Simulation d'un cache et de diverses stratégies de remplacement de bloc

Jean-Paul Rigault

Université de Nice Sophia Antipolis
Polytech'Nice Sophia — Département de sciences informatiques
Sophia Antipolis, France

Email: jpr@polytech.unice.fr

SI3 : projet C-Système, Juin 2015

2 juin 2015

TABLE DES MATIÈRES

1 OBJECTIF

Le but de l'exercice est d'explorer la gestion de cache (et de mémoire virtuelle, car de nombreux problèmes sont identiques) grâce à un **simulateur de cache**. Votre mission est de programmer, d'analyser et de comparer différents **algorithmes de remplacement de bloc**. Vous aurez aussi à implémenter (en fait ré-implémenter) le simulateur de cache lui-même.

En effet, on vous fournit un simulateur opérationnel essentiellement sous forme d'une bibliothèque binaire. Ce simulateur implémente une seule stratégie de replacement (RAND, voir 3.1). Votre travail comporte trois aspects :

- compléter les stratégies de remplacement de blocs en implémentant celles qui sont décrites en 3 ;
- remplacer les modules de la bibliothèque binaire fournie par votre propre implémentation afin d'obtenir un code complètement indépendant de cette bibliothèque ;
- faire tourner le simulateur et ces différentes stratégies de remplacement sur les tests fournis (6.1) ;
- éventuellement inventer et étudier de nouveaux tests ;
- enfin, essayer d'analyser vos résultats et de comparer les différentes stratégies.

L'énoncé semble long, mais il n'est pas nécessaire de tout lire en détail pour commencer à travailler ! Commencez par lire avec soin la section 2 qui présente le problème, la section 4 qui décrit le simulateur fourni et la manière de le regénérer et le modifier ainsi que 7.1 qui décrit le travail à effectuer. Parcourez le reste rapidement. Certes, la section 3 est fondamentale puisqu'elle décrit le cœur du sujet, mais il vous suffit de la lire au fur et à mesure que votre travail avance. La section 5 est une sorte de manuel de référence sur le code fourni, nécessaire puisque le source ne vous en est pas donné ; vous vous y reporterez quand vous aurez besoin des fonctions qui y sont décrites ou lorsque vous les implémenterez vous même. Enfin la lecture de la section 6 ne sera vraiment utile que lorsque vous aurez réalisé toutes les stratégies de remplacement demandées et que vous ferez tourner le simulateur.

2 PRÉSENTATION DU CACHE

On suppose qu'un programme a besoin de lire et d'écrire dans un gros fichier binaire. Vu par le programme, le fichier est composé d'*enregistrements* de taille fixe (`recordsz`). Le programme accède à ce fichier plus ou moins aléatoirement, un enregistrement à la fois, en fournissant d'une part un buffer (de taille `recordsz`) pour cet enregistrement et d'autre part l'indice de l'enregistrement visé (`irfile`) dans le fichier. Le fichier est donc considéré comme un tableau d'enregistrements, chaque enregistrement étant repéré par son index `irfile`).

Le fichier étant vraiment très gros, on décide d'essayer d'améliorer les performances d'accès en interposant un cache entre le buffer de l'utilisateur et le fichier (figure 1).

Afin de minimiser encore le nombre d'entrées-sorties, l'unité de bloc du cache ne sera pas l'enregistrement lui-même, mais un *bloc* de `nrecords` enregistrements consécutifs. Le cache comporte `nblocks` blocs de ce type (pour une taille totale du cache de `nblocks`×`nrecords`

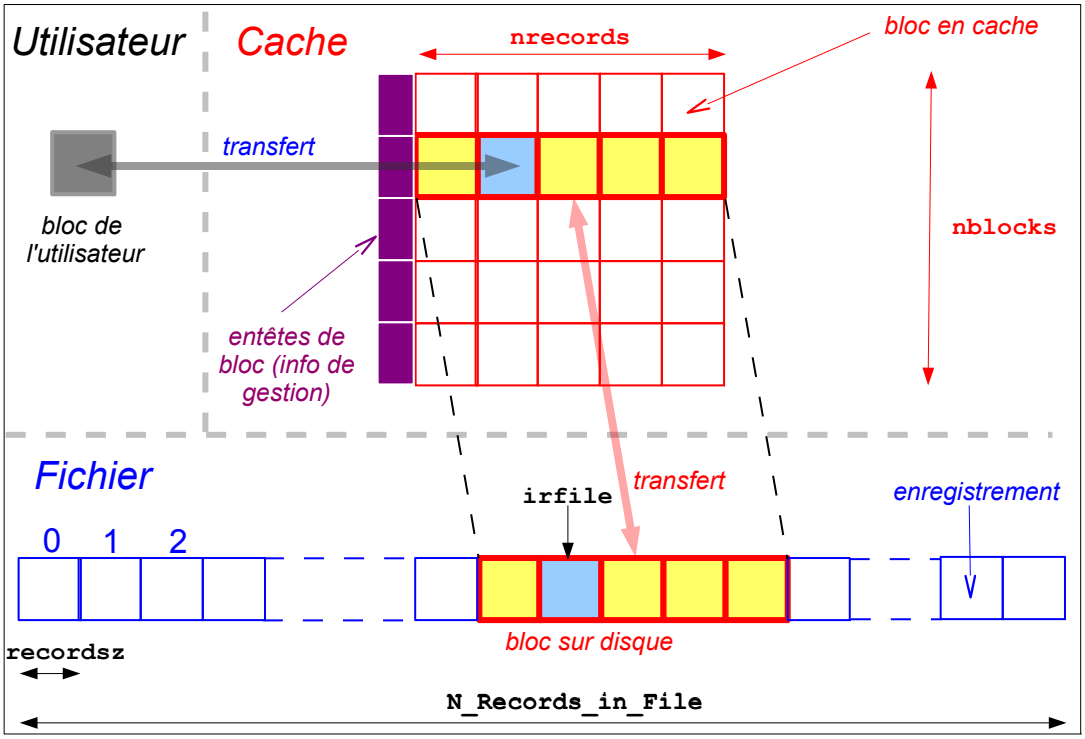

FIGURE 1 – Schéma général du cache et de son fichier.

enregistrements ou encore `nblocks`×`nrecords`×`recordsz` bytes). Le bloc sera l'*unité de transfert* entre le cache et le fichier. Le fichier lui-même est découpé en blocs de même structure et de mêmes tailles. Un bloc du cache peut donc être *affecté* (c'est-à-dire être une copie, une image) d'un bloc sur disque.

