BigGC :
Un ramasse-miettes pour les grands volumes de données

Soutenance de stage

Georgios Boumis

Université Pierre et Marie Curie

2 septembre 2014

Encadrant : Gaël Thomas
Référent : Julien Sopena

Big Data et ramasse-miettes

- à l'époque du Cloud Computing et du Big Data :

  Facebook analyse 30 peta octets *par jour* [1] (en 2012)

- problèmes :
  - traitement des données effectué avec des applications Java
  - grands volumes de données en mémoire, 100+ giga octets
  - énorme pression sur le ramasse-miettes

[1] http ://wikibon.org/blog/taming-big-data/

Big Data et ramasse-miettes

- le coût en performance du ramasse-miette augmente avec la taille du tas
- le ramasse-miettes *doit monter en échelle* avec la taille du tas
- besoin d'un ramasse-miette qui gère les grands volumes de données des applications *sans* les pénaliser

État de l'art : les ramasse-miettes concurrents

- l'application tourne en *parallèle* avec le ramasse-miettes
  $\rightarrow$ évite de mettre en pause l'application pendant la collection

État de l'art : les ramasse-miettes concurrents

- l'application tourne en *parallèle* avec le ramasse-miettes
  $\rightarrow$ évite de mettre en pause l'application pendant la collection
- plusieurs algorithmes proposées : STOPLESS(2006), The Compressor(2007), C4(2011), The Collie(2012), ...

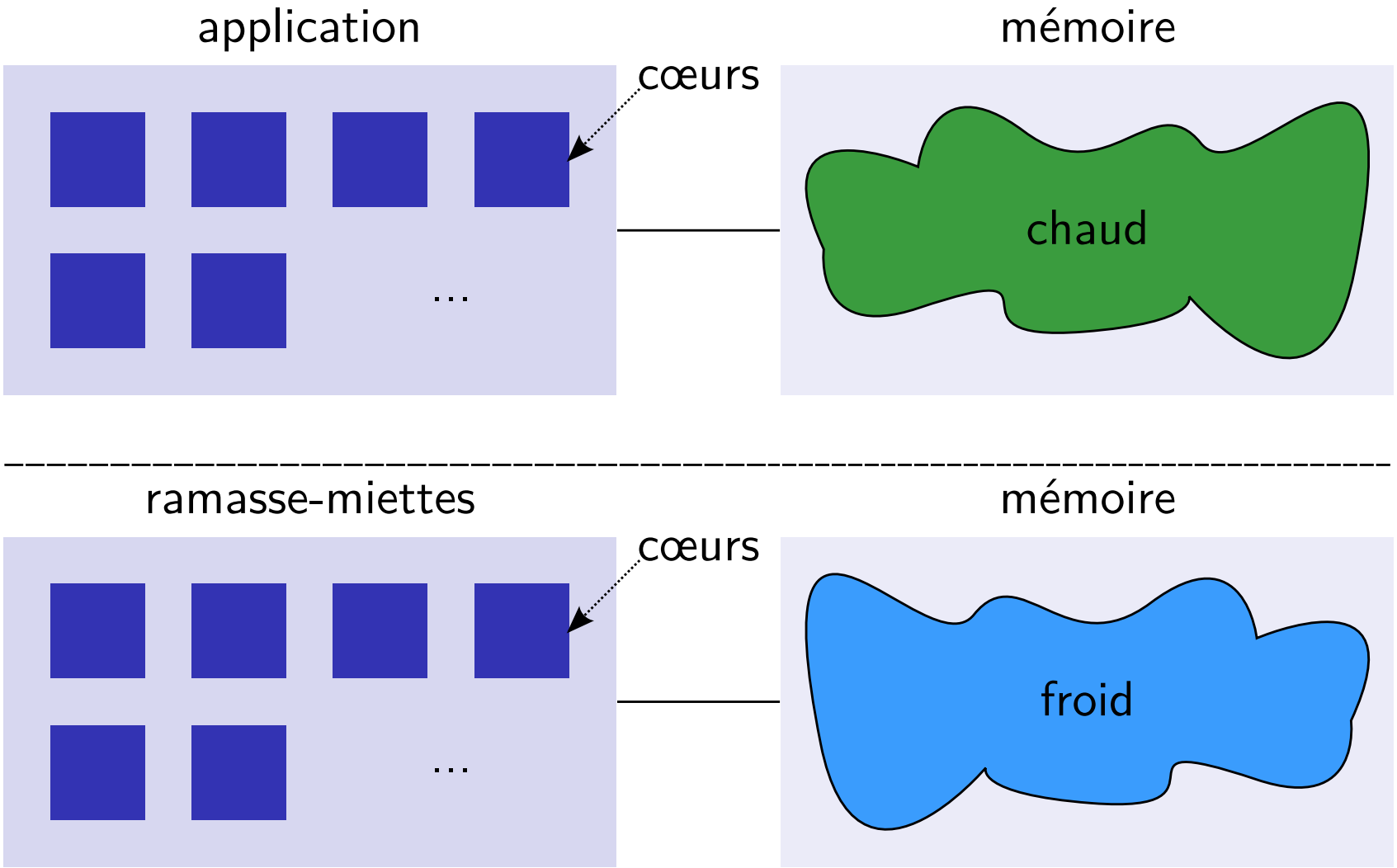
État de l'art : les ramasse-miettes concurrents

