Mesure et analyse des caractéristiques d'une mémoire d'objets

Nicolas Richer

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Mesure et analyse des caractéristiques d'une mémoire d'objets

Nicolas Richer

N ° 3315

Décembre 1997

THÈME 1

ISSN 0249-6399

Mesure et analyse des caractéristiques d'une mémoire d'objets

Nicolas Richer*

Thème 1 — Réseaux et systèmes
Projet SOR

Rapport de recherche n 3315 — Décembre 1997 — 83 pages

Résumé : Nous cherchons à rendre les systèmes de partage d'informations persistantes à la fois simples, efficaces et sûrs. Pour y arriver, il faut comprendre comment se comportent les applications qui en sont potentiellement utilisatrices. Un première étape dans la compréhension du comportement est la mesure des propriétés statiques et dynamiques du graphe des objets. Ce rapport propose une revue des travaux existants dans le domaine, ainsi que des résultats de mesures. Les mesures portent sur la densité et la longueur des références ainsi que sur la proportion de cycles et leur longueur. Un comparatif des résultats obtenus avec différentes politiques d'allocation mémoire est également proposé.

Mots-clé : comportement, cycle d'objets, mémoire, mesures, persistance, ramasse-miettes

(Abstract: pto)

* Email : Nicolas.Richer@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : 01 39 63 55 11 - International : +33 1 39 63 55 11
Télécopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

Measuring and analyzing the characteristics of an objects memory

Abstract: Information sharing systems for persistent data should be simple, efficient and safe. To attain this goal, it is essential to understand the behaviour of applications that use them. The first step is measurement of the static and dynamic properties of the object graph. This report review the state of the art in measurement systems, then provides some figures. Our measurements focus on the density and length of references, and on the length and number of reference cycles. We show the influence of different memory allocation policies on these figures.

Key-words: behaviour, garbage-collector, memory, measure, objects cycle, persistence

Sommaire

Chapitre 1

Introduction

Ce travail s'inscrit dans la cadre du projet de mémoire répartie persistante PerDiS [SKR97]. Le développement de ce système nécessite de connaître le comportement des applications utilisatrices afin de choisir les algorithmes les mieux adaptés.

L'objectif de notre travail est de caractériser le comportement de différents types d'applications en termes de gestion de la mémoire. Il s'agit d'analyser le graphe des objets présents en mémoire durant l'exécution de chaque application afin d'évaluer les longueurs de référence entre objets, la durée de vie des objets, le nombre de cycles et leurs longueurs, la taille des cycles de miettes, leur fréquence ainsi que leur distribution géographique dans la mémoire. Ces informations sont importantes pour le choix de l'algorithme de ramassage des miettes et la politique d'allocation mémoire à mettre en œuvre dans PerDiS.

Avant de s'intéresser au travail effectué durant le stage, nous présentons les concepts de base du projet de mémoire répartie persistante PerDiS afin de mettre en évidence les problèmes rencontrés et les interrogations auxquelles nous devons répondre.

1.1 Mémoire répartie persistante

Une mémoire répartie persistante est la conjonction d'une mémoire virtuellement répartie, d'un ramasse-miettes réparti et d'un système de fichiers. C'est un mécanisme système qui permet à des applications réparties d'allouer ou bien d'accéder par simple référence à des objets distribués sur plusieurs sites, qui deviennent persistants dès lors qu'il sont accessibles par un chemin quelconque depuis une racine de persistance.

Ce type de mémoire permet d'offrir un accès homogène aux objets en assurant la cohérence des données distribuées quelle que soit la localisation de l'application et du stockage. La notion de persistance par atteignabilité[1] permet aux applications de conserver facilement leurs données qui doivent persister. L'utilisation d'un ramasse-miettes libère le programmeur d'application de la gestion manuelle de la mémoire, source de nombreuses erreurs telles que fuites de mémoire et surtout pointeurs vers des objets qui n'existent plus.

1.2 Architecture générale de PerDiS

1.2.1 Grandes lignes de l'API

La mémoire répartie persistante PerDiS offre au programmeur une vision simple de la mémoire, similaire à une Mémoire Partagée Répartie [KCDZ94], mais où les objets accessibles persistent, c'est-

1. Tout objet, accessible par un chemin quelconque dans le graphe des objets depuis une racine de persistance, est persistant [ABC+83].

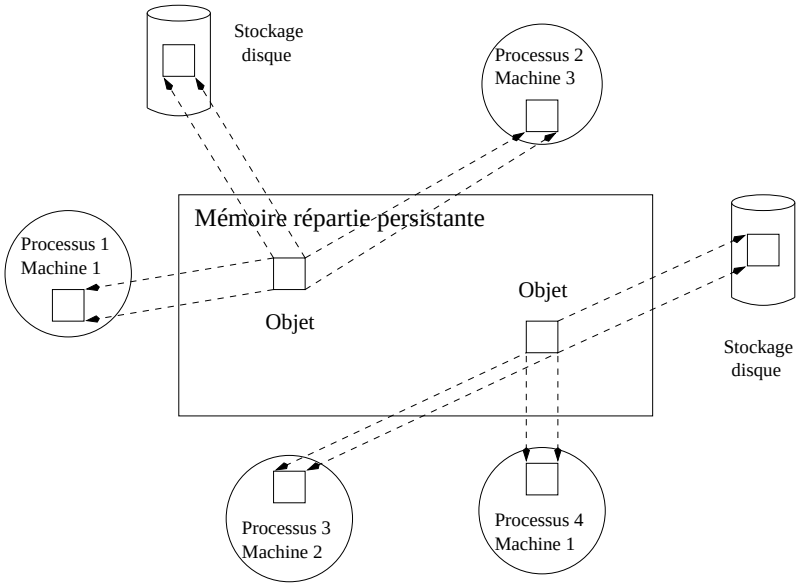

FIG. 1.1 – *Deux objets partagés chacun par deux processus*

à-dire sont stockés sur support secondaire. Un ramasse-miettes réparti assure la libération des objets inaccessibles [FS94]. Les objets persistants sont couplés dans l'espace d'adressage de tout processus qui les utilise (cf figure 1.1). PerDiS fonctionne sur un modèle transactionnel : les opérations effectuées dans une transaction ne sont prises en compte qu'au moment de la validation. Le partage des données est possible, y compris en écriture. Des transactions soit optimistes soit pessimiste sont utilisées. Dans le premier cas, la cohérence est assurée au moment de la validation et si plusieurs transactions sont en conflit, seule la première à valider sera acceptée.[2] Les autres devront alors obtenir une copie à jour des données pour y apporter des modifications, puis engager une nouvelle validation. Dans le cas des transactions pessimistes, les données sont verrouillées en écriture pendant toute la durée de la transaction et les autres applications doivent attendre la fin de cette transaction avant de pouvoir y accéder.

Une autre contrainte de PerDiS est qu'il doit être utilisable à grande échelle, puisque son domaine d'utilisation primaire est le support de logiciels coopératifs de conception assistée par ordinateur pour le bâtiment.[3] Pour cela, PerDiS introduit la notion de grappe.[4] D'un point de vue utilisateur, la grappe est l'unité visible de mémoire, de stockage, de nommage et de protection. Un objet à l'intérieur d'une grappe peut très bien référencer un objet dans une autre grappe. Comme une grappe peut avoir une taille quelconque, PerDiS découpe les grappes en flopées.[5] La flopée est l'unité indivisible sur laquelle peut travailler le ramasse-miettes, et par conséquent, c'est également l'unité d'échange et de regroupement. Une grappe sera composée d'une ou plusieurs flopées. La taille de ces flopées n'est pas déterminée pour l'instant, mais le travail de stage devra justement permettre d'avancer sur ce point. Les flopées sont échangées en une seule fois, ce qui signifie que, plus les objets à l'intérieur d'une flopée seront utilisés ensemble, et moins il y aura de flopées à échanger. Le ramasse-miettes opère sur l'ensemble des flopées présentes sur un site.

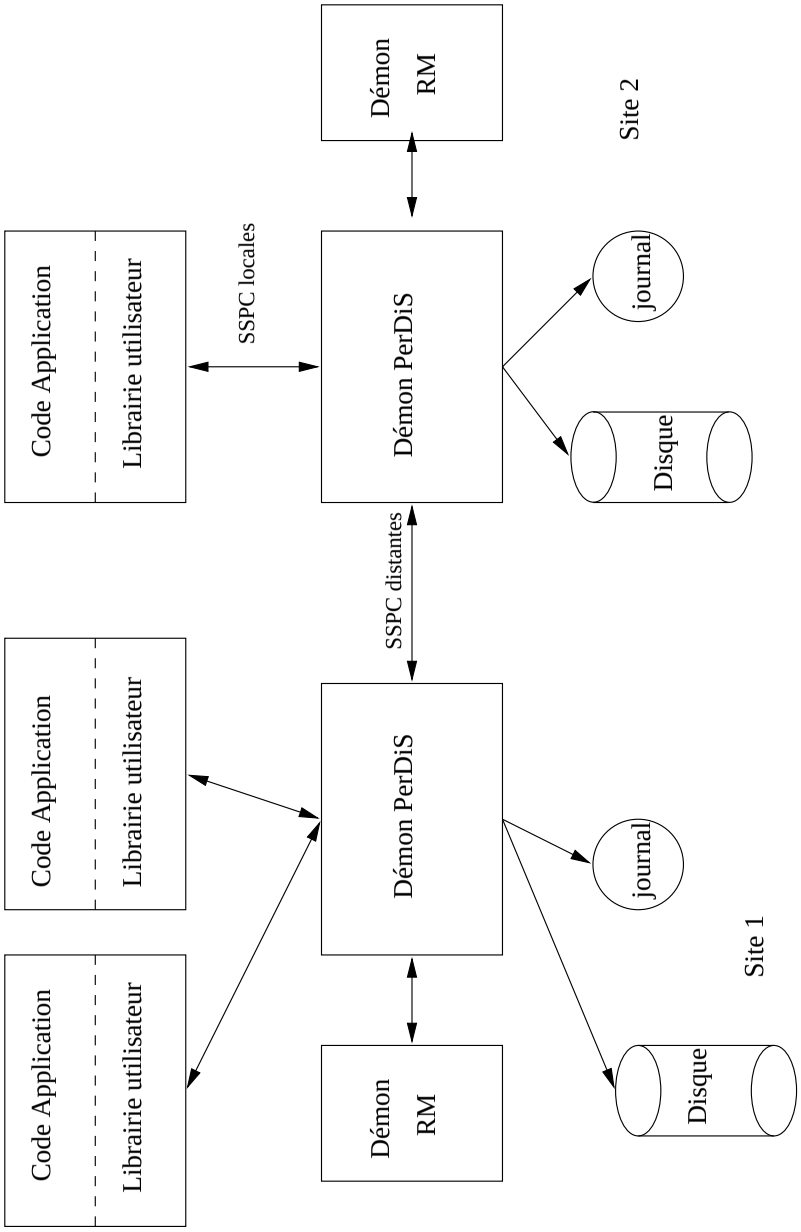

FIG. 1.2 – *Décomposition en modules*

1.2.2 Plate-forme d'accueil

PerDiS tournera au dessus de systèmes d'exploitation existants, sans qu'il soit nécessaire de modifier le noyau de ceux-ci. Il s'agit typiquement de systèmes Unix, tel que Solaris ou Linux, mais un portage sur d'autres plates-formes comme Windows NT est envisagé.

La figure 1.2 illustre le découpage de PerDiS en modules. Le démon PerDiS assure les opérations de validation, de verrouillage et met en cache localement les flopées récemment utilisées sur le site. Le démon RM (ramasse-miettes, dont l'heuristique de déclenchement n'est pas encore définie), parcourt toutes les flopées présentes dans le cache. Les références entre sites sont traitées par listage de références, ce qui autorise un fonctionnement à grande échelle en limitant les échanges entre les différents sites et en permettant à chaque ramasse-miettes local de s'exécuter indépendamment. En revanche, cette approche ne permet pas de ramasser les cycles de miettes répartis entre plusieurs sites. Ce problème est détaillé à la section 1.3.3.

1.3 Principales interrogations

1.3.1 Propriétés du graphe des objets en mémoire

Le premier aspect à étudier pour notre mémoire concerne les propriétés du graphe des objets en mémoire persistante. Cela comprend la mesure des longueurs de référence entre objets, la quantification du nombre de cycles parmis les objets vivant et dans les miettes, la mesure de la longueur de ces cycles ainsi qu'une analyse spécifique de la longueur des cycles de miettes.

Il existe quelques études concernant ce problème. L'une d'entre elles à été réalisée à l'université du Colorado par l'équipe de B. Zorn [GZH93]. Une autre a été faite par Cédric Adjih [Adj96] à l'occasion de son stage dans notre équipe. Les résultats qu'il a obtenus concernent la mesure d'applications centralisées utilisant la politique d'allocation standard de la librairie C au-dessus de laquelle la politique *segregated fit*[6] est mise en œuvre. Notre objectif est de compléter ces travaux par une analyse de l'impact de la politique d'allocation mémoire sur les longueurs de référence et les longueurs de cycle. Ces résultats nous permettront de choisir au mieux la politique d'allocation à mettre en œuvre dans PerDiS.

1.3.2 Qu'est-ce qui est persistant ?

Dans une mémoire persistante, les objets ont des durées de vie très variables. Nous avons, d'une part, des objets à vie très courte qui ne devraient durer que quelques micro-secondes et, d'autre part, des objets contenant les données permanentes qui, eux, vivent jusqu'à plusieurs années. Dans un environnement classique, les données persistantes sont stockées dans des fichiers, les différences entre les durées de vie des objets en mémoire sont donc assez faibles. Avec les mémoires persistantes, le programmeur peut allouer de la même façon les données utiles quelques jours et celles qui resteront plusieurs années. Dans ce contexte, il est capital de savoir quels sont les objets dont la durée de vie est proche afin, par exemple, de les regrouper dans la même flopée. Un autre point important est de déterminer si nous pouvons faire un regroupement automatique ou si le programmeur doit intervenir, comme nous l'envisageons pour l'instant. Les études antérieures, portant sur des objets en mémoire vive ne sont pas directement transposables au problème des objets persistants. Une autre partie du travail consistera donc à mesurer la durée de vie des objets[7].

2. Hypothèse d'une faible concurrence entre transactions.
3. Une activité de CAO coopérative regroupe de nombreuses entreprises aux compétences diverses.
4. *Cluster* dans tous les articles en anglais concernant PerDiS.
5. *Bunch* dans tous les articles en anglais concernant PerDiS.
6. Une brève description des politiques d'allocation mémoire se trouve dans [Ric97].
7. Une analyse de la durée de vie des fichiers serait également intéressante.

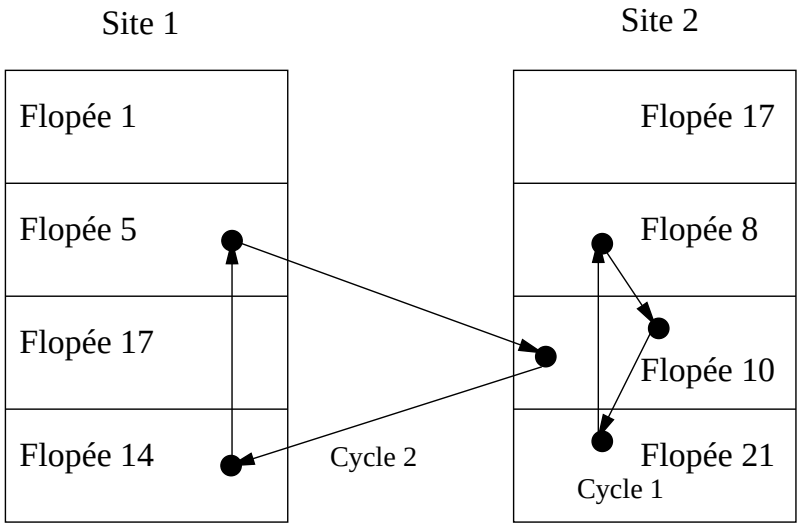

FIG. 1.3 – *Exemple de cycles de miettes*

Pour obtenir des résultats significatifs en terme de persistance, il nous faut étudier des applications persistantes. A l'heure actuelle, les applications persistantes sont soit inexistantes, soit difficiles à obtenir ou à manipuler. Le World-Wide Web est une application persistante potentielement intéressante, mais qui possède une sémantique d'accès différente d'une mémoire classique, aussi, nous avons donc dû nous limiter, au moins dans un premier temps, à l'utilisation de programmes d'évaluation de base de données objet, modifiés pour construire des graphes d'objets persistants en mémoire.

1.3.3 Les cycles de miettes répartis entre sites

Alors que le ramassage des miettes entre les flopées d'un même site s'effectue par traçage, ce qui permet de ramasser le cycle de miettes 1 de la figure 1.3, le ramassage des miettes entre les sites s'effectue par listage de références, ce qui ne permet pas de ramasser les cycles de miettes répartis entre les sites, comme le cycle 2 de la figure 1.3. Cependant, comme les flopées peuvent migrer d'un site à un autre, il est possible que toutes les flopées appartenant à un cycle de miettes se retrouvent sur le même site, ce qui permet le ramassage du cycle. Toute la question est de savoir si ces cycles seront nombreux ou pas et, surtout, si leur nombre restera stable. En effet, l'existence de cycles de miettes réparties n'est pas un problème en soi, mais s'il s'avère que leur nombre croît trop fortement au cours du temps, cela va mettre en péril la stabilité du système (fuite de mémoire).

La quantification des cycles de miettes répartis entre sites n'est pas facile à réaliser sans utiliser d'applications directement portées sur PerDiS. Une solution consiste à tracer des applications traditionnelles, puis caractériser le nombre et la longueur des cycles. Il est alors possible d'extrapoler le nombre de cycles inter-flopées en fonction de la taille des flopées et de la politique d'allocation.

Il faut toutefois rester prudent quant aux conclusions à tirer d'une telle étude sur des applications traditionnelles, qui ne sont pas conçues pour une mémoire persistante et stockent leurs données persistantes dans des fichiers. Les mesures que nous allons obtenir ne concerneront, en effet, qu'une partie des ressources de stockage, et ce n'est pas là que se trouvent les objets persistants.

Pour obtenir des résultat plus significatifs, il faudrait porter des applications sur la plate-forme PerDiS [8] et construire le graphe de tous les objets accessibles afin de quantifier précisément le nombre de cycles de miettes répartis et leur durée de vie. En attendant, l'étude d'applications classiques demeure néanmoins intéressante car elle permet de tester les outils de mesure et de disposer rapidement de

8. ou apporter des modifications sur des applications classiques afin que les données persistantes soit stockées séparément dans un graphe qu'il est possible d'étudier.

résultats que nous extrapolerons. L'expérience passée semble montrer que de telles extrapolations sont en gros correctes et nous pourrons vérifier si c'est bien le cas par la suite.

L'organisation de ce rapport est la suivante : Le chapitre 2 propose une synthèse de toutes les informations qui ont pu être recueillies dans la littérature. Le chapitre 3 expose la méthode que nous proposons pour obtenir le graphe des objets en mémoire durant l'exécution ainsi que les méthodes d'analyse et d'extrapolation qui sont envisageables. Le chapitre 4 présente les applications que nous avons étudiées dans le cadre de nos travaux ainsi que les mesures qui ont été réalisées. Enfin le chapitre 5 conclut ce rapport en proposant une extension du travail de stage dans un avenir proche et dans un avenir plus lointain.

Chapitre 2

Étude bibliographique

La première partie du travail de stage a consisté à rechercher dans la littérature les différentes études déjà existantes concernant l'analyse des allocations et désallocations ainsi que la structure du graphe des objets en mémoire. Parallèlement à cela, une étude sur les méthodes destinées à tracer ce graphe a été menée et a fait l'objet de mon rapport bibliographique [Ric97]. La section 2.3 en reprend la partie concernant les outils de traçage déjà disponibles pour analyser les allocations mémoire afin d'introduire la méthode de traçage et d'analyse du graphe des objets en mémoire que nous proposons au chapitre 3.

2.1 Étude des allocations et désallocations des objets en mémoire

L'étude des allocations et désallocations mémoire a fait l'objet de nombreuses publications au cours de ces dernières décennies. En outre, la plupart des ouvrages de programmation inclut au moins une partie consacrée à l'étude des politiques d'allocation mémoire.

Cependant, force est de constater que peu d'articles se basent sur la mesure de programmes réels. La plupart utilisent des générateurs de requêtes d'allocation synthétiques généralement basés sur des lois de distribution markoviennes. Wilson, Johnstone, Neely et Boles dans [WJNB95], proposent une synthèse détaillée sur l'allocation dynamique de la mémoire et mettent en évidence la faible représentativité des traces synthétiques. Ils proposent simultanément une définition précise des différents critères d'évaluation d'une politique d'allocation mémoire en s'appuyant sur les mesures qu'ils ont réalisées sur des programmes réels. Ces mesures concernent principalement la fragmentation produite par les différentes politiques d'allocation, mais aussi la vitesse des allocateurs. Elles sont publiées dans le mémoire de *Master* de Neely [Nee96] . La conclusion qui ressort de ces travaux est que la portion du tas inoccupé à cause de la fragmentation de la mémoire est relativement faible avec de bons allocateurs. Ainsi, elle est inférieure à 15% pour le *first fit* avec liste des blocs libres ordonnée par adresse (Une brève description des politiques d'allocation mémoire se trouve dans [Ric97]).

Dans la mémoire persistante PerDiS, les critères d'évaluation d'une politique d'allocation mémoire diffèrent de ceux d'une mémoire classique. Dans ce dernier cas, on s'intéresse habituellement à la vitesse d'allocation et dans une moindre mesure, à la fragmentation produite par les différents allocateur. Dans notre cas, une politique d'allocation est intéressante si elle limite la dispersion des objets dans le tas (il faut mesurer la longueur des références et des cycles en octets).

Zorn et Grunwald [ZG92a] proposent une étude des allocations mémoire sur un ensemble de six programmes réalisant de nombreuses allocations dynamiques. Dans [ZG92b], ils analysent ces mesures et proposent une prédiction du comportement des applications du point de vue des allocations mémoire.

Detlefs, Dosser et Zorn ont complété les mesures précédentes sur un plus grand nombre d'applications [DDZ93]. Il ont mis en évidence que toutes les applications mesurées allouent un nombre considérable d'objets.[1]

La conséquence d'une allocation aussi intensive est que les objets ont une durée de vie assez faible. Zorn et Grunwald [BZ93] indiquent que dans 75 % des cas, la durée de vie moyenne d'un objet n'excède pas le temps nécessaire pour allouer 32 Ko.[2] Les résultats sont plutôt intéressants puisque les meilleures prédictions sont fiables à 99 % et les plus mauvaises à 42 % seulement.

On peut se demander si ces résultats sont transposables au problème de la durée de vie des objets dans PerDiS. Les objets persistants peuvent avoir des durées de vie beaucoup plus longues que les objets détruits à la fin de l'exécution d'un programme, et la différence entre les durées de vie de deux objets peut être beaucoup plus importante.

Dans l'ensemble, les programmes mesurés par les différentes études n'excèdent pas la quinzaine, ce qui permet de déduire certaines informations, mais ne permet pas de dégager des classes de comportement. Pour cela, il faudrait étudier un nombre plus important d'applications, réalisant les mêmes types de traitements, afin de déterminer si un comportement en terme d'allocations mémoire selon le type d'application rencontré est prévisible, ou si c'est une propriété spécifique de chaque mise en œuvre.

2.2 Structure du graphe des objets en mémoire

La structure des objets présents en mémoire durant l'exécution d'une application est plutôt mal connue, particulièrement concernant des applications persistantes. Des études ont été réalisées sur des bases de données objets, mais la structure du graphe des objets n'a pas fait partie des propriétés recherchées. Les études les plus courantes portent sur la comparaison des performances de différentes stratégies de regroupement.[3]

Les travaux sur le système Eos [OG92] portent sur un environnement de programmation distribué et persistant. Cependant, bien que le ramasse-miettes utilisé repose sur les mêmes concepts que celui que nous voulons mettre en œuvre dans PerDiS, il n'y a pas d'étude sur la structure du graphe des objets dans la mémoire persistante, mais seulement une étude des performances des différentes stratégies de regroupement, particulièrement le regroupement dynamique.

Le seul article qui donne des informations concernant le graphe des objets en mémoire sur un système réel est celui de DeTreville [Tre90]. Les applications mesurées sont *Taos* (le système d'exploitation de *Topaz*) et *Ivy*, un éditeur de texte, toutes les deux écrites en Modula 2+. Les points à retenir sont les suivants :

- La majorité des pointeurs pointent vers l'arrière[4] (42,8 %) contre 25,6 % vers l'avant, mais de toutes façons à une distance assez faible (le reste pointe vers *nil* ou vers des objets inexistants).
- La proportion d'objets dans les cycles est relativement faible pour les deux applications mesurées (8,1 % pour *Taos* et 11,2 % pour *Ivy*).
- Le degré incident moyen (cf §3.1.3) est de 1,3 dans *Taos* et de 1,8 pour *Ivy* alors que le degré sortant moyen est de, respectivement, 0,9 et 1,7.

Ces résultats sont intéressants pour nous, car ils semblent indiquer que le nombre de cycles est plutôt faible (autour de 10%) et que les longueurs de référence sont également faibles. Cependant, l'étude

1. Les onze applications mesurées allouent entre 200Ko/s et 4 Mo/s.
2. La durée de vie des objets est mesurée en nombre d'octets alloués par le programme durant leur existence.
3. Luciana Arantes en propose une sunthèse dans [LBA97].
4. La mesure concerne ici les adresses des objets dans le tas. Un pointeur vers l'arrière signifie que l'objet pointé est avant l'objet qui contient le pointeur dans le tas.

porte sur seulement deux applications, ce qui est tout à fait insuffisant pour en tirer des conclusions générales. En outre, ces applications sont assez anciennes, et ne reflètent donc pas nécessairement le comportement d'applications récentes.

2.3 Outils de mesure et d'analyse disponibles

Une étude préliminaire des outils de mesure disponible a mis en évidence l'abscence de logiciel standard adapté à l'analyse du graphe des objets en mémoire. Fort de ce constat, nous avons décider de dévelloper notre propre outils sur la base d'Analysis.

Analysis est un outil de traçage et d'analyse du graphe des objets en mémoire développé à l'INRIA SOR par Cédric Adjih [Adj96]. Analysis se décompose en deux parties indépendantes :

1. L'enregistreur dans un fichier des informations sur le graphe des objets en mémoire ;
2. L'analyse, qui exploite ce journal afin d'en déduire les propriétés du graphe.

Les informations sur le graphe sont fournies par une version modifiée du ramasse-miettes de Boehm [Boe91]. Les allocations d'objets sont également enregistrées en redirigeant les appels aux allocateurs classiques (malloc, calloc, realloc, free) vers des allocateurs spécifiques, afin de connaître leur ordre précis de création. Le code de l'enregistreur est ajouté aux applications par édition de liens.

Cela nécessite généralement la recompilation de l'application en ajoutant la bibliothèque contenant l'outils. Toutefois, sur certains systèmes, comme Digital Unix, il est possible d'utiliser directement une bibliothèque dynamique. Pour cela, un mécanisme de redirection peut être mis à profit. Il s'agit de créer une bibliothèque dynamique contenant le code du ramasse-miettes et de spécifier à l'éditeur de lien dynamique d'utiliser cette bibliothèque avant toutes les autres.

L'analyse du graphe est assurée par un outil complètement indépendant de la partie traçage. Seul le format de représentation du graphe des objets est connu de l'analyseur. Des classes de base sont disponibles, permettant de construire rapidement des classes d'analyse de haut niveau pour identifier les différentes propriétés. Actuellement, Analysis permet de caractériser les cycles dans le graphe des objets accessibles ainsi que dans les miettes.

Chapitre 3

Méthodes de mesure et d'analyse d'applications réelles

Dans ce chapitre, nous présentons en détail les mesures que nous allons réaliser dans le cadre de notre travail de caractérisation ainsi que les techniques de traçage et d'analyse à mettre en œuvre pour y parvenir.

3.1 Propriétés à identifier

3.1.1 Durée de vie des objets

La mesure de la durée de vie des objets consiste à dater la création et la libération de chaque objet. Il est ainsi possible de calculer, par exemple, la durée de vie moyenne, ainsi que le nombre d'occurrences de chaque durée de vie.

Unités de mesure :

1. La plus intuitive est la mesure en secondes, mais ce n'est pas celle qui nous paraît la plus intéressante. En effet, elle ne permet pas de savoir ce qui s'est passé au sein du programme pendant la vie de l'objet. De plus, comme les mesures se limitent dans un premier temps à des applications non persistantes, les différences entre les durées de vie exprimées en secondes sont très faibles ;

2. Une métrique plus intéressante est la datation logique en nombre d'allocations mémoire réalisées pendant la vie de l'objet. Elle permet de différencier les objets temporaires de très faible durée de vie des structures du programme qui vivent souvent pendant toute son exécution ;

3. Zorn [BZ93] propose de mesurer la durée de vie en nombre d'octets alloués par l'application pendant la vie des objets.

Dans nos résultats, seule la dernières unités de mesure a été retenue. Il est bien sûr possible d'ajouter d'autre métriques si cela se révèlait nécessaire.

3.1.2 Longueurs des références

La longueur de référence est la mesure qui permet de caractériser l'éloignement de deux objets dont l'un référence l'autre.

Unités de mesure :

1. Nombre d'octets séparant les objets dans la mémoire, ce qui permet de savoir comment les objets en relation les uns avec les autres sont dispersés dans le tas (dispersion spatiale) ;

2. Distance en différence d'octets alloués par le programme. Cette mesure consiste à dater les objets à leur création avec le nombre d'octets déjà alloués par le programme, et à calculer une différence de date pour chaque référence rencontré.

3.1.3 Degré incident et sortant

Le degré incident caractérise le nombre de références qui pointent vers un objet. Le degré sortant indique le nombre de références que contient chaque objet. Ces mesures permettent d'apprécier globalement la densité des références entre objets. Par exemple, si le degré incident moyen est proche de 1, alors il y a peu de cycles (mais ceux-ci sont peut-être trés grands).

3.1.4 Caractérisation des cycles

Le point principal de notre travail de caractérisation porte sur l'étude des cycles du graphe d'une mémoire d'objets. Cela regroupe différentes mesures.

Nombre de cycles, nombre d'objets dans les cycles, volume d'octets dans les cycles

Le nombre de cycles présents dans le graphe des objets doit être mesuré pour le graphe des objets vivants comme pour le graphe des miettes. Ainsi, à chaque déclenchement du ramasse-miettes, il est possible de quantifier le nombre de cycles dans les objets vivants ainsi que dans les miettes.