À chaque bloc du cache est associé un *entête* qui contient les informations nécessaires à la gestion du bloc. Parmi ces informations, on y trouve un certain nombre de *flags*, c'est-à-dire d'indicateurs booléens (en fait des bits). En particulier, deux de ces *flags* jouen un rôle fondamental dans la gestion du cache :

- l'*indicateur de validité* $\mathbf{V}$: $\mathbf{V} = 0$ indique que le bloc est *libre* c'est-à-dire utilisable (on dit aussi *invalide* car le bloc ne contient pas d'information utile) ; si $\mathbf{V} = 1$, le bloc est *valide* c'est-à-dire contient une copie utile d'un bloc analogue du fichier ;
- l'*indicateur de modification* $\mathbf{M}$: $\mathbf{M} = 1$ indique que le bloc a été modifié depuis qu'il réside dans le cache (on dit aussi que le bloc est sale, *dirty*) ; si un tel bloc est réutilisé (remplacé), il devra être réécrit au préalable dans le bloc correspondant du fichier.

Lors de l'initialisation du cache, tous ses blocs sont marqués invalides (libres) et, bien sûr, non modifiés : $\mathbf{V} = 0$ et $\mathbf{M} = 0$,

L'algorithme de gestion du cache est bien connu :

- L'utilisateur fournit le numéro de l'enregistrement (`irfile`) qu'il veut lire ou écrire ainsi que l'adresse d'un buffer dans son propre espace d'adressage ;
- On regarde si cet enregistrement est dans un bloc valide du cache (il ne peut être que dans au plus un seul bloc) ; si oui on transfère une copie de l'enregistrement depuis

le cache vers le buffer de l'utilisateur pour une lecture, ou en sens inverse pour une écriture (dans le second cas on positionne l'indicateur **M** du bloc à 1) ; la requête de l'utilisateur n'induit alors aucune entrée-sortie ;
- Si le cache ne contient pas l'enregistrement demandé, on cherche un bloc libre (i.e., invalide $\mathbf{V} = 0$) dans le cache et y copie tout le bloc du fichier contenant l'enregistrement d'index (`irfile`) ; on peut alors effectuer le transfert de ou vers le buffer de l'utilisateur ; on laisse bien entendu le bloc dans le cas pour le cas où il serait accédé ultérieurement ;
- Si l'opération précédente n'est pas possible car le cache est plein (i.e., tous ses blocs sont valides), on libère un des blocs du cache pour y copier le bloc disque et donc ainsi changer son affectation (le nouveau bloc est alors marqué valide $\mathbf{V} = 1$ et non modifié $\mathbf{M} = 0$) ; sélectionner le bloc à libérer est le rôle de l'*algorithme de remplacement* de bloc ; bien entendu, si le bloc à libérer a été modifié pendant sa durée de résidence (ou, de manière équivalente, de validité) dans le cache, il faut le réécrire sur disque avant de changer son affectation.

Plusieurs stratégies de remplacement de bloc existent et leur choix et leur réglage constituent le critère principal d'efficacité du cache.

Enfin, pour assurer la sécurité des données, de manière régulière, on **synchronise** le contenu du cache avec celui du fichier en écrivant sur disque tous les blocs qui ont été modifiés (et on remet à 0 leur bit **M**). Dans notre cas, ceci s'effectue tous les `NSYNC` (une constante) demandes d'accès (lecture ou écriture) au cache (ou au fichier, c'est pareil) [1].

3 ALGORITHMES DE REMPLACEMENT DE BLOC

L'algorithme de remplacement de bloc joue un rôle évidemment crucial dans les performances du cache. Comme indiqué précédemment, il n'est invoqué que lorsque le cache est plein (tous ses blocs sont valides) et qu'il faut donc faire la place pour un nouveau bloc disque.

La stratégie de remplacement optimale est connue (principe d'optimalité de Peter DENNING) : on doit remplacer le bloc qui sera utilisé à nouveau par le programme au bout du temps le plus long. Malheureusement, cette stratégie n'est pas causale et la détermination de ce bloc optimal est en fait indécidable.

On est donc conduit soit à ignorer complètement le principe d'optimalité, soit à en utiliser des approximations. De nombreux algorithmes ont été proposés, analysés et appliqués sur des systèmes réels. Nous en étudierons quatre ici.

3.1 Remplacement de bloc au hasard (RAND)

Ici, on ignore complètement le principe d'optimalité, puisque l'on tire au sort (*random*) le bloc à remplacer. L'implémentation de cet algorithme vous est donnée à titre indicatif (fichier `NUR_strategy.c`), mais il est clair qu'ils ne donne pas de bons résultats en général et qu'il est très peu utilisé, voire pas du tout.

1. Dans les vrais systèmes c'est plutôt un intervalle de temps qui est utilisé, mais dans notre simulateur, le temps n'est pas vraiment une quantité significative.

L'implémentation de trois autres algorithmes bien connus (FIFO, LRU, NUR) est une de vos missions pour ce projet.

3.2 Remplacement du bloc le plus ancien (FIFO)

Dans l'algorithme FIFO (*First In, First Out*), on joue sur le temps de résidence d'un bloc (son âge) : on remplace le bloc le plus vieux du cache (celui qui y est monté dans le cache depuis le plus longtemps). Ici, les vieux ont donc un avenir assez compromis ! (Mais on pourra quand même aller les rechercher, si besoin, ce sera juste un peu plus long.)

Pour implémenter l'algorithme FIFO, on a besoin d'ajouter au cache une structure de données supplémentaire, une liste de pointeurs sur les blocs valides du cache. Chaque fois qu'un bloc du cache change d'affectation (soit il passe de l'état non valide à valide, soit il est affecté à un autre bloc du disque), on le transfère en queue de liste.

Le bloc le plus anciennement affecté se trouve donc en tête [2] de la liste FIFO et c'est lui qui sera remplacé.

3.3 Remplacement du bloc le moins récemment utilisé (LRU)

Dans l'algorithme LRU (*Least Recently Used*), on remplace le bloc qui a été le moins récemment accédé. C'est une approximation du principe d'optimalité, considérant que le passé proche est une préfiguration du futur pas trop lointain.

Comme pour FIFO, son implémentation requiert une liste de pointeurs sur les blocs valides du cache, et chaque fois qu'un bloc change d'affectation, on le transfère en queue de liste. Mais on effectue également ce transfert vers la queue chaque fois qu'un bloc est utilisé, c'est-à-dire chaque fois qu'il est atteint par une opération de lecture ou d'écriture.

Ainsi le bloc en tête de la liste LRU est-il le bloc le moins récemment utilisé, et c'est lui qui sera remplacé.