- l'application tourne en *parallèle* avec le ramasse-miettes
  → évite de mettre en pause l'application pendant la collection
- plusieurs algorithmes proposées : STOPLESS(2006), The Compressor(2007), C4(2011), The Collie(2012), ...
- problèmes introduits :
  - instrumentation du code de l'application (barrières mémoire)
    → diminution de la performance de l'application
  - le ramasse-miettes et l'application travaillent sur les *mêmes* ressources en parallèle (tas)
    → congestion d'accès (cache, bus mémoire, cœurs, interconnect, ...)

BigGC

partitionner le matériel entre le ramasse-miettes et l'application

- le ramasse-miettes travaille sur une partie et l'application sur l'autre



Stage : le but

- séparer le tas en deux parties :
  - zone chaude : accédée par l'application (`RW` pour l'application et `RW` pour le ramasse-miettes)
  - zone froide : non accédée par l'application (`NONE` pour l'application et `RW` pour le ramasse-miettes)

Stage : les objectifs

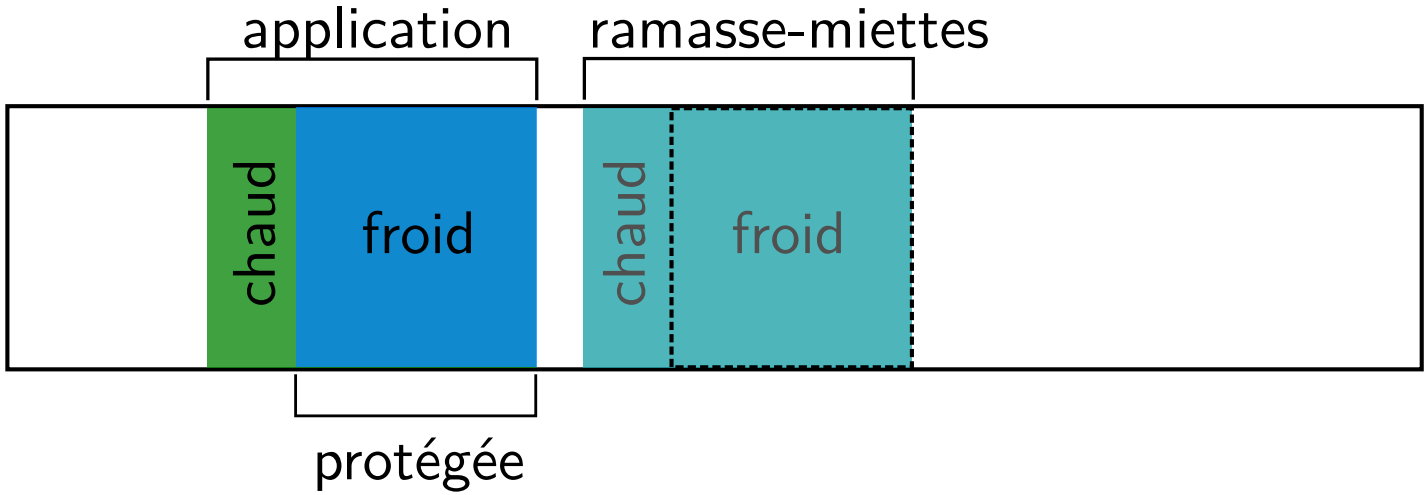
- droits d'accès différents sur le tas si c'est le ramasse-miettes ou l'application qui y accède
- identification des zones froides et les zones chaudes