Ce qui nous intéresse n'est pas le nombre de cycles en lui même, mais la proportion d'objets et d'octets dans les cycles. Ces informations sont importantes pour la conception du ramasse-miettes de PerDiS. En effet, dans sa forme actuelle, l'algorithme utilisé fait l'hypothèse d'un volume de cycles faible par rapport au nombre d'objets total. Si cette hypothèse ne se vérifie pas, il faudra sans doute envisager d'autres mécanismes permettant par exemple de ramasser les cycles de miettes répartis (cf figure 1.3).

Diamètre des cycles

La diamètre des cycles, que ce soit dans les objets vivants ou dans les miettes est une autre caractéristique importante. En effet, pour limiter le nombre de cycles qui pourraient être répartis entre différents sites (cf §1.3.3), il faut qu'un maximum de cycles soit inclus en totalité à l'intérieur d'une même flopée. Pour cela, la taille des flopées pourrait être choisie de façon à être supérieure au diamètre moyen des cycles, et minimiser ce diamètre. La politique d'allocation mémoire qu'il faut utiliser pour allouer les objets de chaque grappe dans les flopées doit limiter la dispersion des objets ayant des références les uns vers les autres.

L'utilisation d'une telle politique d'allocation ne permet pas seulement de limiter la formation de cycles de miettes inter-flopées, elle contribue globalement à l'amélioration des performances. En effet, la flopée étant l'unité d'échange indivisible dans PerDiS, regrouper au maximum les objets ayant des liens dans la même flopée permet de réduire les échanges entre les sites (notons que cela doit aussi permettre de diminuer le faux partage).

Unités de mesure :

1. Nombre d'objets impliqués dans les cycles et plus particulièrement nombre d'occurrences de chaque longueur ;

2. Volume (càd nombre d'octets) inclus dans le cycle, et plus particulièrement nombre d'occurrences de chaque taille ;

3. Diamètre en nombre d'octets, incluant les octets éventuellement intercalés au milieu du cycle. Ce diamètre peut être différent du nombre d'octets inclus dans le cycle parce que l'allocateur disperse les objets dans le tas. Il s'agit en fait de calculer la différence entre la plus petite et la plus grande adresse appartenant au cycle. Nous pourrons ainsi évaluer la dispersion des objets dans le tas en fonction de l'allocateur et choisir la meilleure politique de ce point de vue. La taille des flopées peut être déterminée de façon à être supérieure au diamètre moyen.

Pour extraire ces informations, il faut disposer de techniques pour analyser les applications. Le reste de ce chapitre décrit celles que nous proposons pour y parvenir.

3.2 Comment enregistrer les caractéristiques d'une mémoire d'objets ?

Concrètement, les informations dont nous avons besoin pour réaliser les mesures décrites plus haut sont, d'une part, les allocations et désallocations mémoire afin de calculer la durée de vie des objets, et, d'autre part, la topologie du graphe des objets en mémoire et son évolution durant l'exécution du programme. Pour obtenir la topologie, deux approches sont envisageables : enregistrement de toutes les écritures mémoire ou parcours périodique du graphe.

3.2.1 Allocation et désallocation

Le calcul de la durée de vie des objets nécessite seulement d'enregistrer les allocations et désallocations mémoire. Il est par conséquent très facile de rediriger les appels aux fonctions d'allocations (malloc, calloc, realloc) et à la fonction de libération (free) afin d'enregistrer ces appels dans un journal.

3.2.2 Journalisation de toutes les écritures en mémoire

La technique est difficilement applicable pour la topologie du graphe, car la connaissance des allocations et désallocations n'est plus suffisante. Il faudrait en plus enregistrer toutes les écritures en mémoire, afin de déduire, a posteriori, la topologie du graphe et ses évolutions successives.

La mise en œuvre de cette méthode pose deux problèmes majeurs :

1. Il n'est pas simple de déduire le graphe des objets à partir des seules allocations et écritures mémoire. En effet, avec des programmes écrits en C ou en C++, il n'est pas possible d'identifier précisément les références sans modifier le compilateur. Pour y parvenir, il faut utiliser une méthode inspirée des ramasse-miettes conservatifs [Boe91] qui consiste, pour chaque écriture réalisée, à vérifier si le contenu ne pourrait pas être une référence vers un objet alloué dynamiquement. Une fois le graphe reconstruit, il est alors possible de le parcourir afin d'identifier les cycles qui nous intéressent. Il apparaît clairement que ce type d'analyse est particulièrement coûteux à mettre en œuvre, surtout au vu du nombre élevé de références mémoire qu'il va falloir traiter et donc au nombre très important de lecture/écriture disque à réaliser.

2. Si, malgré les difficultés d'analyse, nous décidions d'utiliser cette méthode, la taille du journal générée serait très importante [1] et le ralentissement de l'application serait également extrêmement

1. Larus dans [Lar90] indique que le traçage des références mémoire sur une machine de 10 Mips nécessite 60 Mo d'espace par seconde.

important.[2] Des mécanismes existent pour réduire la taille du journal [Lar93], mais ils ont, en général, pour conséquence d'augmenter encore la charge de travail nécessaire au moment de l'analyse, alors qu'elle est déjà considérable.

Cette technique, qui ne semble pas être réalisable en pratique, a cependant l'avantage de fournir une vision complète de l'évolution de la topologie du graphe des objets. Nous verrons à la section suivante que ce n'est pas le cas avec la méthode que nous proposons.

3.2.3 Parcours périodique du graphe

L'approche que nous proposons repose sur le parcours du graphe des objets en mémoire grâce à un ramasse-miettes modifié. Le ramasse-miettes dispose en effet de toutes les informations utiles :

- Il identifie les objets vivants et les miettes ;
- Il est capable de détecter les pointeurs à l'intérieur des objets.

A partir d'un algorithme de ramasse-miettes classique par marquage puis balayage, il est possible d'ajouter du code pour enregistrer dans un journal les références entre les objets ainsi que l'état de ces objets (vivants ou miettes) au moment du balayage.

Il y a alors deux possibilités pour mesurer le durée de vie des objets. La première consiste à calculer la différence entre les dates d'appel à malloc et à free. La seconde à considérer la différence entre la date d'allocation et la date à laquelle l'objet aurait été libéré par un ramasse-miettes, c'est-à-dire la durée de vie intrinsèque de l'objet. La première méthode ne semble pas la plus intéressante car elle repose entièrement sur l'intuition du programmeur. Dans le cadre de notre travail de caractérisation du comportement des applications pour la plate-forme PerDiS, il nous semble préférable d'utiliser la deuxième, qui nous permet d'évaluer la durée de vie intrinsèque de l'objet. En outre, la fonction *free* n'effectue aucun traitement, seul le ramasse-miettes peut libérer les objets. De cette façon le programmeur ne peut pas casser les cycles avec des libérations manuelles.

L'utilisation d'un ramasse-miettes permet de limiter les informations contenues dans le journal. Celui-ci ne contient plus que les références et l'état des objets, ce qui réduit sa taille.[3] En outre, le travail d'analyse est plus simple puisque l'identification des références entre objets est déjà faite.

L'inconvénient de cette approche est qu'elle ne permet pas de voir toutes les modifications . Ce n'est pas un inconvénient trop important pour nous, car nous recherchons principalement des statistiques sur le graphe des objets en mémoire, il semble donc raisonnable de penser que des images périodiques du seront représentatives de l'évolution globale.

Il n'en demeure pas moins que le facteur de déclenchement du ramasse-miettes peut influer de façon non négligeable sur nos mesures. En particulier, la politique classique qui consiste à déclencher le ramasse-miettes avant de demander plus de mémoire au système n'est pas adaptée à nos besoin, car cette politique conduit à déclencher de moins en moins souvent à mesure que le tas grandit. Cela a pour conséquence de réduire notre vision du graphe au moment où il devient le plus intéressant. Nous avons donc mis en œuvre un système de déclenchement régulier à chaque fois qu'une quantité déterminée d'octets a été allouée. Cette technique n'est pas bonne en terme de performances, mais permet d'avoir une vision régulière et paramétrable du graphe des objets.[4]

En outre, comme nous l'avons déjà mentionné, le typage dans des langages comme le C ou le C++ est trop faible pour permettre d'identifier précisément les références. Il faut alors utiliser un ramasse-miettes prudent qui considère tout ce qui ressemble à une référence comme telle. Le problème est alors

2. Principalement à cause du coût des Entrées/Sorties

3. Nous verrons cependant à la section 3.3 que cette taille demeure encore très importante et proposerons une solution à la section 3.4.

4. Elle est en outre envisagée pour le ramasse-miettes de PerDiS.

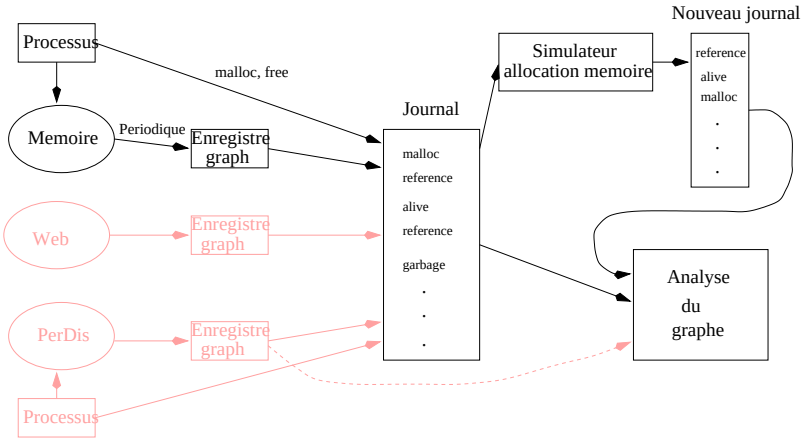

FIG. 3.1 – *Synoptique du système de mesure*

que toutes les miettes ne sont pas nécessairement bien identifiées. En effet, certains objets peuvent être référencés par une zone qui ressemble à un pointeur, mais qui en réalité n'en est pas un. Ces objets sont alors considérés comme vivants alors que ce sont des miettes. Ce problème n'est toutefois pas crucial, car des études sur les ramasse-miettes conservatifs ont montré que l'essentiel des références était bien identifié [Wen90], particulièrement dans le cas d'adresse sur 64 bits comme c'est le cas avec nos machines DEC Alpha.

3.2.4 Représentation du graphe

La méthode décrite ci-dessus a été mise en œuvre par Cédric Adjih en se basant sur le ramasse-miettes de Boehm[5], disponible dans le domaine public. Un soin important a été apporté à complètement séparer la partie journalisation du graphe de la partie analyse. À cet effet, une interface de représentation de graphe, par événements, a été définie. Elle comprend les événements suivants :

- allocation mémoires avec la taille de l'objet, son adresse, et son numéro d'allocation (temps logique) ;
- désallocation de mémoire avec l'adresse de l'objet ;
- déclenchement et fin du ramasse-miettes ;
- état d'objet (vivant ou miette), contenant l'adresse de l'objet ;
- référence entre deux objets. Il contient l'adresse du début de l'objet contenant la référence et la référence elle-même qui désigne l'objet pointé.

3.3 Analyse statique du graphe des objets

Cette section est consacrée à la description des techniques d'analyse du graphe des objets. Nous ne reviendrons pas cependant sur la mesure de la durée de vie des objets, déjà introduite à la section 3.2.2, qui ne pose pas de problèmes d'analyse particuliers.

5. http://reality.sgi.com/employees/boehm_mti/gc.html

3.3.1 Traitement préalable du journal

Les différentes propriétés qui nous intéressent ne peuvent pas être déduites directement du journal. Un traitement préalable est donc nécéssaire avant toute analyse.

Comme indiqué à la section 3.2.4, le journal est composé d'événements. Chaque événement apporte une information qui n'est pas utilisable seule. Par exemple, l'événement *référence* permet de décrire une référence entre deux adresses, mais ne contient pas les dates d'allocation des deux objets. Pour calculer la longueur de référence en date d'allocation, il faut donc établir une correspondance entre l'événement référence et les deux événements allocation correspondant aux deux objets en relation.

Plus généralement, il faut, pour chaque déclenchement du ramasse-miettes, matérialisé dans la trace par un événement *début* et un événement *fin*, construire une table indexée par les sommets du graphe (les objets) et contenant en plus des informations d'allocations, les références sortantes (liste de successeurs) ainsi que l'état des objets (vivants ou miettes). A partir de cette table, toutes les opérations d'analyse sont réalisables.

Il est à noter qu'il serait possible d'éviter cette phase en enregistrant toutes les informations sur les allocations et l'état des objets dans chaque événement *référence*. De cette manière, l'essentiel des informations utiles serait déjà regroupé dans la trace ; il n'y aurait plus qu'à construire la liste des successeurs pour chaque sommet. Cette méthode n'a pas été mise en œuvre car elle introduit beaucoup trop d'informations redondantes dans la trace (plusieurs références peuvent partir d'un même objet).

3.3.2 Les longueurs de référence

La mesure des longueurs de références entre les objets nécessite, pour chaque référence, le calcul d'une différence entre l'adresse de l'objet pointé et l'adresse de l'objet d'origine. Notons que l'opération doit être répétée pour chaque unité de mesure choisie. Dans notre cas, nous avons indiqué à la section 3.1.2 que nous nous intéressions à la mesure en octets et en date logique. Il y a par conséquent deux opérations à réaliser pour chacune des références.

Plusieurs sortes de statistiques sur les longueurs de références peuvent être intéressantes :

- La longueur de référence moyenne vers l'avant et vers l'arrière ;
- Le nombre d'occurrences de chaque longueur de référence ainsi que la proportion qu'il représente parmi le nombre de références totales.

Remarque : La longueur de référence en octets qui est mesurée ne dépend pas seulement de la politique d'allocation, mais aussi de sa mise en œuvre. En effet, selon la taille des informations ajoutées dans le tas, nos résultats ne seront pas les mêmes. De plus la taille des pointeurs utilisé (32 bits ou 64 bits) influe également sur les résultats, car la taille et l'alignement des structures de données n'est plus la même.

3.3.3 Degrés sortant et incident

La mesure des degrés sortant et incident ne pose aucune difficulté une fois que toutes les informations contenues dans la trace concernant les références et les allocations sont regroupées comme indiqué un peu plus haut.

3.3.4 Caractérisation des cycles

La recherche des cycles dans le graphe est l'analyse la plus délicate à réaliser. Le graphe des objets en mémoire est, d'une façon générale, un graphe orienté non connexe. Pour détecter les cycles dans un tel graphe, nous avons utilisé l'algorithme de Tarjan [NSS93, Tar71]. Chaque cycle détecté donne lieu à la création d'une liste d'objets participants. Il est alors possible de recueillir toutes les

informations nécessaires concernant le nombre d'objets, le nombre d'octets, le diamètre de chaque cycle et d'en déduire toutes les statistiques dont nous avons parlé à la section 3.1.4. Notons qu'avec cet algorithme, les objets n'appartenant pas directement à des cycles, mais accessibles uniquement depuis des objets appartenant à des cycles ne sont pas comptabilisés dans les cycles. Pour les prendre en compte, l'algorithme à mettre en œuvre serait beaucoup plus complexe.

3.3.5 Influence de la politique d'allocation mémoire

Pour étudier l'influence de la politique d'allocation mémoire, il y a deux approches possibles. L'une consiste à faire les mesures sur chaque application en utilisant différents allocateurs et l'autre propose de ne faire qu'une mesure et de simuler les résultats qui auraient été obtenus avec les autres politiques. Dans le cas d'une analyse statique, c'est-à-dire après la fin de l'exécution de l'application (post-mortem), la simulation semble la plus intéressante.

Mesure avec plusieurs allocateurs

La mesure avec plusieurs allocateurs consiste à reproduire les mesures à faire sur chaque application pour les différents allocateurs. Elle a l'avantage de fournir des résultats sur une exécution qui a réellement eu lieu, mais surtout, elle autorise la mise en place d'un système d'analyse dynamique pendant l'exécution.

Sa mise en œuvre nécessite de disposer d'allocateurs qui fonctionnent avec un ramasse-miettes modifié pour enregistrer le graphe des objets pendant son exécution. A cette fin, un ramasse-miettes pouvant fonctionner avec différents allocateurs ainsi que quatre allocateurs mettant en œuvre les politiques d'allocation linéaire, *first fit*, *best fit* et *roving pointer* ont été écrits.

Simulation de différentes politiques d'allocation

La simulation de différentes politiques d'allocation mémoire à l'avantage de ne nécessiter qu'une seule exécution réelle de l'application.

Sa mise en œuvre à partir de l'interface de représentation de graphe décrite à la section 3.2.4 a été réalisée comme un pré-processeur de trace. Ainsi, le simulateur reproduit les allocations et libérations par rapport aux informations inscrites dans la trace, et maintient une table réalisant la correspondance entre l'adresse enregistrée dans la trace et l'adresse que la politique d'allocation mémoire simulée aurait affecté à chaque objet. Pour toutes les occurrences des anciennes adresses, un remplacement par les nouvelles est effectué en utilisant la table. Une analyse de cette nouvelle trace peut alors être menée.

Le simulateur qui a été écrit pendant le stage fonctionne avec les allocateurs que nous avons écrit pour les mesures réelles, mais peut également utiliser l'allocateur standard de la bibliothèque C ainsi que n'importe quel allocateur disponible.

3.3.6 Limite de l'analyse statique

Les mécanismes que nous venons de décrire pour obtenir des informations sur le graphe des objets en mémoire sont basés sur une distinction très nette entre (1) la phase de journalisation, qui consiste à recueillir le graphe, et (2) la phase d'analyse qui extrait de ce graphe différentes sortes de statistiques. De fait, la distinction de ces deux phases dans deux outils complètement différents présente de nombreux avantages :

- Le programme d'analyse peut être utilisé avec différent systèmes de journalisation afin d'étudier plusieurs sortes d'applications (par exemple le World-Wide Web) ;
- La trace obtenue après la première phase peut servir de base à une simulation de différentes politiques d'allocation mémoire, sans qu'il soit nécessaire d'exécuter plusieurs fois l'application cible ;

– Un outil en deux parties indépendantes l'une de l'autre est plus simple à réaliser qu'un outil monolithique.

Cependant, il y a deux inconvénients à la séparation en deux phases. Le premier est que cela impose la création d'une trace dont la taille peut devenir importante sur une exécution de longue durée.[6] Le deuxième problème est que le temps d'analyse d'une trace est fréquemment très long, et souvent plus long que le temps d'exécution lui-même.[7]

Forts de cette constatation, nous avons décider d'étudier un mécanisme d'analyse dynamique parallèlement à notre interface journalisation-puis-analyse.

3.4 Mesures dynamiques

L'idée de mettre en œuvre un mécanisme de mesure dynamique a pour origine les difficultés rencontrées lors de la réalisation du mécanisme de mesure post-mortem. En effet, la phase d'analyse est lourde, d'autant plus que le graphe est dense. Ce constat est d'autant plus regrettable qu'une partie importante du travail d'analyse est consacrée à la reconstruction en mémoire du graphe enregistré dans la trace. Dans ces conditions, l'utilisation d'une technique où les mesures se font directement pendant l'exécution du programme semble intéressante. En réalisant les mesures dynamiquement, il est possible d'éviter deux phases de traitements assez lourdes :

– Enregistrement des événements dans une trace, opération qui demeure coûteuse à cause des nombreuses entrées/sorties. Certes, un système d'enregistrement différé et le buffer cache du système limitent le coût, mais ne l'éliminent pas ;

– Reconstruction du graphe des objets en mémoire à partir des informations contenues dans la trace (cf §3.3.3), opération qui nécessite beaucoup de temps d'exécution.

À partir du ramasse-miettes que nous avons développé pour enregistrer le graphe des objets en mémoire, nous avons ajouté des fonctions destinées à enregistrer les informations qui nous intéressent.

3.4.1 Durée de vie

La mesure dynamique de la durée de vie a été réalisée sans trop de difficultés en réservant un champ de datation dans l'en-tête des cellules. La date de création en numéro logique d'allocation est alors enregistrée dans l'entête au moment de la création des cellules, et le ramasse-miettes est en mesure de calculer la durée de vie lorsqu'il libère une cellule qui n'est plus accessible. Un système de statistique qui permet de calculer la durée de vie moyenne ainsi que d'enregistrer le nombre d'occurrences de chaque durée de vie est utilisé pour recueillir les informations.

Le coût de ce mécanisme de mesure en temps d'exécution est négligeable. Il est réel en revanche pour ce qui concerne l'utilisation mémoire puisqu'un champ supplémentaire doit être ajouté, ce qui peut faiblement influencer la mesure de la longueur des références. Toutefois, cela ne semble pas être un problème majeur car nous cherchons seulement à dégager des tendances générales de comportement. Il y a bien d'autres facteurs qui influent sur les mesures comme les différentes mises en œuvre possibles d'une même politique d'allocation ou la taille des types et structures d'un programme, qui peuvent différer selon la machine ou le compilateur utilisé.

6. Des essais sur Ghostscript ainsi que sur OO7 (cf chapitre 4) font apparaître des traces de plusieurs centaines de Mo.

7. Par exemple, pour l'application OO7, l'exécution a pris plusieurs heures, et l'analyse plusieurs jours.

3.4.2 Longueur des références

La mesure dynamique de la longueur des références fait appel au même principe que la mesure de la durée de vie. Le ramasse-miettes intègre les données concernant toutes les références qu'il détecte dans ces statistique. A la fin de l'exécution du programme, les résultats concernant un instantané, c'est-à-dire un déclenchement du ramasse-miettes, et les résultats concernant les miettes sont enregistrés dans des fichiers.

Comme pour la mesure de la durée de vie, le coût en temps d'exécution est négligeable.

3.4.3 Degrés sortant et incident

La mesure dynamique du degré sortant ne pose pas de problème : il suffit de compter les références lorsqu'une cellule est parcourue et enregistrer le résultat à la fin. La mesure du degré entrant, pour sa part, repose sur l'utilisation d'une structure de données auxiliaire, la table des références, déjà construite dans notre ramasse-miettes. Il s'agit d'une table qui contient une entrée pour chaque cellule allouée. Cette structure, qui est à l'origine destinée à déterminer rapidement si une zone mémoire pointe vers un objet alloué, peut être étendue. Un champ représentant le nombre de références entrantes peut y être ajouté. Ainsi, à chaque fois que le ramasse-miettes détecte une référence, le compteur de la cellule référencée est incrémenté. A la fin de l'exécution du ramasse-miettes, les résultats concernant chaque cellule sont recueillis.

Le coût de la mesure en temps d'exécution est un peu plus élevé que précédemment. En effet, à la fin de chaque ramassage des miettes, la table contenant les compteurs doit être parcourue. En revanche, il n'y a aucun problème concernant la mémoire, car la structure de données se trouve dans une zone particulière à la fin du tas et disparaît après l'exécution du ramasse-miettes.

3.4.4 Détection dynamique des arcs multiples

Les premières mesures statiques réalisées sur les applications Ghostscript et OO7 ont fait apparaître un nombre très important de références multiples (plusieurs références entre un objet et un autre objet donné). Ces références ne sont pas intéressantes pour nos mesures, car elles n'influent pas sur les cycles ; seuls les degrés entrant et sortant ainsi que le nombre de longueurs de référence sont concernés. En revanche, dans le cadre de notre système d'analyse classique, utilisant une trace, elles sont embarrassantes car, d'une part, elles occupent de la place dans la trace et, d'autre part, elles augmentent inutilement le travail nécessaire pendant l'analyse.

Un mécanisme d'élimination dynamique des références multiples, qui repose sur le même principe que le comptage des degrés entrant, a donc été mis en œuvre. Un champ réservé à cette fin a été ajouté à la table des références. Il permet de marquer tout les objets déjà accessibles depuis un autre objet lors du parcours de celui-ci. Ainsi, lorsqu'une référence pointe vers un objet marqué, c'est que c'est une référence multiple à éliminer. Le coût d'un tel mécanisme n'est pas négligeable, car il faut initialiser les marques entre chaque parcours de cellules, mais il a permis de considérablement réduire le temps d'analyse statique sur certaines applications.

3.4.5 Caractéristiques des cycles

La caractérisation dynamique des cycles n'a pas encore pu être mise en œuvre par manque de temps. C'est bien sûr la mesure la plus complexe à faire dynamiquement. Il me semble cependant qu'elle est réalisable avec l'algorithme de recherche de cycles de Tarjan [NSS93, Tar71] utilisé dans l'analyse statique et la table des références pour mémoriser les sommets ouverts (sommet qui ont déjà été visité). Le surcoût d'exécution ne sera bien sûr plus du tout négligeable, mais le temps globalement nécessaire pour obtenir des résultats devrait être plus faible qu'avec l'analyse statique. Dans tous les cas les deux approches ne sont pas à opposer, car elles se complètent. L'analyse dynamique permet

d'identifier globalement le comportement d'une application, et une analyse statique plus détaillée peut alors être menée sur certains instantanés de la mémoire.

L'intégration d'un système de mesure dynamique dans plusieurs ramasse-miettes de différentes origines est possible sans avoir à trop les modifier. En particulier, l'intégration du système de traçage statique ainsi que du mécanisme d'analyse dynamique dans le ramasse-miettes de PerDiS pourra être réalisée sans obstacle majeur. Cela permettra alors d'avoir des résultats concernant les données persistantes, ce qui nous intéresse au plus haut point.

Chapitre 4

Résultats des mesures

Un ensemble de cinq applications, réalisant chacune des allocations mémoire intensives, a été mesuré. Les résultats détaillés de ces mesures sont présentés dans les sections suivantes de ce chapitre, à raison d'une section par application. Toutes les mesures ont été réalisées sur des machines DEC d'architecture 64 bits sous Digital Unix 4.0. A chaque fois, plusieurs politiques d'allocation mémoire ont été utilisées ou simulées.

L'analyse de l'ensemble de ces résultats se trouve dans la conclusion de ce rapport, car ils constituent l'aboutissement de notre travail de caractérisation.

Remarque : Le lecteur pourra remarquer dans la suite de ce rapport que les histogrammes comportent parfois des largeurs de colonnes différentes lorsqu'il y a des colonnes aux extremités du graphe. Ce n'est pas un effet volontaire, mais une conséquence de l'utilisation de Gnuplot qui n'a aucune signification particulière.

4.1 Ghostscript

Nombre d'allocations	22891
Nombre d'octets alloués (Mo)	239
Taille moyenne d'une requête (Ko)	10.7
Nombre de libérations (par le GC)	22221
Nombre d'octets libérés (Mo)	236
Nombre d'exécutions du GC	459
Seuil de déclenchement du GC (Ko)	512

TAB. 4.1 – *Résultat global de l'exécution*

Ghostscript est un interpréteur PostScript du domaine public couramment utilisé sur les systèmes Unix. Il permet, typiquement, de visualiser un document PostScript ou de l'imprimer lorsque l'on ne dispose pas d'une imprimante PostScript. La version mesurée est Aladdin Ghostscript 5.03. Il est à noter qu'aucune modification n'a été faite sur le code source de l'application.

Cette application a été retenue car, d'une part, elle fait intensivement appel aux allocations dynamiques et, d'autre part, elle a servi de référence à de nombreuses études sur les politiques d'allocations mémoire [ZG92a, DDZ93]. Le document PostScript utilisé pour les mesures fait 39 pages et contient huit figures. Le tableau 4.1 rassemble des statistiques générales concernant l'exécution.

4.1.1 Durée de vie

Durée de vie	Mo alloués
Durée totale	239
Durée min	0
Durée max	239
Moyenne	6.5

TAB. 4.2 – *Durée de vie des objets en octets alloués pendant leur vie*

Pour les raisons indiquées à la section 3.2.3, la durée de vie est mesurée entre la création des objets et leur libération par le ramasse-miettes. Avec cette méthode, la durée de vie pourrait, en principe, varier avec la politique d'allocation mémoire utilisée, car le ramasse-miettes ne se déclenche plus exactement au même instant. Toutefois, nous n'avons pas constaté de différence significative, aussi nous ne présentons pas les résultats avec chaque politique d'allocation mémoire. Le tableau 4.2 contient des statistiques globales sur la durée de vie des objets et la table 4.1 présente le pourcentage d'occurrence de chaque durée de vie.

