replicated case, we simply show the liveness conditions that must hold and how they are ensured. The proof of liveness is equally trivial, thus not interesting.

The liveness conditions are the same as in the non-replicated case (recall Section 4.10.1.4). Therefore, we do not repeat them here.

We add another assumption to those made for the non-replicated case: we assume that union messages are eventually delivered.

Liveness Condition 3 is obviously ensured because: *(i)* we assumed that every bunch is eventually collected and intra-bunch collection is complete w.r.t. the collected bunch, therefore eventually there will be no stubs for disappearing outgoing pointers, *(ii)* we assumed that union messages are eventually delivered, and *(iii)* messages for deleting scions are eventually delivered. Therefore, a scion representing a incoming cross-bunch pointer no longer existing, will be eventually deleted.

4.11 Summary

In this chapter we described the GC algorithm designed for Larchant. We showed how GC is made scalable and asynchronous to applications, how the coherence interference problem is solved, and formally proved its correctness.

Our GC combines partitioned tracing with reference-listing. A bunch replica is collected independently from the rest of the memory, and concurrently with mutators. The reference-listing algorithm runs asynchronously to applications and manages sets of stubs and scions (describing cross-bunch pointers). We assume causal delivery.

Cycles of garbage are reclaimed by collecting groups of bunches at the same time. Such groups are formed according to a very simple heuristic that does not introduce extra I/O cost: a group contains all the bunches currently cached in the process.

Both mark-and-sweep and copy collectors can be used for intra-bunch collection. With replicated objects, both algorithms are capable of making progress without requiring coherent data, therefore no coherence operation is needed for GC purposes.

The union rule is a fundamental aspect of both intra-bunch and cross-bunch collection; it ensures safety in presence of replicated objects which are not necessarily coherent. Basically, the union rule says that a target object can be reclaimed only if it is not reachable from any replica of its source objects (independently of their location, either in the same or in a different bunch).

Our GC algorithm is widely applicable given its asynchrony to applications, and the minimum assumptions we made concerning objects coherence.

Chapter 5

Implementation of GC in Larchant

The overall goal of this chapter is to present the most important aspects concerning the implementation of the GC algorithm in Larchant. We describe both mark-and-sweep and copy intra-bunch collectors, and the reference-listing cross-bunch collector.

First, we give a high-level view of the architecture of Larchant. Then, we describe the prototype implementation, without getting into much detail. A detailed analysis would necessitate an exhaustive study of the source code, which is out of the scope of this document.

Then, we describe the coherence protocol we implemented: entry-consistency [14]. It follows the intra-bunch collector (mark-and-sweep and copy algorithms) and the cross-bunch collector. We focus on their interaction with entry-consistency. This description shows how the general solutions presented in Chapter 4 are applied to the particular algorithms implemented in the prototype.

Then we make some considerations concerning the impact of the size of the coherence granule on the collection algorithm. We finish this chapter with the description of the programming restrictions imposed by GC when using the C++ language.

5.1 Architecture

In this section we describe the Larchant architecture. We give a high-level view of its major components. The goal of this description is to present the main characteristics of the prototype in which the garbage collector is integrated.

5.1.1 Basics

Applications share a single 64-bit address space spanning every site in the network, including secondary storage. Objects are identified by their virtual ad-

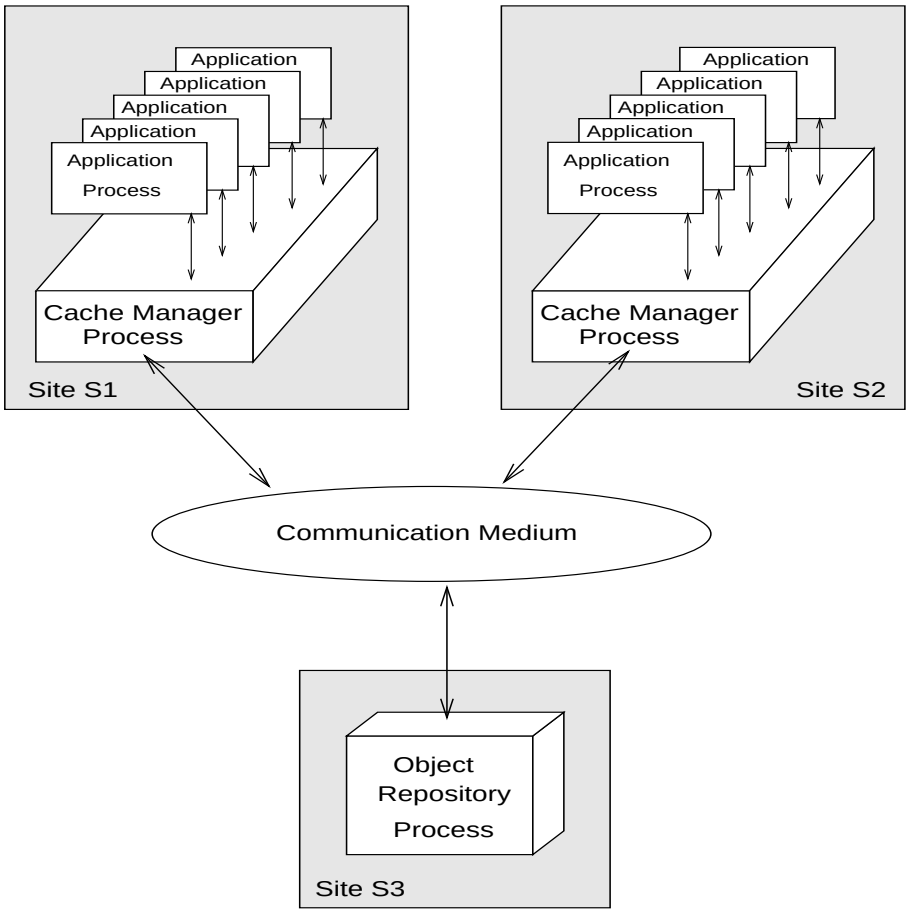


Figure 5.1: *Larchant architecture.*

dress, and are replicated via a DSM mechanism with the entry-consistency protocol.

Applications explicitly start, commit, and abort transactions. We use a very simple transaction model [13]: single-process transactions, two-phase locking. Each transaction updates a cached image of the store in virtual memory. The store goes from one stable state to another by means of a commit operation.

Since transactions are single-process, only one process (among those sharing a bunch at the same time) commits a transaction; we call that process the *principal*. The principal of a bunch is the process that initiated the transaction enclosing that bunch. The other processes sharing the same bunch at the same time are called *associates*. The store is updated with the cached image held by the principal.

The overall architecture of Larchant is illustrated in Figure 5.1. On each site there is a Unix process called *Cache Manager* (CM). Its main task is to serve applications requests, and manage a local cache of bunches.

The *Object Repository* (OBR) process manages the stable store. (As explained afterwards, there can be several OBRs for increased performance and reliabil-

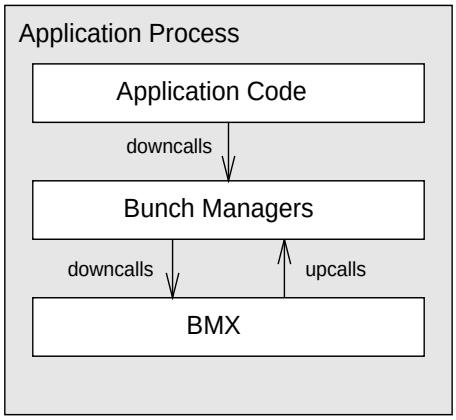


Figure 5.2: *High-level view of an application process.*

ity.) Its mains task is to store (committed) data on disk and serve requests issued by CMs.

We now describe the most important structural aspects of the Larchant architecture. This concerns applications, cache management, object storage, root of persistence, address space management, and garbage collection.

5.1.2 Applications

Figure 5.2 gives an high-level view of the internal organization of an application process. An application process is composed of the application code, one or more *Bunch Managers*, and a *Bunch Manager eXecutive* (BMX), all linked together. We now explain the functionality of bunch managers and BMX.

The concept of bunch manager [53] is key to the flexible support of multiple data management policies in Larchant. (Recall that objects are allocated in bunches; thus, a bunch can be seen as a container for objects.) Such policies can be, for example, the binding protocol (what kind of mechanism can be used to remotely access a bunch, RPC or DSM) [107], a coherence engine that implements an optimized coherence protocol for the bunch, the most appropriate GC algorithm for the bunch, etc.

To support such a variety of policies, each bunch has an associated user-level object, called its bunch manager. The bunch manager implements the policies most appropriate to the objects allocated inside the corresponding bunch. Bunch managers can be improved, modified, or developed by experienced application programmers.

An application programmer links the application code with the bunch manager libraries he thinks are most appropriate for managing its data. For example, if an application only uses objects that are known to be never shared, the application programmer will use a very simple bunch manager, that lacks a coherence engine, to allocate the application objects. A more realistic example is that of an application that generates a large amount of objects whose majority

remains reachable for a very short period of time. In this case, the application programmer will use a bunch manager providing a garbage collector based on the copy algorithm. In fact, a copy algorithm is the most appropriate for this situation (as mentioned in Chapter 2).

The BMX is a library that is linked with each application running on top of Larchant. It allows the application code to interact with the rest of the system (the local CM, more exactly) via bunch managers. BMX provides basic support for the specialized policies implemented by bunch managers. In particular, it implements the minimal generic mechanisms of coherence management and garbage collection that are used by bunch managers. For example, the BMX starts a group garbage collection, including every bunch cached by the process, by up-calling the corresponding GC methods provided by the bunch managers.

5.1.3 Cache Manager

On each client site there is a CM process. It serves applications requests performed through their BMX library, manages the slices of the shared 64-bit address space which are reserved for the creation of new bunches by local applications, and collects cached bunches when they are not being used locally. We describe briefly each one of these functions now.

A fundamental request served by CMs is that of getting a bunch not yet mapped in a process, for application access. In other words, a CM serves bunch faults as explained now. When an application accesses an object in a bunch not yet mapped in the process, a bunch fault is automatically generated in the binding method of corresponding bunch manager (recall the binding policy mentioned in the previous section). The bunch manager issues a request, for the faulted bunch, to the BMX. Then, the BMX forwards the request to the local CM that looks for the bunch in its cache. Then, there are the following possible evolutions:

- The CM finds the bunch in its cache.

  It returns immediately the bunch contents to the BMX that forwards it to the bunch manager. The bunch manager maps the bunch in the process and the application resumes.

- The faulted bunch is not in CM's cache.

  The CM asks the bunch to the OBR, which can reply either with the bunch contents or with the identification of the principal's CM. (The principal's CM is the CM on the site where the principal executes.)[1]

  In the first case, we proceed as in the previous item. In the second case, the CM sends a request to the principal's CM, asking for the bunch. This CM forwards the request to the principal's BMX which upcalls the

[1] Recall the definitions of principal and associate given in Section 5.1.1: the principal of a bunch is the process that initiated a transaction enclosing that bunch; the other processes sharing the same bunch at the same time are called associates.

binding method of the corresponding bunch manager. The reply from the bunch manager depends of the binding policy: it allows the sharing of the bunch via DSM or RPC. Here we only address the DSM case: the bunch manager replies with the bunch contents to the principal's BMX, which forwards it to the principal's CM, which also forwards the reply to the CM serving the bunch fault. Now, we proceed as in the previous item.

Note that, in the above description, when we say that the bunch contents are sent to the application, this does not mean that all the bunch is sent at once. In fact, this is done lazily, *i.e.*, we only send those objects (and some system information) that are effectively needed to allow the application to proceed.

Another important request served by CMs is that of obtaining a coherent replica of an object. When an application needs a coherent replica of an object whose enclosing bunch has already been mapped in the process, the mutator (through the BMX library) sends a request to the local CM. This forwards the request to the CM in the site where the owner process is running.[2] Then, this CM forwards the request to the owner. The reply with the object contents flows in the opposite direction.

Each CM manages the slices of the 64-bit address space reserved for the local applications usage. New bunches created by local applications are allocated in the mentioned slices. When local slices are exhausted, the CM gets more using a procedure described in Section 5.1.6.

Each CM can also collect the bunches kept in its cache. For this purpose, the CM launches a process where the bunches to be collected are mapped. The garbage collection algorithm used is the one implemented by the corresponding bunch manager. The bunches that are candidates for being collected by the CM are those that are not being used by any local application. (More details on this in Section 5.1.7).

5.1.4 Object Repository

The main task of the OBR is to store (committed) data on disk and serve requests issued by CMs. Figure 5.1 shows only one OBR process, however there can be several for increased performance and reliability. Note that the existence of several OBRs may lead to the need of a two-phase commit protocol [13]. We do not address this issue because it is out of the scope of this document.

Recall that if a bunch is requested by a CM, and that bunch is already cached in some application process, the OBR returns the identification of the principal's CM. For this purpose, the OBR keeps for each bunch the identification of its principal's CM.

The OBR can garbage collect any bunch of the store: it launches a process where the bunches to be collected are mapped and invokes the corresponding

[2] Note the difference between principal and owner: the former relates to a bunch while the latter relates to a coherence granule inside a bunch, *i.e.*, an object. The owner of an object in a bunch B is not necessarily B's principal.

methods of their managers. More details on this in Section 5.1.7.

5.1.5 Persistent Root

In Larchant there is a special persistent object called *global-root* which is the persistent root of the Larchant store. The global-root object holds references to other objects which can be associated with human-readable names, also kept in the global-root object. It provides an interface similar to a name service as it offers methods to insert and remove references, to obtain a reference to an object given its human readable name, etc.