Stage et travaux accomplis

- contributions :
  - structuration du tas pour la séparation en chaud/froid
  - identification des parties chaudes/froides du tas
- **mis en œuvre** dans HotSpot/OpenJDK

Plan

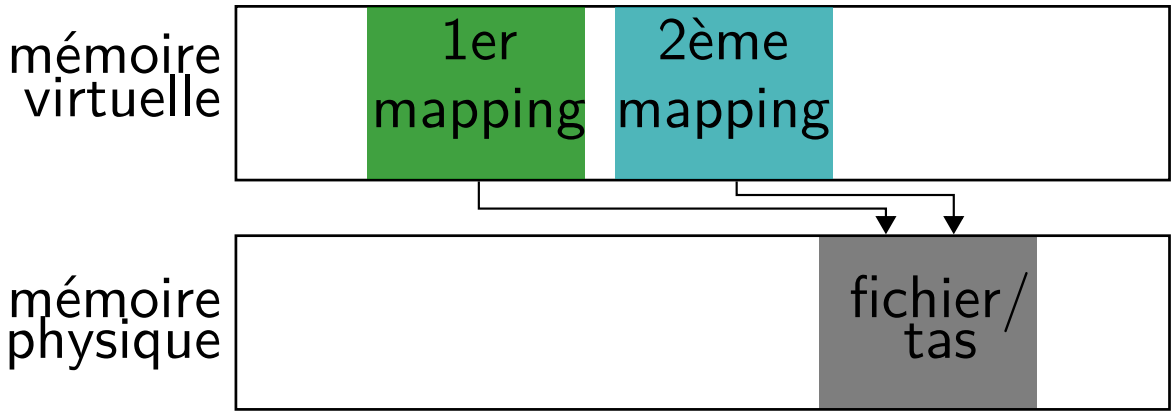
Structuration du tas



- pour structurer le tas
  - besoin de réplication

Réplication du tas

Double Map

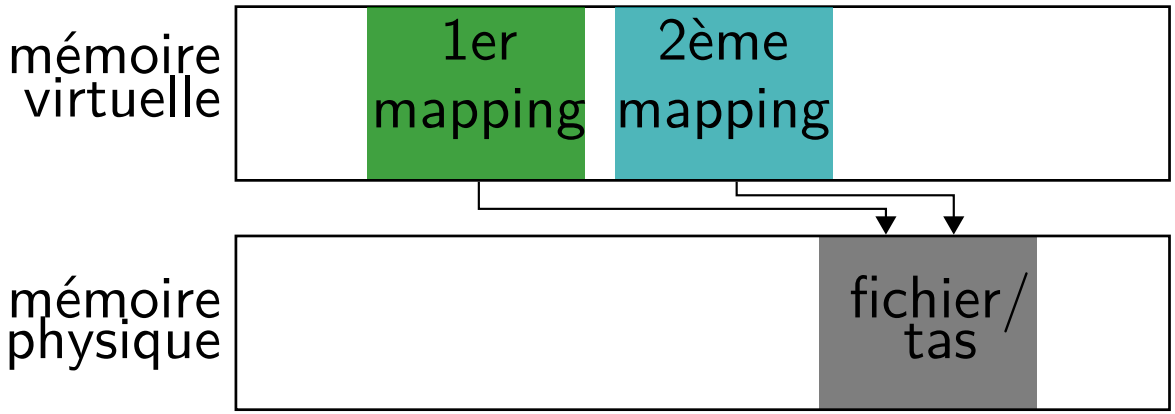


Definition

double map : association d'une plage d'adressage physique vers deux emplacements de la mémoire virtuelle