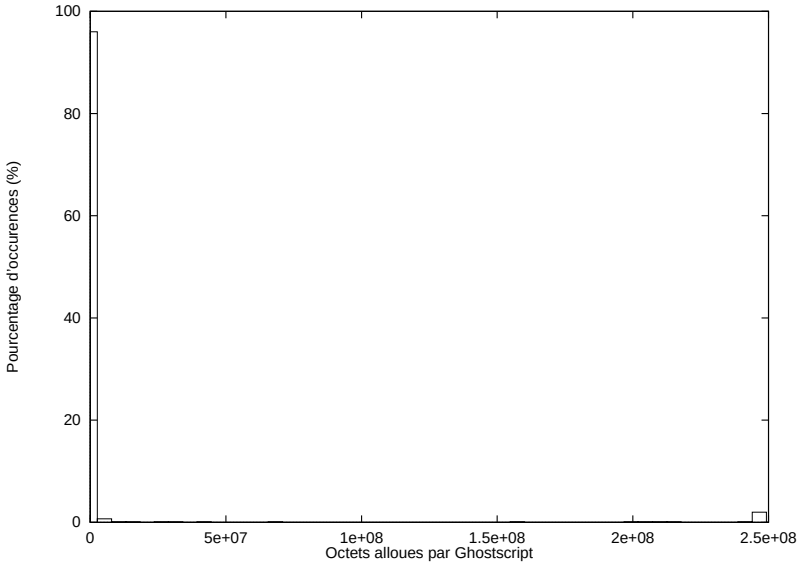

FIG. 4.1 – *Pourcentage d'occurence des durée de vie en octets alloués*

4.1.2 Longueur des références

Politique d'allocation	octets alloués	linéaire	first fit	best fit	roving
Longueur min (octets)	8	56	56	56	120
Longueur max (Mo)	99.8	110	3.3	4.3	5.4
Moyenne (Mo)	11.7	10.4	0.5	0.8	0.8
Référence avant (%)	37.9	36	41.7	43	39.2
Référence arrière (%)	62.1	64	58.3	57	60.8

TAB. 4.3 – *Longueur des références dans un instantané*

Politique d'allocation	octets alloués	linéaire	first fit	best fit	roving
Longueur min (octets)	8	64	64	64	272
Longueur max (Mo)	239	239	4.6	5.1	6.4
Moyenne (Mo)	53.4	59.8	1.5	1.6	1.6
Référence avant (%)	20.1	20.1	30.6	32.9	27.7
Référence arrière (%)	80.1	79.9	69.4	67.1	72.3

TAB. 4.4 – *Longueur des références dans les miettes*

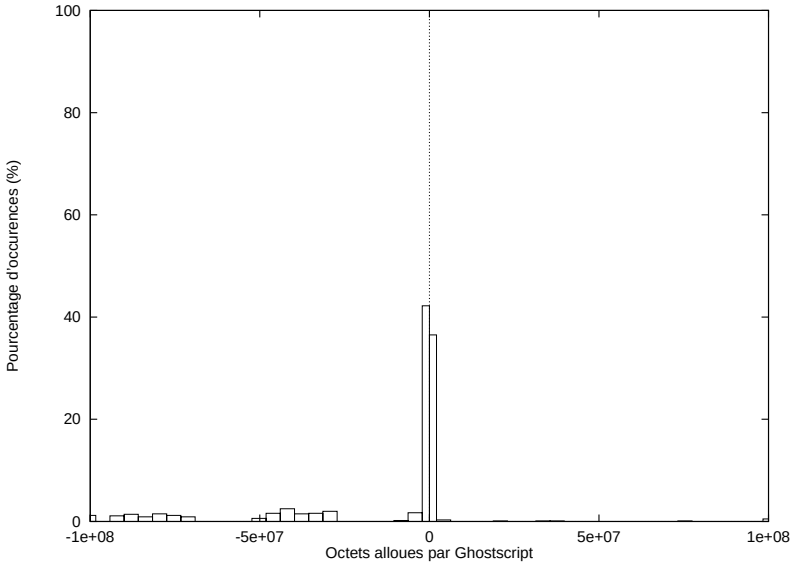

FIG. 4.2 – *Pourcentage d'occurence des longueurs de références en octets alloués dans un instantané*

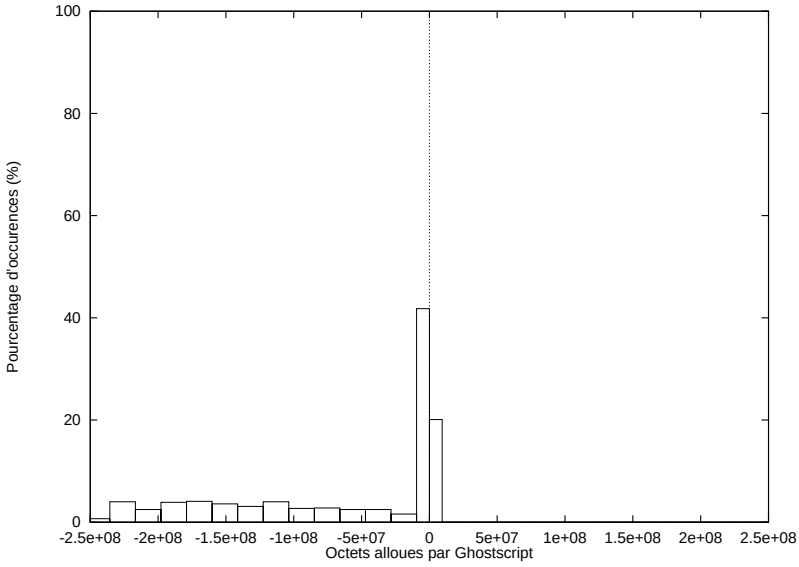

FIG. 4.3 – *Pourcentage d'occurence des longueurs de références en octets alloués dans les miettes*

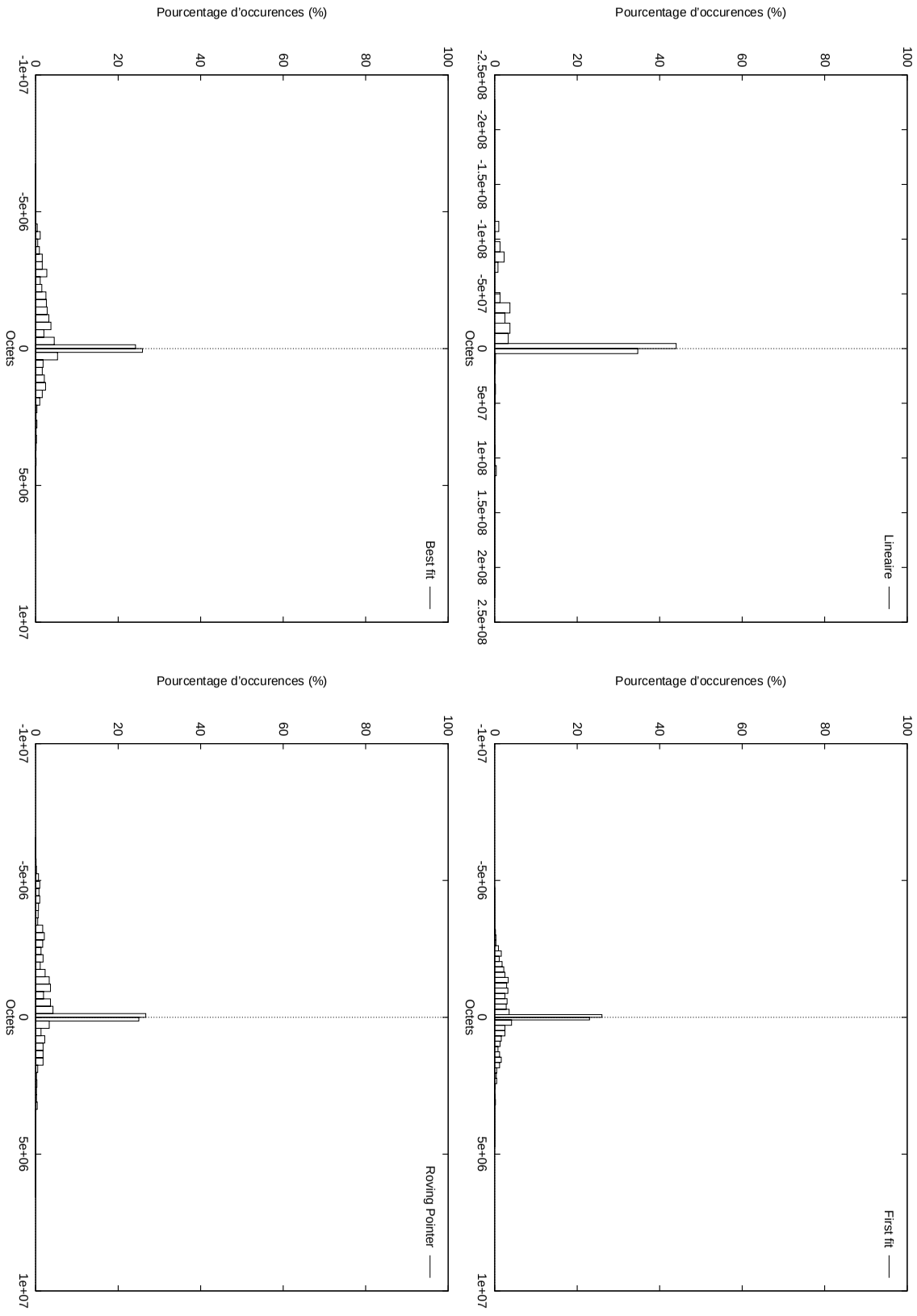

FIG. 4.4 – *Pourcentage d'occurence des longueurs de références en octets dans un instantané en fonction de la politique d'allocation (échelle variable)*

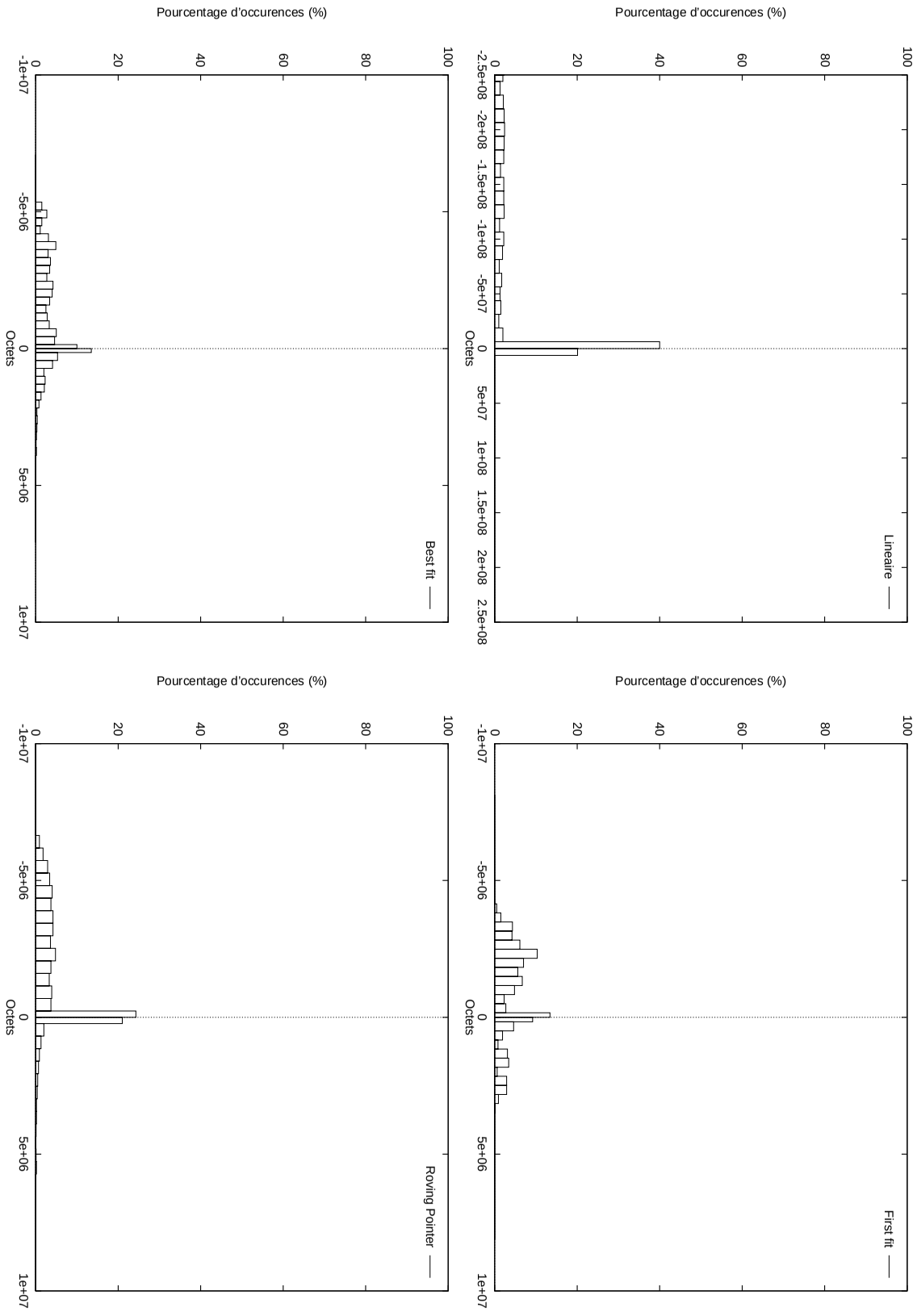

FIG. 4.5 – *Pourcentage d'occurence des longueurs de références en octets dans les miettes en fonction de la politique d'allocation (échelle variable)*

4.1.3 Degrés sortant et incident

Degré	Dans un instantané		Dans les miettes	
	sortant	entrant	sortant	entrant
Degré min	0	0	0	0
Degré max	25	65	46	34
Degré moyen	3	3	4	2

TAB. 4.5 – *Statistiques générales*

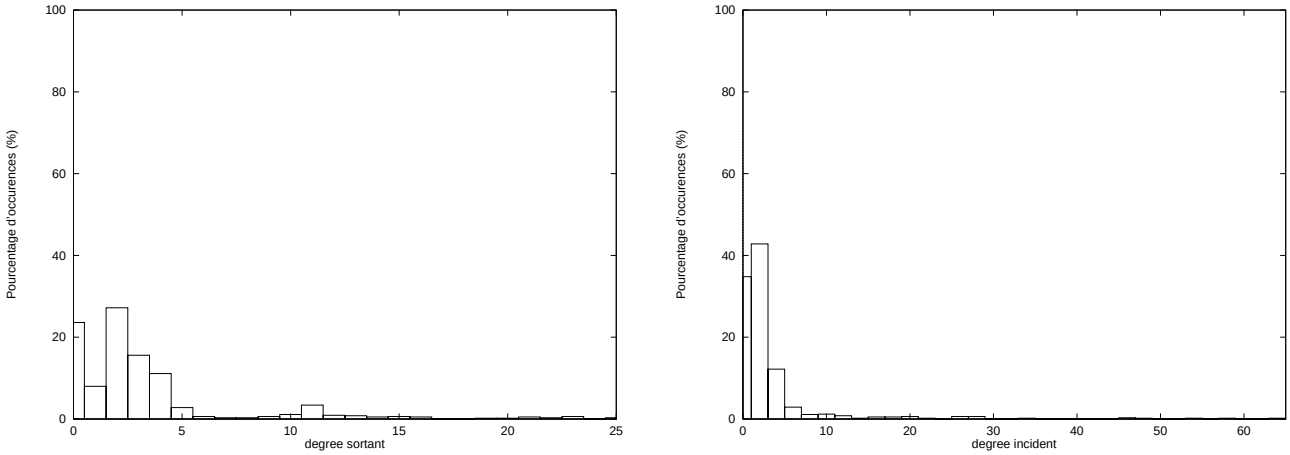

FIG. 4.6 – *Degrés sortant et incident dans un instantané*

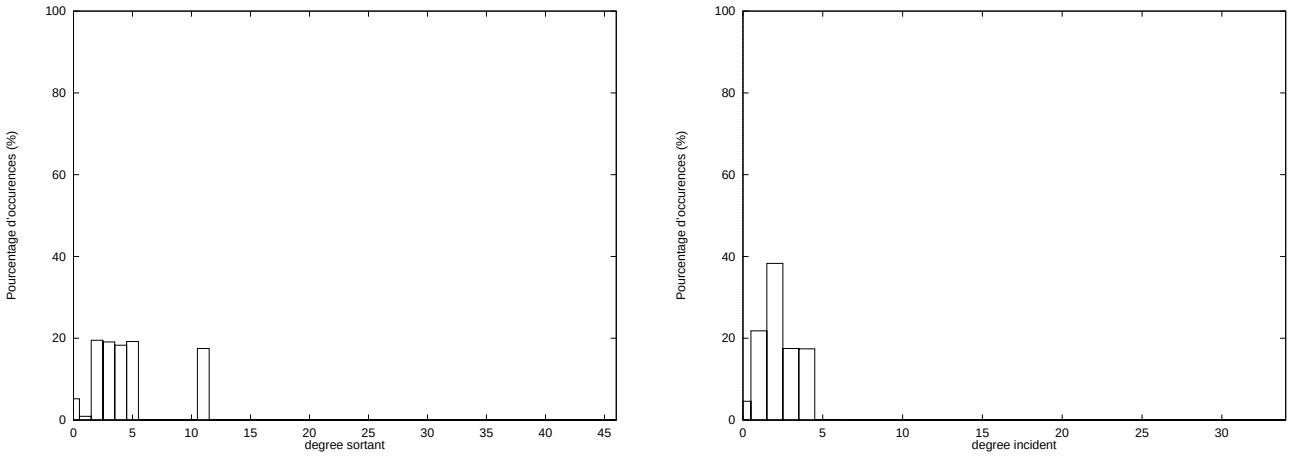

FIG. 4.7 – *Degrés sortant et incident dans les miettes*

4.1.4 Proportion d'objets dans les cycles

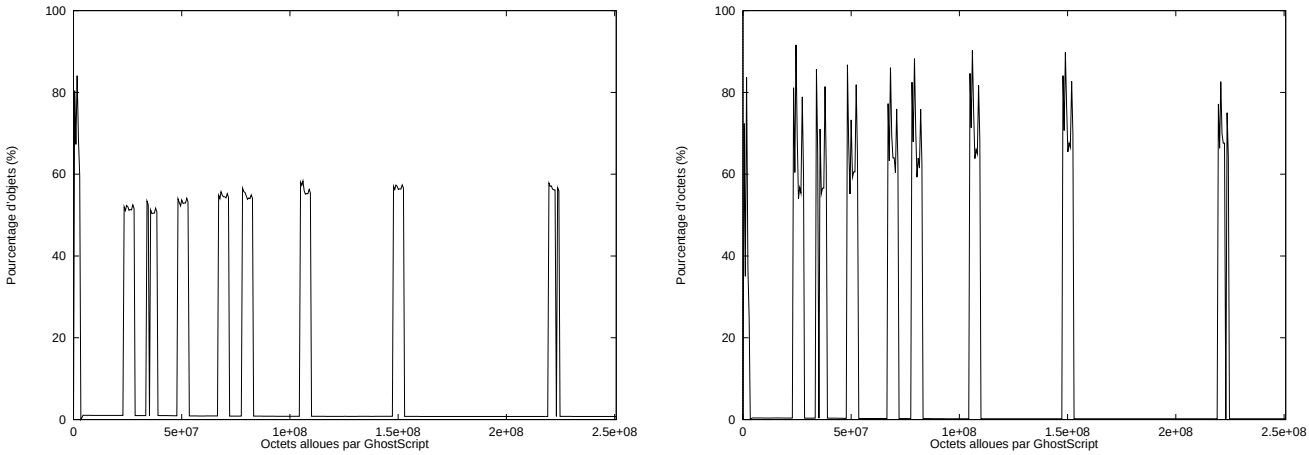

FIG. 4.8 – *Proportion d'objets dans les cycles vivants en fonction du temps*

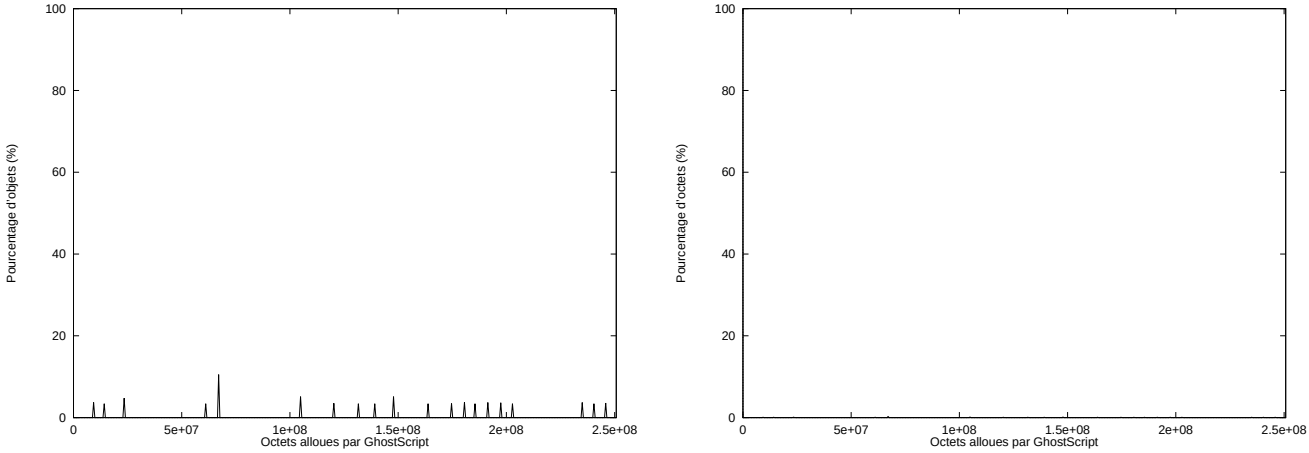

FIG. 4.9 – *Proportion d'objets dans les cycles de miettes en fonction du temps*

4.1.5 Longueur moyennes des cycles

Dans les instantanés :

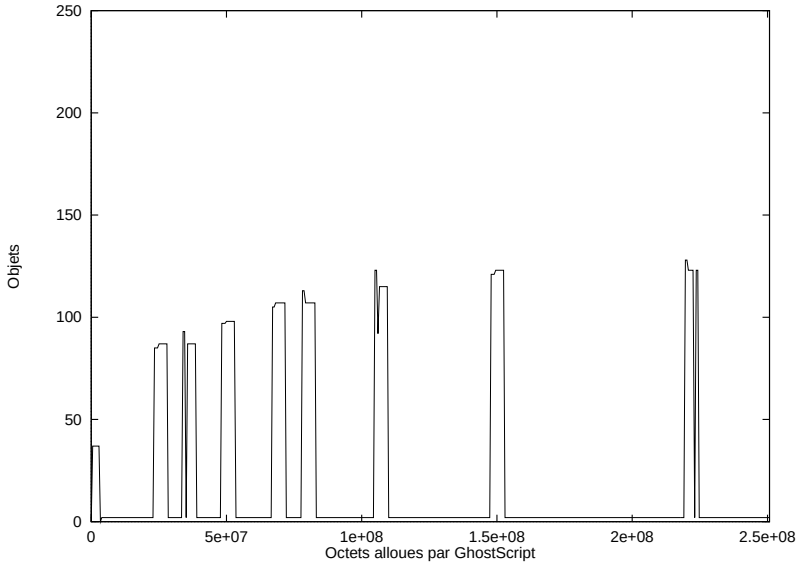

FIG. 4.10 – *Nombre moyen d'objets par cycle en fonction du temps*

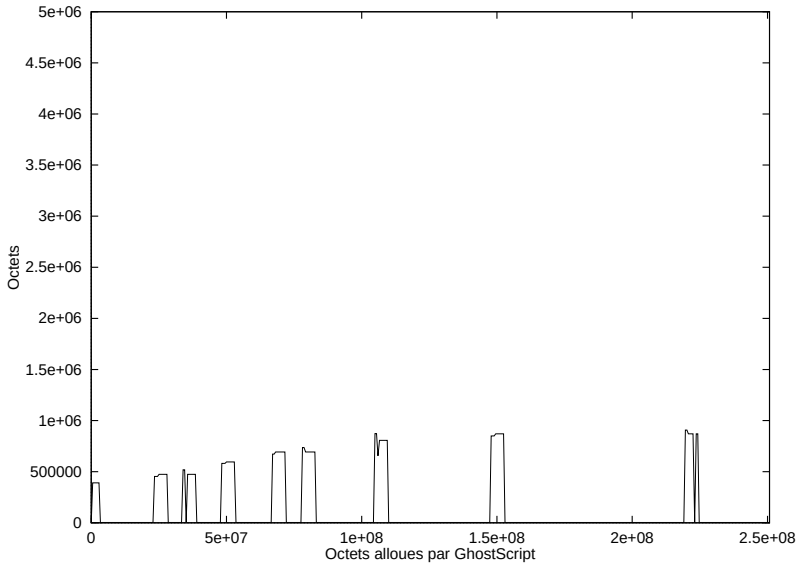

FIG. 4.11 – *Volume moyen des cycles en fonction du temps*

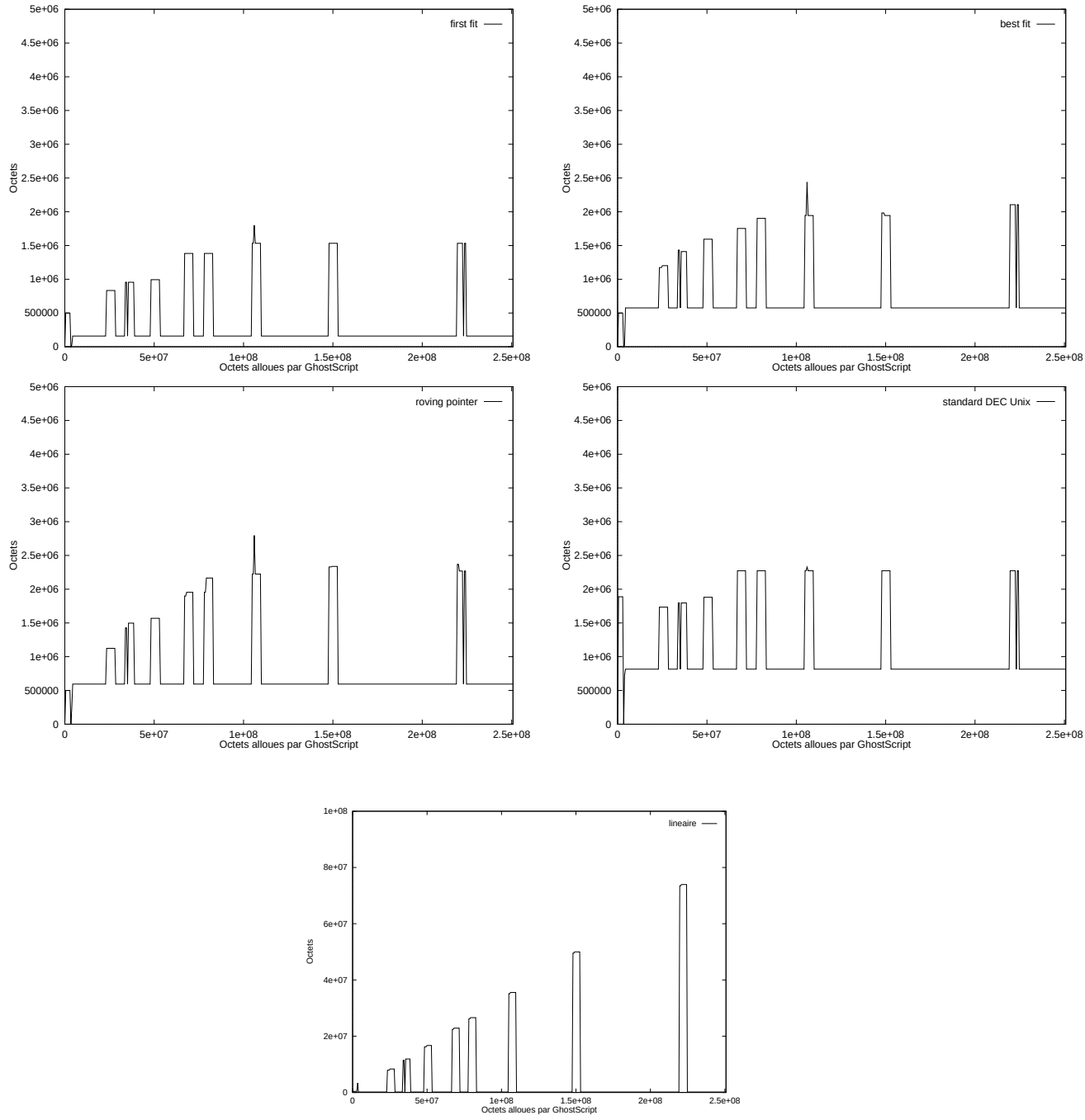

FIG. 4.12 – *Diamètre moyen des cycles en fonction du temps (échelle variable)*

Dans les miettes :

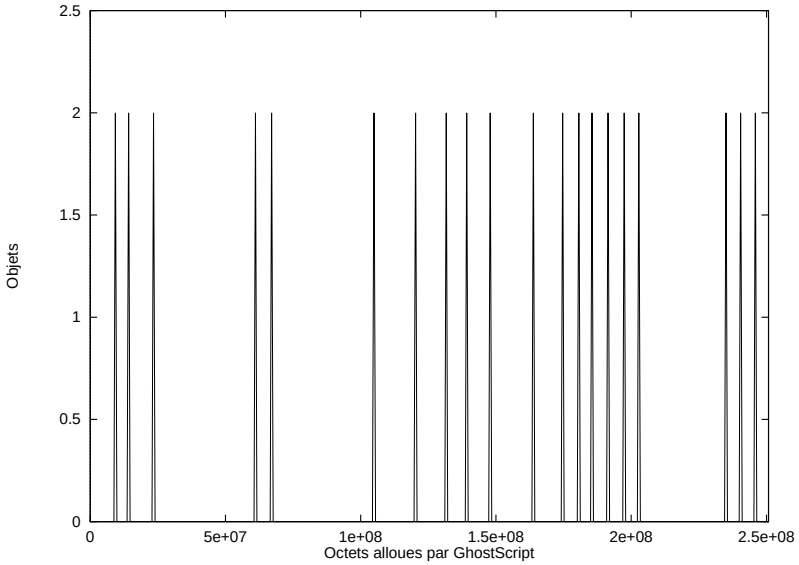

FIG. 4.13 – *Nombre moyen d'objets par cycle en fonction du temps*

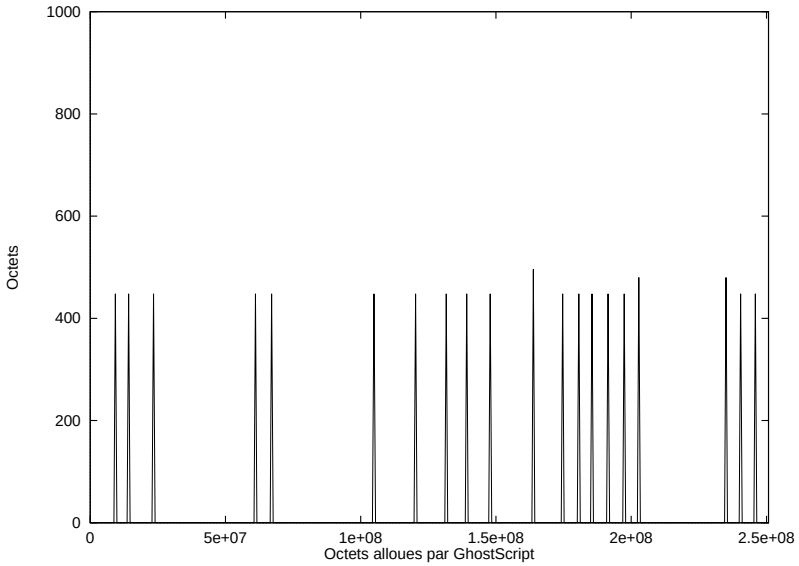

FIG. 4.14 – *Volume moyen des cycles en fonction du temps*

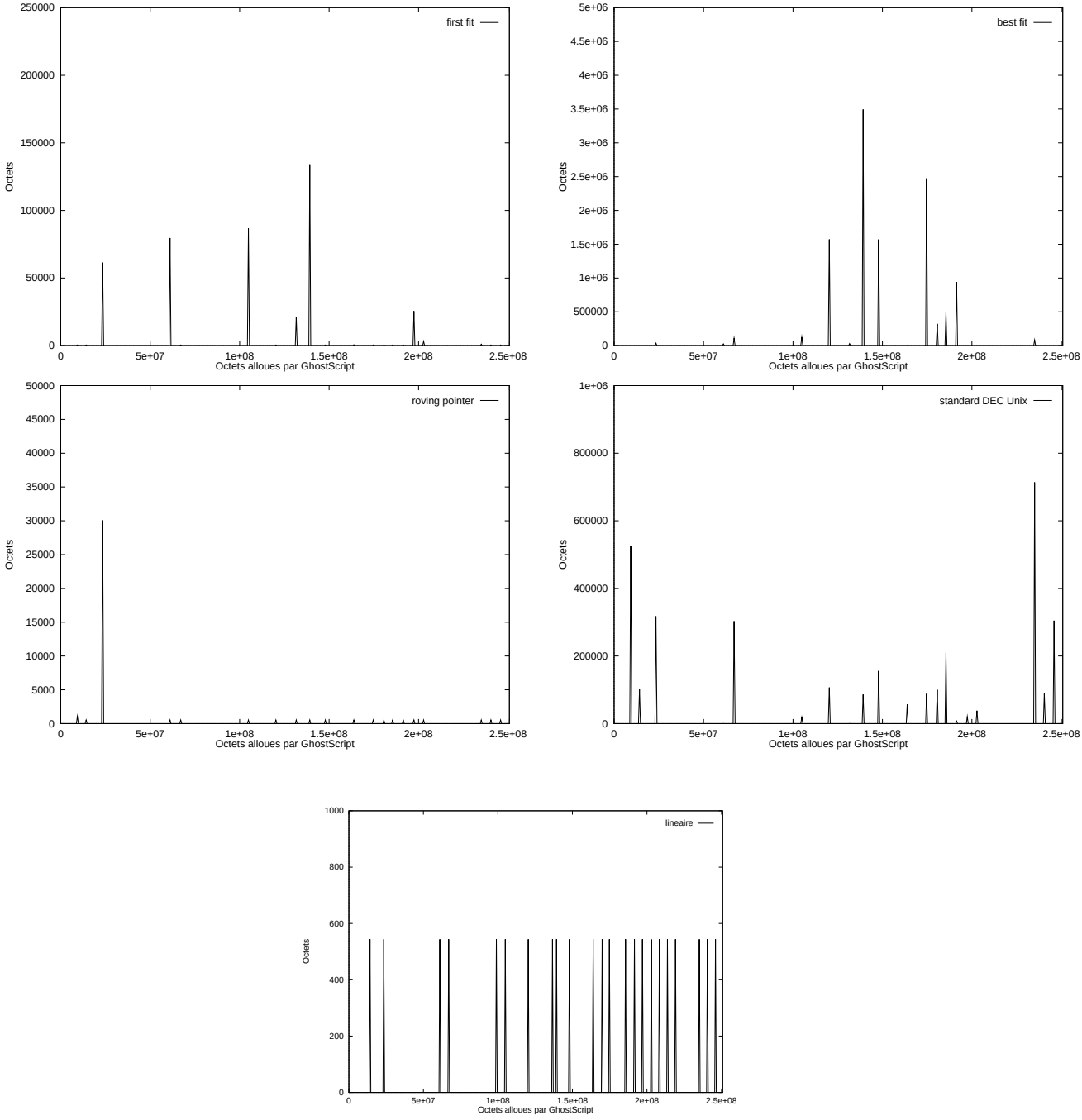

FIG. 4.15 – *Diamètre moyen des cycles en fonction du temps (échelle variable)*

4.1.6 Pourcentage d'occurences des longueurs de cycle

Dans un instantané :