The global-root object is like any other object in the system and can be replicated all over the network. It is allocated in a bunch whose manager implements a coherence protocol that ensures a single-writer multiple-reader sharing policy. Note that for security reasons, the global-root object is inside a bunch that can be accessed for writing only by privileged processes, in particular CMs (see bunch protection in Section 5.2.2).

5.1.6 Address Space Management

Larchant supports a single 64-bit address space spanning every site in the network, including secondary storage. Objects are identified by their 64-bit virtual address.

In order to ensure that bunches have non-overlapping addresses, there is a special persistent object called *Global Address Space Manager* (GASM) which manages a free list of 64-bit address slices.

Each CM holds one (or more) slice(s) for local applications usage, *i.e.*, where local applications allocate new bunches. Each time a CM needs more address space, it accesses a coherent replica of the GASM object, and invokes a method to reserve a slice. A slice is freed by a CM, and returned to the GASM as a free slice, when the bunches allocated inside it become empty, *i.e*, no longer contain any reachable object.

The GASM object is just like any other object in the system. It is referenced from the global-root object, and can be replicated all over the network. It is allocated inside a bunch whose manager implements a coherence protocol that ensures a single-writer multiple-reader sharing policy. Note that for security reasons, the GASM object is inside a bunch that can be accessed only by privileged processes, in particular CMs (see bunch protection in Section 5.2.2).

5.1.7 Garbage Collection

Each bunch is collected by a specific GC algorithm which is implemented by the corresponding bunch manager. A bunch can be collected in the application process, or in a process launched by a CM or by the OBR. Note that in the last two cases, the process in which the collector runs does not contain a mutator. This is equivalent to run the collector in GC-only mode.

The GC of a bunch being used by an application, can be started either by the bunch manager or by the process's BMX. In the first case, the goal is to collect a single bunch. The collection starts because the bunch manager detected that the bunch size has reached some threshold, or because the application explicitly required so.

In the second case (collection started by the BMX) the goal is to collect every bunch in the process at the same time, in order to reclaim cross-bunch cycles. This collection starts when the amount of memory allocated by the bunches mapped in the process reaches some threshold, or when the application requires it explicitly through a bunch manager.

The CM collects the bunches kept in its cache. A goal of this GC is to reclaim cross-bunch cycles in the group of bunches cached by the CM. Another goal is to collect those bunches whose size has increased considerably since they were cached, and were not collected while being used by applications. These bunches are suspected to contain a large amount of unreachable objects. Thus, the bunches that are candidates for being collected by the CM are those that are not being used by any local application, and those that are suspected to have a large amount of unreachable objects.

The OBR collects bunches that are not being cached by any CM. The goal is to collect large groups of bunches in order to reclaim a maximum amount of cross-bunch cycles.

In summary, a GC started by the BMX reclaims small-scale cross-bunch cycles; a GC launched by a CM reclaims medium-scale cross-bunch cycles; a GC launched by the OBR reclaims large-scale cross-bunch cycles. We do not go into more details concerning when such collections are launched or, in the case of the OBR, which heuristics can be used to form groups of bunches in order to maximize the amount of cycles reclaimed.

5.2 Prototype Implementation

We implemented a prototype of Larchant, which is as simple as possible, yet sufficiently complete to test the GC algorithms and to execute distributed applications with persistent objects. In this section we present the main aspects of the prototype.

Larchant is programmed in C++. The total size of the system is about 45000 lines. In fact, many of these lines correspond to comments and to probing methods. The latter monitor the system internal data structures and check internal invariants. These monitoring code is used only for debugging.

Thus, the core of Larchant (including the mark-and-sweep collector, the copy collector, the DSM mechanism with the entry-consistency protocol, the OBR, the CM, and the global-root and GASM objects) comprises about 35000 lines of C++ code.

The prototype runs on a network of DEC Alpha workstations connected by

FDDI. The host operating system is OSF/1 which offers a 64-bit address space.[3] We take advantage of this large address space by mapping objects at fixed addresses, therefore avoiding the cost of swizzling [128].

In the rest of this section we describe the most relevant high-level aspects concerning application processes, their threads and ports, and the internal structure of bunches and objects.

5.2.1 Processes, Threads, and Ports

The current prototype implements a simplified version of the Larchant architecture previously described. There is a single Unix process per site that includes some application program, linked with the BMX library and the CM code (recall Figure 5.1). The OBR runs in a separated process.

Within each application process there are the following threads, each one with a particular functionality and using one or more communication ports (see Figure 5.3):

- *Collector Thread.*

  It executes the code of the intra-bunch and cross-bunch collectors; uses port bmxUdpSend to send messages related to GC; in particular, the move messages of a copy collector as described in Section 4.4.2.3.

- *Executive Thread.*

  It executes the code of the BMX library; it uses the ports bmxTcpRecv and bmxUdpRecv on which it waits for messages requesting a bunch (TCP protocol) or a coherence operation (UDP protocol) as described in Section 5.1.3.

- *Mutator thread.*

  A mutator contains one or more threads that execute the code of the application program. They use (through the BMX library, thus transparently to the application code) the ports bmxTcpSend and bmxUdpSend to send requests for bunches (TCP protocol) or for coherent objects (UDP protocol), and receive the corresponding replies.

- *Manager Thread.*

  It executes the code of the CM; waits for messages on ports cmTcpRecv and cmUdpRecv requesting a bunch (TCP protocol) or a coherence operation (UDP protocol); sends messages to local application processes or to remote CMs requesting bunches (cmTcpSend) or a coherence operation (cmUdpSend).

[3] In fact, the implementation of the Alpha microprocessor we use supports a 43-bit virtual address space with a page size of 8 Kbytes [113].

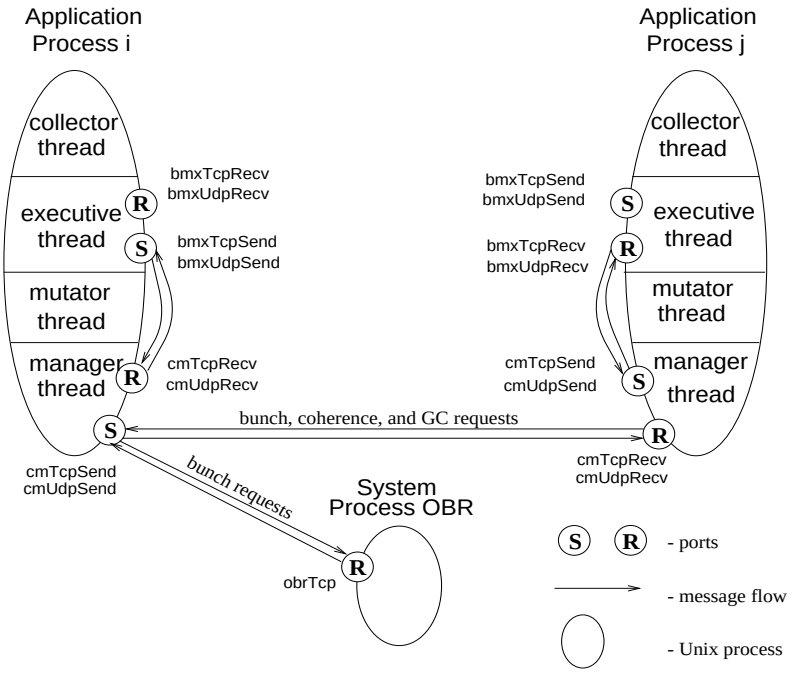


Figure 5.3: *Sketch of Larchant implementation. The ports whose name encloses* Tcp *use the TCP protocol; those whose name encloses* Udp *use UDP. A port marked* **R** *means that there is a thread mostly waiting on it in order to serve requests; a port marked* **S** *means that a thread uses it to send messages.*

Suppose that a mutator of some application process i, needs to access a bunch B, which is not yet locally mapped. (Recall the mechanism described in Section 5.1.3). Then, the mutator thread requests the faulted bunch, through the BMX library, to the local CM (by sending a message from bmxTcpSend to cmTcpRecv).[4]

If the faulted bunch is not locally cached, the CM requests it to the OBR by sending a message from port cmTcpSend to obrTcp. If the local CM requests the faulted bunch to the principal's CM, it sends a message from port cmTcpSend to principal's cmTcpRecv. This CM will forward the request from cmTcpSend to bmxTcpRecv. The reply will flow in the opposite direction.

When a mutator needs a coherent replica of some object (located inside a bunch that has already been mapped in the enclosing process) it uses the bmxUdpSend port (through the BMX library) to send a request message to the local CM (port cmUdpRecv). This forwards the request to the CM local to the object's owner (from cmUdpSend to cmUdpRecv) according to the coherence protocol (see Section 5.3). The latter CM forwards the request to the object's owner (to port bmxUdpRecv). The reply flows in the opposite direction with the object's contents.

[4] In the current release, the mutator and the CM run within the same Unix process. Thus, the message exchange is just a procedure call.

Finally, the bmxUdpRecv port also receives move messages from the intra-bunch copy collector thread of another process caching a replica of the same bunch (recall Section 4.4.2.3).

5.2.2 Bunches and Objects

A bunch contains a unlimited number of segments. A *segment* is a set of contiguous virtual memory pages; it has a fixed (but arbitrary) size. Segments in a bunch do not need to be contiguous.

Each segment is backed by a Unix file. All files backing segments in a bunch have the same owner and protections. Special objects like the global-root and the GASM are allocated in bunches owned by the administrator of the Larchant store; therefore, they can be accessed only by privileged processes like the OBR and CMs (recall Sections 5.1.5 and 5.1.6).

A segment can be mapped using Unix V shared memory, memory mapped files, or by reading the file into an anonymous memory mapped region. The memory mapped file mechanism is primarily intended to be used within a local area network since it can take advantage of existing distributed file systems (such as NFS [117]).

Due to garbage collection, some segments in a bunch may become empty, *i.e.*, with no reachable objects inside. In this case, the virtual memory space occupied by such a segment is freed to the corresponding CM. The CM adds that memory space to its list of free address space slices, for future utilization.

Each object has a header which contains system-specific information such as its size, its state (if it is mapped, for example), a pointer to the bunch manager corresponding to the enclosing bunch, etc. Note that accessing this information does not cost any extra swapping. In fact, when Larchant needs to access it, the corresponding object is already in main memory due to application needs.

On the contrary, GC specific information such as the color of an object might be accessed by the collector when the corresponding object is swapped out. Thus, to avoid extra swapping costs, the color of each object is stored in a bitmap separated from the corresponding object.

5.3 Coherence Engine

The Larchant prototype supports a single bunch manager type, which coherence engine implements the *entry-consistency* protocol [14], described in Section 3.4. We recall here only the most important aspects.

Every shared object has a owner, which is either the process currently holding the object's write token, or the process that last held the write token. (There can either be several read tokens, or one exclusive write token associated with a shared object.)

A write token can only be obtained from the object's owner. In the current prototype a read token is also obtained from the object's owner; in the future,

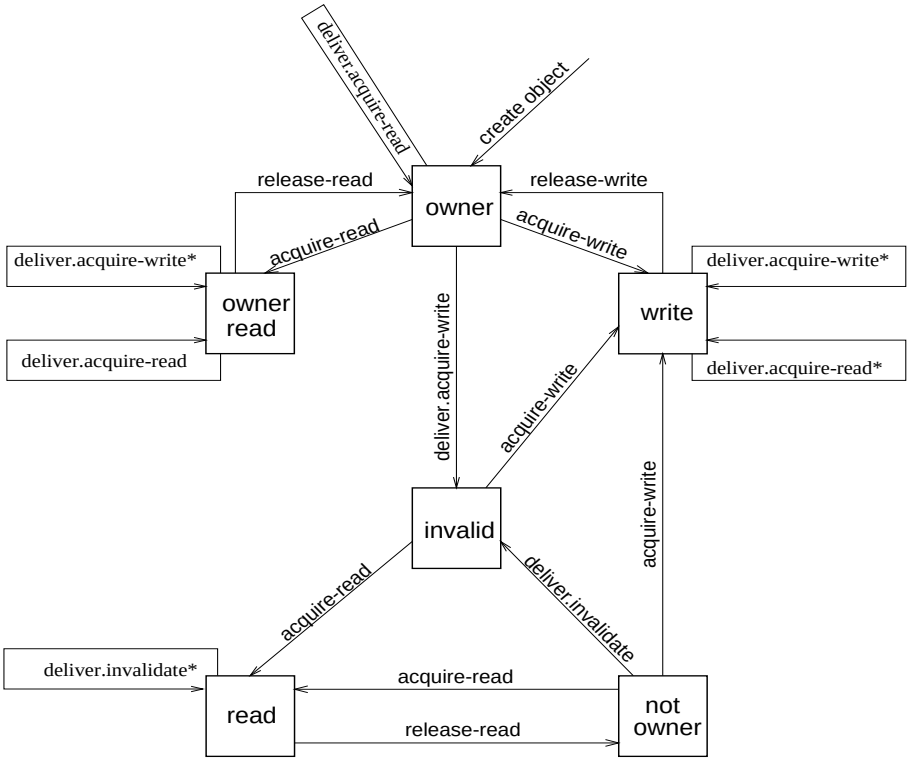


Figure 5.4: *State-machine for the entry-consistency protocol.*

we intend to optimize this by allowing a read token to be obtained from any process already holding a read token.

The acquisition of a write token for an object implies the invalidation of every readable replica of that object in other processes. Each process receiving an invalidation message acknowledges it via a reply.

Figure 5.4 illustrates the state-machine that represents the entry-consistency protocol just described. The acquisition of a write token or of a read token is made by operations acquire-write and acquire-read, respectively. When the object's owner is remote, acquire-write designates the execution of send.acquire-write, deliver.acquire-write, and the corresponding reply. (The same reasoning applies to the operation acquire-read.)