Réplication du tas

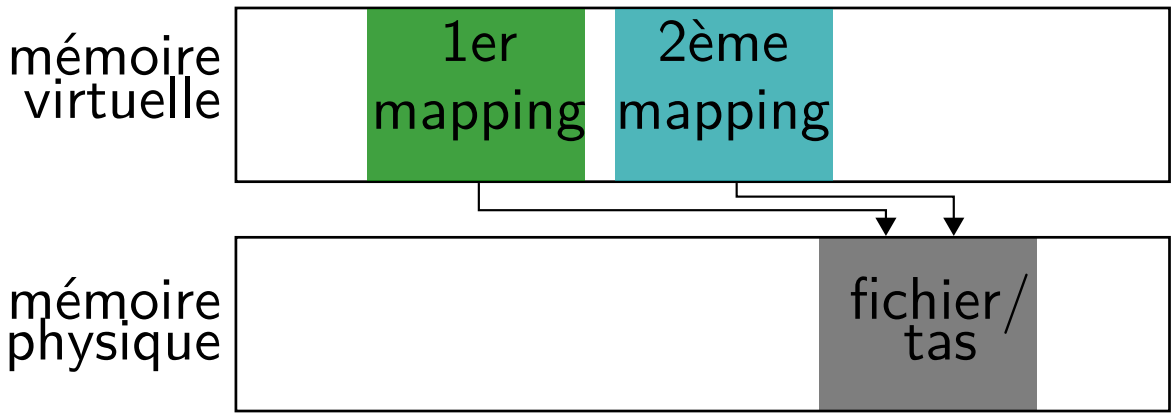
l'intérêt du double map



- les deux parties peuvent avoir des droits indépendants
- les modifications faites dans un mapping sont visibles dans l'autre

Réplication du tas

aspects techniques



- mise en œuvre :
  1. créer/ouvrir un fichier avec `open(2)`
  2. tronquer le fichier avec `ftruncate(2)`
  3. mapper le fichier deux fois avec `mmap(2)`
     utiliser comme flag `MAP_SHARED`
  4. donner le 1er mapping à l'application
     garder le second pour le ramasse-miettes