3.4 Remplacement d'un bloc non utilisé récemment (NUR)

C'est une variation sur l'idée de l'algorithme précédent et cela en constitue une approximation. NUR signifie *Not Used Recently* : on choisit donc un bloc qui n'a pas (de préférence, *pas du tout*) été utilisé récemment. Celui qui semble ne plus servir à rien est viré ! (Peut-être reviendra-t-il, mais, là encore, ce sera plus long.)

L'avantage de cet algorithme est sa légèreté de mise en œuvre [3]. En effet, comparé aux deux algorithmes précédents qui demandent une structure de données supplémentaire assez lourde à gérer, NUR ne demande qu'un seul bit par bloc du cache, que nous noterons **R**. Ce *bit de référence* est utilisé conjointement avec le bit de modification **M** déjà présent dans l'en-tête du bloc (voir 2) [4].

Le bit **R** est mis à 1 chaque fois que le bloc est référencé (accédé en lecture ou en écriture). De plus, pour réaliser la notion de « récemment », le bit **R** de tous les blocs du cache est remis

2. Dans cette description ainsi que dans celle de l'algorithme LRU, on peut sans dommage échanger le rôle de la tête et de la queue de la liste.

3. Tellement légère que cet algorithme est implémenté dans certaines MMU (*Memory Management Unit*), donc au niveau matériel.

4. Noter que le bit **V** ne joue aucun rôle dans les algorithmes de remplacement puisque, par définition, ces algorithmes ne sont activés que lorsque tous les blocs sont valides.

à 0 à intervalle régulier. Dans notre implémentation, c'est tous les `nderef` accès, `nderef` étant un paramètre constant de configuration, que cette remise à 0 se fera [5]. Donc, à un instant donné, nous avons dans le cache quatre types de blocs valides :

R	M	$2 \times \mathbf{R} + \mathbf{M}$	**État du bloc**
0	0	0	pas utilisé pendant le dernier intervalle, pas modifié
0	1	1	pas utilisé pendant le dernier intervalle, modifié
1	0	2	utilisé pendant le dernier intervalle, pas modifié
1	1	3	utilisé pendant le dernier intervalle, modifié

L'algorithme NUR consiste à choisir de remplacer en priorité les blocs dont le bit **R** est nul (ils n'ont pas été utilisé « récemment », c'est-à-dire pendant le dernier intervalle) ; dans ceux-là, on préférera ceux dont le bit **M** est nul, car on gagne ainsi une écriture sur disque. Évidemment, si on ne trouve pas de tels blocs, on est contraint de remplacer un des blocs avec $\mathbf{R} = 1$ (toujours en donnant la priorité à ceux qui n'ont pas été modifiés) [6]. Dit de manière compacte, l'algorithme NUR consiste donc à remplacer l'un des blocs du cache qui minimise le nombre entier $n = 2 \times \mathbf{R} + \mathbf{M}$.

4 CODE FOURNI

4.1 Architecture générale et structure des fichiers

Le programme de gestion du cache est organisé de manière modulaire, en quatre couches (figure 2). Cette modularité permet à l'application d'ignorer les détails d'implémentation du cache, et à la gestion générale de ce dernier d'être indépendante de la stratégie de remplacement utilisée. Cette dernière propriété autorise la substitution d'une stratégie par une autre sans modifier le reste du code.

Les fichiers suivants sont fournis dans l'archive `Cache_Project.zip` :

- une `Makefile` ;
- les fichiers-sources, en bleu sur la figure 2, `tst_Cache.c` (programme principal et tests) et `RAND_strategy.c` (implémentation de la stratégie de remplacement au hasard ;
- les fichiers-objets, en noir sur la figure 2, `cache.o` (interface utilisateur et gestion générale du cache, indépendante de la stratégie de remplacement), `low_cache.o` (implémentation interne du cache, interface avec la stratégie de remplacement mais indépendante de l'implémentation de cette dernière) et `cache_list.o` (une simple liste de blocs de cache pour les stratégies LRU et FIFO) sont regroupés dans une bibliothèque binaire statique `libCache.a` ;
- les fichiers d'entête (`*.h`), en vert sur la figure 2, assurent l'interfaçage entre ces différents modules ; vous n'êtes pas censés les modifier mais vous devez respecter les spécifications qu'ils constituent de fait.

5. Dans un vrai système, cette remise à 0 se fait avec une période temporelle constante (plusieurs fois par seconde), mais on se rappelle que le temps n'est pas significatif dans notre simulateur.

6. Dans ce dernier cas, peu importe le bloc que l'on retourne, du moment qu'il correspond à un minimum de $2 \times \mathbf{R} + \mathbf{M}$. On pourrait évidemment tenter d'être plus sélectif, mais alors il faudrait sans doute mettre en place des structures de données supplémentaires, ce qui risquerait de réduire l'avantage de NUR, sa légèreté.

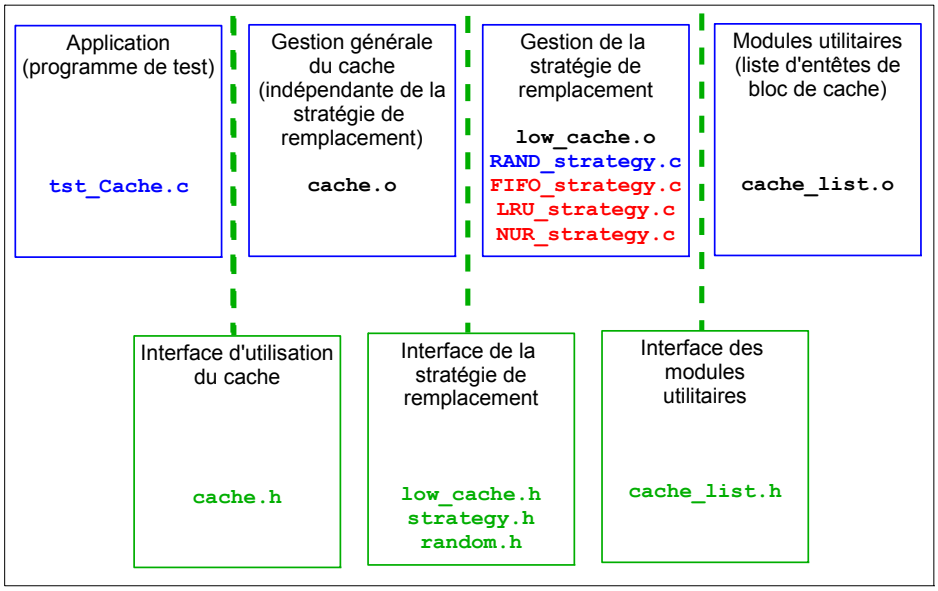

FIGURE 2 – Architecture générale du simulateur de cache.