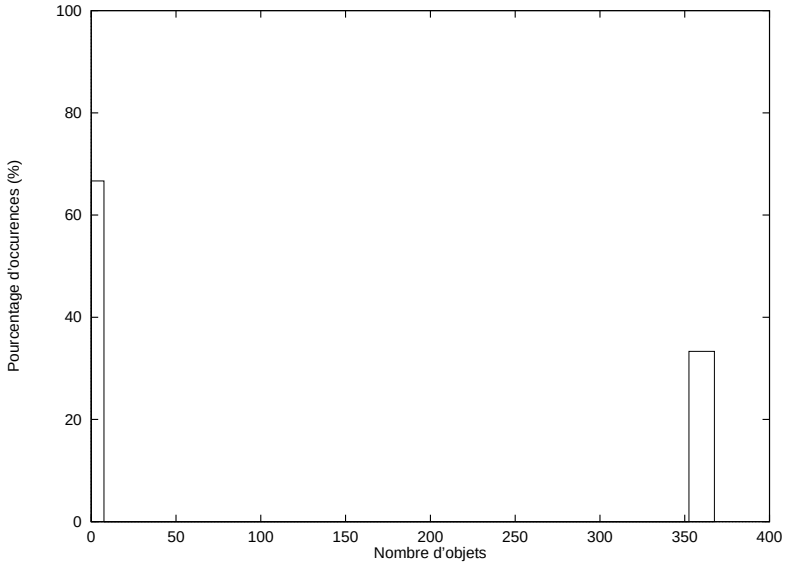

FIG. 4.16 – *Pourcentage d'occurrences des longueurs de cycles (objets)*

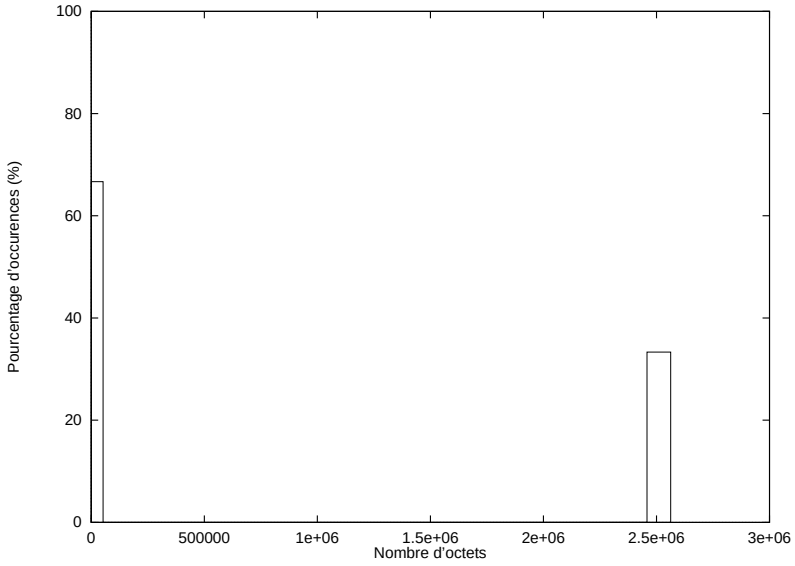

FIG. 4.17 – *Pourcentage d'occurrences des volumes de cycles (octets)*

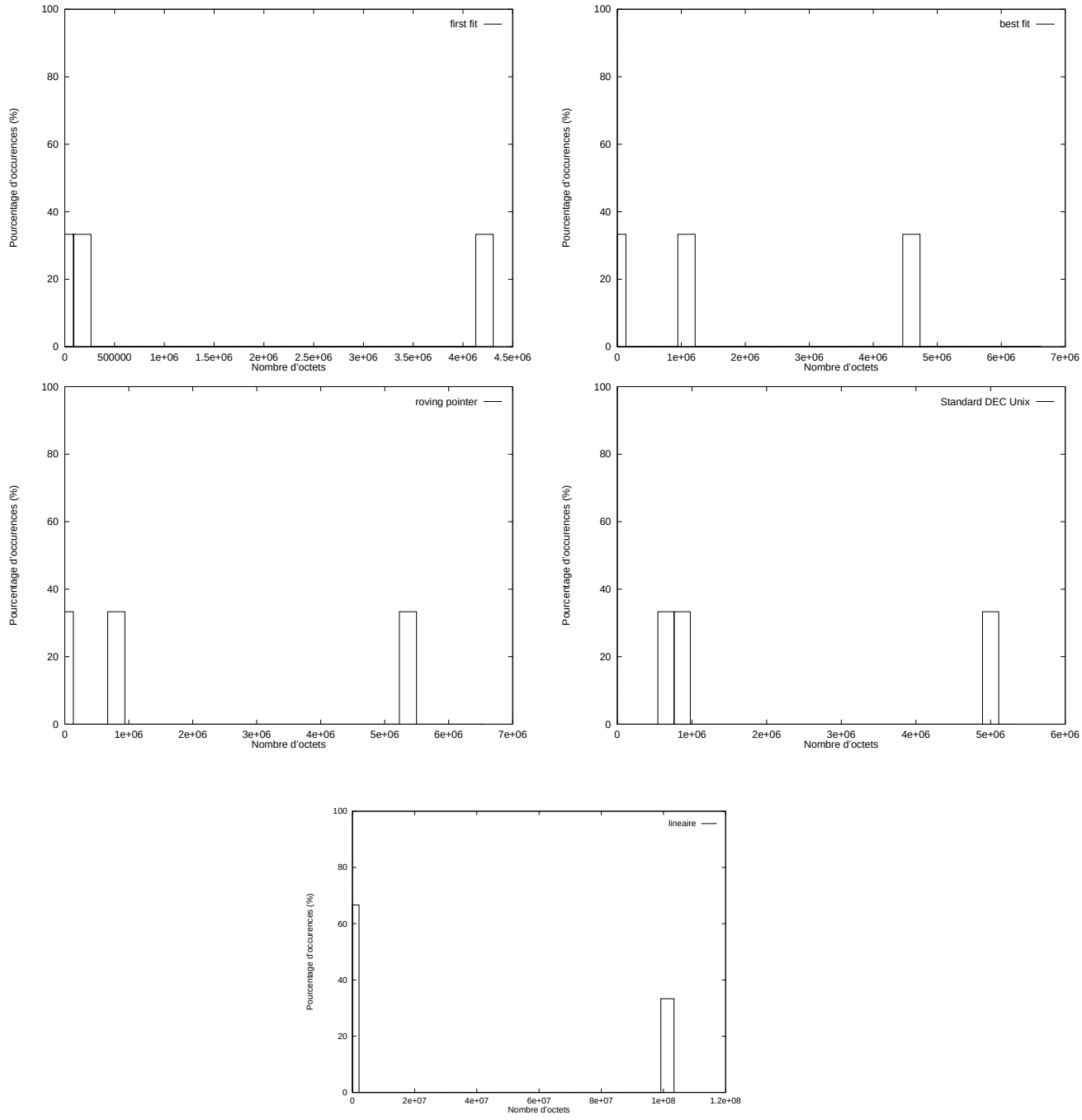

FIG. 4.18 – *Pourcentage d'occurences des diamètres de cycles (échelle variable)*

Dans les miettes :

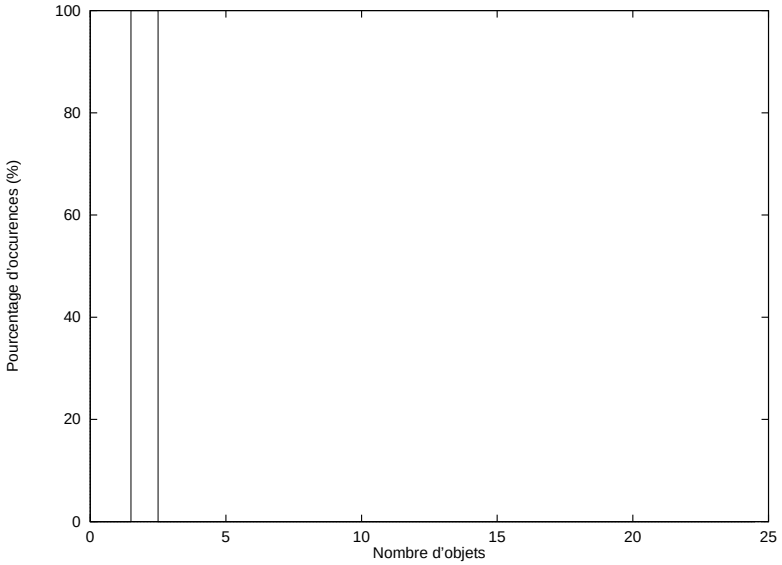

FIG. 4.19 – *Pourcentage d'occurrences des longueurs de cycles (objets)*

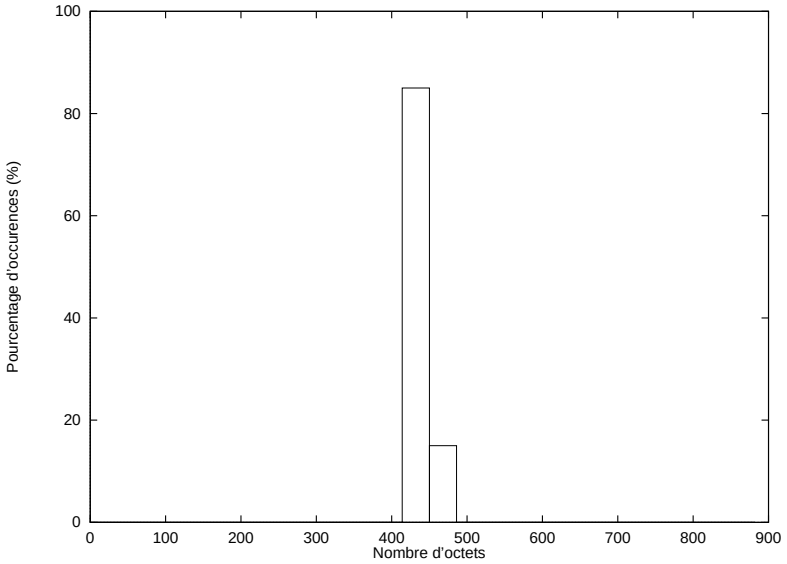

FIG. 4.20 – *Pourcentage d'occurrences des volumes de cycles (octets)*

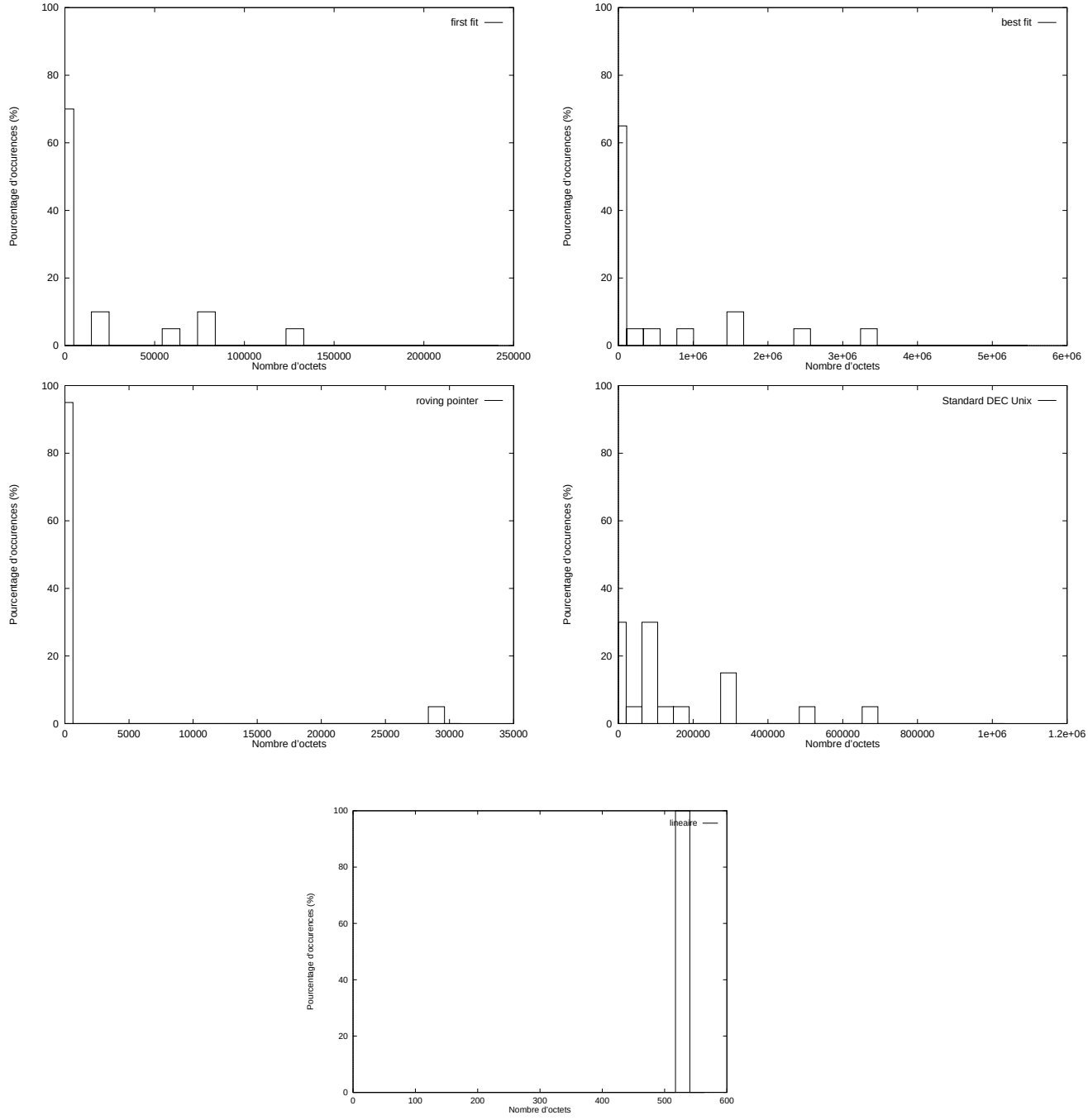

FIG. 4.21 – *Pourcentage d'occurences des diamètres de cycles (échelle variable)*

4.2 XV

Nombre d'allocations	1331
Nombre d'octets alloués (Mo)	57
Requête moyenne (Ko)	44.1
Nombre de libérations (par le GC)	614
Nombre d'octets libérés (Mo)	48
Nombre d'exécutions du GC	30
Seuil de déclenchement du GC (Ko)	512

TAB. 4.6 – *Résultat global de l'exécution*

XV est une application interactive permettant de visualiser, de modifier et de convertir des images dans différents formats. La version mesurée est XV 3.10a, sans aucune modification du code source.

Cette application a été retenue car elle est interactive, ce qui est une caractéristique des futures applications de PerDiS. En outre, elle utilise abondamment l'allocation dynamique. L'exécution a consisté à lire une image de 780 Ko au format GIF, à appliquer plusieurs transformations dessus puis à sauver l'image résultat au format JPEG. Le tableau 4.6 rassemble des statistiques générales concernant l'exécution.

Pour cette application, les mesures n'ont pas révélées la présence de cycle dans les miettes, aussi les résultats concernant les cycles ne portent que sur les objets vivants.

4.2.1 Durée de vie

Durée de vie	Mo alloués
Durée totale	57
Durée min	0
Durée max	57
Moyenne	27

TAB. 4.7 – *Durée de vie des objets en octets alloués pendant leur vie*

Comme pour Ghostscript, la durée de vie est mesurée entre la création des objets et leur libération par le ramasse-miettes afin de recueillir des résultats significatifs lorsqu'une gestion automatique de la mémoire est mise en œuvre. Le tableau 4.7 contient des statistiques globales sur la durée de vie des objets et la figure 4.22 présente le pourcentage d'occurrence de chaque durée de vie.

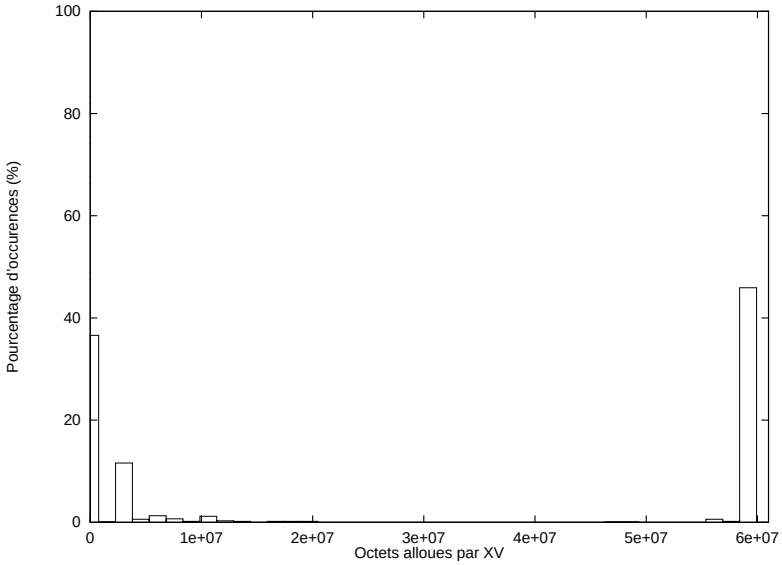

FIG. 4.22 – *Pourcentage d'occurrence des durées de vie en octets alloués*

4.2.2 Longueur des références

Politique d'allocation	octets alloués	linéaire	first fit	best fit	roving
Longueur min (octets)	8	56	56	56	64
Longueur max (Mo)	45.7	45	6.8	10.6	16.8
Moyenne (Ko)	245	270	27.7	37.6	86.9
Référence avant (%)	48.6	48	50.2	51.3	48.4
Référence arrière (%)	51.4	52	49.8	48.7	51.6

TAB. 4.8 – *Longueur des références dans un instantané*

Politique d'allocation	octets alloués	linéaire	first fit	best fit	roving
Longueur min (octets)	16	80	80	64	56
Longueur max (Ko)	85	113	125	114	17304
Moyenne (Ko)	3.5	5.9	16	11	856
Référence avant (%)	28.6	28.4	27.1	17.8	26.8
Référence arrière (%)	71.4	71.6	72.9	82.2	73.2

TAB. 4.9 – *Longueur des références dans les miettes*

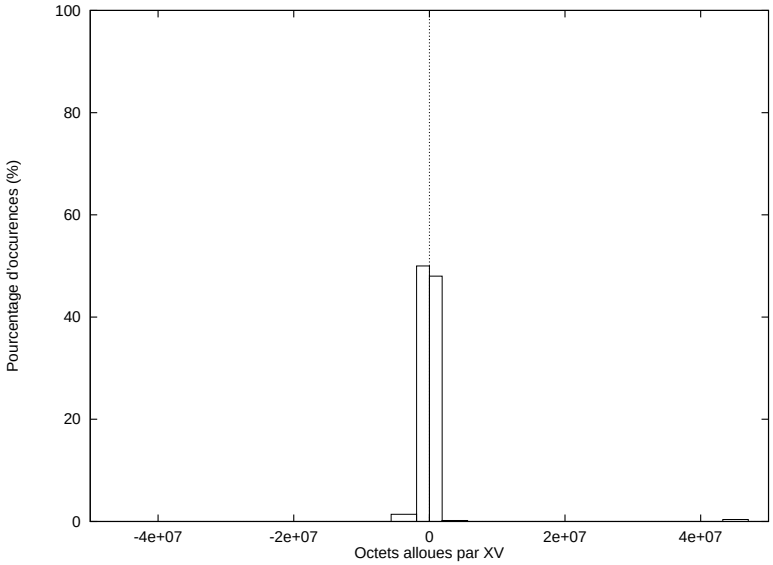

FIG. 4.23 – *Pourcentage d'occurence des longueurs de références en octets alloués dans un instantané*

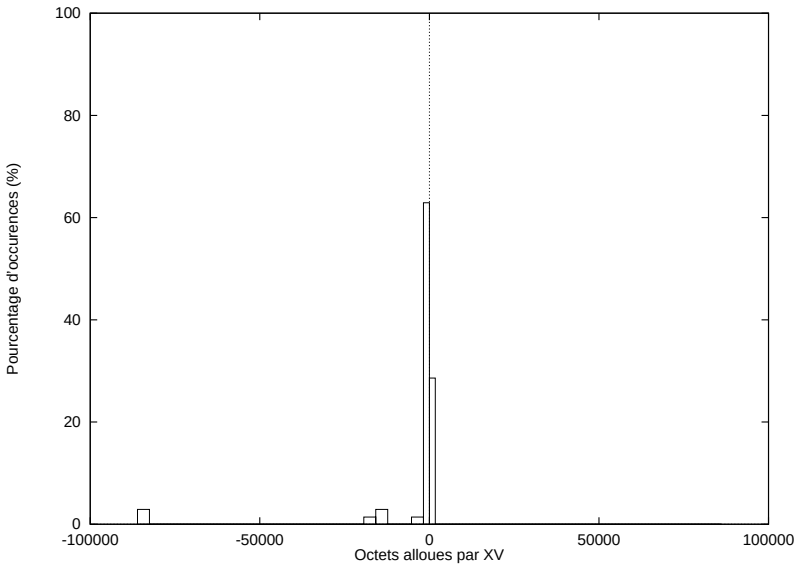

FIG. 4.24 – *Pourcentage d'occurence des longueurs de références en octets alloués dans les miettes*

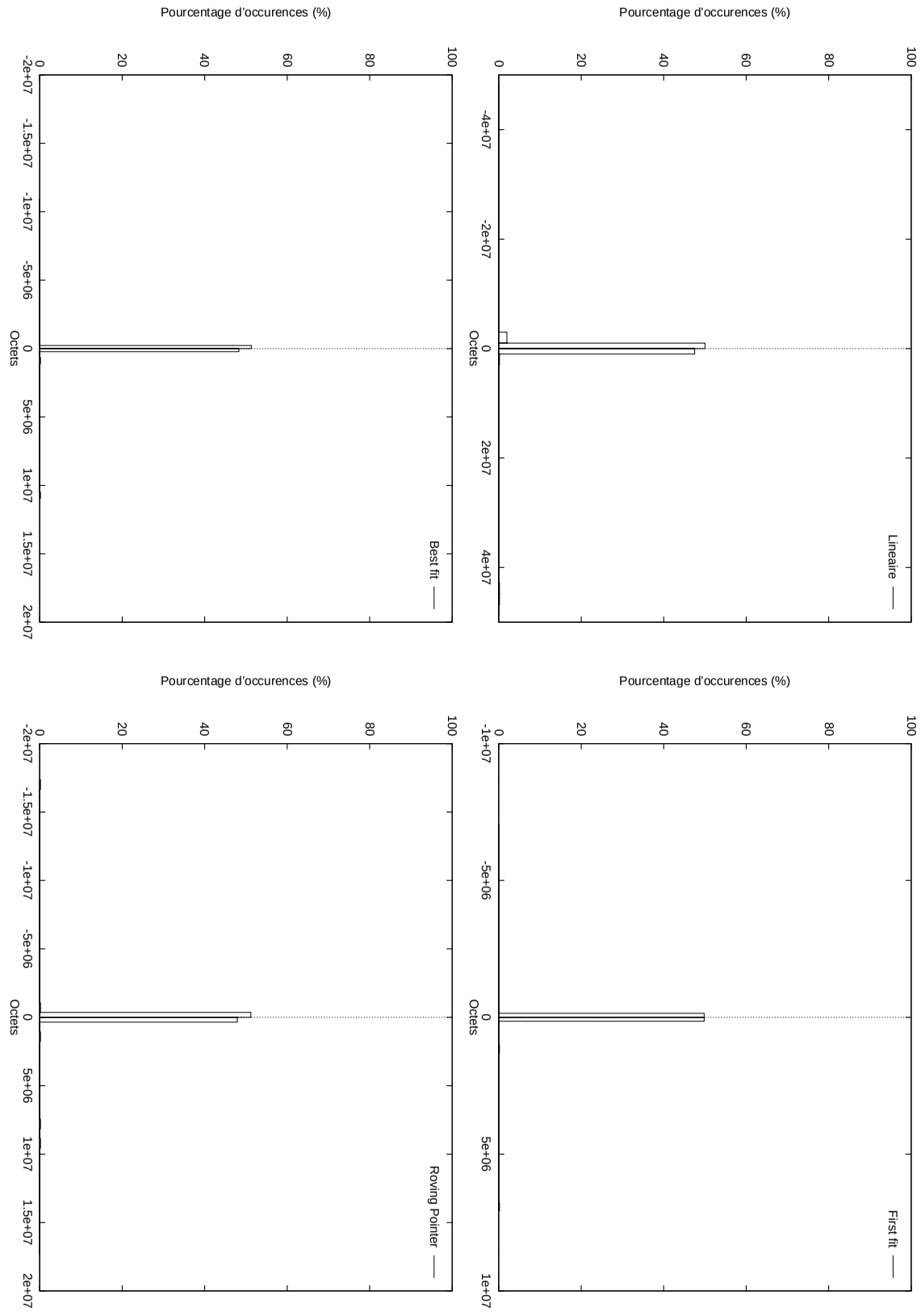

FIG. 4.25 – *Pourcentage d'occurence des longueurs de références en octets dans un instantané (échelle variable)*

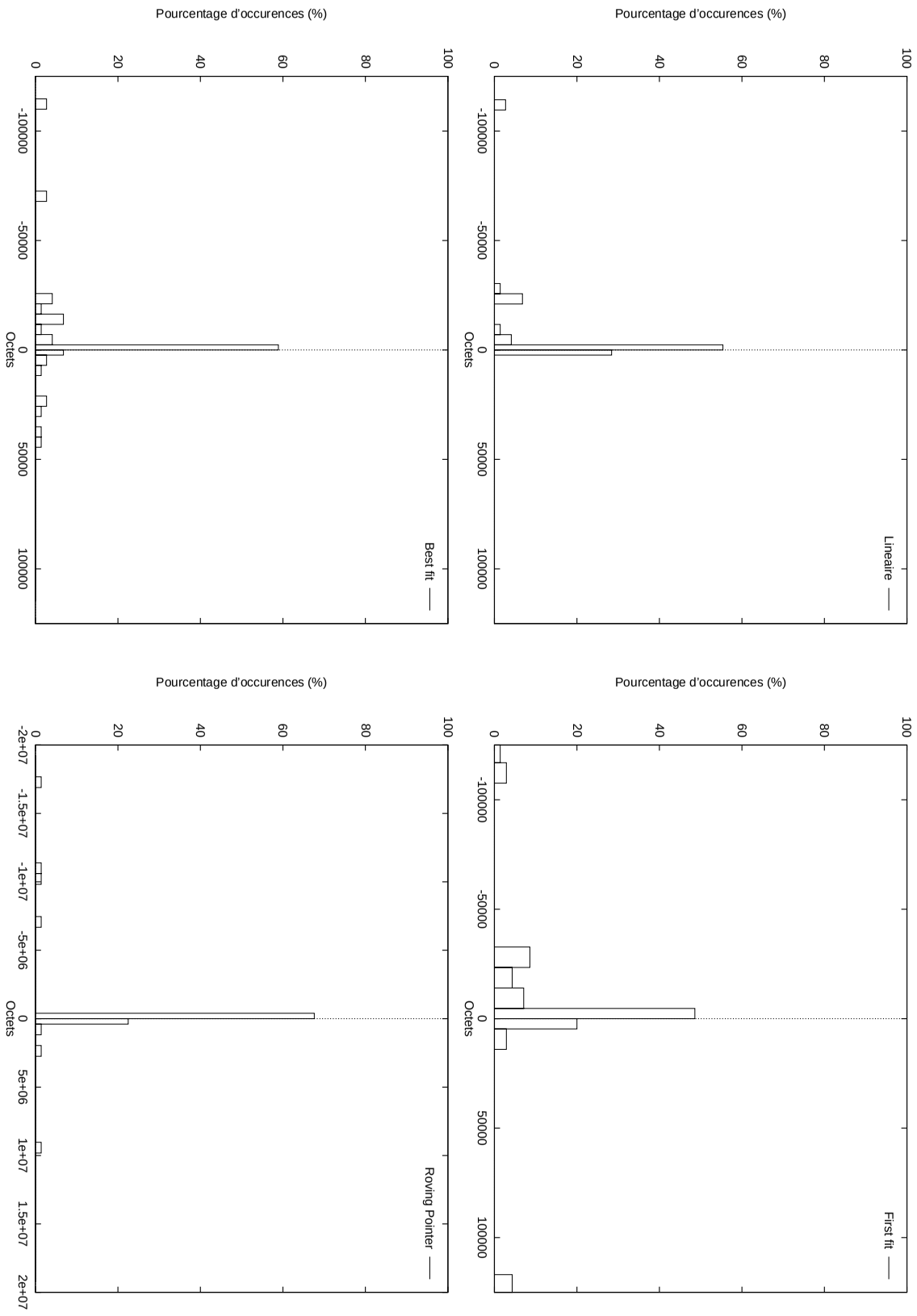

FIG. 4.26 – *Pourcentage d'occurence des longueurs de références en octets dans les miettes (échelle variable)*