Réplication du tas

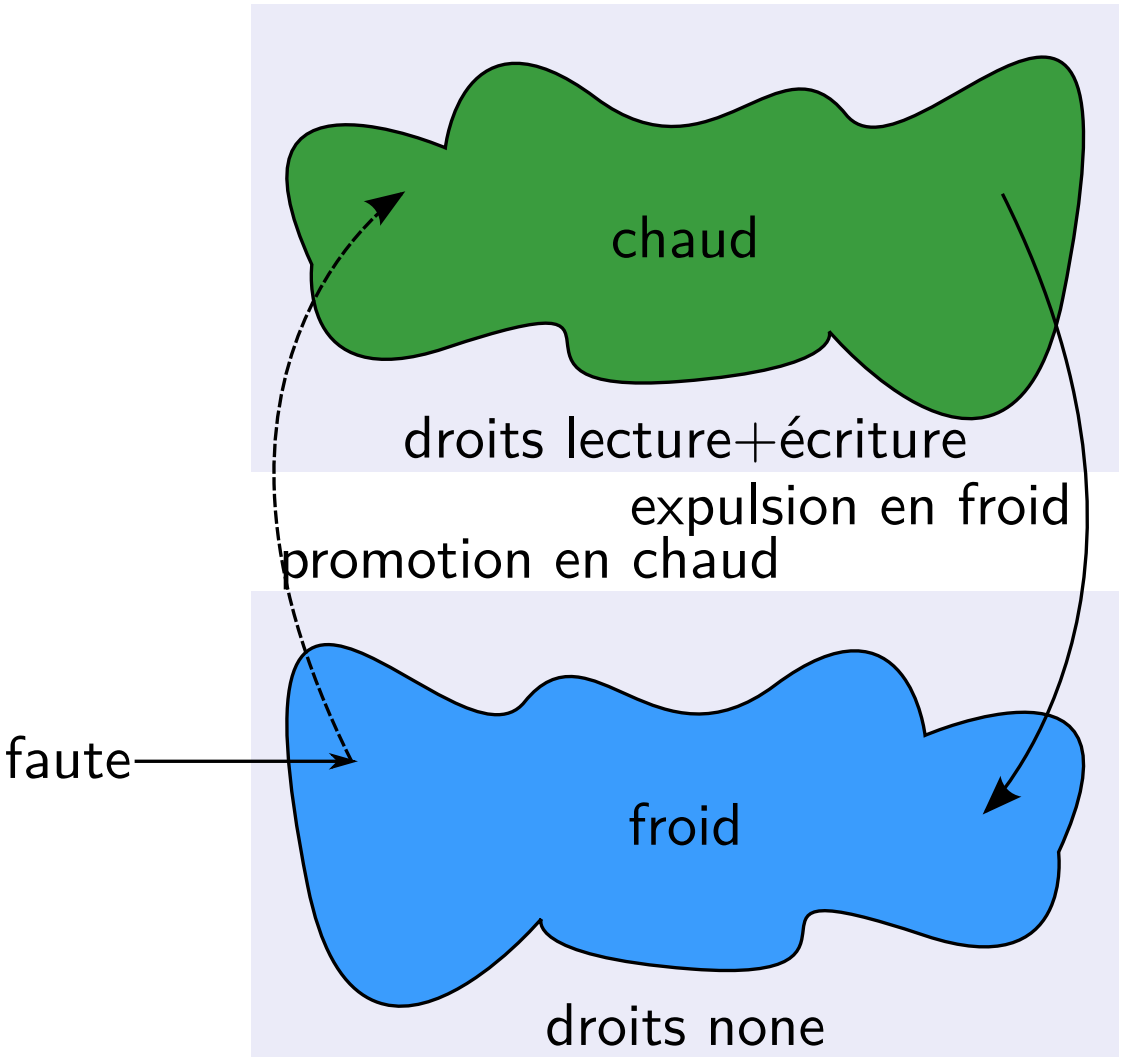
les problèmes

1. le support du `mmap(2)` est un fichier, donc les modifications sont propagées sur le disque
   $\Rightarrow$ coût d'accès au disque est colossal avec des grands tas
   Solution : allouer le fichier en mémoire
2. les fichiers sont traités différemment par Linux
   $\Rightarrow$ les pages sont swappées en priorité
   Solution : utiliser des fonctions de verrouillage de pages

Identification des parties chaudes/froides du tas

Identification des parties chaudes/froides du tas

- identifier l'ensemble des pages utiles à l'application à un instant donné

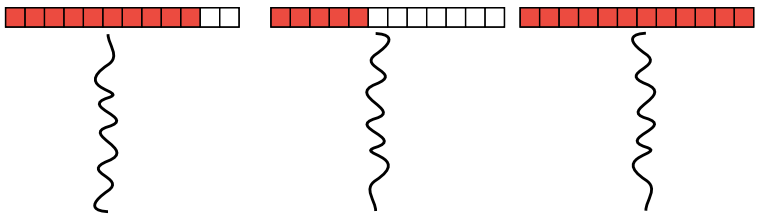


Identification des parties chaudes/froides du tas

- deux mises en œuvres :
  - avec une liste FIFO
  - avec une liste LRU

Identification des parties chaudes avec une liste FIFO

- utiliser une liste FIFO dans le contexte suivant :
  1. protéger la totalité du tas, avec `mprotect(2)` (`PROT_NONE`)
  2. dé-protéger des pages à la demande
     en les ajoutant dans la liste (`PROT_READ|PROT_WRITE`)
  3. si la liste est pleine
     alors enlever une page de la tête et la protéger



- les pages qui sont dans la liste constituent la partie chaude du tas

Identification des parties chaudes avec une liste FIFO

aspects techniques

dé-protection à la demande

- attraper le signal `POSIX` de violation de segment (`SIGSEGV`)
- vérifier que l'adresse fait partie des pages du tas
  très rapide, calcul arithmétique sur l'intervalle des adresses du tas
- modifier les protection avec `mprotect(2)`
- mettre à jour la liste FIFO

Identification des parties chaudes avec une liste LRU

- pages récemment accédées probablement accédées dans le futur
- utilisation une liste LRU à la place d'une liste FIFO
- idée :
  - Le noyau Linux gère déjà une liste LRU des pages pour le swapper :
    $\rightarrow$ utiliser la liste LRU du swapper noyau
- développement d'un module Linux pour communiquer la liste au ramasse-miettes

Identification des parties chaudes avec une liste LRU

aspects techniques (1)

- interface `sysfs` du module (`/sys/module/<module>/state/`) :
- Commandes

`register`	enregistrement un `pid`
`unregister`	désenregistrement un `pid`
`update`	demande d'une mise à jour de la LRU

- Résultats

`clients/`	le répertoire contient les `pids` enregistrés
`clients/<pid>`	fichier contient la dernière version de la LRU mise à jour par `update`

Identification des parties chaudes avec une liste LRU

aspects techniques (2)

- astuce = utilisation d'un `cgroup` :
  - offre une liste LRU dédiée pour un groupe de processus
  - liste plus petite que la liste globale
  - liste ne contenant que des pages de l'application