En ce qui concerne les stratégies de remplacement, c'est à vous d'en écrire les fichiers de source de noms respectifs `FIFO_strategy.c`, `LRU_strategy.c` et `NUR_strategy.c` (en rouge sur la figure 2). De même, vous devrez écrire vos propres versions des fichiers `cache.c`, `low_cache.c`, et `cache_list.c` dont les fichiers-objets se substitueront à ceux de la bibliothèque `libCache.a`.

Remarque

D'autres fichiers sont fournis dans l'archive `Cache_Project.zip` :

- les fichiers `*_default.out` correspondent aux résultats de ma propre version du simulateur, pour les différentes stratégies ; ils sont fournis à titre indicatif ;
- un script **shell** principal `plot.sh` et son compagnon `cache_size-inv.sh` permettent de tracer des courbes ; l'usage de `plot.sh` est décrit en 6.4 [a] ;

a. `cache_size-inv.sh` est invoqué uniquement par `plot.sh`

Attention

Respectez les noms des fichiers et le contenu des fichiers d'entête, sinon la `Makefile` fournie ne fonctionnera pas ou la construction des exécutables échouera.

> **Remarque**
> La `Makefile` reconstruit automatiquement les dépendances entre fichiers dans un fichier `depend.out` (voir la règle `depend`). Ignorer le warning sur l'absence de `depend.out` la première fois que vous exécutez **make**.

4.2 Reconstruction des exécutables : utilisation de la `Makefile`

À chaque stratégie de remplacement correspond un binaire exécutable de nom

`tst_Cache_`*XXX*

où *XXX* représente la stratégie de remplacement choisie (RAND, FIFO, LRU, NUR). Dans la `Makefile` ces exécutables sont construits grâce à une règle par défaut d'édition de liens qui a la forme suivante :

```
tst_Cache_% : tst_Cache.o %_strategy.o $(USRFILES)
	$(CC) -o $@ $^ libCache.a
```

Le « joker » `%` désigne ici encore la stratégie de remplacement choisie. La variable de **make** `USRFILES` est initialement vide. L'entrée `all` de ma `Makefile` a initialement la forme suivante :

```
all: tst_Cache_RAND
```

et donc la simple commande au shell

```
make
```

va construire l'exécutable `tst_Cache_RAND`. Lorsque vous aurez implémenté les autres stratégies de remplacement, il faut suffira d'ajouter les exécutables correspondant à la suite de la définition de `all`. À la fin, vous aurez

```
all: tst_Cache_RAND tst_Cache_FIFO tst_Cache_LRU tst_Cache_NUR
```

Par ailleurs, lorsque vous aurez réimplémenté un ou plusieurs des fichiers-sources de la bibliothèque `libCache.a`, il vous suffira d'ajouter le ou les noms des fichiers-objets(`*.o`) correspondants à la suite définition de la variable `USRFILES`

```
USRFILES = cache.o low_cache.o
```

et ces nouveaux fichiers-objets seront choisis par l'éditeur de liens de préférence à ceux de `libCache.a`.

5 RÉFÉRENCE DU CODE DU SIMULATEUR DE CACHE

5.1 Interface entre le cache et l'application (`cache.h` et `cache.o`)

5.1.1 Interface de base

L'interface entre l'application et le cache est définie dans le fichier `cache.h`, qui contient essentiellement une liste de prototypes de fonctions externes :

`struct Cache *Cache_Create(const char *fic, unsigned nblocks,`
`unsigned nrecords, size_t recordsz, unsigned nderef);`

Crée un cache associé au fichier de nom `fic` : la cache comporte `nblocks`, chaque bloc contenant `nrecords` enregistrements de taille `recordsz` caractères.

Le dernier paramètre (`nderef`) n'est utilisé que pour la stratégie NUR ; pour les autres stratégies sa valeur est ignorée. Dans le cas de NUR, le bit de référence **R** devra être remis à 0 (pour tous les blocs du cache) tous les `nderef` accès (lecture ou écriture).

La fonction ouvre le fichier en le créant et le remplissant si nécessaire, alloue et initialise les structures de données du cache, et retourne un pointeur sur le nouveau cache.

`int Cache_Close(struct Cache *pcache);`

Détruit le cache pointé par `pcache` : synchronise le cache et le fichier grâce à `Cache_Sync()`, ferme le fichier et détruit toutes les structures de données du cache.

`int Cache_Sync(struct Cache *pcache);`

Synchronise le contenu du cache avec celui du fichier : écrit sur disque tous les blocs dont le bit **M** vaut 1 et remet à 0 ce bit. L'application peut appeler `Cache_Sync()` quand elle le souhaite, mais il y a un appel automatique tous les `NSYNC` accès (par défaut `NSYNC` vaut 1000, défini dans `low_cache.c`).

`int Cache_Invalidate(struct Cache *pcache);`

Invalide le cache, c'est-à-dire met à 0 le bit **V** de tous les blocs. C'est donc comme si le cache était vide : aucun bloc ne contient plus d'information utile.

Noter que cette fonction `Cache_Invalidate()` ne devrait pas faire partie de l'interface utilisateur du cache. Néanmoins, elle est nécessaire au simulateur, puisqu'elle permet d'enchainer des tests différents sans avoir à réallouer le cache.

`int Cache_Read(struct Cache *pcache, int irfile, void *precord);`
`int Cache_Write(struct Cache *pcache, int irfile, const void *precord);`

Lecture (resp. écriture) à travers le cache de l'enregistrement d'indice `irfile` dans le fichier. Le paramètre `precord` doit pointer sur un buffer fourni par l'application et au moins de taille `recordsz`. L'enregistrement sera transféré du cache dans ce buffer pour une lecture (resp. du buffer vers le cache pour une écriture).

5.1.2 Instrumentation du cache

Dans le fichier `cache.h` est également définie la structure `Cache_Instrument` contenant un certain nombre de compteurs permettant de collecter des statistiques sur le fonctionnement interne du cache :

```
struct Cache_Instrument
{
    unsigned n_reads;    // Nombre de lectures
    unsigned n_writes;   // Nombre d'écritures
    unsigned n_hits;     // Nombre de fois où l'enregistrement
                         // était déjà dans le cache
    unsigned n_syncs;    // Nombre d'appels à Cache_Sync()
    unsigned n_deref;    // Nombre de déréférençage (NUR)
};
```

Le plus important de ces compteurs est `n_hits`, le nombre de succès, c'est-a-dire le nombre d'accès pour lesquels l'enregistrement était déjà dans le cache. Ainsi que nous l'avons déjà mentionné, le taux de succès (*hit rate*) est le principal critère d'évaluation des performances du cache :

```
hit_rate = n_hits / (n_reads + n_writes)
```

Noter que `hit_rate` est un nombre réel, toujours inférieur ou égal à 1.