4.2.3 Degrés sortant et incident

	Dans un instantané		Dans les miettes	
Degré	sortant	entrant	sortant	entrant
Degré min	0	0	0	0
Degré max	19	27	4	1
Degré moyen	0.4	0.2	0.08	0.01

TAB. 4.10 – *Statistiques générales*

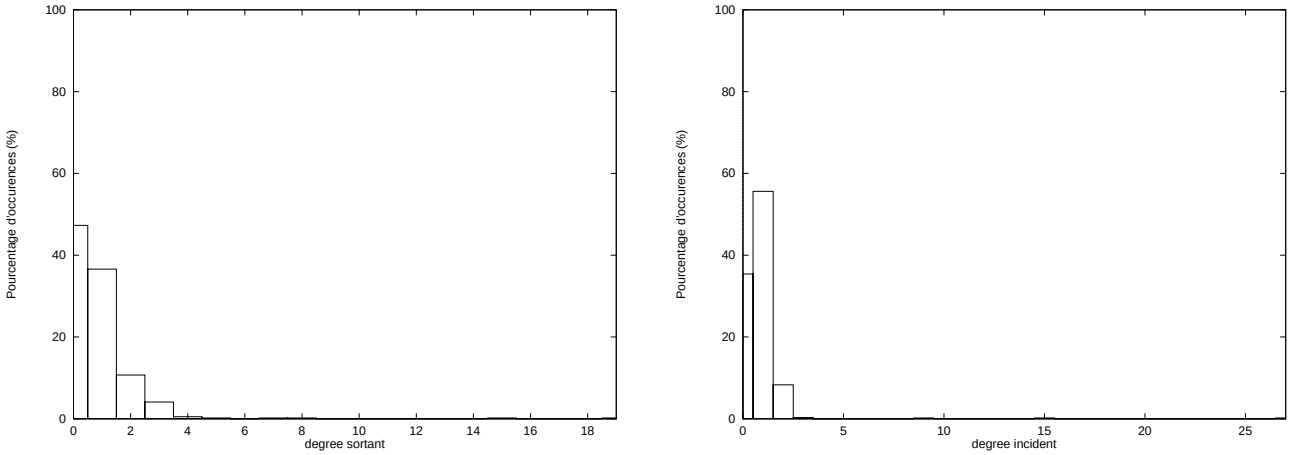

FIG. 4.27 – *Degrés sortant et incident dans un instantané*

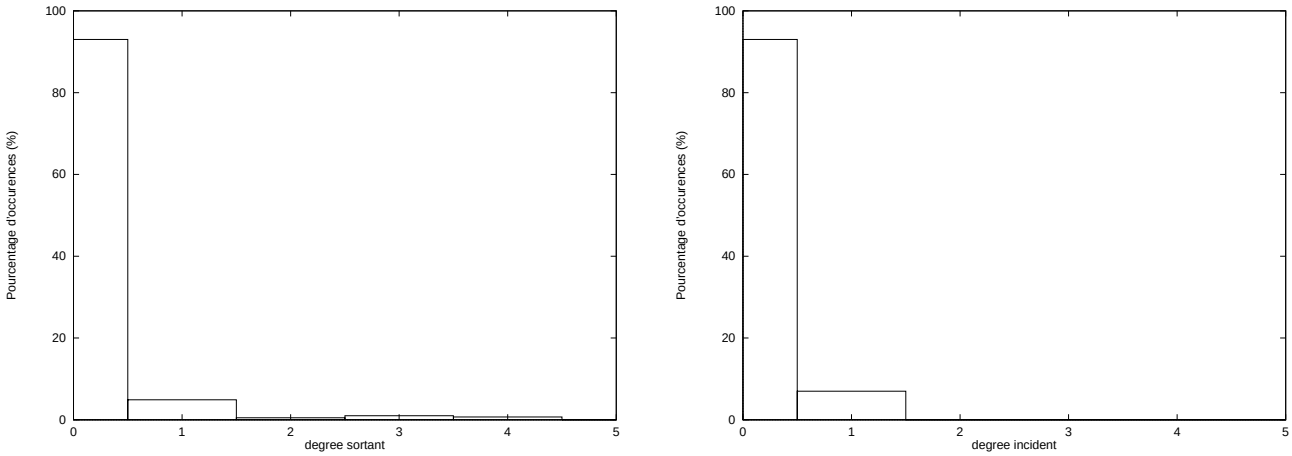

FIG. 4.28 – *Degrés sortant et incident dans les miettes*

4.2.4 Proportion d'objets dans les cycles

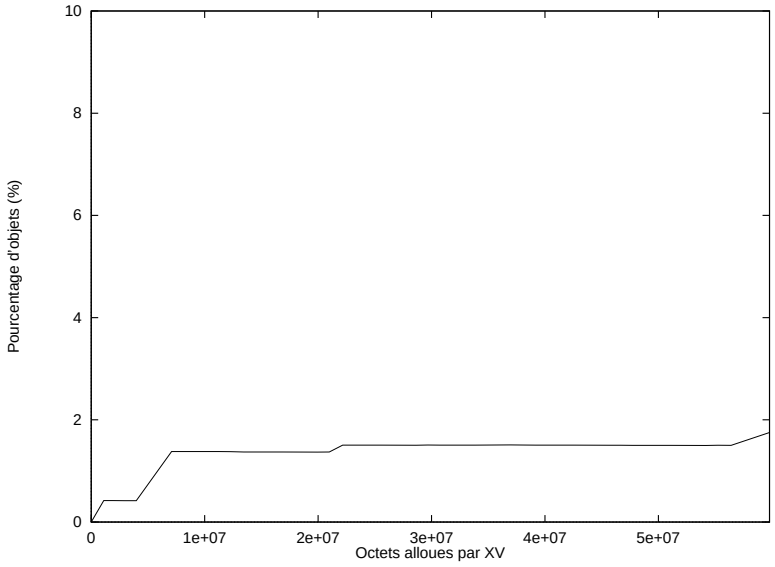

FIG. 4.29 – *Proportion d'objets dans les cycles vivants en fonction du temps*

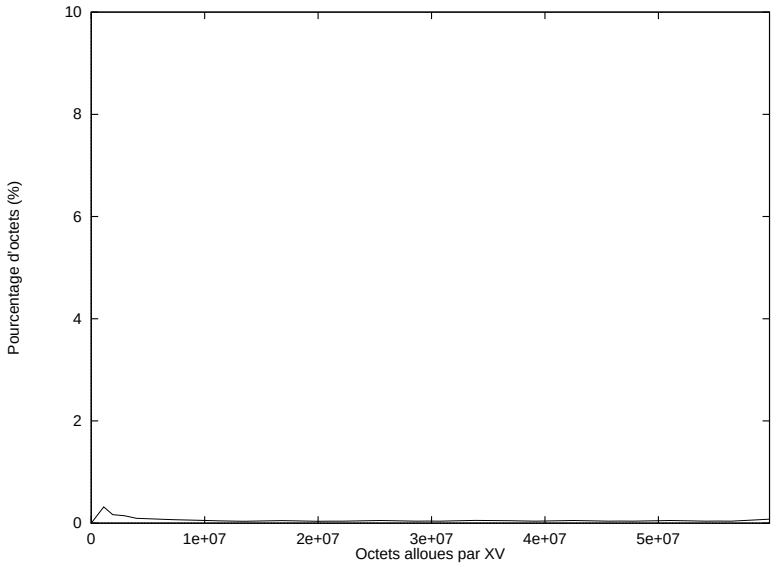

FIG. 4.30 – *Proportion d'octets dans les cycles vivants en fonction du temps*

4.2.5 Longueur moyennes des cycles

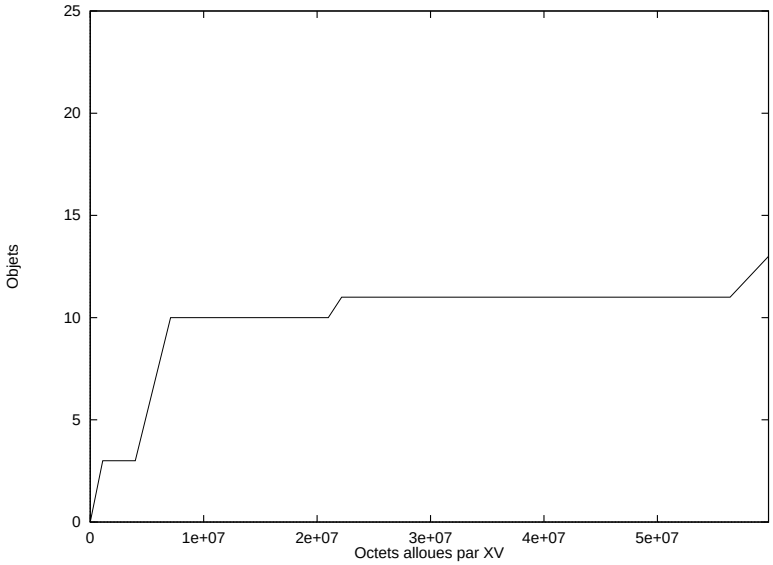

FIG. 4.31 – *Nombre moyen d'objets par cycle en fonction du temps*

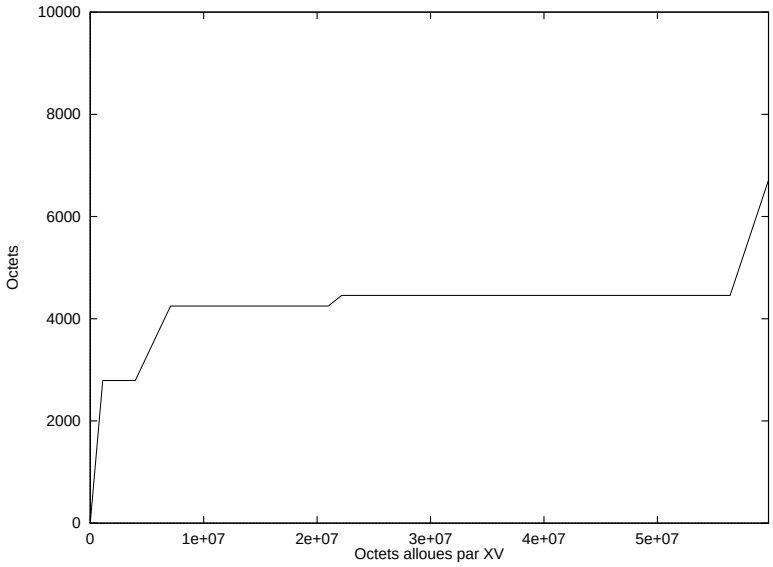

FIG. 4.32 – *Volume moyen des cycles en fonction du temps*

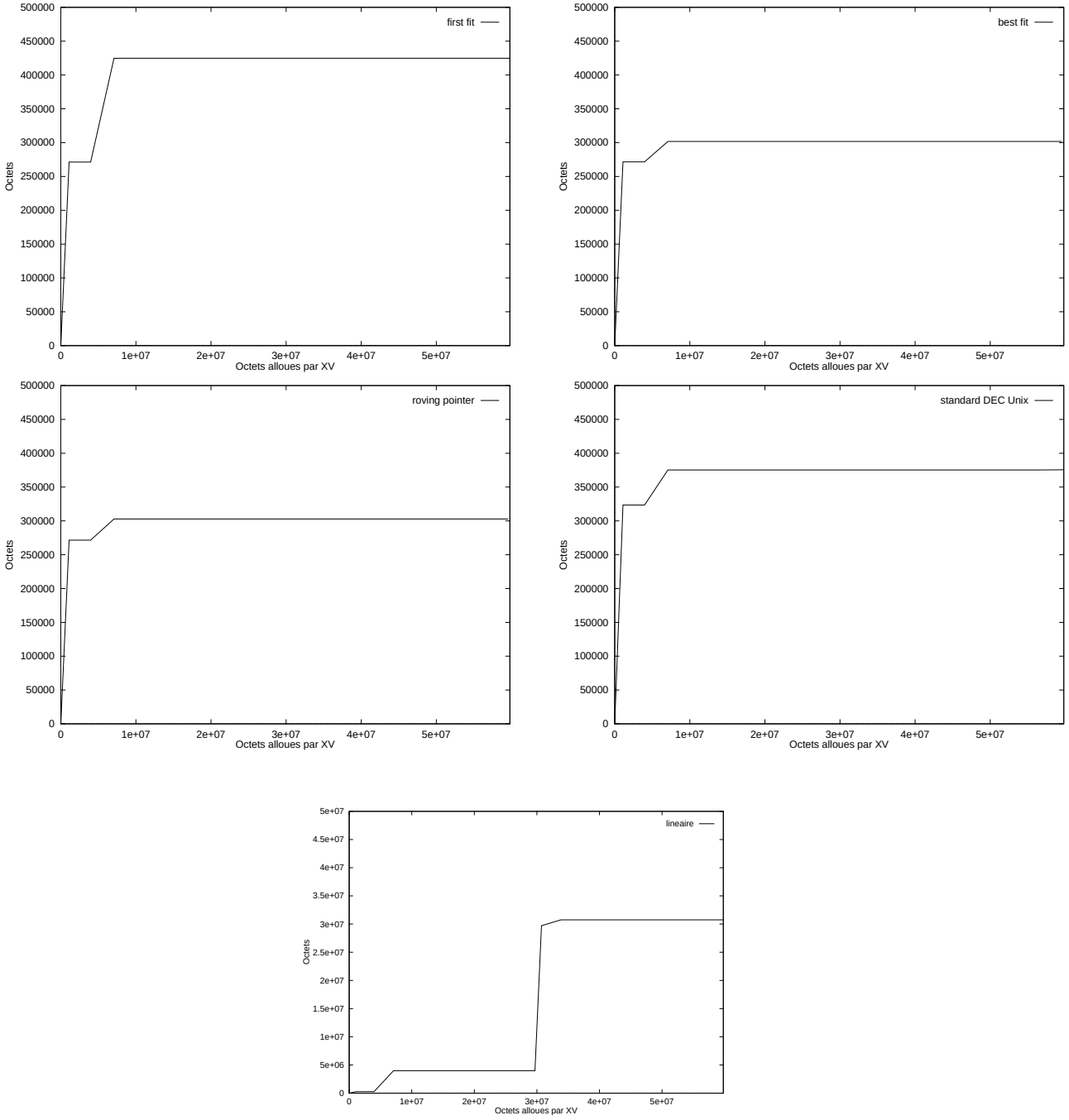

FIG. 4.33 – *Diamètre moyen des cycles en fonction du temps*

4.2.6 Caractéristiques détaillées des cycles

Pour l'application XV, il n'y a jamais plus d'un cycle dans les instantanés. Le tableau 4.11 contient les caractéristiques du cycle pour l'instantané correspondant au 25ème déclenchement du ramasse-miettes.

Caractéristiques	Linéaire	First fit	Best fit	Roving Pointer	Std Unix
Nb objets	11	11	11	11	11
Volume (octets)	4456	4456	4456	4456	4456
Diamètre (Ko)	30030	414	294	295	366

TAB. 4.11 – *Caractéristiques du cycle*

4.3 OO7

Nombre d'allocations	104648
Nombre d'octets alloués (Mo)	5.7
Requête moyenne (octets)	57
Nombre de libérations (par le GC)	496
Nombre d'octets libérés (Ko)	39
Nombre d'exécutions du GC	11
Seuil de déclenchement du GC (Ko)	512

TAB. 4.12 – *Résultat global de l'exécution*

OO7 est un banc d'essais couramment utilisé pour évaluer les bases de données objet. Il a été retenu, bien que ce ne soit pas réellement une application, car il permet la construction d'un graphe d'objets persistants. Le programme mesuré est une version modifiée afin de construire le graphe d'objets en mémoire au lieu d'utiliser une base de donnée.

Les mesures sur OO7 ont été très difficiles à réaliser en raison du très grand nombre d'objets présent dans le tas. La plus petite base de donnée que OO7 propose de simuler est la seule que nous ayons pu mesurer. Cela a nécessité plusieurs heures d'exécution pour recueillir la trace et plusieurs jours d'analyse sur les plus puissantes machines ALPHA dont nous disposions. Le tableau 4.12 rassemble des statistiques générales concernant l'exécution.

Pour cette application, les mesures n'ont pas révélé la présence de cycle dans les miettes (il n'y a que très peu de miettes). En conséquence, toutes les caractéristiques de cycle présentées ne concerne que les objets vivants.

4.3.1 Durée de vie

Durée de vie	Mo alloués
Durée totale	5.6
Durée min	0
Durée max	5.6
Moyenne	2.8

TAB. 4.13 – *Durée de vie des objets en octets alloués pendant leur vie*

Comme pour Ghostscript, la durée de vie est mesurée entre la création des objets et leur libération par le ramasse-miettes afin de recueillir des résultats significatifs lorsqu'une gestion automatique de la mémoire est mise en œuvre. Le tableau 4.13 contient des statistiques globales sur la durée de vie des objets et la figure 4.34 présente le pourcentage d'occurrence de chaque durée de vie.

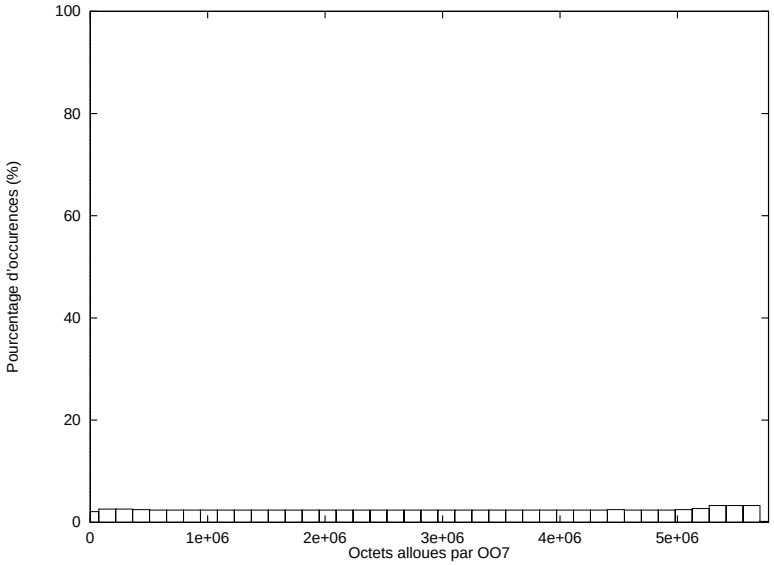

FIG. 4.34 – *Pourcentage d'occurrence des durées de vie en octets alloués*

4.3.2 Longueur des références

Politique d'allocation	octets alloués	linéaire	first fit	best fit	roving
Longueur min (octets)	40	88	80	80	80
Longueur max (Mo)	4.5	8.3	8.5	8.5	8.5
Moyenne (Ko)	166	308	313	313	313
Référence avant (%)	53.0	53.4	53.0	53.0	52.9
Référence arrière (%)	47.0	46.6	47.0	47.0	47.1

TAB. 4.14 – *Longueur des références dans un instantané*

Politique d'allocation	octets alloués	linéaire	first fit	best fit	roving
Longueur min (octets)	112	128	128	128	128
Longueur max (Ko)	4.3	5.6	872	872	763
Moyenne (Ko)	1.4	2.5	5.5	5.3	4.84
Référence avant (%)	100	100	100	100	87.8
Référence arrière (%)	0	0	0	0	12.2

TAB. 4.15 – *Longueur des références dans les miettes*

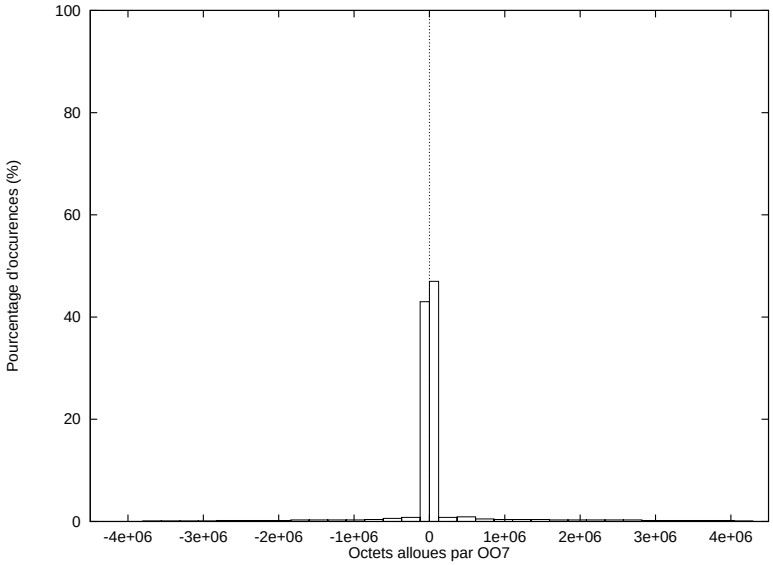

FIG. 4.35 – *Pourcentage d'occurence des longueurs de références en octets alloués dans un instantané*

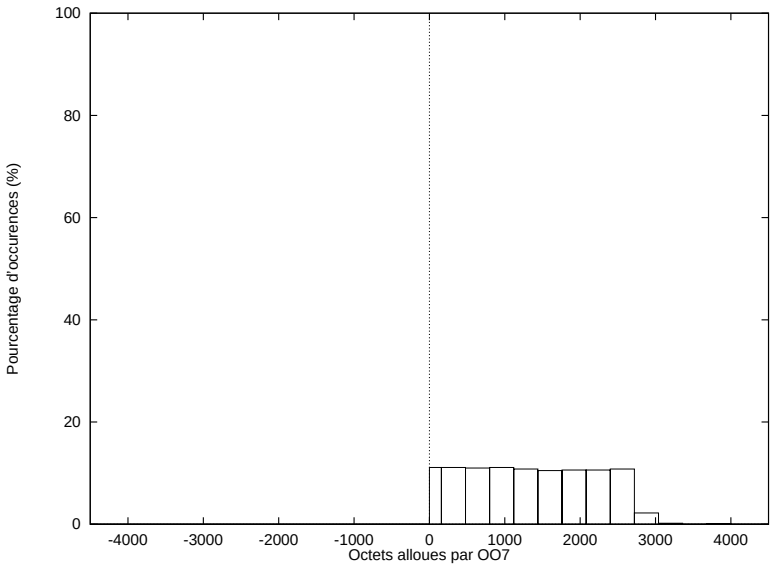

FIG. 4.36 – *Pourcentage d'occurence des longueurs de références en octets alloués dans les miettes*

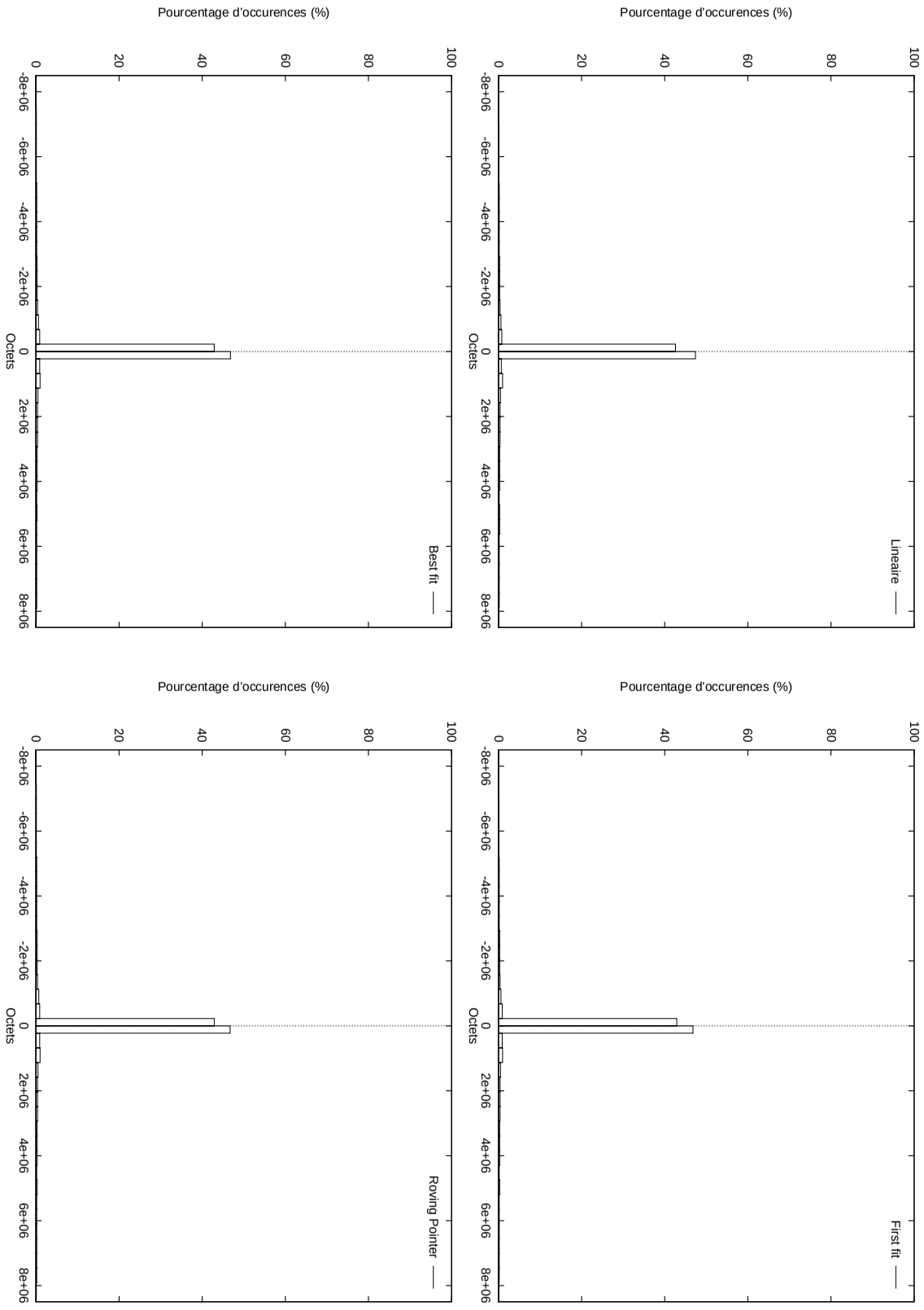

FIG. 4.37 – *Pourcentage d'occurence des longueurs de références en octets dans un instantané en fonction de la politique d'allocation (échelle variable)*

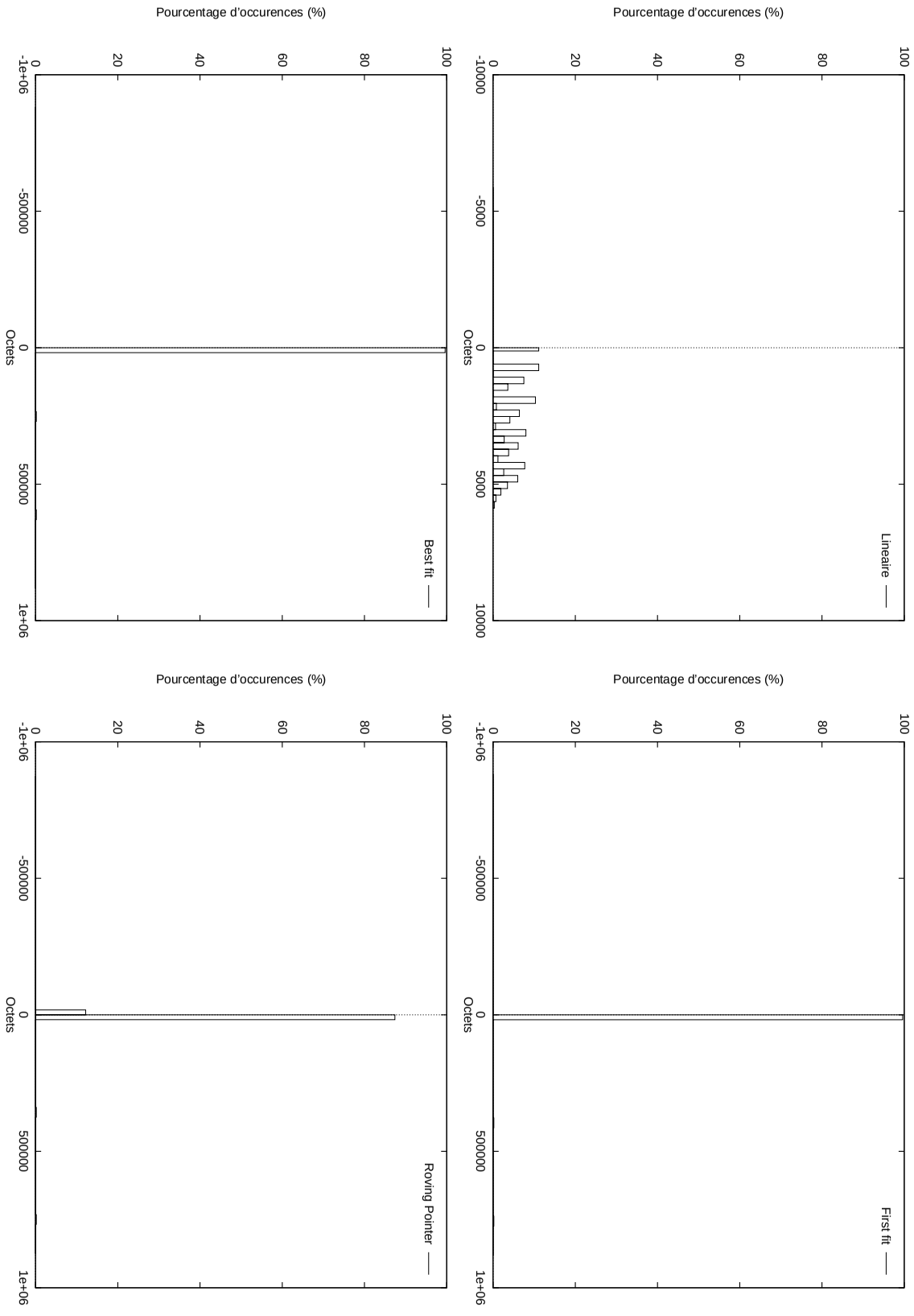

FIG. 4.38 – *Pourcentage d'occurence des longueurs de références en octets dans les miettes en fonction de la politique d'allocation (échelle variable)*