  $\rightarrow$ plus rapide de trouver les pages de l'application

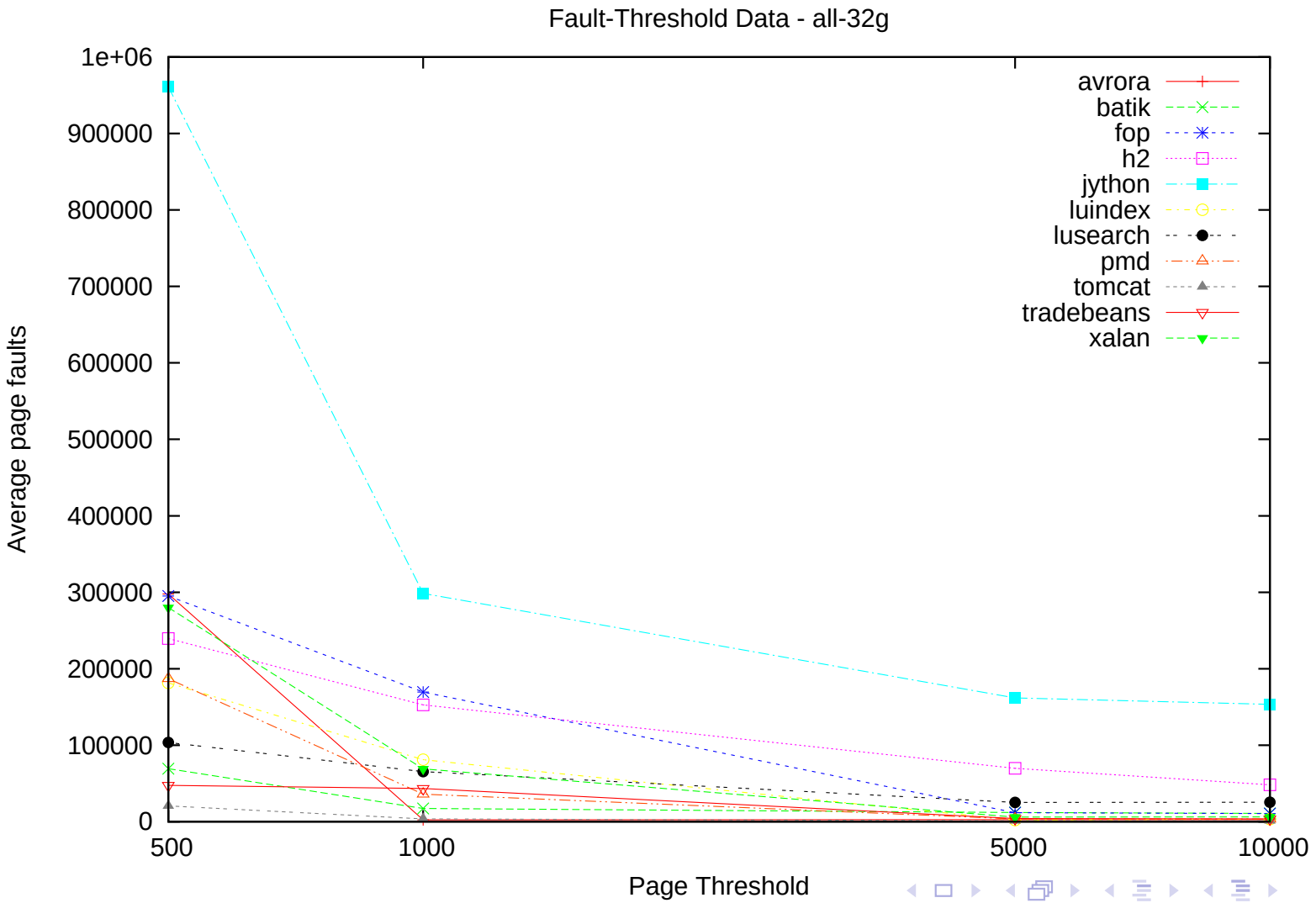
Étude de la localité temporelle sur le benchmark DaCapo

La suite DaCapo

- un outil de benchmark pour Java
- un ensemble des applications avec de charges mémoires non-triviales