Tous ces compteurs, sauf un, sont mis à jour par l'algorithme général de gestion du cache (dans `cache.c`). L'exception est `n_deref`, qui compte le nombre de fois où l'on a remis à 0 le bit R pour l'ensemble du cache, dans la stratégie NUR. Il est souhaitable que vous le gériez (cependant, ce n'est pas indispensable pour le fonctionnement de l'ensemble).

La fonction `Get_Cache_Instrument()` récupère (un pointeur sur) une copie des statistiques courantes :

`struct Cache_Instrument *Get_Cache_Instrument(struct Cache *pcache);`

Retourne une copie de la structure d'instrumentation du cache pointé par `pcache`. **Attention** : tous les compteurs de la structure courante sont remis à 0 par cette fonction.

> **Remarque**
>
> Dans votre propre réalisation de `cache.c` vous avez le droit d'ajouter à cette structure des informations supplémentaires qui vous semblent pertinentes. Dans ce cas, vous devrez aussi modifier la fonction `Print_Instrument()` de `tst_Cache.c`.

5.2 Structure interne du cache (`low_cache.h` et `cache.o`)

Les détails internes du cache sont définis dans le fichier `low_cache.h` sous forme de deux structures C. La structure `Cache_Block_Header` défini un bloc du cache avec son en-tête et ses données :

```
struct Cache_Block_Header
{
    unsigned int flags; // Indicateurs d'état
    int ibfile;         // Index de ce bloc dans le fichier
    int ibcache;        // Index de ce bloc dans le cache
    char *data;         // Les données (nrecords enregistrements)
};
```

Les flags servent à ranger en particulier les deux bits **M** et **V**. Ceux-ci correspondent respectivement aux constantes `VALID` et `MODIF` définies dans le même fichier. Vous pouvez également utiliser ce champ flags pour y mettre le bit **R** correspondant à la stratégie NUR (mais n'empiétez pas sur **M** et **V** !).

Le cache lui-même correspond à la structure `Cache` et contient un tableau d'en-têtes de bloc (alloué dynamiquement et nommé `headers`), plus toutes les informations de configuration :

```
struct Cache
{
    char *file;              // Nom du fichier
    FILE *fp;                // Pointeur sur fichier
```

```
    unsigned int nblocks;    // Nb de blocs dans le cache
    unsigned int nrecords;   // Nombre d'enregistrements par bloc
    size_t recordsz;         // Taille d'un enregistrement
    size_t blocksz;          // Taille d'un bloc
    unsigned int nderef;     // Période de déréférençage pour NUR
    void *pstrategy;         // Structure de données dépendant de la stratégie
    struct Cache_Instrument instrument; // Instrumentation du cache
    struct Cache_Block_Header *pfree;   // premier bloc libre
    struct Cache_Block_Header *headers; // Les données elles-mêmes
};
```

Le champ `headers` pointe sur un tableau alloué dynamiquement d'en-têtes de bloc. Comme c'est un tableau, il suffit de l'indexer pour le parcourir (`pcache->headers[i]`).

La fonction `Get_Free_Block()` déclarée dans `low_cache.h` et implémentée dans `cache.c`, permet à la stratégie de demander un bloc libre (c'est-à-dire non valide), s'il en reste :

`struct Cache_Block_Header *Get_Free_Block(struct Cache *pcache);`
: Retourne le premier bloc libre du cache ou le pointeur `NULL` si le cache est plein. Cette fonction doit être invoquée par la stratégie de remplacement avant de considérer l'utilisation d'un bloc valide.

5.3 Interface de la stratégie de remplacement (`strategy.h`)

Comme indiqué en 4.1, la stratégie de remplacement est indépendante de la gestion générale du cache. Cependant cette dernière a besoin d'une interface avec l'algorithme de remplacement. Notons que le seul rôle de l'algorithme de remplacement est de retourner un bloc utilisable par l'algorithme de gestion général (dans `cache.c`) quand celui-ci lui demande. Tout le reste (accès au fichier, entrées-sorties, gestion des bits **M** et **V**) est du ressort de la gestion générale.

L'interface est définie dans le fichier `strategy.h` et ce sont ces fonctions que vous devez réaliser, pour les trois stratégies proposées. Mêmes vides, toutes ces fonctions doivent être implémentées pour chaque stratégie.

`void *Strategy_Create(struct Cache *pcache);`
: Crée et initialise les structures de données spécifiques à la stratégie. Évidemment invoqué par `Cache_Create()`. Dans `Cache_Create()` le pointeur retourné par cette fonction doit être affecté au champ `pstrategy` du cache pointé par `pcache`). Noter que `pstrategy` est un **`void`** `*` ce qui lui permet de pointer sur n'importe quel type. Bien entendu, dans le fichier de stratégie, il devra être forcé (« casté ») dans le type idoine.

`void Strategy_Close(struct Cache *pcache);`
: Libère et détruit les structures de données spécifiques à la stratégie. Évidemment invoqué par `Cache_Close()`.

`void Strategy_Invalidate(struct Cache *pcache);`
: Appelé lors de l'invalidation du cache pour effectuer les éventuelles action spécifiques à la stratégie. Cette fonction est appelée par `Cache_Invalidate()` après que tous les blocs du cache aient été marqués invalides.

`struct Cache_Block_Header *Strategy_Replace_Block(struct Cache *pcache);`
: Retourne un bloc à remplacer. Cette fonction est évidemment invoquée lorsque l'enregistrement cherché ne se trouve pas dans le cache. Le bloc retourné doit être soit

(et en priorité) un des blocs invalides, soit un bloc valide choisi en fonction de la stratégie. Pour chercher un bloc invalide, cette fonction peut utiliser `Get_Free_Block()` décrit précédemment (voir 5.2).

`void Strategy_Read(struct Cache *pcache, struct Cache_Block_Header *pbh);`
`void Strategy_Write(struct Cache *pcache, struct Cache_Block_Header *pbh)`
Effectue les actions spécifiques de la stratégie en cas d'écriture (resp. lecture) du bloc pointé par `pbh`. Ces fonctions sont invoquées tout à fait à la fin des fonctions de lecture et d'écriture du cache (`Cache_Read()` et `Cache_Write()`).

`char *Strategy_Name();`
Retourne une chaîne de caractères identifiant la stratégie, quelque chose comme `"LRU"` ou `"NUR"`.