4.3.3 Degrés sortant et incident

Degré	Dans un instantané		Dans les miettes	
	sortant	entrant	sortant	entrant
Degré min	0	0	9	0
Degré max	494	1094	10	0
Degré moyen	1	1	9	0

TAB. 4.16 – *Statistiques générales*

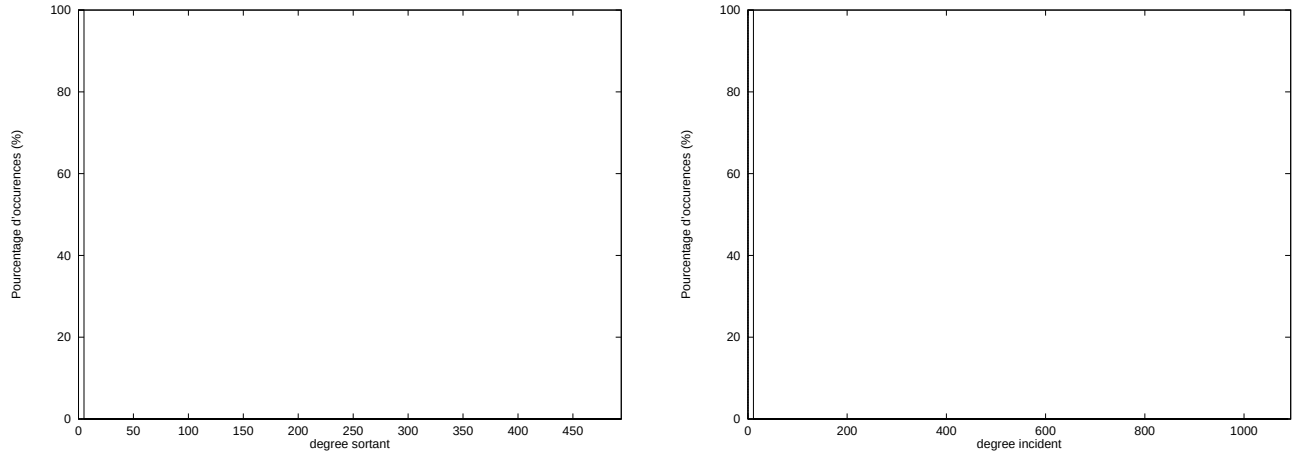

FIG. 4.39 – *Degrés sortant et incident dans un instantané*

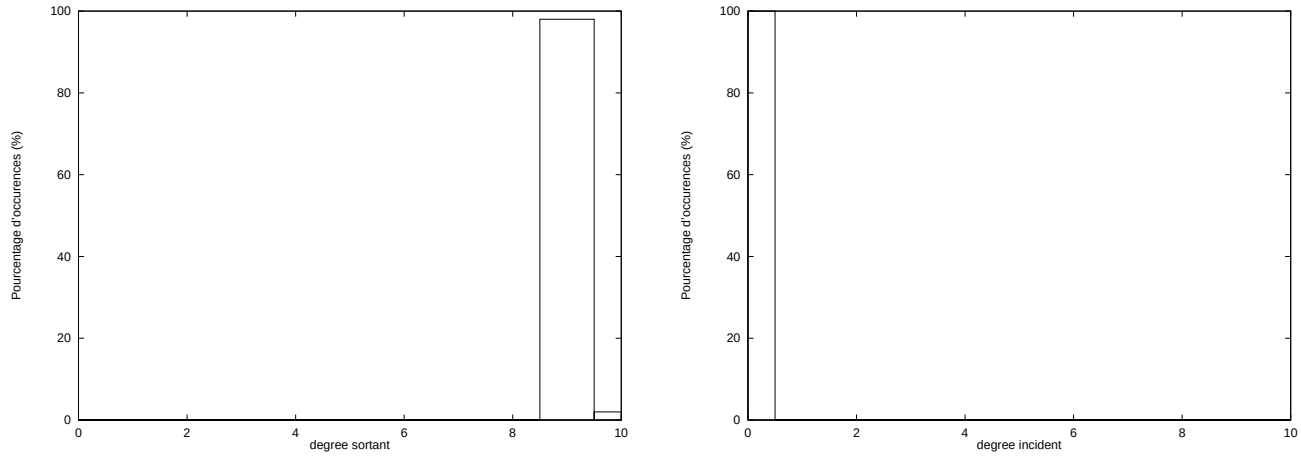

FIG. 4.40 – *Degrés sortant et incident dans les miettes*

4.3.4 Proportion d'objets dans les cycles

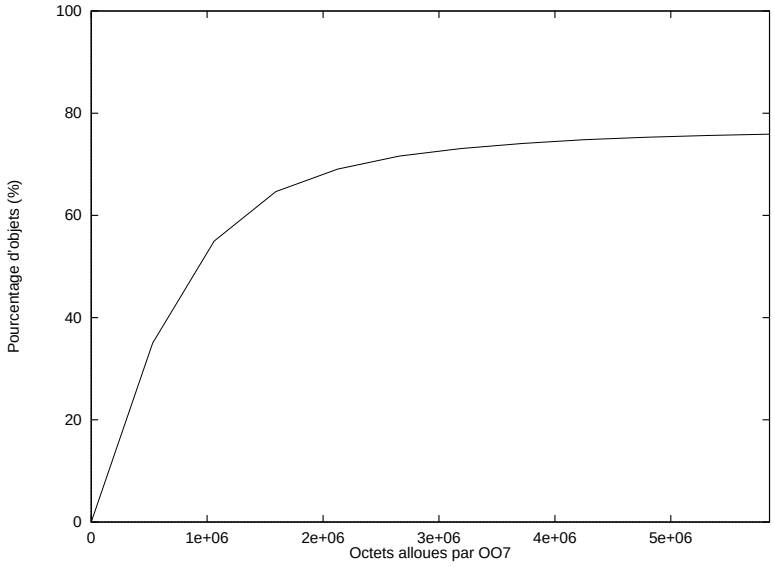

FIG. 4.41 – *Proportion d'objets vivants dans les cycles en fonction du temps*

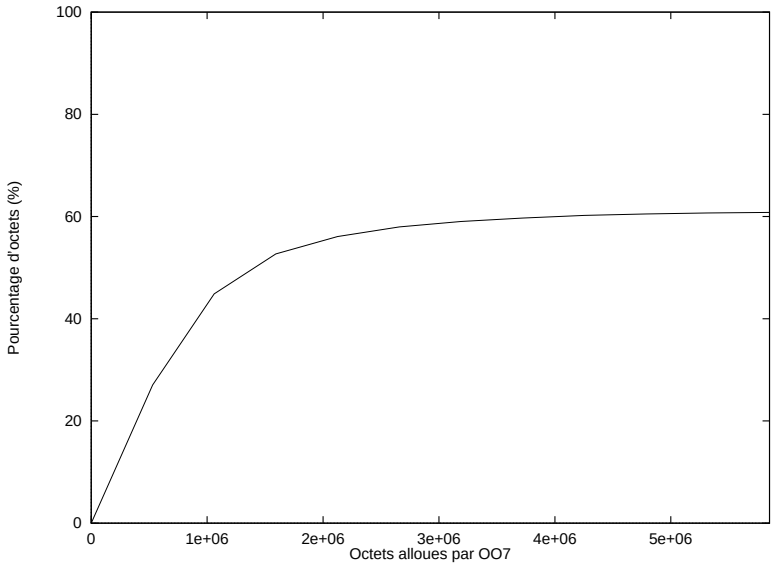

FIG. 4.42 – *Proportion d'octets dans les cycles vivants en fonction du temps*

4.3.5 Longueur moyennes des cycles

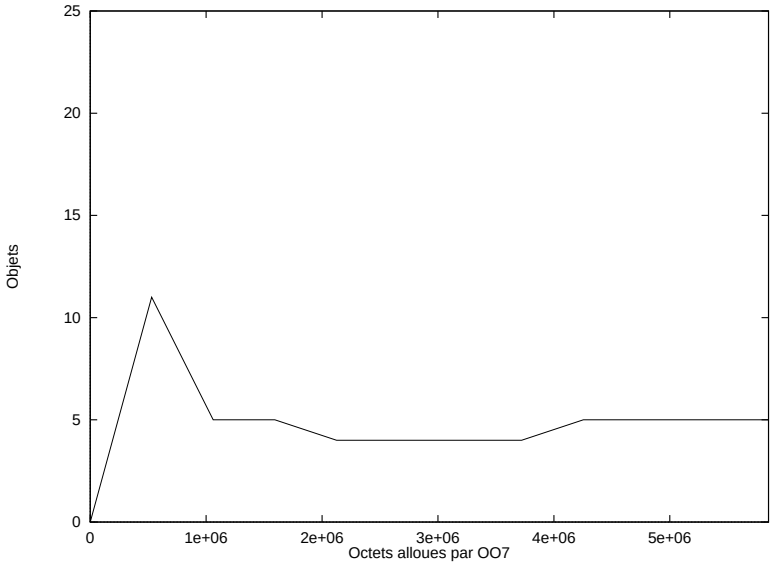

FIG. 4.43 – *Nombre moyen d'objets par cycle en fonction du temps*

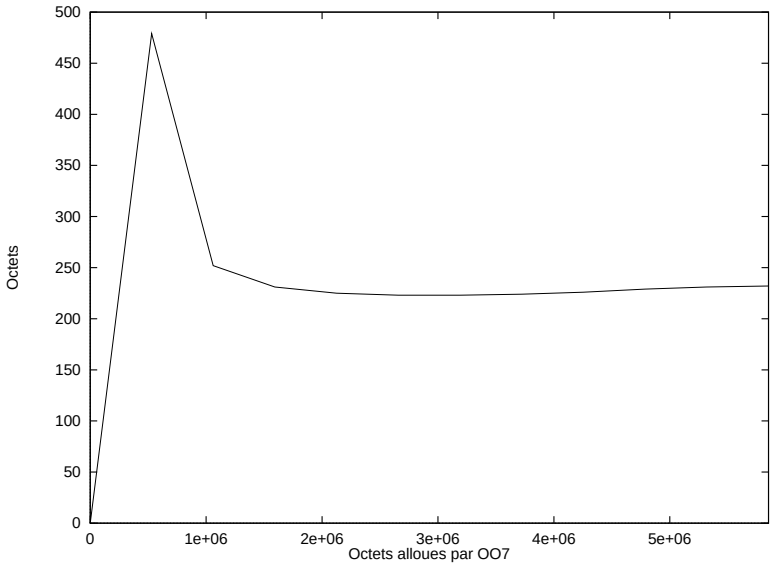

FIG. 4.44 – *Volume moyen des cycles en fonction du temps*

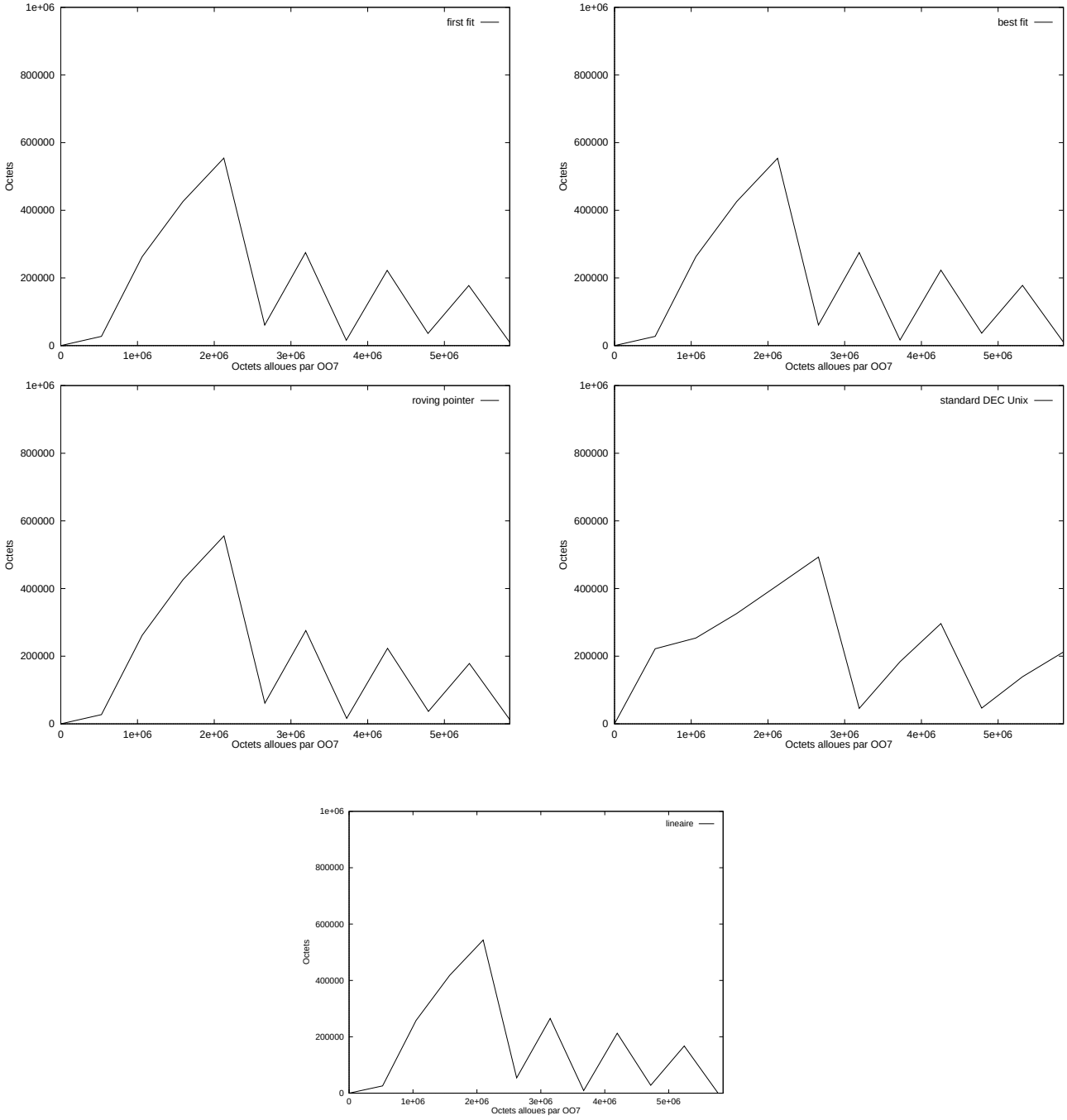

FIG. 4.45 – *Diamètre moyen des cycles en fonction du temps*

4.3.6 Pourcentage d'occurences des longueurs de cycle

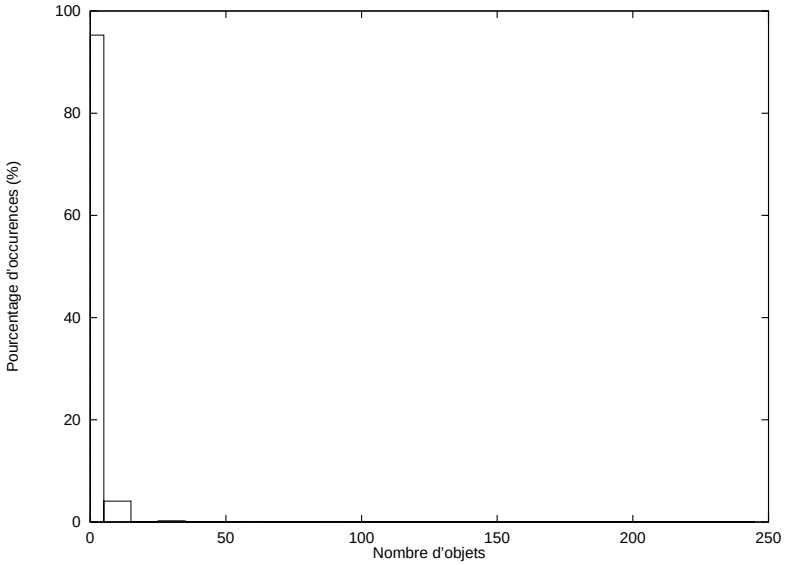

FIG. 4.46 – *Pourcentage d'occurrences des longueurs de cycles (objets)*

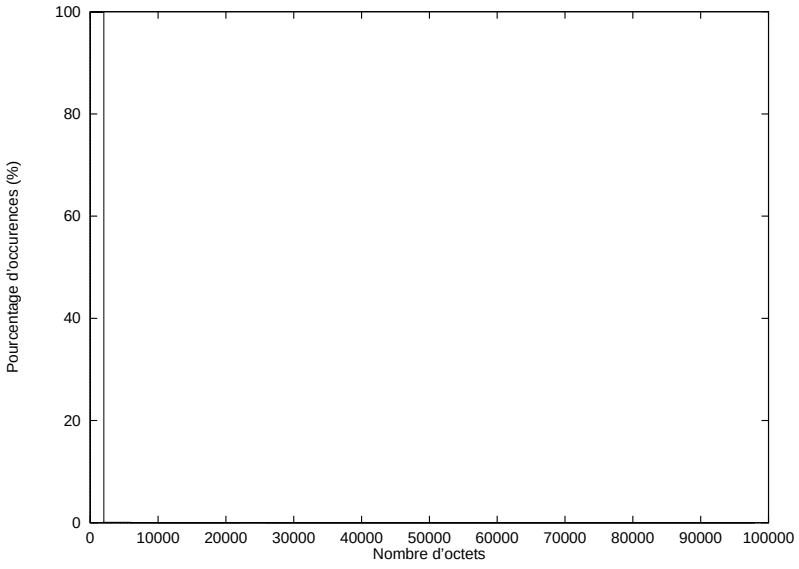

FIG. 4.47 – *Volume moyen des cycles en fonction du temps*

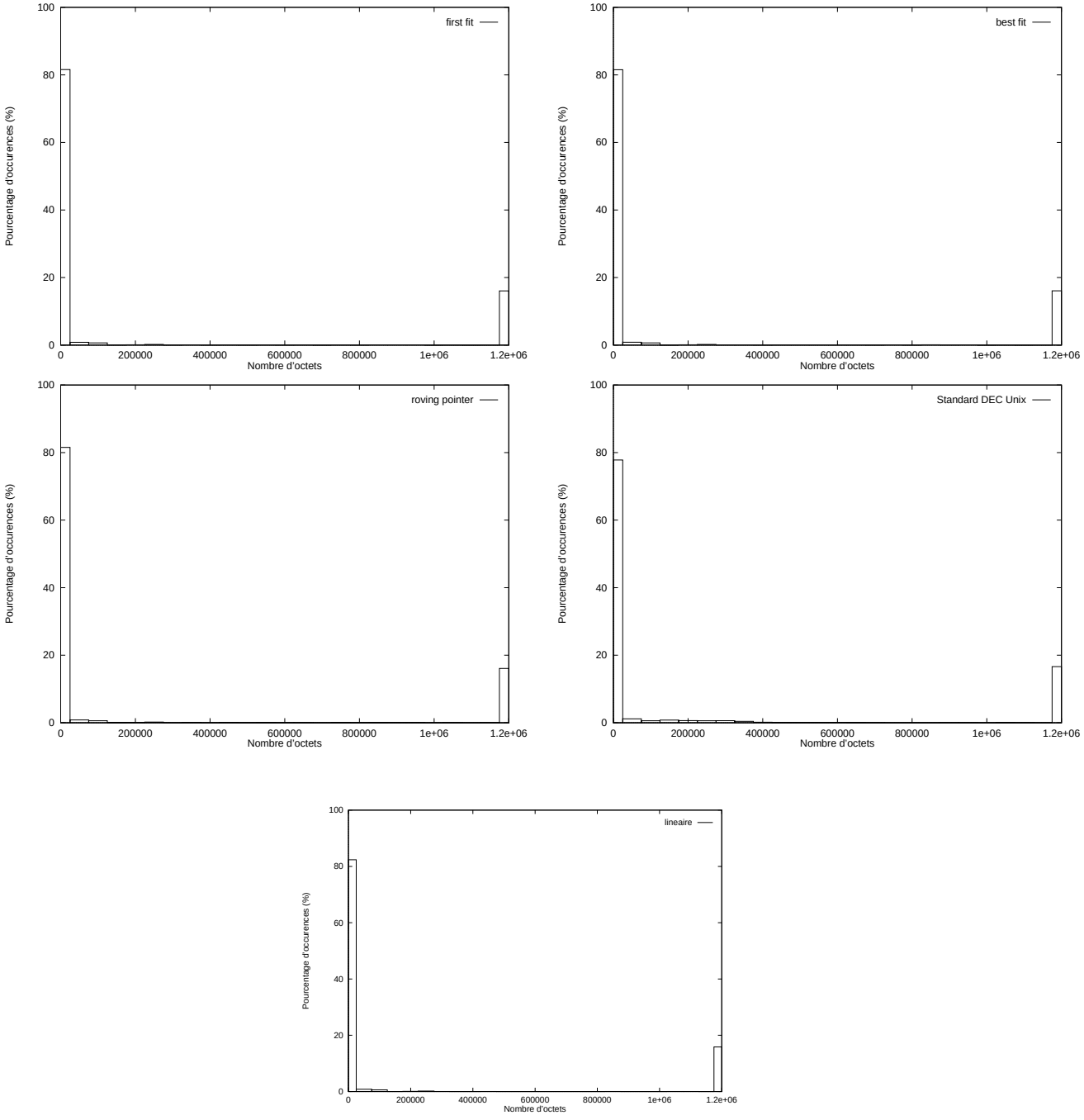

FIG. 4.48 – *Pourcentage d'occurences des diamètre de cycles*

4.4 LaTeX

Nombre d'allocations	13125
Nombre d'octets alloués (Ko)	1185
Taille moyenne d'une requête (octets)	92
Nombre de libérations (par le GC)	5119
Nombre d'octets libérés (Ko)	802
Nombre d'exécutions du GC	14
Seuil de déclenchement du GC (Ko)	64

TAB. 4.17 – *Résultat global de l'exécution*

LaTeX est un formateur de texte d'usage très courant développé à l'origine par Leslie Lamport [Lam86] en se basant sur TeX de Donald Knuth [Knu84]. La version mesurée est TeX 3.14159 (C version 6.1) qui est incluse dans la distribution teTeX 0.4 de LaTeX. Comme pour les autres applications, aucune modification n'a été faite sur le code source.

Cette application a été retenue car elle réalise beaucoup d'allocations dynamiques, même si celles-ci sont de petites tailles. Le fichier d'entrée utilisé pour les tests fait 39 pages et contient 8 références à des figures. Le tableau 4.17 rassemble des statistiques générales concernant l'exécution. Les résultats présentés concerne la durée de vie des objets, les longueurs de références ainsi que les degrés sortant et incident. Il n'y a pas de résultat sur les cycles, car les différentes mesures réalisées n'ont pas permis d'identifier le moindre cycle, que ce soit dans les miettes ou dans les objets vivants.

4.4.1 Durée de vie

Durée de vie	Mo alloués
Durée totale	1185
Durée min	0
Durée max	1065
Moyenne	481

TAB. 4.18 – *Durée de vie des objets en octets alloués pendant leur vie*

Comme pour Ghostscript, la durée de vie est mesurée entre la création des objets et leur libération par le ramasse-miettes afin de recueillir des résultats significatifs lorsqu'une gestion automatique de la mémoire est mise en œuvre. Le tableau 4.18 contient des statistiques globales sur la durée de vie des objets et la figure 4.49 présente le pourcentage d'occurrence de chaque durée de vie.

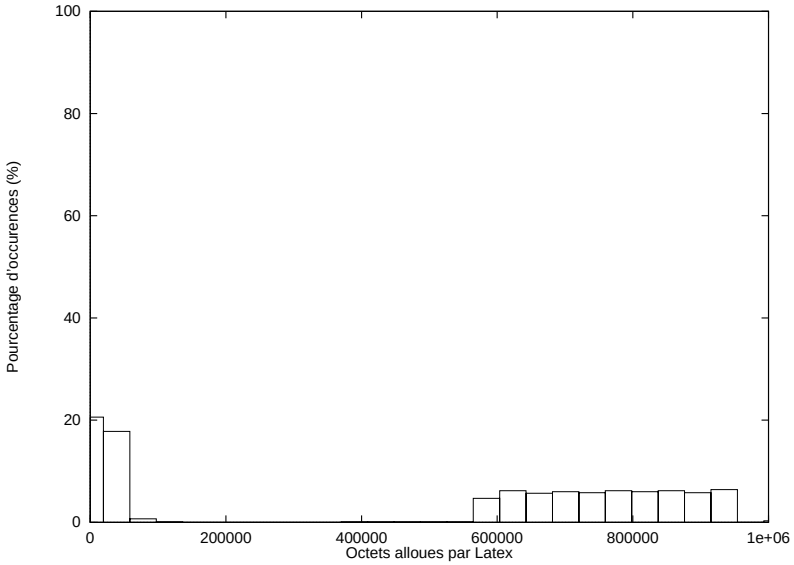

FIG. 4.49 – *Pourcentage d'occurrence des durées de vie en octets alloués*

4.4.2 Longueur des références

Politique d'allocation	octets alloués	linéaire	first fit	best fit	roving
Longueur min (octets)	24	56	56	56	56
Longueur max (Ko)	617	1155	864	720	866
Moyenne (Ko)	50.3	129	96.1	98.8	99.2
Référence avant (%)	28.2	26.3	35.9	44.3	34.1
Référence arrière (%)	71.8	73.7	64.1	55.7	65.9

TAB. 4.19 – *Longueur des références dans un instantané*

Politique d'allocation	octets alloués	linéaire	first fit	best fit	roving
Longueur min (octets)	8	56	56	56	56
Longueur max (Ko)	700	1154	863	861	864
Moyenne (Ko)	107	224	235	217.8	165.8
Référence avant (%)	7.5	10.4	68.9	27.2	38.2
Référence arrière (%)	92.5	89.6	31.1	72.8	61.8

TAB. 4.20 – *Longueur des références dans les miettes*

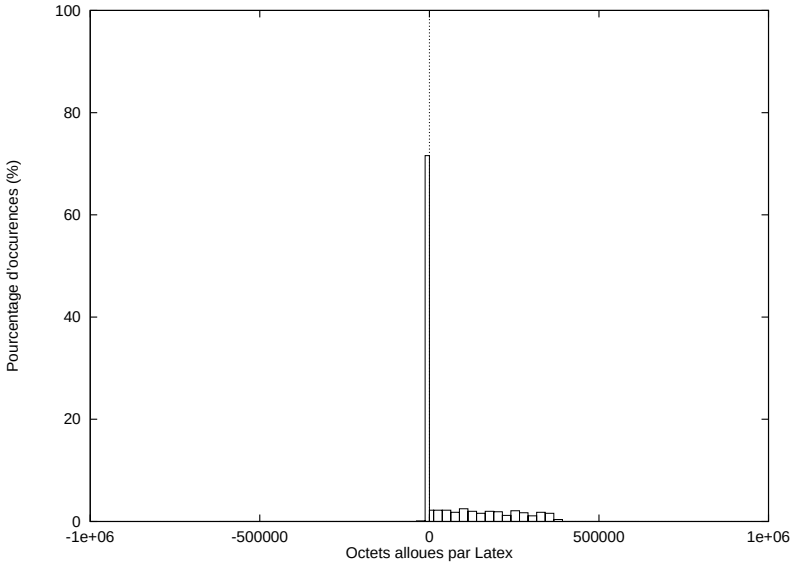

FIG. 4.50 – *Pourcentage d'occurence des longueurs de références en octets alloués dans un instantané*

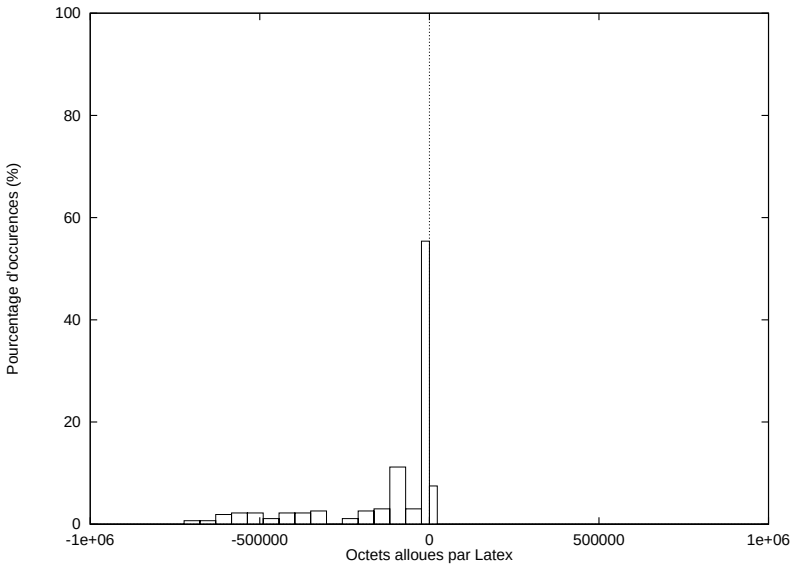

FIG. 4.51 – *Pourcentage d'occurence des longueurs de références en octets alloués dans les miettes*