In Figure 5.4, the character * identifies those requests (read or write) that are not immediately satisfied, *i.e*, they will be served once the concerned object is released. For example, when an object is in state owner-read, a request for a write token (deliver.acquire-write) will be served only after the object's state becomes owner; until then, the requesting thread remains blocked.

5.4 Intra-Bunch Garbage Collector

Our bunch manager implements two GC algorithms: mark-and-sweep and copy. Both algorithms can be executed concurrently with the mutator, or in GC-only

mode, *i.e.*, the mutator is halted while the collector runs. When running in concurrent mode, both algorithms are based on the replication technique from Nettles and O'Toole [91] (recall Section 2.2.3.1).

In this section we present the most important implementation aspect of both intra-bunch GC algorithms: their state-machines.

The state-machine shows the states, *i.e.*, the colors of each object (recall the tricolor algorithm described in Section 4.9) and the transitions between states. It also shows how the GC algorithm interacts with entry-consistency.

The C++ code of the collectors resulted from a simple mapping (almost automatic) of the corresponding state-machine into: *(i)* a variable (stored in a bitmap) associated to each object defining its color, *(ii)* a few C++ switch instructions that for each initial color sets the new color of the concerned object, depending of the transition, as defined by the state-machine.

We terminate this section with a description of how GC-dirty objects are detected by taking advantage of the entry-consistency protocol.

5.4.1 Mark-and-Sweep

The mark-and-sweep algorithm in presence of the entry-consistency protocol is described by the state-chart[5] [60] of Figure 5.5. It results from the combination of the state-machines of Figures 4.13-(a) and 5.4. Some aspects of the state-chart deserve special attention:

- A reachable object can be made black by the collector no matter the object coherence state, except when it is being written (write state). In this case, after being scanned, the object goes from gray to gray-log which means that the object will have to be scanned again.
- G and H are auxiliary states used only for clarity. For example, when going from coherence state invalid to write due to an acquire-write operation, the object GC state H or G is maintained. This means that from black or gray-log (state H) the object color goes to gray-log (state H); the other colors, white and gray (state G) do not change. Similarly, the transition from coherence state not-owner to write implies that the object's color goes from black or gray-log to gray-log (auxiliary state H is kept); in the same coherence transition, a white object remains white, a gray object remains gray.
- The only coherence state transitions that may imply a color change are those that end in the write state: from owner to write, from invalid to write, and from not-owner to write. The color change is always from black to gray-log. Any other coherence state transition does not influence an object's color.

[5] A state-chart is a visual formalism similar to, but more powerful than a state-machine. It extends the latter with the notions of depth, orthogonality, and broadcast communication leading to more concise specifications.

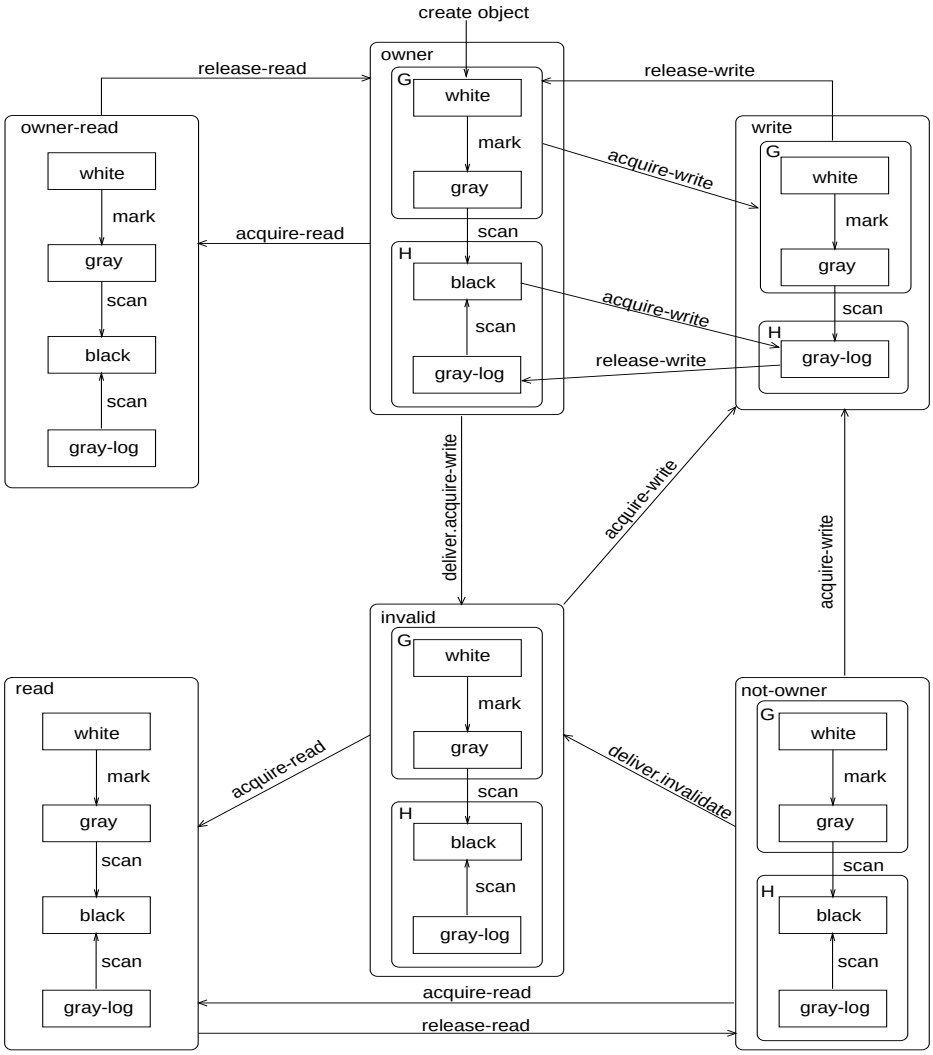


Figure 5.5: *State-chart of an object located in a bunch which manager supports the entry-consistency protocol and the mark-and-sweep GC algorithm.*

From the state-chart it is clear that only when an object is acquired for writing, and the intra-bunch collector is running, it is necessary to perform a GC operation: possibly changing the object's color from black to gray-log. In addition, it is clear that the GC algorithm never requires a coherence operation (*e.g.*, acquiring a read or a write token). Thus, there is no coherence interference.

5.4.2 Copy

In this section we describe the state-machine of the copy collector. We pay special attention to the interaction between the GC algorithm and the entry-consistency protocol.

Given the replication technique, the state-machine describing the colors and

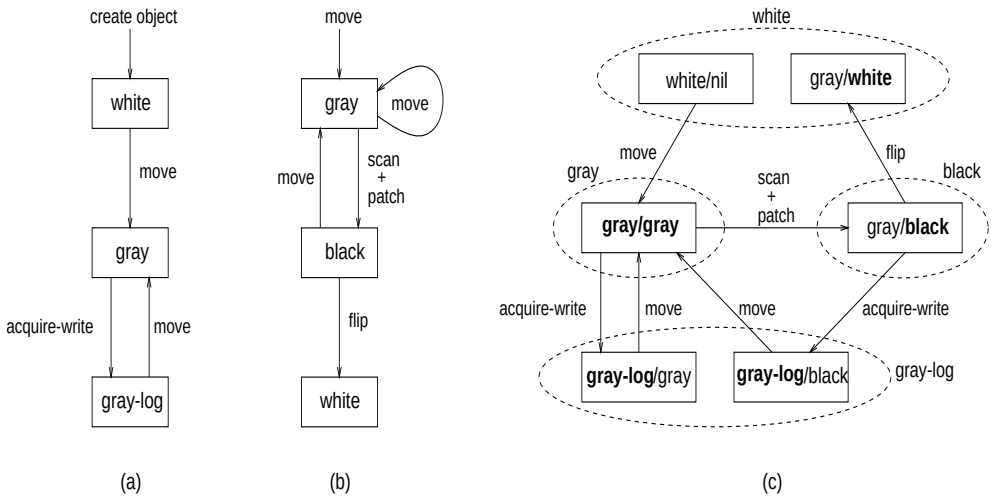


Figure 5.6: *State-machine of a copy collector: (a) from-space object; (b) to-space object; (c) merge of (a) and (b). In the latter, the color of an object (including both "replicas" in from-space and in to-space) is represented as* color-of-replica-in-from-space/color-of-replica-in-to-space*; to make clear which replica is affected in each transition, the changed color is shown in bold.*

transitions of an object located in from-space, is different from the state-machine of its "replica" in to-space. Since the collector only scans and patches to-space objects, only these will become black. Objects located in from-space can only become gray as they are moved to to-space; they are never scanned nor patched.

Figure 5.6 illustrates the state-machines describing: (a) the colors and transitions of an object in from-space, (b) the colors and transitions of an object's replica in to-space, and (c) the merged color and transitions of both replicas of an object (in from-space and in to-space). In the latter state-machine, the dashed ellipses and their state labels highlight the equivalence with the general state-machine presented in Figure 4.13-b.

The operations that trigger the color transitions are executed either by the collector (move, scan, patch, flip) or by the mutator (acquire-write). There is no mutator operation in the state-machine for a to-space replica (Figure 5.6-(b)) because the mutator only accesses from-space replicas. Hence, the mutator operation acquire-write is applied only on the from-space replica. Conversely, move affects both replicas in from-space and in to-space; the scan, patch and flip operations affect only the to-space replica. Note that when the from-space replica's color is gray-log and its to-space replica is black, the moving of the former by the collector forces the transition of the to-space replica to gray. This is due to the fact that the to-space replica must be scanned again.

The copy algorithm in presence of the entry-consistency protocol is described by the state-chart of Figure 5.7. It results from the combination of the state-machines of Figures 5.4 and 5.6-(a). Some aspects of the state-chart deserve to

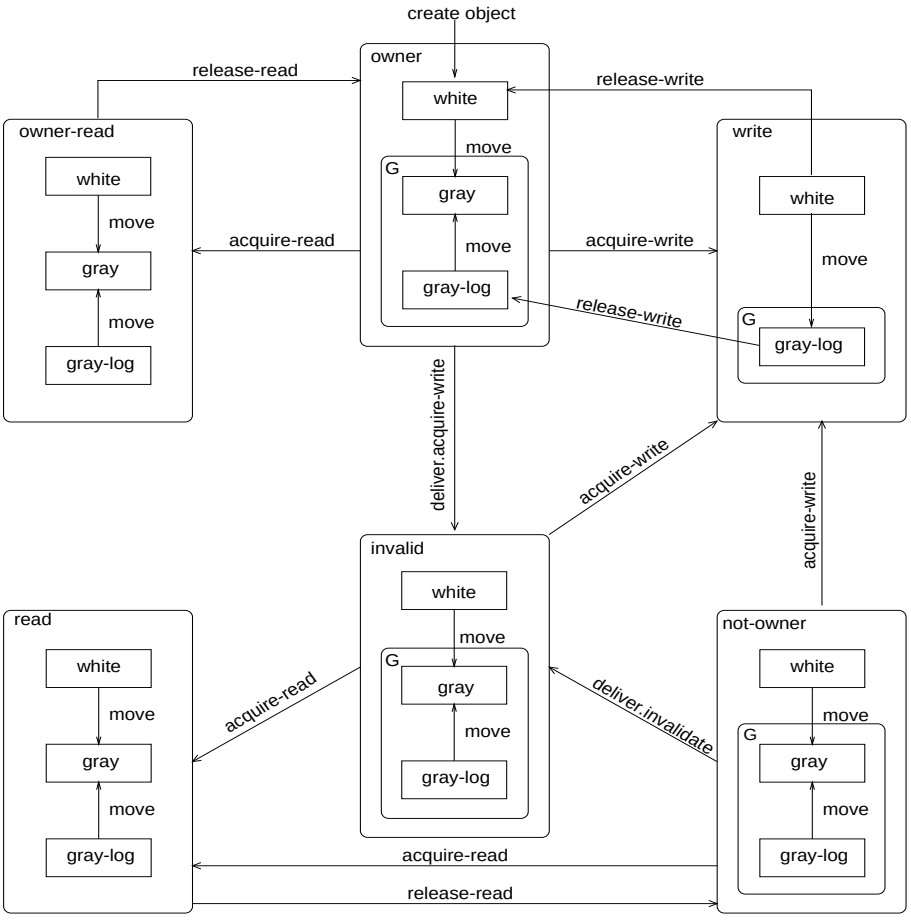


Figure 5.7: *State-chart of a from-space object located in a bunch which manager supports the entry-consistency protocol and the copy GC algorithm.*

be mentioned:

- G is an auxiliary state used only for clarity. For example, when going from coherence state invalid to write due to an acquire-write operation, the object GC state G is maintained. This means that from gray or gray-log (state G) the object color goes to gray-log (state G); the other color (white) does not change. Similarly, the transition from coherence state not-owner to write implies that the object's color goes from gray or gray-log to gray-log (auxiliary state G is kept); in the same coherence transition, a white object remains white. The same happens when going from coherence state owner to write.

- When the collector finds a not-owned object (states read, invalid, not-owner) it can only move it to to-space after its new address has been locally delivered (sent by the object's owner). To avoid halting the collection while waiting for the object's to-space address, the collector thread sends a message to the object's owner requesting its new address and does not wait for the reply. Once the message is sent, the collector scans the

from-space object (that can even be invalid) and continues tracing; the object color remains unchanged. Later, when the object's new address is delivered the object will be moved. In the current implementation, new address messages are batched and sent by the owner process into a single message.