5.4 Liste de blocs (`cache_list.h` et `cache_list.o`)

Les algorithmes de remplacement FIFO et LRU ont besoin d'une liste de blocs, en fait d'une liste de pointeurs sur entête de bloc. Une telle liste est définie pour vous, avec son interface d'utilisation dans `cache_list.h` :

`struct Cache_List *Cache_List_Create();`
Crée et initialise une nouvelle liste (vide) et retourne un pointeur dessus.

`void Cache_List_Delete(struct Cache_List *list);`
Détruit la liste pointée par `list`.

```
void Cache_List_Append(struct Cache_List *list,
                       struct Cache_Block_Header *pbh);
void Cache_List_Prepend(struct Cache_List *list,
                       struct Cache_Block_Header *pbh);
```
Crée une nouvelle cellule de liste contenant le pointeur `pbh` et l'insère en queue (resp. en tête) de la liste.

```
struct Cache_Block_Header*Cache_List_Remove_First(
                       struct Cache_List *list);
struct Cache_Block_Header *Cache_List_Remove_Last(
                       struct Cache_List *list);
```
Retourne le pointeur sur le premier (resp. le dernier) entête de la liste et détruit la cellule correspondante de la liste. Retourne `NULL` si la liste est vide.

```
struct Cache_Block_Header *Cache_List_Remove(struct Cache_List *list,
                       struct Cache_Block_Header *pbh);
```
Cherche la cellule contenant le pointeur `pbh` et la retire de la liste. Retourne `pbh` (ou `NULL` si la liste est vide ou si elle ne contient pas `pbh`).

`void Cache_List_Clear(struct Cache_List *list);`
Détruit toutes les cellules de la liste, qui redevient donc vide.

`bool Cache_List_Is_Empty(struct Cache_List *list);`
Retourne **`true`** si la liste est vide, **`false`** sinon.

```
void Cache_List_Move_To_End(struct Cache_List *list,
                       struct Cache_Block_Header *pbh);
```

```
void Cache_List_Move_To_Begin(struct Cache_List *list,
                              struct Cache_Block_Header *pbh);
```

Cherche la cellule contenant le pointeur `pbh` et la transfère en queue (resp. en tête) de liste. Si la liste ne contient pas `pbh`, ce dernier est ajouté à la fin (respectivement au début) de la liste. Si `pbh` est déjà à sa position de destination, rien ne se produit.

5.5 Divers (`random.h`)

Le fichier `random.h` contient la fonction (**`inline`**) `RANDOM(m, n)` qui constitue la manière correcte de tirer un nombre entier au hasard dans l'intervalle `[m, n[` (`m` inclus, `n` exclus) en utilisant la fonction `rand()` de la bibliothèque C. Affichez donc **man 3 rand** si vous souhaitez des détails.

Cette fonction est utilisée par certains des tests du programme principal (`main_Cache.c`) ainsi que par la stratégie de remplacement au hasard (`RAND_strategy.c`).

6 SIMULATIONS

L'intérêt de cet exercice est de comparer les performances de la gestion de cache suivant plusieurs critères : dimensionnement du fichier et du cache, comportement local ou non des programmes, paramètres de la stratégie, etc.

À cette fin, le programme principal exécute un certain nombre de boucles de test qui visent à simuler divers comportements de programme. Il a aussi été doté de nombreuses options qui permettent de faire varier certains des paramètres de dimensionnement ou de stratégie.

6.1 Boucles de test du programme de simulation

Il y a autant de programmes binaires exécutables de test que de stratégies de remplacement, mais ils exécutent les mêmes tests. Chacun exécute consécutivement cinq boucles correspondant à différentes caractéristiques de localité (la propriété d'un programme à agréger ses références à la mémoire — ici ses références au fichier — dans la même zone).

Test 1 : boucle de lecture séquentielle La boucle parcourt complètement et séquentiellement les enregistrements du fichier en écrivant l'enregistrement courant et en lisant le précédent. Ce test possède une bonne localité (la séquentialité est un élément d'icelle) et vos taux de succès devraient être bons (de l'ordre de 95 % ou mieux, avec les paramètres par défaut).

Test 2 : boucle d'écriture aléatoire On tire au sort (uniformément) le numéro de l'enregistrement à lire. On effectue cela un certain et grand nombre de fois (`N_Loops`). La localité est évidemment mauvaise et votre taux de succès devrait être très bas, surtout avec un petit cache. En fait vous devriez même être capable de prédire de taux de succès a priori.

Test 3 : boucle de lecture/écriture aléatoire Ce test est analogue au précédent avec cependant deux différences :

— on mélange des lectures et des écritures, en effectuant une écriture toutes les `Ratio_Read_Write` lectures (`Ratio_Read_Write` vaut par défaut 10) ;
— à chaque requête, au lieu d'accéder à un seul enregistrement, on accède à un nombre aléatoire (entre 1 et `N_Seq_Access - 1`, `N_Seq_Access` valant par défaut 5) d'enregistrements consécutifs suivant l'enregistrement courant.

Dans ce test, la localité est améliorée à cause de l'anticipation juste évoquée, mais elle n'est pas vraiment suffisante pour un fonctionnement optimal. Vos taux de succès devraient être de l'ordre de 50 % avec les paramètres par défaut.

Test 4 : boucle de lecture/écriture aléatoire avec localité améliorée Ce test vise à se rapprocher un peu du comportement local des programmes réels. Pour cela, nous découpons notre nombre d'itérations `N_Loops` en un certain nombre de phases de travail ou *Working Sets*. Nous désignons par `N_Working_Sets` ce nombre de phases (100 par défaut). À chaque phase correspondent donc un nombre d'accès `nlocal` tel que

```
nlocal = N_Loops / N_Working_Sets
```

Pour chaque phase, on tire au sort, d'abord le numéro `irfile` d'un enregistrement de base, puis `nlocal` fois un incrément `incr` entre 1 et `N_Local_Window - 1` et on accède à l'enregistrement `irfile + incr`. On reste donc un certain temps (pendant `nlocal` accès) dans la même zone du fichier (voisine de `irfile`) ce qui améliore la localité. Cette zone, de longueur `N_Local_Window`, est appelée ici la *fenêtre de localité*. Dans ce test aussi, le paramètre `Ratio_Read_Write` permet de décider si l'accès est une écriture ou une lecture.

Test 5 : boucle de lecture/écriture séquentielle avec localité améliorée Ce test est analogue au test précédent avec cependant deux différences qui doivent améliorer la localité :
— au lieu de tirer au sort l'enregistrement de base `irfile`, on parcourt le fichier séquentiellement ;
— au lieu de lire parmi enregistrements suivants `irfile`, on lit parmi ceux le précédant.