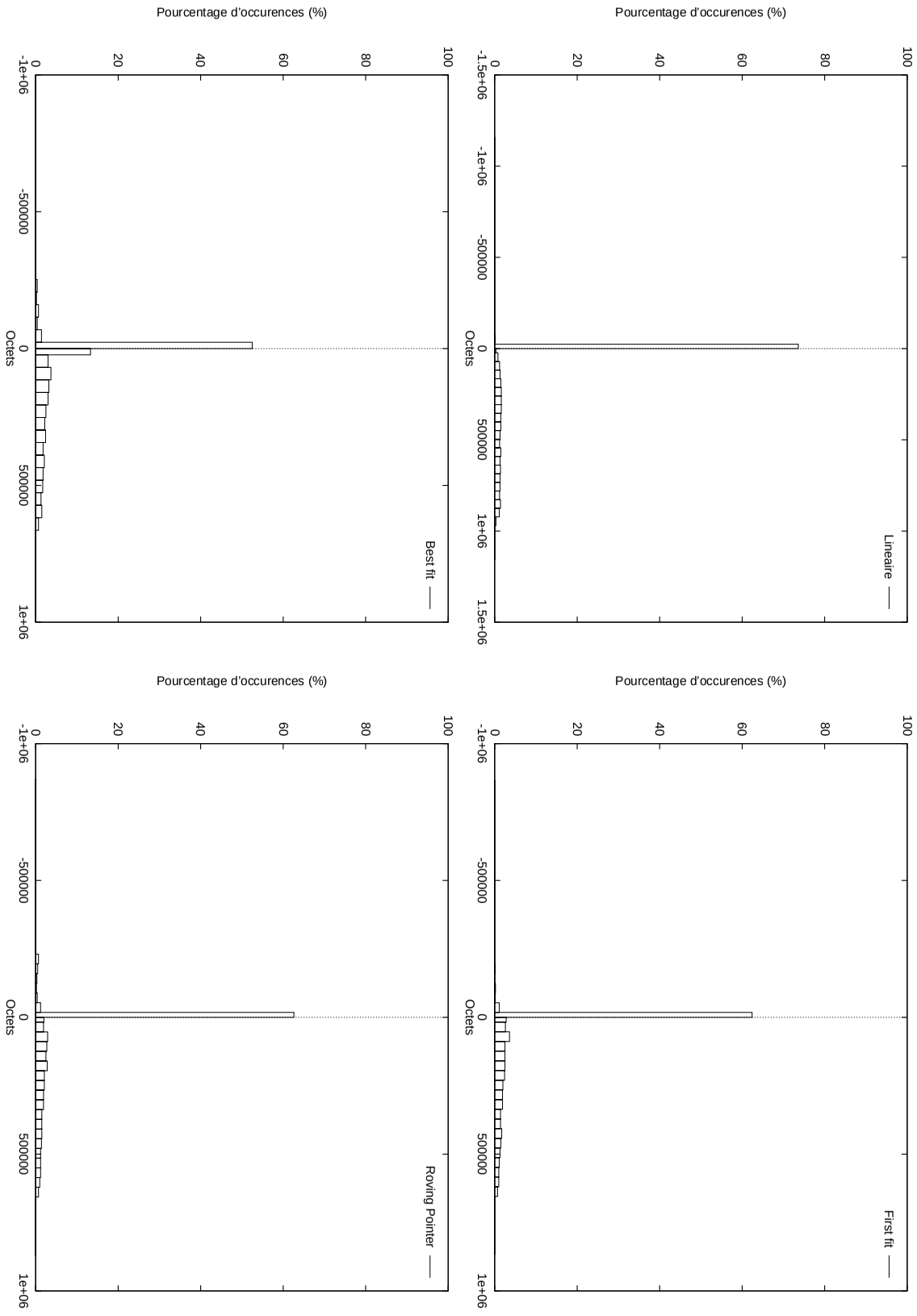

FIG. 4.52 – *Pourcentage d'occurence des longueurs de références en octets dans un instantané en fonction de la politique d'allocation (échelle variable)*

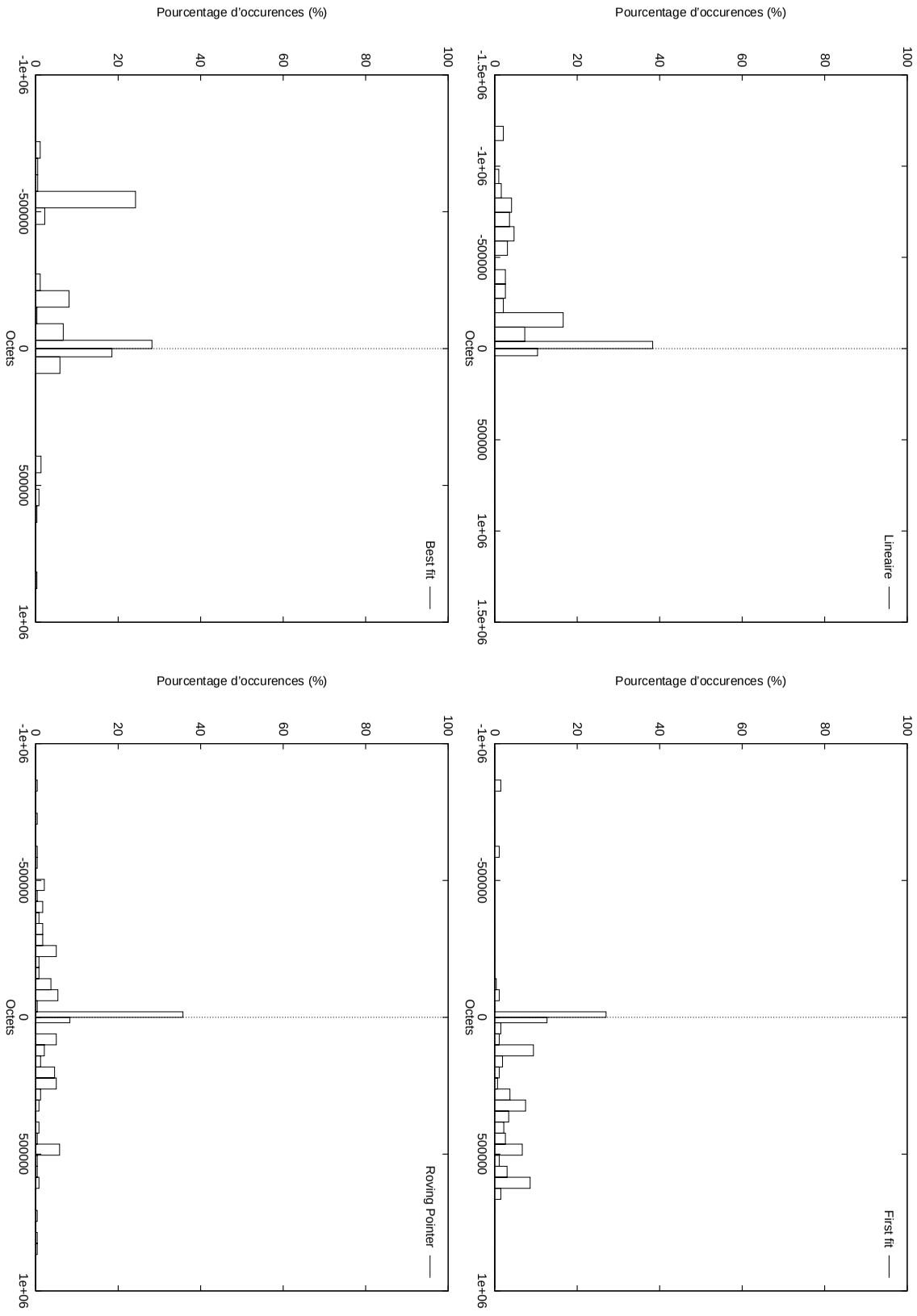

FIG. 4.53 – *Pourcentage d'occurence des longueurs de références en octets dans les miettes en fonction de la politique d'allocation (échelle variable)*

4.4.3 Degrés sortant et incident

	Dans un instantané		Dans les miettes	
Degré	sortant	entrant	sortant	entrant
Degré min	0	0	0	0
Degré max	2664	142	11	5
Degré moyen	1	1	0.1	0.2

TAB. 4.21 – *Statistiques générales*

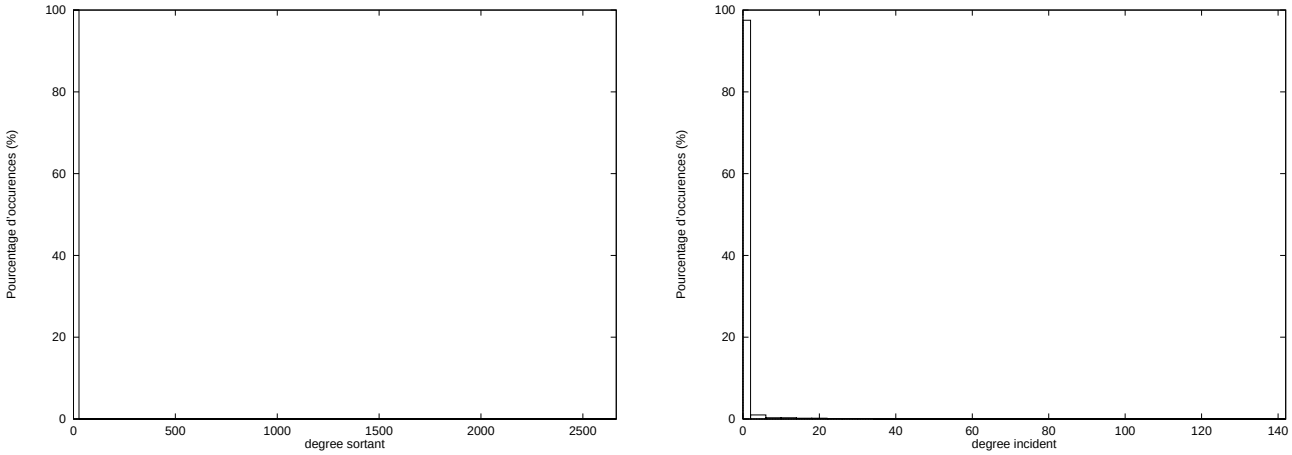

FIG. 4.54 – *Degrés sortant et incident dans un instantané*

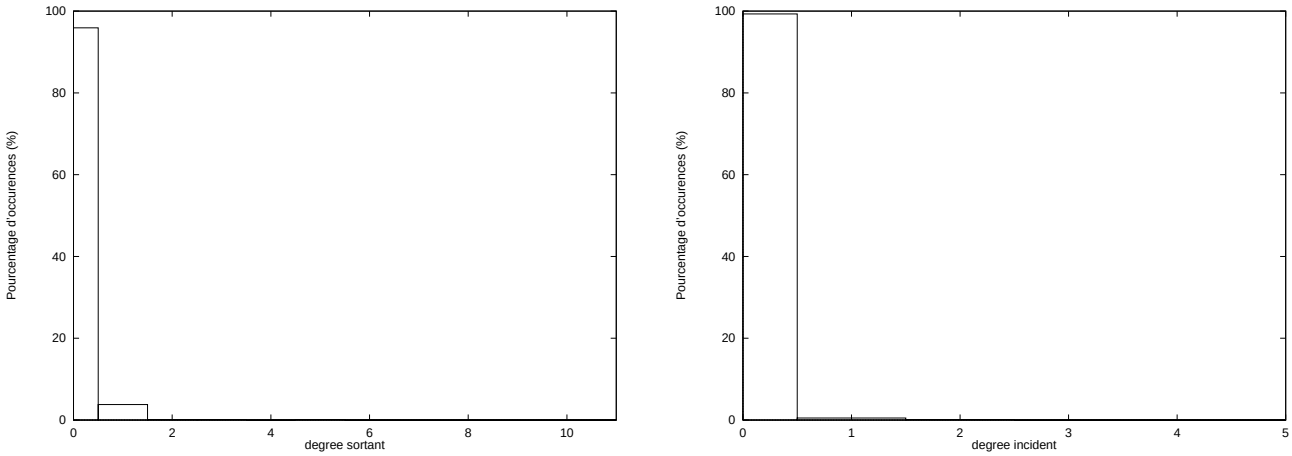

FIG. 4.55 – *Degrés sortant et incident dans les miettes*

4.5 Xfig

Nombre d'allocations	10088
Nombre d'octets alloués (Mo)	2.12
Requête moyenne (octets)	220
Nombre de libérations (par le GC)	5349
Nombre d'octets libérés (Mo)	1.4
Nombre d'exécutions du GC	32
Seuil de déclenchement du GC (Ko)	64

TAB. 4.22 – *Résultat global de l'exécution*

Xfig est une application interactive permettant de créer des dessins afin, par exemple, d'illustrer un article ou un rapport. La version mesurée est Xfig 3.2 sans aucune modification du code source.

Cette application a été retenue car elle est interactive et, de même que pour XV, c'est particulièrement ce type d'application qui nous intéresse dans PerDiS. L'exécution a consisté à créer un dessin composé de rectangles, de cercles et de texte, puis à appliquer des rotations et symétries à l'ensemble du dessin. A la fin, le tout a été enregistré au format Xfig. Le tableau 4.22 rassemble des statistiques générales concernant l'exécution.

Pour cette application, comme pour XV et OO7, les mesures n'ont pas révélé la présence de cycle dans les miettes. En conséquence, toutes les caractéristiques de cycle présentées ne concerne que les objets vivants.

4.5.1 Durée de vie

Durée de vie	Mo alloués
Durée totale	2.12
Durée min	0
Durée max	2
Moyenne	0.6

TAB. 4.23 – *Statistiques générales sur la durée de vie des objets*

Comme pour Ghostscript, la durée de vie est mesurée entre la création des objets et leur libération par le ramasse-miettes afin de recueillir des résultats significatifs lorsqu'une gestion automatique de la mémoire est mise en œuvre. Le tableau 4.23 contient des statistiques globales sur la durée de vie des objets et la figure 4.56 présente le pourcentage d'occurrence de chaque durée de vie.

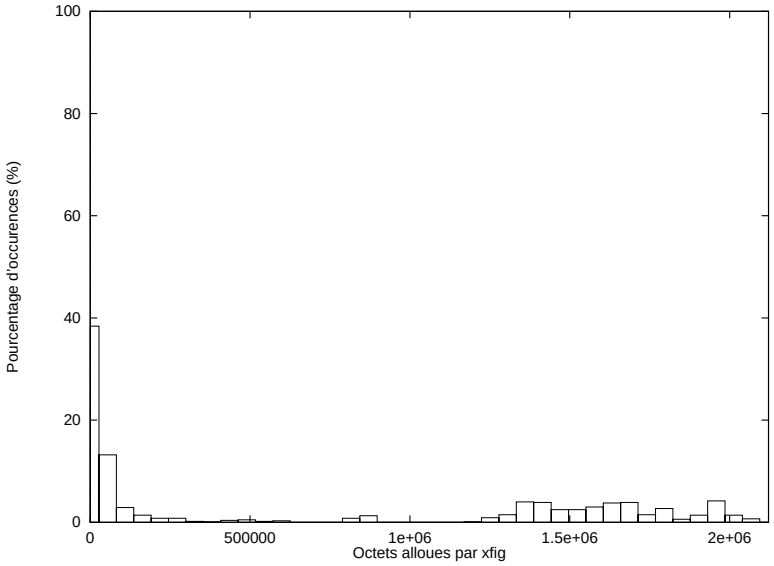

FIG. 4.56 – *Pourcentage d'occurrence des durées de vie en octets alloués*

4.5.2 Longueur des références

Politique d'allocation	octets alloués	linéaire	first fit	best fit	roving
Longueur min (octets)	8	56	64	64	64
Longueur max (Ko)	1548	1892	718	757	922
Moyenne (Ko)	187.2	254	134	143	149.3
Référence avant (%)	41.4	40.8	38.2	39.7	37.8
Référence arrière (%)	58.6	59.2	61.8	60.3	62.2

TAB. 4.24 – *Longueur des références dans un instantané*

Politique d'allocation	octets alloués	linéaire	first fit	best fit	roving
Longueur min (octets)	8	64	80	56	72
Longueur max (Ko)	1985	2345	816	770	981
Moyenne (Ko)	301	398.1	176.9	174.8	187.9
Référence avant (%)	13	11.9	32.9	28.5	23.1
Référence arrière (%)	87	88.1	67.1	71.5	76.9

TAB. 4.25 – *Longueur des références dans les miettes*

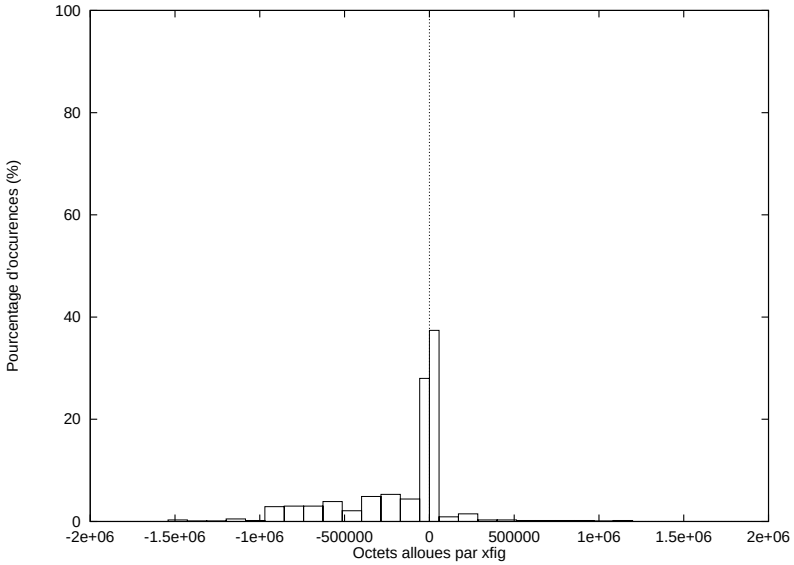

FIG. 4.57 – *Pourcentage d'occurence des longueurs de références en octets alloués dans un instantané*

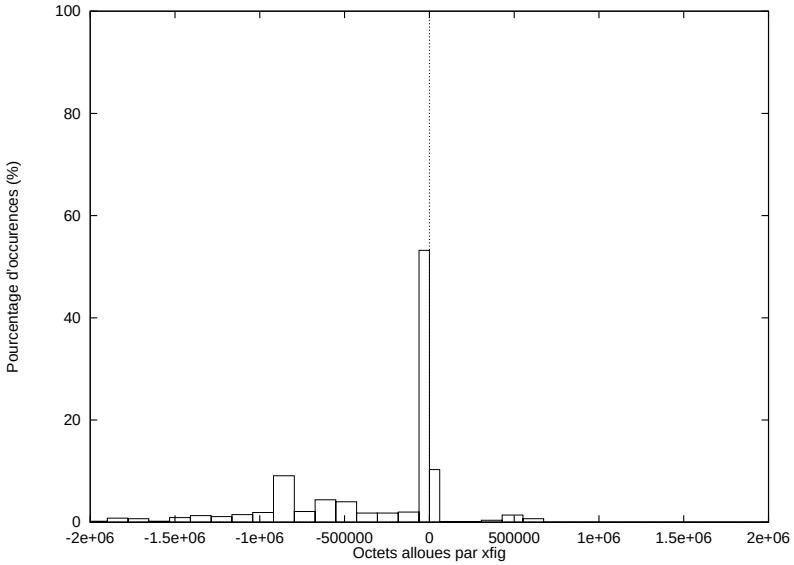

FIG. 4.58 – *Pourcentage d'occurence des longueurs de références en octets alloués dans les miettes*

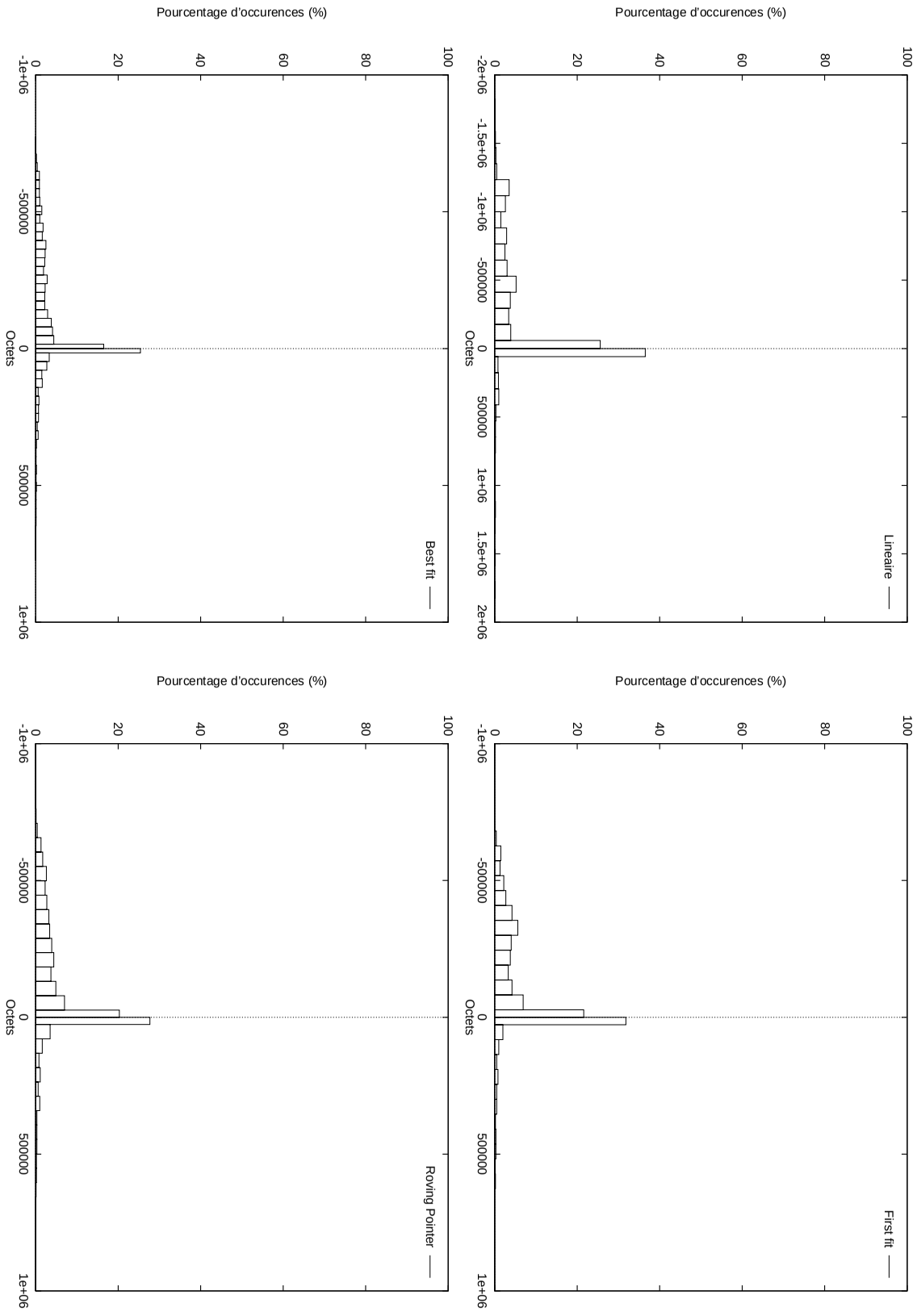

FIG. 4.59 – *Pourcentage d'occurence des longueurs de références en octets dans un instantané en fonction de la politique d'allocation (échelle variable)*

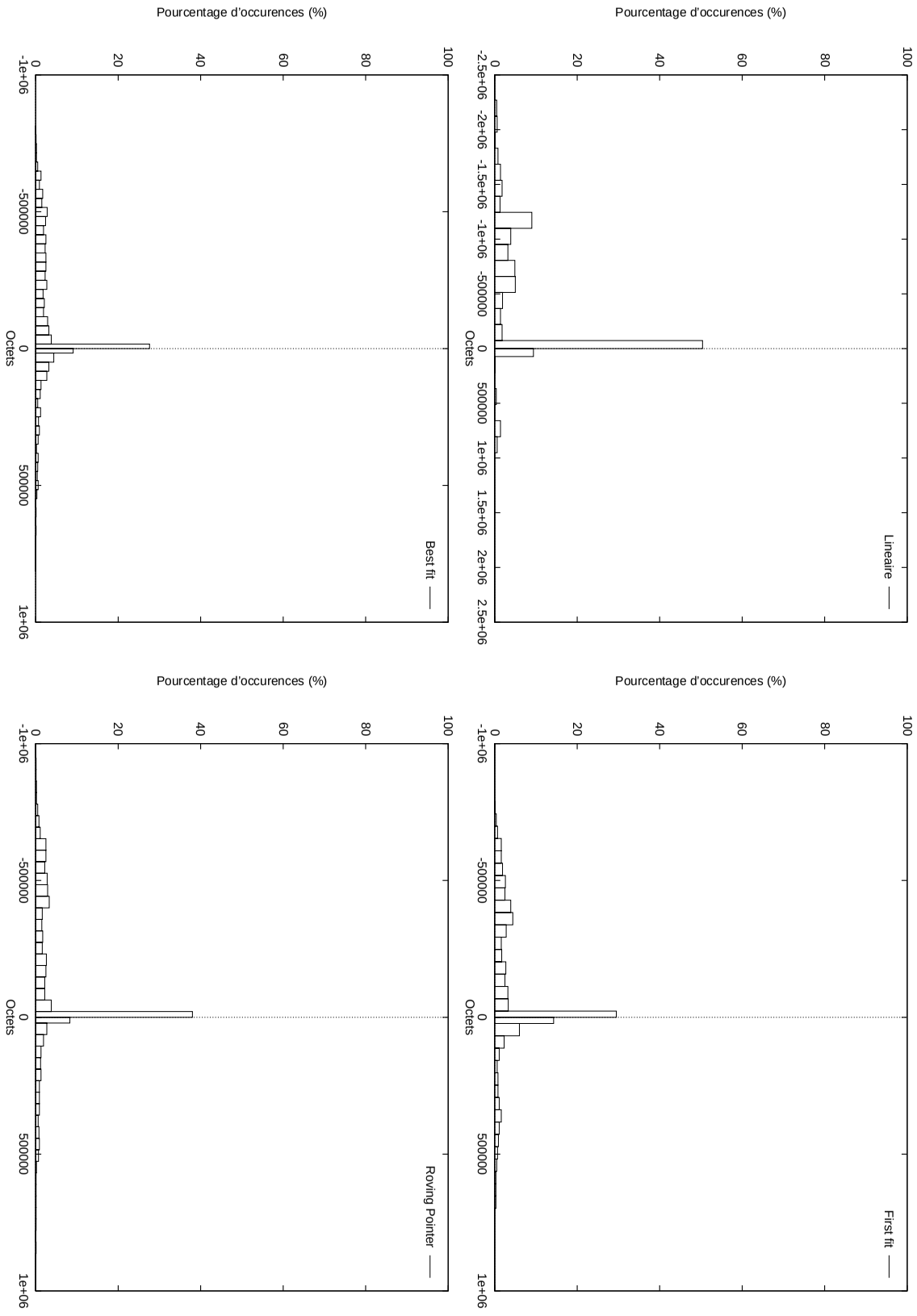

FIG. 4.60 – *Pourcentage d'occurence des longueurs de références en octets dans les miettes en fonction de la politique d'allocation (échelle variable)*

4.5.3 Degrés sortant et incident

	Dans un instantané		Dans les miettes	
Degré	sortant	entrant	sortant	entrant
Degré min	0	0	0	0
Degré max	204	233	204	9
Degré moyen	1	1	0.1	0.2