- The only coherence state transitions that may imply a color change are those that end in the write state: from owner to write, from invalid to write, and from not-owner to write. The color change is always from gray to gray-log. Any other coherence state transition does not influence an object's color.

From the state-chart it is clear that only when an object is acquired for writing, and the intra-bunch collector is running, it is necessary to perform a GC operation: possibly changing the object's color from gray to gray-log. In addition, it is clear that the GC algorithm never requires a coherence operation (*e.g.*, acquiring a read or a write token). Thus, there is no coherence interference.

5.4.3 Detection of GC-dirty Objects

As discussed in Chapter 4, the collector must know which objects are GC-dirty. There are numerous ways to implement the detection of GC-dirty objects (or bunches).

A compiler may insert a write-barrier instrumenting every pointer write, that sets a GC-dirty bit associated with the object [127]; collecting a bunch resets the GC-dirty bits of its objects.

If the coherence protocol only allows the owner of an object to modify it, then we may instead conservatively assume that any object the current process owns, is GC-dirty.

Another alternative, assuming objects are protected by locks, is to set the bit when taking a write lock; in this case, the scan resets the GC-dirty bit only if the lock has been released [54]. This is the alternative we implemented in Larchant as it takes advantage of the entry-consistency protocol.

To modify an object x, the entry-consistency protocol requires that the process where the mutator is running, holds the write token of x. So, when the mutator performs acquire-write of x, and before the acquisition completes, x's address is inserted in a list of GC-dirty objects.

Once a GC-dirty object has been scanned, and its write token is not being held, it becomes GC-clean and is removed from the list of GC-dirty objects. Note that during the flip, the intra-bunch collector scans every object whose write token is being held. Thus, after being scanned such an object remains GC-dirty.

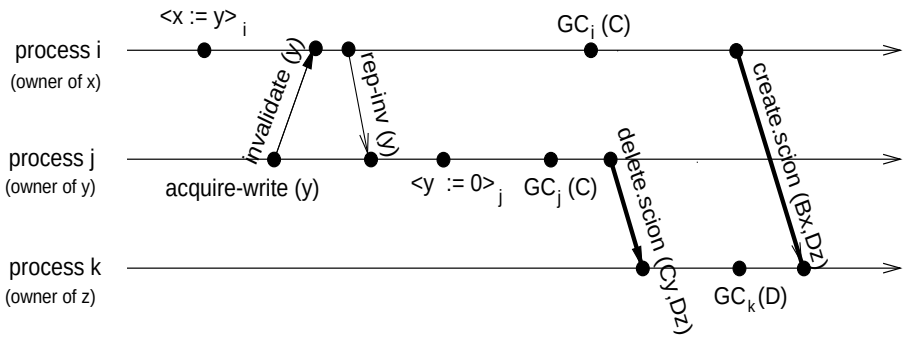


Figure 5.8: *Timeline showing a possible problem with the asynchronous creation of scions in presence of the entry-consistency protocol: the stub-scion pair describing* Bx→Dz *is created too late (for safety purposes).*

5.5 Cross-Bunch Garbage Collector

In this section we present the most important implementation aspect of the cross-bunch collector: how the promptness condition is fulfilled (recall Section 4.5) in presence of the entry-consistency protocol: *(i)* all create messages resulting from an intra-bunch collector run must be sent before any union or delete message in the same run, and *(ii)* create and delete messages must be delivered in causal order.

Figure 5.8 illustrates the prototypical situation of Figure 4.8, restricted to the case of the entry-consistency protocol, and with the difference that $\langle y := 0 \rangle_i$ is not executed. This difference is not relevant and makes our description more clear. In fact, when $\langle y := 0 \rangle_j$ is executed, Cy→Dz is immediately "deleted" everywhere (including from y_i); indeed, j holds the only valid replica of y.

An interesting aspect is that the cross-bunch collector in j cannot take advantage of this implicit information (that Cy→Dz no longer exists once $\langle y := 0 \rangle_j$ is executed) because this could lead to the premature deletion of scion(Cy, Dz), as explained now.

Consider Figure 5.8: object x is owned by process i, and object y is owned by process j. After the execution of $\langle y := 0 \rangle_j$, the $GC_j(C)$ runs enabling the detection that Cy→Dz no longer exists; as process j is the owner of y, this means that no other replica of y points to z. Thus, apparently, scion(Cy, Dz) could be deleted at once. Consequently, $\langle$send.delete.scion(Cy, Dz)$\rangle_j$ could be performed before $\langle$send.create.scion(Bx, Dz)$\rangle_i$. Then, z could be unsafely reclaimed by $GC_k(D)$ (note that Bx→Dz still exists). Therefore, in spite of the knowledge an object's owner has concerning the object's outgoing pointers, the union message (and the corresponding union rule) is still needed.

Figure 5.9 extends Figure 5.8 with the union rule: process j, the owner of y, can perform $\langle$send.delete.scion(Cy, Dz)$\rangle_j$ only after having made the union of all y's stubs, and the resulting set is empty. Note that we add the propagation of y from j to i; otherwise, the new set of stubs generated by $GC_i(C)$ would contain stub(Cy, Dz) and by the union rule, scion(Cy, Dz) would not be deleted.

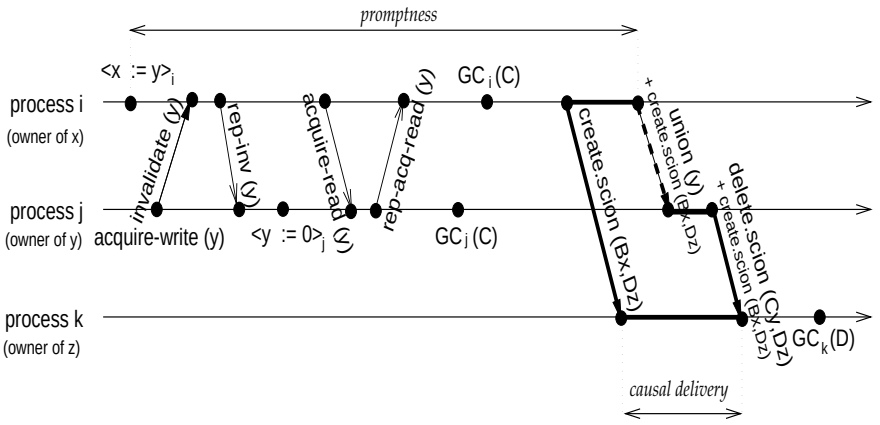


Figure 5.9: *Figure 5.8 extended with the union message. Event* <send.create.scion(Bx, Dz)>$_i$ *can be delayed at most until* <send.union(y, i⇝j)>$_i$ *is sent. Note that the union message carries the causal dependency (shown in bold lines) between the create and delete messages through the use of piggy-backing.*

According to the Cross-Bunch Collection Algorithm described in Section 4.10.2.2, before sending the union message, the cross-bunch collector in process i issues <send.create.scion(Bx, Dz)>$_i$. This ensures condition *(i)* (presented at the beginning of this section).

Note that there is still the need for causal delivery. As we do not have a communication protocol that ensures causality, we implement it using the technique of piggy-backing: every <send.create.scion> not yet acknowledged is piggy-backed on both the union and on the <send.delete.scion> messages, as illustrated in Figure 5.9. Another possibility would be to delay the sending of the union message until every <send.create.scion> has been acknowledged. However, this may delay indefinitely the union message therefore compromising the liveness of the GC algorithm.

5.6 GC and Coherence Granularity

In this section we make some considerations concerning how the GC algorithms are affected by the size of the granule of coherence. Recall Section 3.3.1: by definition, an object is the granule of coherence.

Basically, the GC algorithms are exactly the same independently of the number of application objects inside a coherence granule. An important aspect is the following. When a coherence granule encloses several application objects, any GC operation done on the granule must be applied to every application object

inside the granule. For example, when a granule is propagated, by the GC-Clean Propagation Rule (recall Section 4.10.2.2) the granule contents must be GC-clean; this implies that every application object enclosed in the granule must be GC-clean.

Another important aspect concerns the detection of GC-dirty objects. If an acquire-write is performed on a granule that contains several application objects, then every application object in the granule becomes GC-dirty.

If a coherence granule contains every object in the bunch, the copy intra-bunch algorithm can be optimized: since at any moment every application object is owned by one process, all move messages flow from the owner. Thus, the collector running in the owner process never receives the addresses of objects in to-space.

If the granule of coherence includes every object in a bunch, the union rule is easier to implement because every union message sent by a process goes to the same target: the owner of every object in the bunch.

Finally, note that with a large coherence granule, such as an entire bunch, serious performance problems may arise due to false sharing.

5.7 Programming Restrictions

In this section we describe the C++ programming restrictions due to the current implementation of GC in Larchant. These are related to the need of knowing where pointers are stored: in the stacks (which is part of the GC-root) and inside objects.

We first consider the mark-and-sweep algorithm. The collector adopts a conservative approach (recall Section 2.2.4) concerning pointers in the stack. In other words, any properly aligned bit pattern that could be the address of an object is actually considered to be a pointer to that object. Such objects are considered reachable, therefore marked and scanned.

Concerning pointers inside objects there are two possible approaches. The first is conservative, *i.e.*, just like with pointers in the stack, the collector treats anything that might be a pointer as a pointer. This approach requires no programming language restriction.

The second approach requires the exact knowledge of pointers location inside objects. For this purpose, the programmer must insert a macro in each class in order to inform Larchant about pointers location. In the future we intend to eliminate this restriction by integrating in Larchant a tool that automatically analyses object files (*i.e.*, **.o files*) and discovers where pointers are [62].

Concerning pointers in the stack, the copy collector adopts a conservative approach. Thus, a bit pattern in the stack whose value corresponds to a valid object address is considered as a pointer and will be patched. This is the cause of a strong C++ programming restriction: the application programmer must ensure that in the stack there are no non-pointer bit patterns whose value cor-

respond to objects addresses. Otherwise, the value will be corrupted when patched by the collector.

At first glance, it seems that the above restriction could be avoided simply by not moving to to-space objects that are directly pointed from the stack. However, this is would be very costly in terms of communication and synchronization, for the following reason. Remember that an object's new address in to-space is chosen by its owner (recall Section 4.4.2.3). Not moving an object is equivalent, in terms of the GC algorithm, to implicitly decide what is going to be its new address. Thus, there would be a conflict between an object's owner that decides to move an object, and some other process with a replica of the same object that decides not to move it, because locally it is directly reachable from the stack. In other words, the former decides that the object is moved to to-space while the latter decides that its address does not change. Therefore, deciding if an object should be moved or not, which depends if it is pointed from the stack in some process, would incur into a high communication and synchronization cost because every process caching a replica of the concerned object would have to agree on the same decision.

Note that if there is run-time type information enabling the differentiation of pointers from other types of variables, such as in Smalltalk [67], the above C++ programming restriction no longer holds. Also note that if the copy collector is used on bunches that are not being concurrently accessed by mutators, there is no programming restriction because there is no stack holding pointers to objects in the bunch being collected. This is the case when the collection of a bunch is started by initiative of either the CM or the OBR (recall Section 5.1.7).

The copy collector needs to know where pointers are allocated inside objects. For this purpose the application programmer must use a macro just as with the mark-and-sweep collector (second approach).

Finally, both collectors allow the existence of pointers to the interior of objects.

5.8 Summary

In this chapter we gave a general overview of the Larchant architecture and described our prototype. This is as simple as possible, yet sufficiently complete to test the GC algorithms and to execute distributed applications with persistent objects.

We presented the state-machines for the intra-bunch garbage collectors we implemented: mark-and-sweep and copy algorithms. The programming of the collectors resulted from a simple mapping (almost automatic) of the state-machines to C++. The collectors can be executed either concurrently with mutators or in GC-only mode.

We also described how the promptness condition of the cross-bunch collector is ensured in presence of the entry-consistency protocol by using the union rule and the piggy-backing technique (for causality).

Finally, we made some considerations concerning application programming restrictions due to garbage collection. These restrictions are mostly imposed by the copy collector given its need for patching pointers.

Chapter 6

Performance of GC Algorithms

In this chapter we study the performance of the GC algorithms described in the previous chapters. This study has two goals. First, to prove the feasibility of the GC algorithms. Second, to prove that our goals of good performance (non-disruptiveness, in particular), scalability, and coherence orthogonality are in fact achieved.

The performance results we present in this chapter are aimed at validating the design decisions regarding the GC algorithms. In particular, we show that:

- GC is non-disruptive.
- GC is scalable.
- There is no coherence interference due to GC.
- An asynchronous cross-bunch collector has better performance than a synchronous one.

The most important performance aspect concerning the intra-bunch collector is the GC pause time, because this is the time interval during which a mutator is halted. The GC pause time must be as small as possible in order to avoid disruption of applications. Scalability and coherence orthogonality means that GC pause time remains acceptable whatever the bunch size and the number of sites caching a bunch being collected.

Concerning the cross-bunch collector, the most important performance aspect is related to its asynchrony w.r.t. applications. We measure the performance advantage of the asynchronous creation of stub-scion pairs, and the overhead of the GC-Clean Propagation Rule when a coherent replica is acquired (recall Section 4.10.2.2).

There is no standard (or widely accepted) benchmark for GC. This is an important aspect that makes GC performance study a difficult issue and prevents a fair comparison with other algorithms and systems [4]. In addition, there is

a lack of knowledge on the behavior of real distributed applications with persistent objects, *e.g.*, amount of garbage, rate of object creation, rate of garbage generation, etc. Such data, possibly gathered in the form of trace files, would be extremely valuable for GC performance evaluation, simulation, tuning, and optimization.