6.2 Arguments du programme de simulation

Les exécutables de test sont nommés `tst_Cache_`*XXX*, où *XXX* désigne la stratégie de remplacement. Chacun peut être invoqué depuis le **shell** avec un certain nombre d'arguments de la ligne de commande décrits ici. Tous ces arguments ont des valeurs par défaut.

6.2.1 Options générales

-h affiche un message d'aide.

-p affiche les valeurs des paramètres mais n'exécute pas de test

-S format de sortie court, pour les tracés de courbe avec **plot.sh** (voir 6.4).

6.2.2 Options de configuration du cache

-f nom du fichier (défaut : `"foo"`) ; si ce fichier n'existe pas, il est automatiquement créé et rempli.

-N *nrf* *nrf* est le nombre d'enregistrements dans le fichier (défaut : 30 000).

-R *nrb* *nrb* est le nombre d'enregistrements par bloc du cache (`nrecords`) (défaut : 10).

-r *rfc* *rfc* est le rapport (entier) entre la taille du fichier et la taille du cache (défaut ; 100). Donc par défaut, le cache a une taille qui est 1 % de celle du fichier.

6.2.3 Options des boucles de test

-t *n* valide le test de numéro *n*. Il peut y avoir plusieurs options **-t**, auquel cas seuls les tests mentionnées seront réalisés. En l'absence de cette option, tous les tests sont exécutés. Au départ 5 tests sont définis, mais vous pouvez en ajouter d'autres : il faut pour cela modifier `tst_Cache.c`.

-l *v* conditionne le nombre d'itérations (d'accès) dans les tests 2 à 5 (défaut : 3). Ce nombre d'itérations, `N_Loops`, sera $nrf \times v$ (défaut : 90 000).

-w *nrw* pour les tests 3 à 5, il y aura une écriture toutes les *nrw* (`Ratio_Read_Write`) lectures (défaut : 10).

-s *nsa* *nsa* (`N_Seq_Access`) est le nombre maximum de blocs locaux à accéder à proximité du bloc de base, dans le test 3 (défaut : 5).

-W *nws* *nws* (`N_Working_Sets`) est le nombre de phases, ou *Working Sets*, dans les tests 4 et 5 (défaut : 100).

-L *nlw* *nlw* (`N_Local_Window`) est la largeur de la fenêtre de localité dans les tests 4 et 5 (défaut : 300).

-d *ndr* *ndr* (le champ `nderef` du cache, voir 5.2) est la période avec laquelle le bit **R** est remis à 0 dans la stratégie NUR (voir 3.4) (défaut : 100). Cette option n'a de sens que pour la stratégie NUR et elle est silencieusement ignorée pour les autres.

6.3 Exemples de résultats

Les fichiers `tst_Cache_`*XXX*`.out` du répertoire `Cache` (où *XXX* désigne la stratégie de remplacement) présentent la sortie des différents tests avec ma propre réalisation des quatre stratégies. Sauf bug de ma part, toujours possible, vos propres résultats devraient être du même ordre.

6.4 Tracé de courbes

Il est intéressant de comparer les différentes stratégies en faisant varier certains des paramètres, et même de tracer les courbes correspondantes. Tout d'abord, il est conseillé que la taille du fichier soit assez grande afin d'avoir des résultats significatifs tout en ayant des temps d'exécution acceptables sur nos machines. Le défaut de 30000 semble raisonnable pour établir ce compromis. Les autres paramètres peuvent être modifiés librement (et raisonnablement).

Cependant, vous constaterez que les résultats sont délicats à interpréter, et ceci d'autant plus qu'on fait tout bouger en même temps. Il est conseillé de ne faire varier qu'un petit nombre de paramètres à la fois (1 par exemple).

Pour faciliter le tracé de courbes de comparaison, il est fourni un script **shell**, **plot.sh**, qui permet de faire varier un parmi les paramètres mentionnés ci-dessus (6.2) et de tracer le

taux de succès (*hit rate*) correspondant pour les quatre stratégies. Ces courbes sont en fait produites sous forme d'un fichier en format EPS (Postscript encapsulé) que l'on peut donc visualiser avec **gv** ou **ghostview**.

Ce script ne peut fonctionner que si **gnuplot** est installé [7] et il s'utilise de la manière suivante :

```
plot.sh options -- option_test suite_valeurs...
```

Les *options* sont les suivantes :

-o *nom* le nom de base du fichier de courbes (son nom complet sera *nom*`.eps` ; sera également produit un fichier *nom*`.gp` de commandes à **gnuplot**.

-T *titre* titre général du graphique.

-x *étiq* étiquette de l'axe des x (la quantité que l'on fait varier).

-L "*x,y*" les coordonnées du titre entre doubles quotes (il est cadré à gauche) ; si ces valeurs sont incorrectes, le titre peut ne pas être affiché.

-t *n* *n* est le numéro du test (1 à 5) ; ici il ne doit y avoir qu'une seule option **-t**.

-i mode interactif : au lieu du fichier EPS [8], les graphiques sont affichés directement dans une fenêtre ; vous devez alors taper la commande `q` pour quitter **gnuplot**.

Le paramètre *option_test* de la ligne de commande est la désignation d'une des options décrites précédemment (6.2) et *suite_valeurs* la liste des valeurs que l'on veut donner au paramètres correspondant (leur ordre n'a pas d'importance, **gnuplot** les triera).

La Makefile contient plusieurs exemples d'invocation de **plot.sh** (que l'on peut exécuter par la commande `make plots`) (les résultats sont alors dans le répertoire `Plots` créé pour l'occasion). On peut aussi exécuter `plot.sh` directement dans le répertoire `scripts` :

```
./plot.sh -T "Effet de la taille du cache" \
        -o cache_size \
        -x "taille fichier/taille cache" \
        -t 5 -L "400,70" \
        --  -r 150 125 100 75 50 25 10
```

exécute-t-il le test 5 en faisant varier le rapport (option `-r`) entre la taille du fichier et la taille du cache [9] de 150 à 10 (le cache varie donc de 0,67 % à 10 % de la taille du fichier). Les autres paramètres ont leurs valeurs par défaut. Le résultat est envoyé dans le fichier `cache_size.eps` et est présenté sur la figure 3.

Pour ce test au moins, on peut constater que, à part RAND qui n'est vraiment pas terrible (est-ce une surprise ?), les trois autres stratégies se comportent bien pour des caches de l'ordre de 1 % de la taille du fichier (`r = 100`) et au delà (`r < 100`). En revanche, les performances s'effondrent rapidement pour des caches plus petits.