TAB. 4.26 – *Statistiques générales*

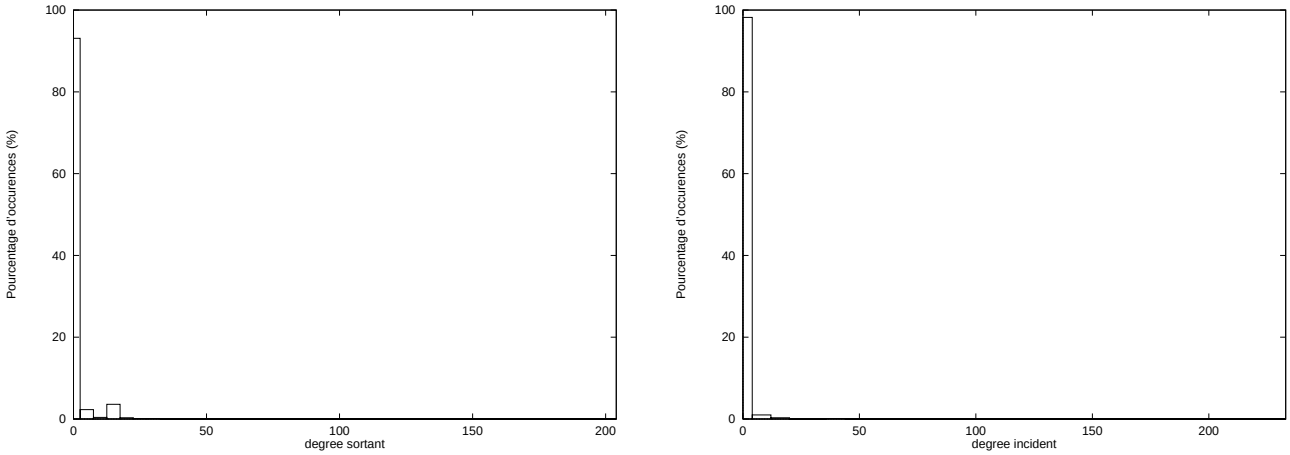

FIG. 4.61 – *Degré sortant et incident dans un instantané*

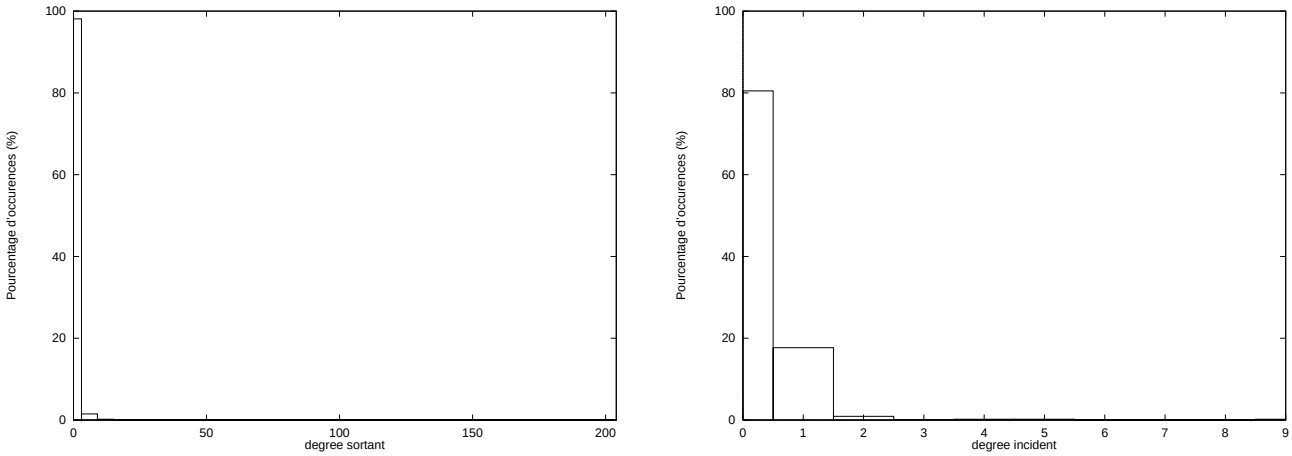

FIG. 4.62 – *Degré sortant et incident dans les miettes*

4.5.4 Proportion d'objets dans les cycles

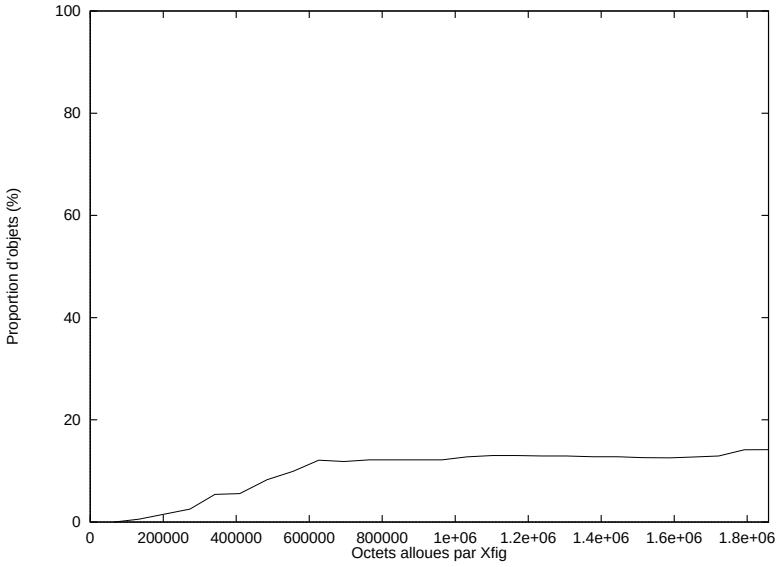

FIG. 4.63 – *Proportion d'objets dans les cycles vivants en fonction du temps*

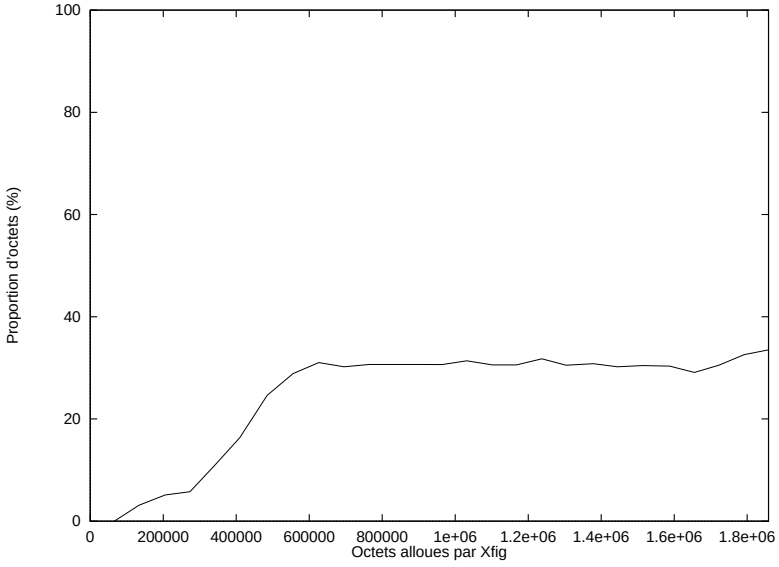

FIG. 4.64 – *Proportion d'octets dans les cycles vivants en fonction du temps*

4.5.5 Longueur moyennes des cycles

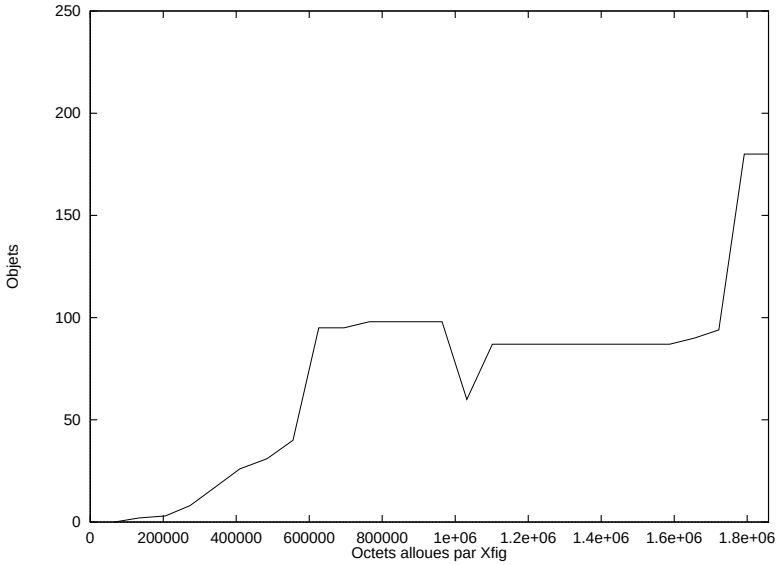

FIG. 4.65 – *Nombre moyen d'objets par cycle en fonction du temps*

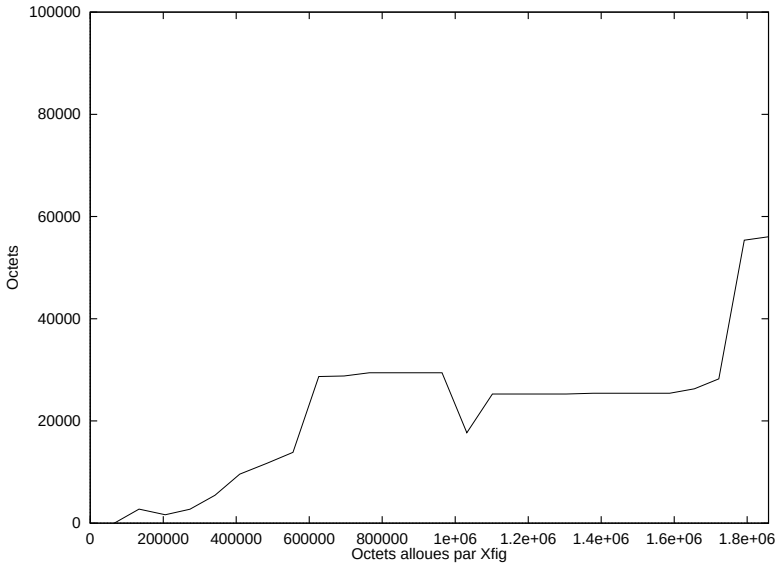

FIG. 4.66 – *Volume moyen des cycles en fonction du temps*

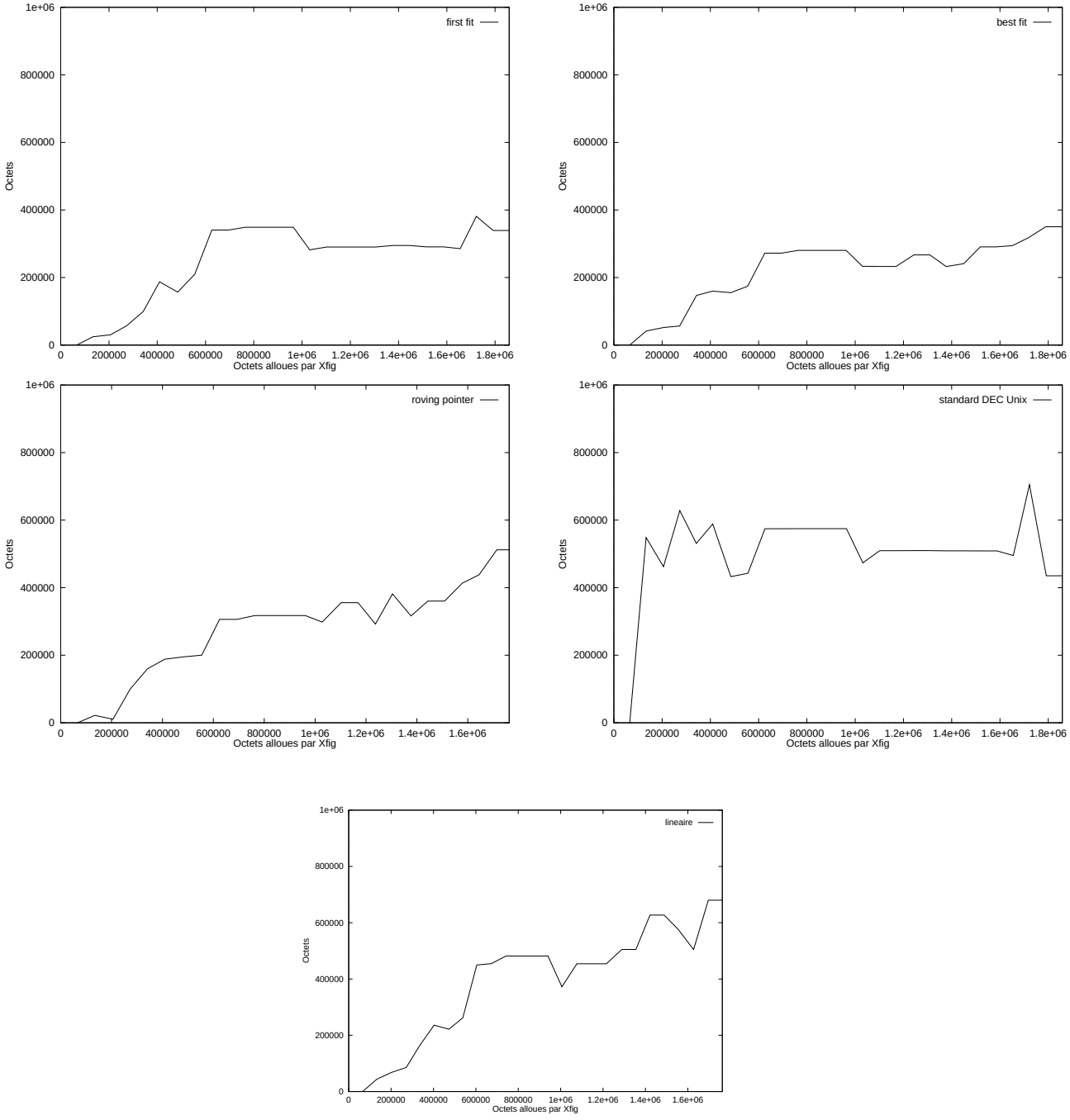

FIG. 4.67 – *Diamètre moyen des cycles en fonction du temps*

4.5.6 Pourcentage d'occurences des longueurs de cycle

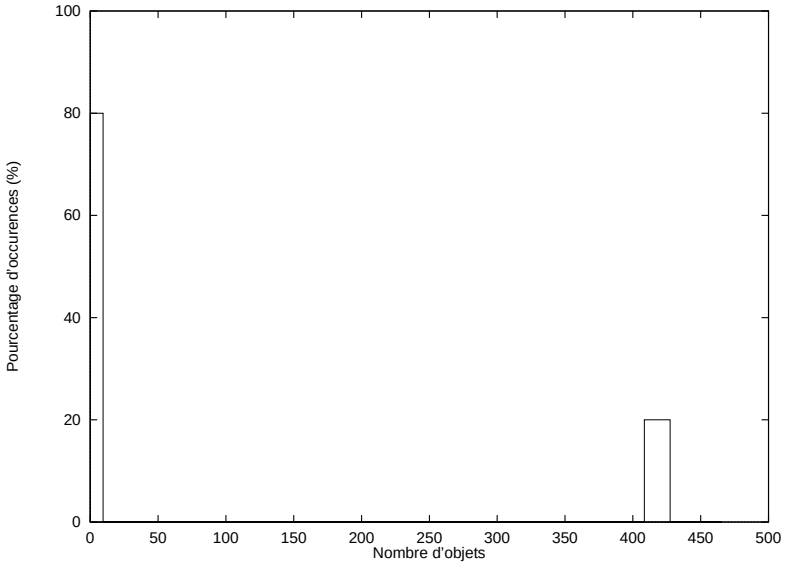

FIG. 4.68 – *Pourcentage d'occurrences des longueurs de cycles (objets)*

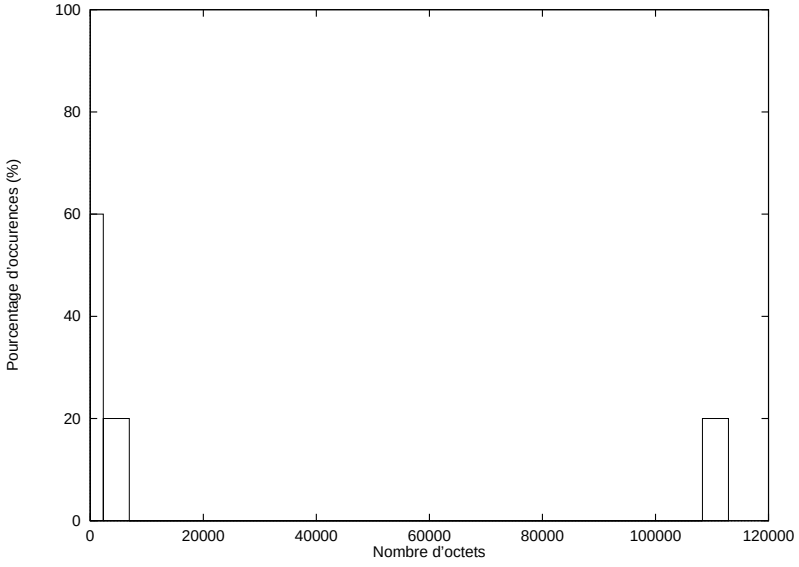

FIG. 4.69 – *Volume moyen des cycles en fonction du temps*

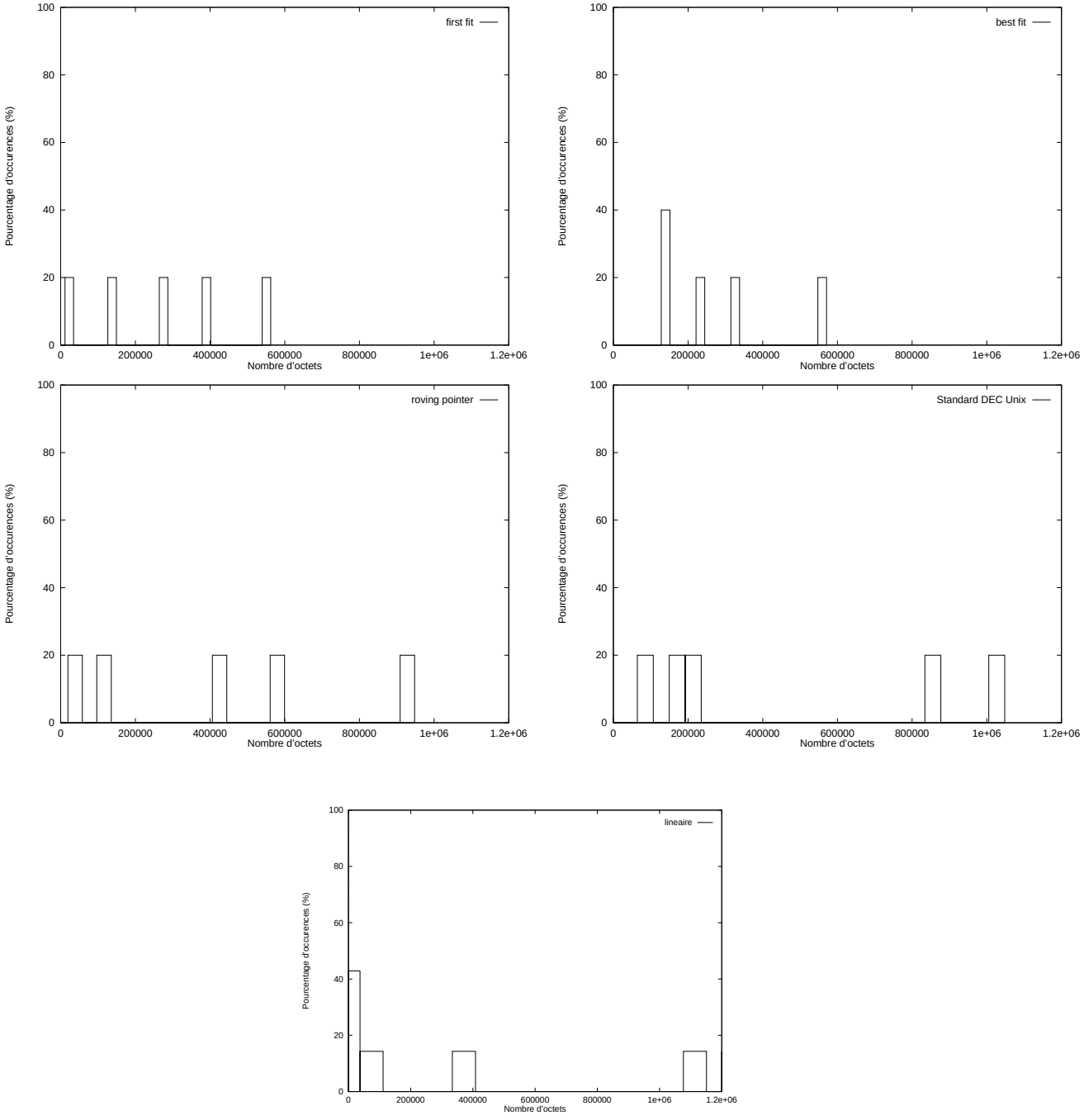

FIG. 4.70 – *Pourcentage d'occurences des diamètres de cycles*

Chapitre 5

Conclusion

5.1 Durée de vie

La durée de vie moyenne des objets varient beaucoup d'une application à l'autre. Dans nos mesures, deux classes de comportements se dégagent. Il y a d'une part Ghostscript et OO7 ou la durée de vie moyenne est courte comparée au temps d'exécution total (n'excèdant pas 10 %) et d'autre part toutes les autres applications (XV, LaTeXet Xfig) ou cette moyenne est supérieure à 25 %. Concernant la répartition des différentes durées de vie, la seule tendance qui apparaisse clairement est qu'une majorité d'objets a une durée de vie courte. La différence relevée sur les moyennes s'explique en fait par la quasi absence de données permanentes[1] allouées dans le tas pour Ghostscript et OO7 alors qu'il y en a dans les autres applications.

Plus généralement, les mesures que nous avons réalisées ont mis en évidence qu'il y avait principalement deux types d'objets pour ce qui concerne la durée de vie :

- Les objets temporaires dont la durée de vie est très faible (inférieure à 5 % du temps total d'exécution).
- Les objets permanents, qui sont créés au début et qui vivent pendant toute l'exécution du programme.

Les objets à durée de vie intermédiaire sont globalement très peu nombreux, à l'exception notable du banc d'essais OO7, pour lequel la répartition des durées de vie est très régulière (cf figure 4.34).

Remarque : L'ensemble de ces mesures de durée de vie portent sur des applications non persistantes, il faut donc être prudent quand à leurs transpositions dans le contexte de la mémoire répartie persistante PerDiS.

5.2 Longueur et densité des références

La densité des références, c'est-à-dire le nombre de références entrant et sortant des objets, est globalement assez faible pour les applications que nous avons mesurées. De façon générale, les valeurs moyennes sont d'une ou deux références entrantes et d'une ou deux références sortantes. En outre, la grande majorité des objets comporte moins de cinq références.

Concernant la longueur des références, il apparaît assez nettement pour toutes les applications que la majorité des références (50 à 70 %) pointe à une distance relativement faible (en octets comme en différence de date d'allocation) ce qui indique clairement que les objets alloués ensemble ont effectivement plus de référence les uns avec les autres que les objets quelconques.

1. Objets alloués pendant toute la durée de l'exécution du programme.

L'impact de la politique d'allocation mémoire sur la longueur des références est très important pour certaines applications, comme Ghostscript ou XV, alors qu'il est à peu près nul pour OO7. Cela vient du fait que OO7 ne génère que très peu de miettes ; la mémoire n'est par conséquent que très peu réutilisée et les résultats des différents allocateurs sont sensiblement les mêmes. De façon plus générale, la politique d'allocation linéaire génére des longueurs de référence plus grandes que les autres politiques. En effet, comme il n'y a pas de réutilisation de la mémoire, le tas devient rapidement très grand. Dans ces conditions, les références entre objets de générations différentes (0 à 20 % des références) sont très longues, ce qui pèse considérablement sur la longueur moyenne des références. La politique qui limite le plus la longueur des références est le *first fit* suivi de très près par le *best fit*. Nous verrons que les choses sont un peu plus complexe en ce qui concerne la longueur des cycles.

5.3 Caractéristiques des cycles

5.3.1 Quantification des cycles

La première remarque concernant les cycles est qu'ils sont peu nombreux si l'on ne s'intéresse qu'aux miettes. En effet, nous n'avons pu trouver des cycles de miettes que pour Ghostscript et il s'agit de cycles mettant en cause un petit nombre d'objets (deux au maximum). En revanche, les cycles sont beaucoup plus nombreux dans les objets vivants. La proportion d'objets dans les cycles vivants atteint parfois 60 % pour Ghostscript et 80 % pour OO7. Cette différence flagrante entre le nombre de cycles dans les objets vivants et le nombre de cycles dans les miettes n'est toutefois pas très surprenante. En effet, la plupart des cycles correspondent à des listes doublement chaînées ou simplement chaînées circulaires. Dans ces conditions, il est plutôt rare de voir l'ensemble d'une liste libérée en une fois. Un comportement beaucoup plus classique consiste à ajouter ou retirer les objets un par un dans une liste et à détruire toutes les références à la liste dans l'objet qui en est retiré[2]. Une exception toutefois est l'application Ghostscript pour laquelle nous observons huit pics dans la proportion d'objets et d'octets dans les cycles (cf figures 4.8). Ils correspondent aux huit dessins que contient notre document, car, pour interpréter ces dessins, Ghostscript construit une vaste liste chaînée circulaire. Le point intéressant est que la liste est complètement détruite après le traitement de chaque dessin, mais qu'elle ne se retrouve pas dans les miettes. Cela tient au fait que Ghostscript fait une gestion de la mémoire interne par dessus malloc, et qu'il efface le contenu de la mémoire avant de la libérer.

5.3.2 Longueur des cycles

La longueur des cycles, une fois encore, n'est pas du même ordre dans les objets vivants et dans les miettes. Le nombre moyen d'objets impliqués dans des cycles de miettes ne dépasse pas deux et la taille moyenne en octets se limite à 500 octets. Dans ce contexte, c'est la politique d'allocation linéaire qui obtient les meilleurs résultats avec un diamètre de cycle toujours inférieur à 600 octets (cf figure 4.15). Les résultats concernant la longueur des cycles de miettes sont toutefois à prendre avec précaution au vu du petit nombre de cycles mesurés et sur une application seulement. La principale information à retenir de notre étude des cycles de miettes est leur faible occurrence pour les applications qui ont été mesurées. Il n'est toutefois pas encore possible de généraliser cette affirmation en raison du faible nombre d'applications mesurées et de l'environnement mono-langage.

Pour les cycles d'objets vivants, les longueurs en objets ainsi qu'en octets sont parfois plus importantes. En particulier avec l'application Ghostscript ou Xfig, des cycles de plusieurs centaines d'objets ont été rencontrés et des cycles de plus d'un mégaoctets ont été mesurés avec Ghostscript. Dans ce contexte, la politique d'allocation linéaire ne donne pas de bons résultats. La politique *first fit* donne les meilleurs résultats en limitant le diamètre à 2Mo pour Ghostscript et à 400 Ko pour Xfig. La politique

2. Il sera intéressant par la suite de regarder si un tel comportement se retrouve avec des applications écrites dans d'autres langages commme Smalltalk ou Java.

best fit obtient des résultats assez similaires dans la plupart des cas, mais sa mise en œuvre plus lourde la rend globalement moins intéressante. Un point intéressant concernant les politiques d'allocation est que l'allocateur standard de Digital Unix, qui met visiblement en œuvre une politique *segregated fit* obtient des diamètres de cycle plus importants que les deux allocateurs précédents, mais pas dans des proportions aussi importantes que nous pouvions le craindre. Une explication à ce phénomène est que les objets dans les cycles vivants que nous avons rencontrés dans Ghostscript et dans OO7 ont une taille relativement importante et que l'allocateur de Digital Unix traite les objets au-delà d'une certaine taille avec une politique différente (sans doute, un *first fit* comme c'est souvent le cas dans les mises en œuvre de la politique *segregated fit*).

5.4 Limite de l'étude

- peu d'applications mesurées ;
- pas de véritable application utilisant des données persistantes (OO7 n'est qu'un banc d'essai) ;
- pas d'application multi-utilisateurs ;
- les objets accessible uniquement depuis un cycle ne sont pas comptabilisés dans le total des octets dans les cycles.

5.5 Perspectives et extensions

Le travail de stage a permis d'identifier différentes caractéristiques pour chacune des applications qui ont été mesurées. En outre, le comportement de plusieurs politiques d'allocation mémoire à pu être évalué. Les extensions possible de ce travail peuvent prendre différentes formes :

- Amélioration de l'outil de mesure dynamique pour qu'il puisse caractériser les cycles, et utilisation du mécanisme de bibliothèques dynamiques pour permettre la mesure d'applications dont le code source n'est pas disponible (sous Digital Unix et Linux).
- Simulation d'un plus grand nombre d'allocateurs afin de déterminer si les propriétés déjà identifiées sont liées à la politique d'allocation elle-même, ou si elles sont liées à une mise en œuvre particulière.
- Mesures de nouvelles applications et en particulier d'applications interactives utilisant encore plus largement l'allocation dynamique.
- Mesures d'applications persistantes portées sur la plate-forme PerDiS. Pour ce faire, l'outil de traçage et d'analyse dynamique peut être facilement intégré dans la plate-forme. C'est le point essentiel à réaliser de mon point de vue, car il permettra d'obtenir des résultats sur des applications persistantes.
- L'utilisation de la plate-forme PerDiS nous donnera accés aux informations de type. Celles-ci nous permettrons d'effectuer de nouvelles mesures visant particulièrement la définition de la stratégie de placement dans les flopées.
- Réaliser une série de mesure sur le WWW, car c'est un bon exemple d'application répartie persistante de très grande échelle.

Bibliographie

[ABC+83] 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.

[Adj96] Cédric Adjih. Mesure et caractérisation d'applications réparties. Master's thesis, Université Paris XI Orsay, 1996. Stage au projet SOR - Responsable Marc Shapiro.

[Boe91] Hans-Juergen Boehm. Hardware and operating system support for conservative garbage collection. *1991 International Workshop on Object Orientation in Operating Systems*, pages 61–67, October 1991. IEEE Computer Society Press Order Number 2265.

[BZ93] David A Barrett and Benjamin Zorn. Using lifetime predictors to improve memory allocation performances. Technical report, University of Colorado at Boulder, June 1993.

[DDZ93] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. Technical report, University of Colorado at Boulder, August 1993.

[FS94] Paulo Ferreira and Marc Shapiro. Garbage collection of persistent objects in distributed shared memory. In *Proc. of the 6th Int. Workshop on Persistent Object Systems*, pages 176–191, Tarascon (France), September 1994. Springer-Verlag. http://www-sor.inria.fr/SOR/docs/GC-PERS-DSM_POS94.html.

[GZH93] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *Proc. ACM SIGPLAN'93 Conf. on Programming Language Design and Implementation (PLDI)*, Albuquerque, New Mexico, June 1993. ftp://ftp.cs.colorado.edu/pub/cs/techreports/zorn/PLDI-93-locality.ps.Z.

[KCDZ94] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Winter Usenix*, pages 115–131, San Francisco CA, January 1994.

[Knu84] Donald E. Knuth. *The Texbook*. Addison-Westley, 1984.

[Lam86] Leslie Lamport. *Latex users guide & reference manual*. Addison-Westley, 1986.

[Lar90] James R. Larus. Abstract execution : A technique for efficiently tracing programs. Technical report, University of Wisconsin-Madison, 1990.

[Lar93] James R. Larus. Efficient program tracing. Technical Report 25 5, IEEE Computer, May 1993.

[LBA97] Luciana Bezerra Arantes. Object clustering. Technical report, INRIA - Project SOR, April 1997. http://www.perdis.esprit.ec.org/deliverables/docs/tracing/cluster.ps.gz.

[Nee96] Michael Shannon Neely. An analysis of the effect of memory allocation policy on storage fragmentation. Master's thesis, The University of Texas at Austin, 1996. OOPS Research Group - Supervisor : Paul R. Wilson.

[NSS93] Esko Nuutila and Eljas Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters 49 (1994)*, pages 9–14, October 1993.

[OG92] Olivier Gruber. *Eos, an Environment for Persistent and Distributed Applications over a Shared Object Space*. PhD thesis, Université Paris VI, Paris (FRANCE), Décembre 1992. Projet Rodin - INRIA Rocquencourt.

[Ric97] Nicolas Richer. Etude de la traçabilité du graphe des objets en mémoire. Technical report, INRIA - SOR, June 1997. http://www.perdis.esprit.ec.org/deliverables/docs/tracing/tracage.ps.

[SKR97] Marc Shapiro, Sytse Kloosterman, and Fabio Riccardi. PerDiS - A persistent distributed store for cooperative applications. Technical report, INRIA - SOR, February 1997. http://www.perdis.esprit.ec.org/papers/skr97.ps.gz.

[Tar71] R. Tarjan. Depth-first search and linear graph algorithms. In *Proceedings IEEE International Symposium on the Theory of Switching*, 1971.

[Tre90] John De Treville. Heap usage in the Topaz environment. Technical report, digital Systems Research Center, August 1990.

[Wen90] E. P. Wentworth. Pitfalls of conservative garbage collection. *Software—Practice and Experience*, 20:719–727, July 1990.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), September 1995. ftp://ftp.cs.utexas.edu/pub/garbage/allocscr.ps.

[ZG92a] Benjamin Zorn and Dirk Grunwald. Empirical measurements of six allocation-intensive C programs. Technical report, University of Colorado at Boulder, July 1992.

[ZG92b] Benjamin Zorn and Dirk Grunwald. Evaluating models of memory allocation. Technical report, University of Colorado at Boulder, July 1992.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399