Given the lack of such traces and of a widely accepted GC benchmark, we decided to study the performance of our GC algorithms mainly with synthetic micro-benchmarks. These benchmarks have the non-negligible advantage of being easy to implement. Note that these synthetic micro-benchmarks are not assumed to model the behavior of real applications. They simply enable the study of the GC algorithms relative to important parameters such as bunch size and percentage of unreachable objects, for example. They are relevant in the sense that they allow us to understand how the collector works and to simulate worse-case situations (*e.g.*, maximum object creation rate).

In this chapter we start by presenting the environment in which the performance results were obtained, and the benchmarks that were used. Then, we show the GC pause time of the intra-bunch GC algorithms: first without replication relative to the percentage of unreachable objects and bunch sizes, then with replication relative to the distribution of owned objects and number of bunch replicas. We finish this chapter with some GC performance results obtained with two small macro-benchmarks. The first, called TX1 [52], simulates a cooperative text editor. The second, called OO7 [26], is a comprehensive test of object-oriented database performance.

6.1 Basics

We ran our benchmarks on a network of DEC Alpha workstations. Each has 64 Mb of main memory, 1 Gb of disk, an 8 Kb data cache, a 512 Kb secondary cache, and a 150 MHz clock. They are connected by FDDI.

The micro-benchmarks for the intra-bunch collector were performed with an application that keeps creating objects inside a single bunch, *i.e.*, fills up a bunch with objects, and inserts some of them in a list (with a periodic distribution); objects in the list are reachable, objects not in the list are garbage. Every object has the same size, 128 bytes.

In the rest of this chapter we present the following performance results concerning the intra-bunch collector. We show the GC pause time of both mark-and-sweep and copy intra-bunch collectors in both GC-only and concurrent modes. First we consider the collection of a single non-replicated bunch; we vary the bunch size and the percentage of unreachable objects. Then we consider the case where the bunch being collected is replicated on several sites and vary the bunch size, the percentage of unreachable objects, and the percentage of owned objects on each site.

Concerning the cross-bunch collector, we study the performance advantage of the asynchronous (versus synchronous) creation of stub-scion pairs. We show

how long it takes to create a local stub, a local scion, and a remote scion.

6.2 Mark-and-Sweep without Replication

In this section we study the performance of the mark-and-sweep algorithm when applied to a single non-replicated bunch. We study the GC pause time of the intra-bunch collector in two functioning modes: GC-only and concurrent. We show how GC pause time varies with the bunch size and the percentage of unreachable objects.

6.2.1 GC Pause Time in GC-only Mode

With the GC-only mode, the application is halted while the collector marks and scans all reachable objects, and sweeps all the bunch.

In Figures 6.1 and 6.2 we show the GC pause time of the intra-bunch collector relative to the percentage of unreachable objects, for several bunch sizes. These graphs confirm the classic results: *(i)* GC pause times are disruptive, *(ii)* GC pause time increases with bunch size, and *(iii)* GC pause time increases as the percentage of unreachable object decreases.

Disruption is due to the GC-only functioning mode of the intra-bunch collector.

Obviously, GC pause time increases with the bunch size because there are more objects to scan and mark, and more memory to sweep.

Smaller percentage of unreachable objects means that there are more reachable objects to mark and scan; thus, the GC pause time increases.

To summarize, the mark-and-sweep algorithm in GC-only mode is extremely disruptive, and does not scale to large bunch sizes and arbitrary number of reachable objects.

6.2.2 GC Pause Time in Concurrent Mode

With the concurrent functioning mode, GC adds no pauses on top of time-slicing. The application is halted only while the collector flips.[1]

In Figure 6.3 we show the GC pause time of the intra-bunch collector relative to the percentage of unreachable objects for several bunch sizes. Note that with the concurrent functioning mode, GC pauses are caused only by flipping and hence are very short.

From Figure 6.3 it is clear that the mark-and-sweep intra-bunch collector is much less disruptive when running in concurrent mode than in GC-only mode: in fact, GC pause time is always under 60 milliseconds.

[1] Recall that, in the case of a mark-and-sweep collector, the flip is the instant when the mark phase is finished and the sweep can start.

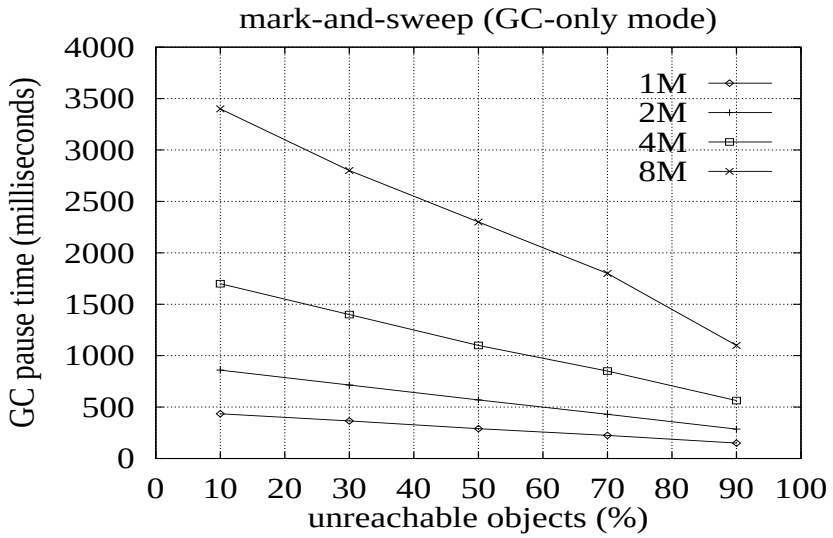


Figure 6.1: *GC pause time of the intra-bunch collector relative to the percentage of unreachable objects, for several bunch sizes.*

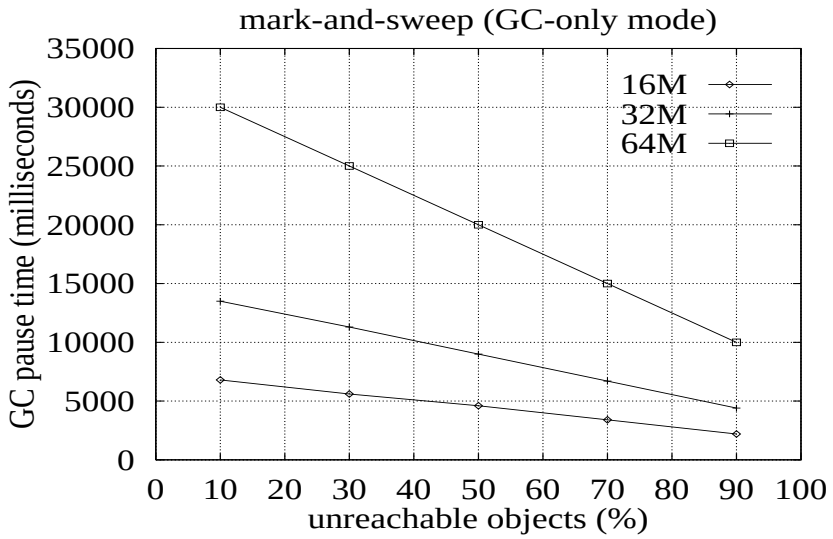


Figure 6.2: *GC pause time of the intra-bunch collector relative to the percentage of unreachable objects, for several bunch sizes.*

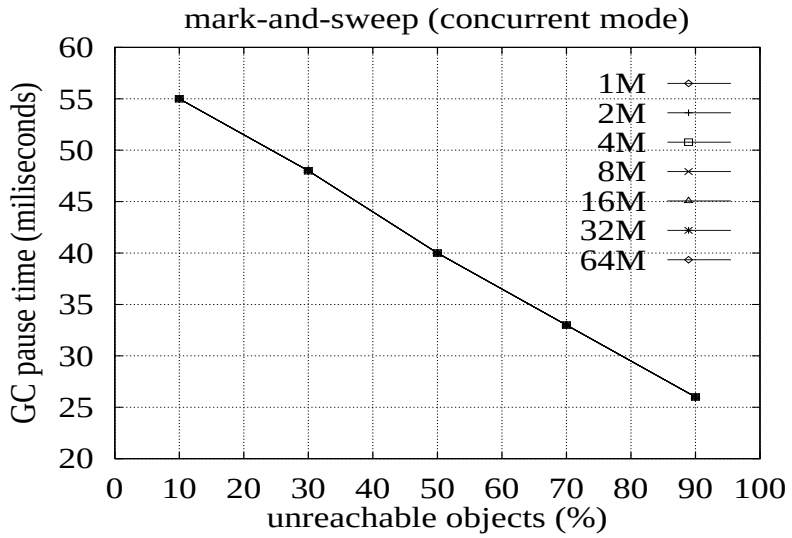


Figure 6.3: *GC pause time of the intra-bunch collector relative to the percentage of unreachable objects, for several bunch sizes.*

GC pause time is independent of bunch size. This is due to the fact that the majority of the GC work is done concurrently to the mutator.

GC pause time is not independent of the percentage of unreachable objects. This behavior is explained as follows. When flipping, the collector sweeps a certain amount of memory until it reclaims 1024 objects (a configurable parameter). This threshold ensures that once the mutator is resumed, object allocation will not catch up the GC sweeping, *i.e.*, the mutator does not yield the processor to the collector to allow it to reclaim unreachable objects, and hence obtain free space for new objects. Thus, for smaller percentages of unreachable objects, the memory that must be swept is larger; consequently, the GC pause time increases with decreasing percentage of unreachable objects.

Note that GC pause time can be made smaller if the number of objects to be reclaimed while flipping decreases (*i.e.*, less than 1024); however, in this case, if the mutator allocates objects at the maximum rate (as is the case of our benchmark) it will have to yield its processor to the collector because the sweeping phase is not fast enough. Another possibility, instead of the threshold mentioned above, would be to sweep the memory during the flipping for a maximum pre-defined time interval (sufficiently small to be non-disruptive). This solution would ensure that GC pause times would always be non-disruptive. The price to pay would be (possibly) the mutator yielding the processor to the collector, to allow the latter to sweep.

To summarize, the mark-and-sweep algorithm in concurrent functioning mode is not disruptive, and it scales well with bunch size.

6.3 Copy without Replication

In this section we study the performance of the copy algorithm when applied to a single non-replicated bunch. We study the GC pause time of the intra-bunch collector in two functioning modes: GC-only and concurrent. We show how the GC pause time varies with bunch size and with the percentage of unreachable objects.

6.3.1 GC Pause Time in GC-only Mode

With the GC-only mode, the application is halted while the collector moves, scans, and patches all reachable objects.

In Figures 6.4 and 6.5 we show the GC pause time of the intra-bunch collector relative to the percentage of unreachable objects, for several bunch sizes. These graphs confirm the classic results: *(i)* GC pause times are disruptive, *(ii)* GC pause time increases with bunch size, and *(iii)* GC pause time increases as the percentage of unreachable objects decreases.

Disruption is due to the GC-only functioning mode of the intra-bunch collector.

Obviously, GC pause time increases with bunch size because there are more reachable objects to move and scan.

A smaller percentage of unreachable objects means that there are more reachable objects to move and scan; hence, GC pause time increases.

The copy collector provides better (*i.e.*, smaller) GC pause times than the mark-and-sweep collector (see Figures 6.1 and 6.2). This difference becomes more significant as bunch size and percentage of unreachable objects increase. This is due to the fact that the copy collector only "sees" the reachable objects, while the mark-and-sweep algorithm "sees" them all.

To summarize, although its performance is better than mark-and-sweep, the copy algorithm in GC-only mode remains disruptive, and it does not scale well to large bunch sizes and arbitrary number of reachable objects.

6.3.2 GC Pause Time in Concurrent Mode

In Figure 6.6 we show the GC pause time of the intra-bunch collector relative to the percentage of unreachable objects, for several bunch sizes. Note that in the concurrent mode, the GC pause time corresponds to the flip time which is the only time interval during which the mutator is halted.

From Figure 6.6 it is clear that the copy intra-bunch collector is not disruptive when running in concurrent mode.

GC pause time is independent of bunch size and of percentage of unreachable objects. This is due to the fact that the majority of the GC work is done concurrently to the mutator. When flipping, the collector simply has to re-scan and move again those objects that were modified by the mutator after having been scanned and moved (recall Section 4.4).