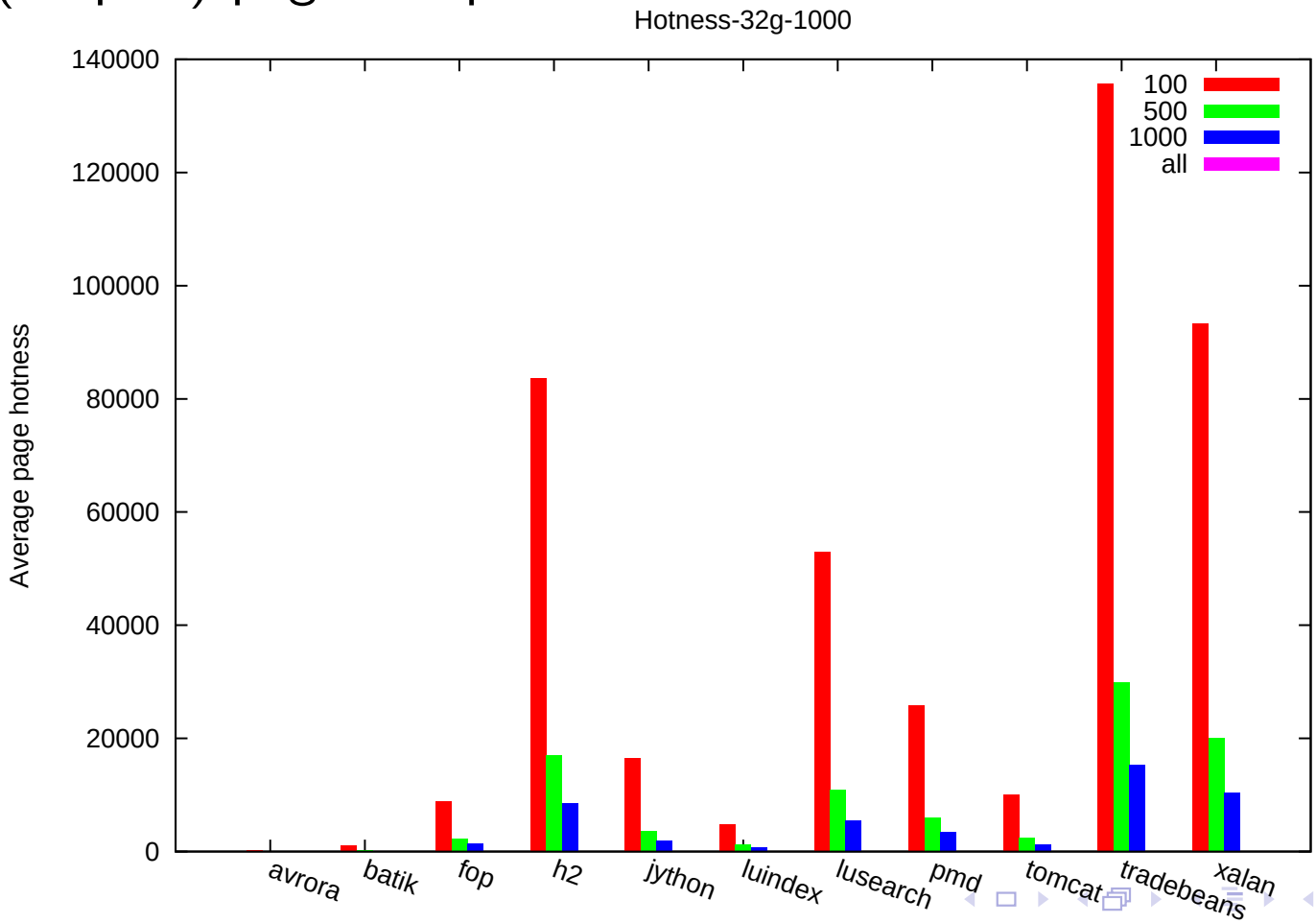
Étude de la localité temporelle sur le benchmark DaCapo

- les applications possèdent une bonne localité temporelle : environ 1000 pages seulement sont utiles par thread à un instant donné