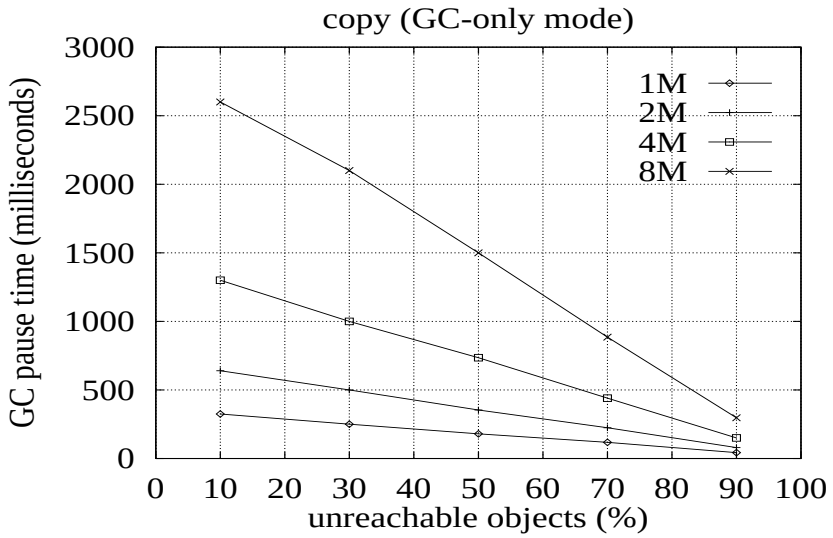


Figure 6.4: *GC pause time of the intra-bunch collector relative to the percentage of unreachable objects, for several bunch sizes.*

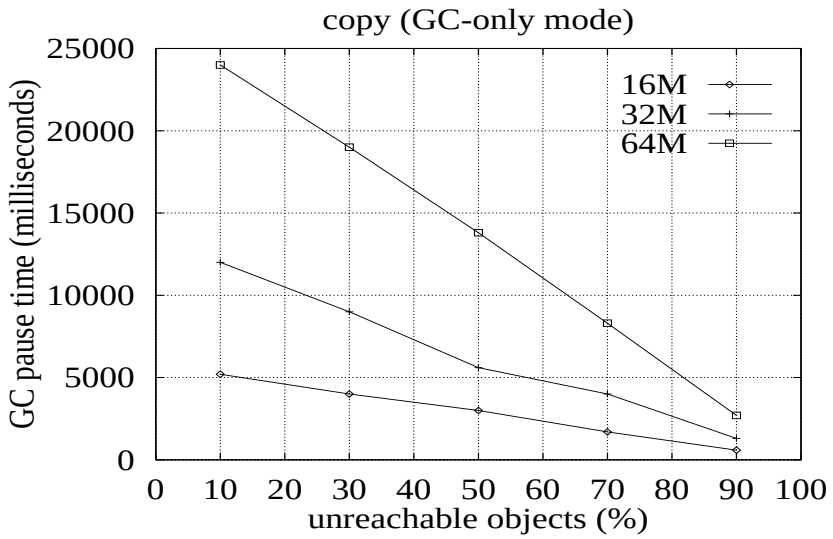


Figure 6.5: *GC pause time of the intra-bunch collector relative to the percentage of unreachable objects, for several bunch sizes.*

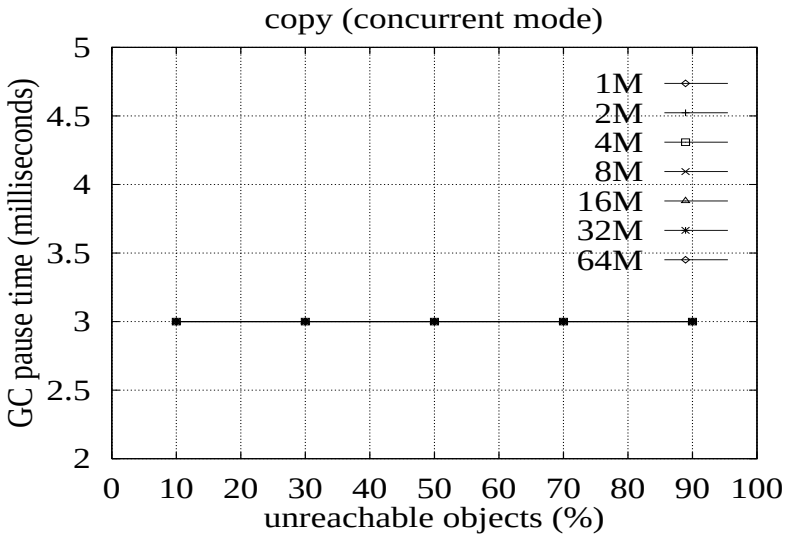


Figure 6.6: *GC pause time of the intra-bunch collector relative to the percentage of unreachable objects, for several bunch sizes.*

To summarize, the copy algorithm in concurrent mode is not disruptive, and it scales well to large bunch sizes and arbitrary number of reachable objects. The GC pause time is smaller with the copy collector than with the mark-and-sweep.

6.4 GC Pause Time with Replication

In this section we study the performance of both mark-and-sweep and copy algorithms when applied to a single replicated bunch. We study the GC pause time of the intra-bunch collector in two functioning modes: GC-only and concurrent. We show how the GC pause time varies with the bunch size, number of bunch replicas, distribution of objects ownership among the sites, and percentage of unreachable objects.

We ran the benchmark used in the non-replicated case (recall Section 6.1) on several sites all sharing the same bunch. We varied the number of sites from 2 to 6 (the maximum available in our laboratory) and ran the intra-bunch collector in both GC-only and concurrent modes.

6.4.1 Mark-and-Sweep in GC-only and Concurrent Modes

The values of GC pause time that were obtained for each site, in both functioning modes, are the same as in the non-replicated case (see Figures 6.1 and 6.2 for GC-only mode, and Figure 6.3 for concurrent mode).

These results confirm our expectations that our GC algorithm is orthogonal to coherence. In fact, given that the collector does not need coherent objects, there is no communication between sites on behalf of GC. Each site collects independently from any other site.

time to create a local stub	0.022 milliseconds
time to create a local scion	0.022 milliseconds
time to create a remote scion	1.2 milliseconds

Table 6.1: *Time it takes to create (synchronously) stubs and scions.*

6.4.2 Copy in GC-only and Concurrent Modes

The values of GC pause time that were obtained for each site, in both functioning modes, are the same as in the non-replicated case (see Figures 6.4 and 6.5 for GC-only mode, and Figure 6.6 for concurrent mode).

These results confirm our expectations that our GC algorithm is orthogonal to coherence. In fact, given that the collector does not need coherent objects, the only messages between sites on behalf of GC are exchanged in the background (these correspond to move operations; recall Section 4.4.2.3).

6.5 Cross-Bunch GC

In this section we study the most important performance aspect of the cross-bunch collector, *i.e.*, we show the time it takes to create a stub, and to create a scion. We quantify the advantage of asynchronous, versus synchronous, execution of the cross-bunch collector. We also measure the overhead of the GC-Clean Propagation Rule when a coherent replica is acquired.

If the cross-bunch collector runs synchronously, every assignment operation is instrumented in order to detect if a new cross-bunch pointer has appeared. In that case, the corresponding local stub and (possibly remote) scion are synchronously created. Creating a remote scion implies sending a message to the remote site, and to wait an acknowledgement. During all this time, the mutator is halted. In Table 6.1, we show the time it takes to create synchronously: a stub, a local scion, and a remote scion.

The advantage of making the cross-bunch collector asynchronous to mutators, comes from the substantial performance gains since the mutator is not halted, it might avoid work, and enables message batching. To quantify this advantage we ran the following benchmark: with an object containing 16 pointers, we performed an assignment operation for each one in order to create 16 cross-bunch pointers (all pointing to the same target bunch). We measured the time the assignment operations take and the corresponding creation of stub-scion pairs, in both synchronous and asynchronous modes. We show these values in Table 6.2.

In the synchronous case, the cross-bunch pointer is detected when the corresponding assignment operation is executed, and the stub-scion pair is immediately created. With the asynchronous mode, assignment operations are not instrumented; the object is scanned while the intra-bunch collector executes,

synchronous	24 milliseconds
asynchronous	1.6 milliseconds

Table 6.2: *Time it takes to create 16 cross-bunch pointers and the corresponding stub-scion pairs.*

the stubs are created locally, and the creation of remote scions is batched in a single message. (There is a single message to acknowledge the creation of all scions.) This batching is the reason for the difference of performance.

From these values, the advantage of running the cross-bunch collector asynchronously to applications is clear. If the creation of a stub-scion pair was done synchronously with the appearance of the corresponding cross-bunch pointer (due to an assignment operation) the performance of the mutator would be penalized accordingly.

The GC-Clean Propagation Rule described in Section 4.10.2.2 says that an object y can be propagated from its owner process i only if y_i is GC-clean, and the messages to create the scions corresponding to y_i's outgoing cross-bunch pointers have been sent. Thus, this rule introduces an overhead to the acquisition of a coherent replica.

We measured this overhead by acquiring a coherent replica of an object (with 16 outgoing cross-bunch pointers inside) with and without performing the GC-Clean Propagation Rule. The results that we obtained are the following: acquiring a coherent replica of the object without executing the above rule takes 1,25 milliseconds; when executing the above rule, the acquisition of a coherent replica takes 1,65 milliseconds. Thus, there is an overhead of 400 microseconds (*i.e*, 32%) due to the GC-Clean Propagation Rule: these are spent on scanning the object and creating the corresponding stubs (350 microseconds) and sending a message to create the corresponding scions (it's a single message as all the cross-bunch pointers point to the same target bunch: 30 microseconds for building the message buffer and 20 microseconds to send it).

6.6 Macro-Benchmarks

In addition to the synthetic micro-benchmarks, we ran two macro-benchmarks. The first one, called TX1, simulates a cooperative text editor. A TX1 document is a hierarchical structure containing objects for sections, subsections, paragraphs, and lines. The simulator creates objects, assigns pointers, reads and writes lines of text, all randomly.

We measured the GC pause times of the intra-bunch mark-and-sweep collector with TX1 running concurrently on one, two, and three sites, with a 4 Mbytes bunch and various amounts of garbage. GC pause times are never greater than 40 milliseconds. The same benchmark for the intra-bunch copy collector, presents GC pause times which are never greater than 5 milliseconds.

In addition, the total GC overhead for both collectors, lays between 10% and 20%, with a mean value of 14%. This value was obtained by running TX1 first with GC and then without it, and comparing the total execution time. The values obtained with several replicas do not differ significantly from the non-replicated case.

We have also ported the well-known OO7 benchmark, widely used by the object-oriented database community. Our main goal was to test the reliability of the Larchant prototype. The standard OO7 benchmark runs on a single site.

The mean GC pause time with the concurrent mark-and-sweep intra-bunch collector is 1 second. This value is bigger than the GC pause times obtained with the synthetic benchmark for the following reason: with OO7, 99% of objects are reachable, thus while flipping all the memory must be swept. If the number of objects to be reclaimed while flipping is made smaller (currently, the value is 1024) than the GC pause times become similar to those obtained with the synthetic benchmark.

For the OO7, the GC pause times with the concurrent copy intra-bunch collector are all less than 5 milliseconds.

6.7 Summary

The performance results presented in this chapter show that our implementation of the GC algorithms is non-disruptive, is orthogonal to coherence, and is scalable.

The concurrent functioning mode ensures that applications are not disrupted by long pauses, which would be annoying to the user. This is due to the fact that the majority of the GC work is done concurrently to the mutator; thus, GC pause time is independent of the bunch size, and of the percentage of unreachable objects. This aspect is clearly observed by comparing the GC pause times of the intra-bunch collectors, for both mark-and-sweep and copy algorithms, in GC-only and concurrent modes.

We also showed that GC pause times are independent of the number of replicas. This is due to the fact that the collector does not need coherent data to make progress.

The copy collector in concurrent mode scales to large bunch sizes. Both the copy and mark-and-sweep collectors scale to large number of replicas. This scalability comes from the fact that a bunch is collected independently from the rest of the memory, and from coherence orthogonality.

In general, the copy collector provides smaller GC pause times than the mark-and-sweep collector. However, the copy algorithm requires precise knowledge of pointer locations, which might be impracticable with some programming languages.

We did not show any performance results concerning group collection because we have strong reasons indicating that they are similar to the collection of a bunch whose size is equal to the sum of the sizes of the bunches in the group.

We showed how much we gain by executing the cross-bunch collector asynchronously to the mutator, instead of doing it synchronously. The mutator freely assigns pointers without being slowed down by the creation of stub-scion pairs.

We tested and measured the intra-bunch collector with two macro-benchmarks: a simulator of a cooperative text editor (TX1) and a standard comprehensive test of object-oriented database performance (OO7). For both benchmarks, GC pause time is non-disruptive. However, OO7 poses some problems to the mark-and-sweep algorithm because almost every object is reachable. Thus, when flipping, the intra-bunch collector cannot reclaim the 1024 objects required.

It is important to stress the lack of widely accepted benchmarks for measuring GC performance. In particular, our version of the OO7 benchmark is not indicated for measuring GC because every object used in the benchmark is created when the application starts running, and there are almost no unreachable objects. In other words, the pointer graph remains mostly unchanged. We believe that this behavior is typical of databases which are optimized towards searching for relevant data (through indexing, for example) with modest updates and minimal cooperative read/write interaction. This is not the kind of applications we are interested in (recall Sections 3.1 and 3.2). However, note that even for such database systems, our fundamental GC design aspects are still valid: independent collection of bunches, and orthogonality to coherence.

Finally, note that the source code of the current prototype is far from being optimized. We can achieve better results with some more effort. For example, we could scan the stack concurrently to the mutator instead of doing it only while flipping.

Chapter 7

Conclusion

The basic motivation for the work described in this document is to make data sharing simple and efficient (recall Chapter 1). This is an important issue because an essential part of tomorrow's computing will be workers and organizations carrying out cooperative tasks interacting via shared data.

Larchant is a Cached Distributed Shared Store based on the model of a Shared Address Space with Persistence By Reachability. To provide the illusion of a shared address space across the network, although site memories are disjoint, Larchant implements a distributed shared memory mechanism. Data is replicated in multiple sites for performance and availability. Reachability is accessed by tracing the pointer graph, starting from the persistent root, and reclaiming unreachable data. This is the task of Garbage Gollection (GC).

GC automatically ensures referential integrity, therefore improving program reliability and programming productivity. In addition, it may compact memory thus reducing fragmentation and improving locality.

If in a centralized short-lived program GC might be considered as an expensive luxury, in contrast, in a cached distributed shared store such as Larchant, the nightmare of manual memory-management increases as the number of objects, pointers and users scales up. Then GC becomes indispensable.

In the rest of this chapter we summarize the most important issues concerning distributed GC in a persistent shared store, for which we proposed innovative solutions. We analyze the achievements in the light of the GC algorithm characteristics and performance results. We conclude this chapter with some perspectives for future research.

7.1 GC Issues and Solutions

We addressed the following fundamental issues faced by GC in a large-scale distributed shared store: correctness, completeness, performance, scalability, and interference with applications' coherence needs. A common thread to our solutions, described in detail in Chapter 4, is that GC is opportunistic, *i.e.*, it

does not cause events that mutators would eventually cause; the collector waits for them to happen and then takes advantage of them.

Our GC algorithm combines partitioned tracing (within one or more bunches) with reference-listing (across bunch group boundaries). Each bunch replica is collected independently of other bunches and of other replicas of the same bunch. A bunch replica is collected in the process where it is currently cached, with a tracing algorithm: either mark-and-sweep or copy.

Concerning performance, applications overhead due to GC must be minimized. In particular, interactive applications should not be disrupted by the collector. Our solution is to perform GC concurrently and asynchronously w.r.t. mutators. The intra-bunch collector runs concurrently to mutators and performs most of its work without adding pauses on top of time-slicing. The mutator is halted only when the collector flips; thus, GC pauses are non-disruptive. The cross-bunch collector runs asynchronously w.r.t. mutators, by safely delaying the creation of stub-scion pairs describing cross-bunch pointers. The advantage of making the cross-bunch collector asynchronous to mutators, comes from the substantial performance gains since the mutator is not halted, it might avoid work, and enables message batching.

Scalability means that the GC algorithm must be capable of collecting a huge pointer graph without incurring into high performance costs due to computation, I/O, and distributed synchronization. In particular, the collection of the whole graph in a single phase would clearly not be scalable. In addition, we claimed that perfect completeness is not feasible in a large-scale system. Thus, we proposed an approximate solution that is not provably complete, but which we believe adequate for most real-life situations because it takes advantage of locality.

As already mentioned, each bunch is collected independently from the rest of the memory. This solution is not complete, since it does not reclaim cross-bunch cycles of garbage. Apparently, the only way of reclaiming such cycles is to perform a global trace, which is not scalable, and would generate a huge amount of extra I/O. Our solution to reclaim cross-bunch cycles of garbage, without incurring into extra I/O, consists of extending the tracing performed by the intra-bunch collector to include several bunches at once. Bunches are grouped and collected as if they were a single bunch. Thus, groups of bunches change dynamically and seamlessly, and independently in each process, in order to reclaim cycles of unreachable objects.

Groups are formed according to a locality-based heuristic: a group contains all the bunches currently cached in the process. This heuristic avoids extra I/O costs. However, it does not enable the reclamation of cross-bunch cycles enclosed in bunches that reside partially on disk. Also, cycles of garbage where some bunches, but not all, are replicated in multiple processes, are not reclaimed. This garbage might be reclaimed with a more aggressive grouping heuristic. However, the extra I/O cost involved needs to be balanced against the expected gain.

To prevent coherence interference, the GC algorithm must not compete with

applications for holding coherent data replicas. Such competition would interfere with applications' coherence needs. For example, if the collector on some site were to require coherent access to some object, that would prevent an application running at another site from writing into another replica (of that object) at the same time. To avoid the coherence interference problem, the collection of a bunch replica makes progress even if the objects it contains are not known to be coherent in the process where the intra-bunch collection is taking place. In other words, our GC algorithm is orthogonal to coherence. Thus, the garbage collector can work with objects that are incoherent for applications' purposes, while remaining safe and live. This minimizes the assumptions underlying GC and ensures wide applicability of the algorithms.

We designed our GC algorithm for the Larchant model (presented in Chapter 3). This model is that of a cached distributed shared store with persistence by reachability. The model is general and we made minimal assumptions. Therefore, the GC algorithm is widely applicable.

In Chapter 4, we described the GC algorithm, first intuitively, then rigorously, and proved formally its safety and liveness. A fundamental aspect of safety is the union rule: a target object can be reclaimed only if it is not reachable from the union of all replicas of its source objects (independently of their location, whether in the same or in a different bunch). We showed how the creation of scions can be safely delayed by respecting the promptness condition: deliver a message to create a scion before the corresponding object could otherwise be reclaimed. For safety, causal delivery is needed.

We implemented a prototype of Larchant and showed how the intra-bunch collector is specified with a state-machine formalism (both mark-and-sweep and copy algorithms). Then, the programming of the collectors resulted from a simple (almost automatic) mapping of the state-machines to C++.

We measured the performance of the most important aspects of the GC algorithms: mark-and-sweep and copy intra-bunch collectors (in both GC-only and concurrent functioning modes) and of the cross-bunch collector. The results confirm our expectations in terms of performance, scalability, and coherence orthogonality.

7.2 Achievements

Our goals for the GC algorithm in Larchant were (recall Chapter 1): correctness, scalability, orthogonality to coherence, and good performance. The GC algorithm fulfills these goals. It is correct, we proved its safety and liveness formally. It is scalable, orthogonal to coherence, and has good performance; from the performance results obtained with our prototype (recall Chapter 6) we clearly see that GC pause times are independent from: the size of a bunch, the percentage of reachable objects, the number of replicas, and ownership distribution.

7.3 Future Work

An obvious task for future work is to improve the current prototype. This implies an engineering effort in order to increase the robustness and the usability of Larchant. The development of other bunch managers, possibly with different coherence protocols and GC algorithms, is another important aspect.

In the future we intend to study how the GC algorithms interact with a checkpoint mechanism that will be integrated in Larchant. This research is part of the more general issue of fault-tolerance. The GC algorithm must be capable of dealing with site crashes and unreliable communication, *i.e.*, GC must remain safe and live under adverse conditions. In particular, we intend to make the GC algorithm safe even if the communication medium does not ensure causal communication, by using the SSP Chain mechanism [108]. This issue of fault-tolerance is fundamental for large geographically sparse networks or for mobile computing.

Another direction for future research is related to the benchmarking of GC algorithms. As stated in Chapter 6, there is no standard (or widely accepted) benchmark for GC. In addition, there is insufficient knowledge concerning the behavior of real distributed applications with persistent objects (*e.g.*, amount of garbage, rate of object creation, rate of garbage generation, etc). These two aspects prevent the fair comparison of GC algorithms and makes difficult their optimization and tuning. Thus, a very interesting direction for future research consists of tracing real distributed applications in order to model their memory behavior. Once such data is available, different GC algorithms can be designed accordingly, compared fairly, and optimized.

The garbage collection of distributed (cross-bunch) cycles of garbage would benefit enormously from the study of applications' memory behavior mentioned above. In fact, it would be extremely helpful to characterize such cycles in real applications: size, geographical distribution, creation rate, typical mutator operations that generate them, etc. The difficulty with such tracing is that real distribution applications with persistent objects are not easily available.

The completeness of our GC algorithms is a very important issue for future research. We believe that the locality-based heuristic for grouping bunches is well suited for the majority of applications. However, a practical study is needed. Then, if experimental results mandate it, more complex heuristics will be developed and their cost carefully evaluated either through simulation or integration within the current prototype.

Finally, note that we proved the correctness of our GC algorithm, but we did not prove formally the correctness of its implementation. This is also an interesting aspect for future work: from the specification of the GC algorithm (proved to be correct) generate automatically the corresponding source code.

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for unix development. In *Proceedings of Summer Usenix*, July 1986.

[2] Guy Almes, Andrew Black, Edward Lazowska, and Jerry Noe. The Eden system: a technical review. *IEEE Transactions on Software Engineering*, SE-11(1), January 1985.

[3] L. Amsaleg, M. Franklin, and O. Gruber. Efficient Incremental Garbage Collection for Client-Server Object Database Systems. In *Proc. of the 21th VLDB Int. Conf.*, Zürich, Switzerland, September 1995.

[4] Laurent Amsaleg, Paulo Ferreira, Michael Franklin, and Marc Shapiro. Evaluating garbage collectors for large persistent stores. In *Conf. on Object-Oriented Programming Systems, Languages, and Applications*, Austin TX (USA), October 1995.

[5] Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2:153–162, 1989.

[6] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, February 1989.

[7] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent garbage collection on stock multiprocessors. In *Proceedings of the 1988 SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–20, Atlanta, Georgia, June 1988. ACM Press.

[8] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, 1983.

[9] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[10] Stoney Ballard and Stephen Shirron. The design and implementation of vax/smalltalk-80. In *Smalltalk-80: Bits of History, Words of Advice*, pages 127–150. Addison-Wesley, MA, 1983.

[11] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Equipment Corporation Western Research Laboratory, Palo Alto, California, February 1988.

[12] John Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proc. of the 17th Annual International Symp. on Comp. Architecture*, pages 125–135, Seattle, WA (USA), May 1990.

[13] Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman, editors. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Massachusetts, 1987.

[14] Brian N. Bershad and Matthew J. Zekauskas. The Midway distributed shared memory system. In *Proceedings of the COMPCON'93 Conference*, pages 528–537, February 1993.

[15] D. I. Bevan. Distributed garbage collection using reference counting. In *Parallel Arch. and Lang. Europe*, pages 117–187, Eindhoven, The Netherlands, June 1987. Spring-Verlag Lecture Notes in Computer Science 259.

[16] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[17] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Programming Languages and Systems*, 2(1), February 1984.

[18] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 217–230, Asheville, NC (USA), December 1993.

[19] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection.* PhD thesis, Massachusetts Institute of Technology Laboratory for Computer Science, May 1977. Technical report MIT/LCS/TR-178.

[20] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation* [98], pages 157–164.

[21] Hans-Juergen Boehm. Space-efficient conservative garbage collection. In *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–206, Albuquerque, New Mexico, June 1993. ACM Press.

[22] Hans-Juergen Boehm and David Chase. A proposal for garbage-collector-safe compilation. *The Journal of C Language Translation*, 4(2):126–141, December 1991.

[23] Rodney A. Brooks. Trading data space for reduced time and code space in real-time collection on stock hardware. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming* [79], pages 108–113.

[24] P. Butterwoth, A. Otis, and J. Stein. The GemStone object database management system. *Com. of the ACM*, 34(10):64–77, October 1991.

[25] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalso, and Greg Nelson. Modula-3 report (revised). Research Report 52, Digital Equipment Corporation Systems Research Center, November 1989.

[26] Michael Carey, D. J. DeWitt, and J. F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *Proceedings of the Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94)*, pages 414–426, Portland OR (USA), October 1994.

[27] Michael J. Carey and David DeWitt. The architecture of the EXODUS extensible DBMS. In *Proc. Int. Workshop on Object-Oriented Database Systems*, pages 52–65, Pacific Grove, CA (USA), September 1986. IEEE.

[28] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, volume 25 of *Operating Systems Review*, pages 152–164, Pacific Grove CA (USA), October 1991.

[29] J. S. Chase, H. E. Levy, M. J. Feely, and E. D. Lazowska. Sharing and adressing in a single address space system. *ACM Transactions on Computer Systems*, 12(3), November 1994.

[30] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[31] Thomas W. Christopher. Reference count garbage collection. *Software Practice and Experience*, 14(6):503–507, June 1984.

[32] Jacques Cohen. Garbage collection of linked data structures. *Computing Surveys*, 13(3):341–367, September 1981.

[33] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 2(12):655–657, December 1960.

[34] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, David Hulse, Anders Lindstöm, Stephen Norris, John Rosenberg, and Francis Vaughan. Protection in Grasshopper: A persistent operating system. In *Proc. of the 6th Int. Workshop on Persistent Object Systems*, pages 54–72, Tarascon (France), September 1994.

[35] Alan Demers, Mark Weiser, Barry Hayes, Daniel 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*, pages 261–269, San Francisco, California, January 1990. ACM Press.

[36] David Detlefs. Garbage collection and run-time typing as a C++ library. In *C++ Conference*, pages 37–56, Portland OR (USA), August 1992. Usenix.

[37] David L. Detlefs. *Concurrent, Atomic Garbage Collection.* PhD thesis, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1991. Technical report CMU-CS-90-177.

[38] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[39] O. Deux et al. The 0_2 system. *Communications of the ACM*, 34(10):34–48, October 1991.

[40] E. W. Dijkstra, L. 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):966–975, November 1978.

[41] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically-typed language. In *Proceedings of the 1992 SIGPLAN Conference on Programming Language Design and Implementation*, pages 273–282, San Francisco, California, June 1992. ACM Press.

[42] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Proc. of the 21th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Lang.*, pages 70–83, Portland, OR (USA), January 1994.

[43] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proc. of the 20th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Lang.*, pages 113–123, Charleston SC (USA), January 1993.

[44] Daniel R. Edelson. Precompiling C++ for garbage collection. In *Proc. 1992 International Workshop on Memory Management*, pages 299–314, Saint-Malo (France), September 1992. Springer-Verlag.

[45] Daniel R. Edelson. Smart pointers: They're smart, but they're not pointers. In *C++ Conference*, pages 1–19, Portland, OR (USA), August 1992. Usenix.

[46] John R. Ellis and David L. Detlefs. Safe, efficient garbage collection for C++. Technical Report 102, Digital Equipment Corporation Systems Research Center, 1993.

[47] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[48] Vaughan F. and A. Dearle. Supporting large persistent stores using conventional hardware. In *Proc. of the Fifth International Workshop on Persistent Object Systems Design, Implementation and Use*, pages 29–50, San Miniato Pisa (Italy), September 1992.

[49] Michael J. Feeley, Jeffrey S. Chase, Vivek R. Narasayya, and Henry M. Levy. Integrating coherency and recovery in distributed systems. In *Proc. Symp. on Operating Systems Design and Implementation*, pages 215–228, Monterey CA (USA), November 1994.

[50] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[51] Paulo Ferreira. Reclaiming storage in an object oriented platform supporting extended C++ and Objective-C applications. In *Proc. of the International Workshop on Object Orientation in Operating Systems*, Palo Alto (USA), October 1991.