Étude de la localité temporelle sur le benchmark DaCapo

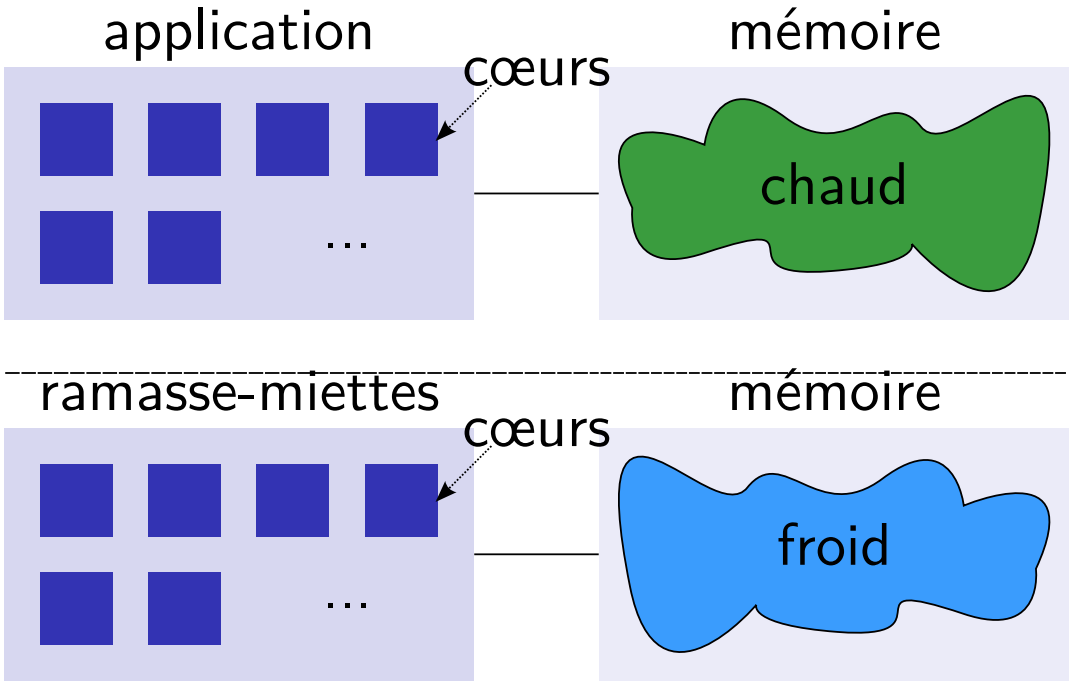
- nombre de fois où les N pages les plus chaudes sont entrées dans la liste FIFO
  $\Rightarrow$ les 100 pages plus chaudes sont beaucoup plus accédées que les 500 (ou plus) pages les plus chaudes



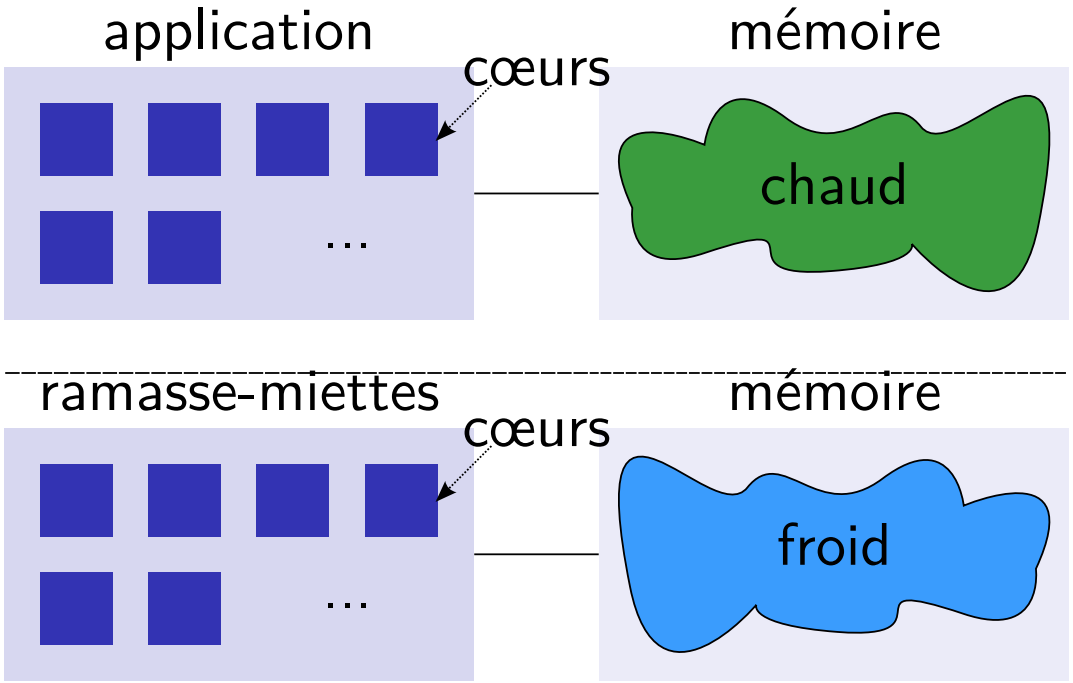
Conclusion

5 Conclusion

Conclusion


application
cœurs
mémoire
chaud
...
ramasse-miettes
cœurs
mémoire
froid
...

Conclusion



- ce qui reste à faire :
  - modifier HotSpot pour communiquer avec le module
  - modifier le ramasse-miettes pour collecter dans le tas répliqué

Merci