[52] Paulo Ferreira. TX1 benchmark for GC performance evaluation. Personal communication, October 1995.

[53] Paulo Ferreira and Marc Shapiro. Distribution and persistence in multiple and heterogeneous address spaces. In *Proc. Int. Workshop on Object-Orientation in Operating Systems*, Asheville NC (USA), December 1993.

[54] Paulo Ferreira and Marc Shapiro. Garbage collection and DSM consistency. In *Proc. of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 229–241, Monterey CA (USA), November 1994. ACM.

[55] Paulo Ferreira and Marc Shapiro. Larchant: Persistence by reachability in distributed shared memory through garbage collection. In *International Conference on Distributed Computing Systems (ICDCS'96)*, Hong Kong, May 1996. IEEE.

[56] Ross S. Finlayson and David R. Cheriton. Log files: An extended file service exploiting write-once storage. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 139–148, Austin TX (USA), November 1987. ACM.

[57] B. Fleisch and G. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 211–223, Litchfield Park AZ (USA), December 1989. ACM.

[58] Benjamin Goldberg. Generational reference counting: A reduced-communication distributed storage reclamation scheme. In *Programming Languages Design and Implementation*, number 24(7) in SIGPLAN Notices, pages 313–321, Portland, OR (USA), June 1989. SIGPLAN, ACM Press.

[59] Benjamin Goldberg. Tag-free garbage collection for strongly-typed programming languages. In *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation* [98], pages 165–176.

[60] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[61] Barry Hayes. Using key object opportunism to collect old objects. In Andreas Paepcke, editor, *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '91)*, pages 33–46, Phoenix, Arizona, October 1991. ACM Press.

[62] Yann Herve. OCI User Manual. Personal communication, October 1995.

[63] Lorenz Huelsbergen and James R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 73–82, San Diego, California, May 1993. ACM Press. Published as *SIGPLAN Notices 28*(7), July 1993.

[64] John Hughes. A distributed garbage collection algorithm. In Jean-Pierre Jouannaud, editor, *Functional Languages and Computer Architectures*, number 201 in Lecture Notes in Computer Science, pages 256–272, Nancy (France), September 1985. Springer-Verlag.

[65] Avadis Tevanian Jr. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Application-Approach*. PhD thesis, Dept. of Comp. Sc., Carnegie-Mellon University, Pittsburgh (USA), December 1987.

[66] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[67] T. Kaelher and G. Krasner. *LOOM-Large Object-Oriented memory for Smalltalk-80 Systems, Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, MA, 1983.

[68] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proc. Winter Usenix Conf.*, pages 115–131, San Francisco CA (USA), January 1994.

[69] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[70] Bett Koch, Tracy Schunke, Alan Dearle, Francis Vaughan, Chris Marlin, Ruth Fazakerley, and Chris Barter. Cache coherency and storage management in a persistent object system. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 99–109, Martha's Vineyard, MA (USA), September 1990.

[71] E. Kolodner and W. Weihl. Atomic incremental garbage collection and recovery for large stable heap. In *Proc. of the ACM SIGMOD Int. Conf*, pages 177–185, Washington D.C. (USA), June 1993.

[72] R. Kordale, M. Ahamad, and J. Shilling. Distributed/concurrent garbage collection in distributed shared memory systems. In *Proc. Third Int. Workshop on Object Orientation and Operating Systems*, Asheville NC (USA), November 1993.

[73] Rivka Ladin and Barbara Liskov. Garbage collection of a distributed heap. In *Int. Conf. on Distributed Computing Systems (ICDCS'92)*, pages 708–715, Yokohama (Japan), June 1992. IEEE.

[74] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[75] B. Lampson. Atomic transactions. In *Distributed Systems—Architecture and Implementation*, pages 246–265. Spinger-Verlag, New York, 1981. Lecture Notes in Computer Science vol. 105.

[76] Bernard Lang and Francis Dupont. Incremental incrementally compacting garbage collection. In *SIGPLAN 1987 Symposium on Interpreters and Interpretive Techniques*, pages 253–263, Saint Paul, Minnesota, June 1987. ACM Press. Published as *SIGPLAN Notices 22*(7), July 1987.

[77] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In *ACM Symposium on Principles of Programming Languages*, pages 39–50, Albuquerque, New Mexico, January 1992.

[78] T. Le Sergent and B. Berthomieu. Incremental multi-threaded garbage collection on virtually shared memory architectures. In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 179–199, Saint-Malo (France), September 1992. Springer-Verlag.

[79] *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, Austin, Texas, August 1984. ACM Press.

[80] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[81] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[82] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in Thor. In *Proc. Int. Workshop on Distributed Object Management*, pages 1–15, Edmonton (Canada), August 1992.

[83] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in Thor. In *Proc. Int. Workshop on Distributed Object Management*, pages 1–15, Edmonton (Canada), August 1992.

[84] Barbara Liskov, Robert Gruber, Paul Johnson, and Liuba Shrira. A highly available object repository for use in a heterogeneous distributed system. In *Proc. of the Fourth International Workshop on Persistent Object Systems Design, Implementation and Use*, Martha's Vineyard MA (USA), September 1990.

[85] U. Maheshwari and B. Liskov. Fault-tolerant distributed garbage collection in a client-server, object database. In *Proceedings of the Parallel and Distributed Information Systems*, pages 239–248, Austin, Texas (USA), September 1994.

[86] U. Maheshwari and B. Liskov. Collecting cyclic distributed garbage by controlled migration. In *Proceedings of the Principles of Distributed Computing*, Ottawa (Canada), September 1995.

[87] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, April 1960.

[88] Marvin Minsky. A LISP garbage collector algorithm using serial secondary storage. A.I. Memo 58, Massachusetts Institute of Technology Project MAC, Cambridge, Massachusetts, 1963.

[89] David Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming* [79], pages 235–246.

[90] J. Eliot B. Moss. Addressing large distributed collections of persistent objects: The Mneme project's approach. In *Proceedings of the Second International Workshop on Database Programming Languages*, Gleneden Beach, OR (USA), June 1989. Also available as COINS Technical Report 89–68, Object-Oriented Systems Laboratory, Dept. of Computer and Information Science, University of Massachusetts, Amherst.

[91] S. Nettles, J. O'Toole, D. Pierce, and N. Haines. Replication-based incremental copying collection. In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 357–364, Saint-Malo (France), September 1992. Springer-Verlag.

[92] S. C. North and J. H. Reppy. Concurrent garbage collection on stock hardware. In Gilles Kahn, editor, *ACM Conference on Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 113–133. Springer-Verlag, September 1987.

[93] E.I. Organick. *The Multics system: an examination of its structure*. MIT Press, Cambridge, Massachusetts, 1972.

[94] James O'Toole, Scott Nettles, and David Gifford. Concurrent compacting garbage collection of a persistent heap. In *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, Asheville, North Carolina (USA), December 1993. ACM Press. Published as *Operating Systems Review 27*(5).

[95] José M. Piquer. Indirect reference-counting, a distributed garbage collection algorithm. In *PARLE'91—Parallel Architectures and Languages Europe*, volume 505 of *Lecture Notes in Computer Science*, pages 150–165, Eindhoven (the Netherlands), June 1991. Springer-Verlag.

[96] David V. Pitts and Partha Dasgupta. Object memory and storage management in the Clouds kernel. In *Int. Conf. on Distributed Computing Systems (ICDCS'88)*, pages 10–17, San José, CA(USA), June 1988. IEEE.

[97] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, Lecture Notes in Computer Science, Kinross (Scotland), September 1995. Springer-Verlag.

[98] *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, Ontario, June 1991. ACM Press. Published as *SIGPLAN Notices 26*(6), June 1992.

[99] Christian Queinnec, Barbara Beaudoing, and Jean-Pierre Queille. Mark DURING Sweep rather than Mark THEN Sweep. In Eddy Odijik, M. Rem, and Jean-Claude Sayr, editors, *Parallel Architectures and Languages Europe*, number 365, 366 in Lecture Notes in Computer Science, Eindhoven, The Netherlands, June 1989. Springer-Verlag.

[100] Michel Raynal, André Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343–350, September 1991.

[101] Mendel Rosenblum and John K. Ousterhout. The LFS storage manager. In *Proc. Summer 1990 USENIX Conf.*, pages 315–324, Anaheim, CA (USA), June 1990. USENIX.

[102] Vincent Russo. Garbage collecting an object-oriented operating system kernel. Workshop on GC in Object-Oriented Systems at Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91) - available by anonymous FTP from cs.utexas.edu in /pub/garbage/GC91, October 1991.

[103] A. D. Samples. Garbage collection-cooperative C++. In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 315–329, Saint-Malo (France), September 1992. Springer-Verlag.

[104] M. Satyanarayanan. A survey of distributed file systems. Technical Report CMU-CS-89-116, Department of Computer Science, Carnegie-Mellon University, Pittsburgh PA (USA), February 1989.

[105] Michael L. Scott, Thomas J. LeBlanc, Brian D. Marsh, Timothy G. Becker, Dubnicki Cezary, and Evangelos P. Markatos. Implementation issues for the Psyche multiprocessor operating system. *Computing Systems*, 3(1):101–137, 1990.

[106] Marc Shapiro. A fault-tolerant, scalable, low-overhead distributed garbage detection protocol. In *Tenth Symp. on Reliable Distributed Sytems*, Pisa (Italy), October 1991.

[107] Marc Shapiro. A binding protocol for distributed shared objects. In *Proc. Int. Conf. on Distributed Computing Systems*, Poznan (Poland), June 1994.

[108] Marc Shapiro, Peter Dickman, and David Plainfossé. Robust, distributed references and acyclic garbage collection. In *Symp. on Principles of Distributed Computing*, pages 135–146, Vancouver (Canada), August 1992. ACM. Superseded by [109]: corrects a bug, more elegant, more informative.

[109] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapport de Recherche 1799, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), nov 1992. Also available as Broadcast Technical Report #1.

[110] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object-oriented operating system — assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.

[111] Marc Shapiro, Olivier Gruber, and David Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Rapport de Recherche 1320, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), November 1990.

[112] K. Singhal, S. Kakkad, and P. Wilson. Texas: An efficient, portable persistent store. In *Proc. of the Fifth International Workshop on Persistent Object Systems Design, Implementation and Use*, pages 13–28, San Miniato Pisa (Italy), September 1992.

[113] Richard L. Sites. *Alpha Architecture Reference Manual.* Digital Press, 1992.

[114] Pedro Sousa, Manuel Sequeira, André Zúquete, Paulo Ferreira, Cristina Lopes, José Pereira, Paulo Guedes, and José Alves Marques. Distribution and persistence in the IK platform: Overview and evaluation. *Computing Systems*, 6(4):391–424, November 1993.

[115] Guy L. Steele Jr. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[116] Peter Steenkiste and John Hennessy. Tags and type checking in Lisp. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 50–59, Palo Alto, California, October 1987.

[117] Sun Microsystems, Inc. NFS: Network file system protocol specification. RFC 1094, Network Information Center, SRI International, March 1989.

[118] G. Tel and F. Mattern. The derivation of distributed termination detection algorithms from garbage collection schemes. *ACM Transactions on Programming Languages and Systems*, 15(1):1–35, January 1993.

[119] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In Norman Meyrowitz, editor, *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '88) Proceedings*, pages 1–17, San Diego, California (USA), September 1988. ACM Press.

[120] David M. Ungar. Generation scavenging: A non-disruptive high-performance storage reclamation algorithm. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167. ACM Press, April 1984. Published as *ACM SIGPLAN Notices 19*(5), May, 1987.

[121] Francis Vaughan, Tracy Lo Basso, Alan Dearle, Chris Marlin, and Chris Barter. Casper: a cached architecture supporting persistence. *Computing Systems*, 5(3):337–359, 1992.

[122] Stephen C. Vestal. *Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming.* PhD thesis, Dept. of Comp. Sc., U. of Washington, Seattle WA (USA), January 1987. U. of Washington Tech. Report 87-01-03.

[123] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In *PARLE'87—Parallel Architectures and Languages Europe*, number 259 in Lecture Notes in Computer Science, Eindhoven (the Netherlands), June 1987. Springer-Verlag.

[124] Mark Weiser, Alan Demers, and Carl Hauser. The portable common runtime approach to interoperability. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, Litchfield Park, AZ (USA), December 1989. ACM Press.

[125] Brent Welch. A comparison of three distributed file system architecture: Vnode, Sprite, and Plan 9. *Computing Systems*, 7(2):175–199, 1994.

[126] E. P. Wentworth. Pitfalls of conservative garbage collection. *Software Practice and Experience*, 20(7):719–727, July 1990.

[127] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, Saint-Malo (France), September 1992. Springer-Verlag.

[128] Paul R. Wilson and Sheetal V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *International Workshop on Object Orientation in Operating Systems*, pages 364–377, Dourdan (France), October 1992.

[129] Paul R. Wilson and Thomas G. Moher. Design of the Opportunistic Garbage Collector. In *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '89) Proceedings*, pages 23–35, New Orleans, Louisiana, 1989. ACM Press.

[130] V. Yong, J. Naughton, and J. Yu. Storage reclamation and reorganization in client-server persistent object stores. In *Proc. Data Engineering Int. Conf.*, pages 120–133, Houston TX (USA), February 1994.

[131] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11:181–198, 1990.

[132] S. Zdonik and D. Maier. *Readings in Object-Oriented Database Systems.* Morgan-Kaufman, San Mateo, California (USA), 1990.

[133] Benjamin Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms.* PhD thesis, University of California at Berkeley, EECS Department, December 1989. Technical Report UCB/CSD 89/544.