7 Au travail !

7.1 Qu'y a-t-il à faire ?

Il est temps d'y venir !

7. **gnuplot** est généralement disponible sous forme d'un *package* dans les diverses distributions de LINUX
8. En fait le fichier EPS est bien créé, mais son contenu est inexploitable. Bug ou feature de **gnuplot** ?
9. **Attention !** N'oubliez pas les `--` avant le paramètre à faire varier et sa liste de valeurs.

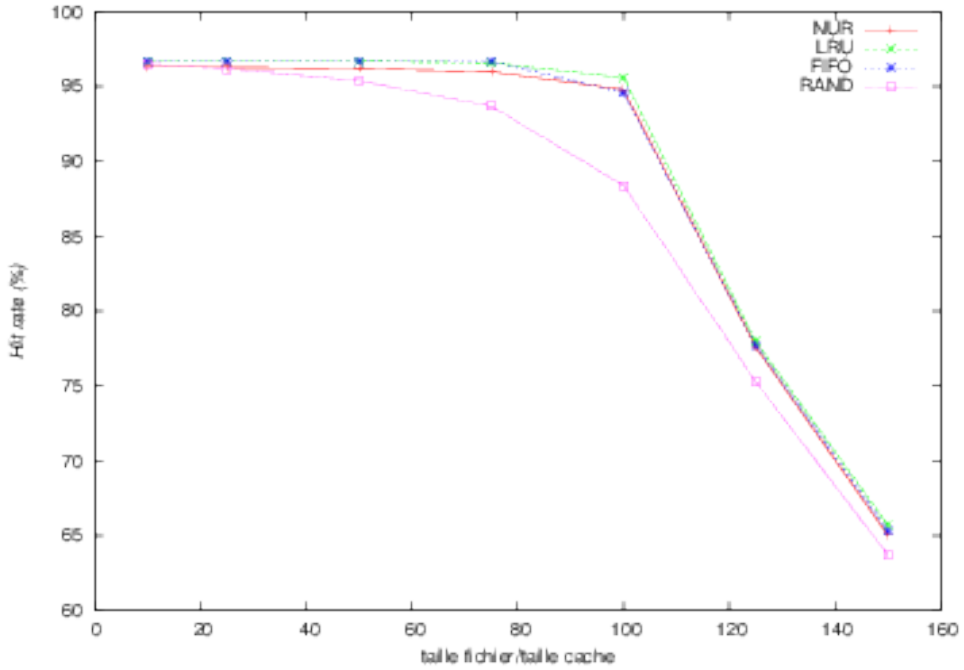

FIGURE 3 – Exemple de graphiques produits par le script **plot.sh** : comparaison des stratégie en fonction du rapport taille du fichier / taille du cache, ici pour le test 5.

1. Extraire le fichier `Cache_Project.zip` : ceci crée un répertoire `Cache_Project` avec les sources fournis ainsi que le PDF de ce document ;
2. Essayez de compiler l'application fournie avec la stratégie donnée par défaut (RAND). Il suffit de taper la commande **make** qui produit le fichier `tst_Cache_RAND` que, bien entendu, vous vous empresserez d'exécuter. Après un temps relativement court, vous devez obtenir quelque chose qui ressemble à ce qui suit (**kheops%** étant mon prompt de **zsh**). Même si les chiffres sont différents, ils devraient être s du même ordre :

```
kheops% tst_Cache_RAND
================ Configuration du cache ================
Paramètres du fichier :
    30000 enregistrements 360000 octets totaux
Paramètres du cache :
    30 blocs 10 enregistrements/bloc 12 octets/enregistrement
    120 octets/bloc 3600 octets totaux
    Rapport cache/fichier : 1.00 %
    Stratégie : RAND
Paramètres des tests :
    Nombre d'accès : 90000
    Rapport lectures/écritures : 10
    Nombre maximum accès séquentiels : 5
    Nombre de Working Sets : 100
    Largeur de la fenêtre de localité : 300
    Fréquence de déréférençage pour NUR : 100
========================================================
Test_1 : boucle de lecture séquentielle :
```

```
    29999 lectures 30000 écritures 56887 succès (94.8 %)
    60 syncs 0 déréférençages
Test_2 : boucle écriture aléatoire :
    0 lectures 90000 écritures 888 succès (1.0 %)
    91 syncs 0 déréférençages
Test_3 : boucle lecture/écriture aléatoire :
    81000 lectures 9000 écritures 44736 succès (49.7     91 syncs 0 déréféren-
çages
Test_4 : boucle lecture/écriture aléatoire avec localité :
    80965 lectures 9035 écritures 79406 succès (88.2 %)
    91 syncs 0 déréférençages
Test_5 : boucle lecture/écriture séquentielle avec localité :
    81047 lectures 8953 écritures 79449 succès (88.3 %)
    91 syncs 0 déréférençages
kheops%
```

Le programme de simulation s'est exécuté correctement avec toutes les options par défaut (voir 6.2) et avec la stratégie choisie (RAND). Il affiche d'abord les paramètres de configuration puis le résultat des cinq boucles de test décrites plus loin (voir 6.1).

3. Écrivez les fichiers nécessaires aux différentes stratégies, `FIFO_strategy.c`, `LRU_strategy.c` et `NUR_strategy.c`. Voir la section 4.2 pour construire les exécutables correspondants.
4. Essayer de comparer les différentes stratégies en variant les paramètres de configuration. N'essayez pas de tout faire varier à la fois ! Essayez d'utiliser le script **plot.sh** (voir 6.4) dont des exemples d'utilisation se trouvent dans `Makefile.plots` (exemples que vous pouvez exécuter grâce à la commande **make plots**) de la `Makefile` principale.
5. Implémenter les modules du cache indépendant de la stratégie de remplacement (`cache.c`, `low_cache.c`, `cache_list.c` afin de remplacer (et donc de vous passer de la bibliothèque `libCache.a`.
6. Optionnellement, ajouter de nouveaux tests de simulation de comportement de programme dans le fichier `tst_Cache.c` pour mettre en évidence les propriétés des diverses stratégies. Si nécessaire vous pouvez étendre l'instrumentation du cache.
7. Rédiger un bref rapport sur vos observations : efficacité du cache avec les diverses stratégie et les divers tests de simulation de comportement des programmes.

Vous êtes libres de tenter toutes les expériences que vous estimez pertinentes. Cependant, il ne suffit pas d'expérimenter, encore convient-il aussi d'analyser ! Vous allez donc tenter d'étudier l'influence de la variation de certains paramètres sur le taux de succès (*hit rate*) du cache.

7.2 Quelques éléments de réflexion et d'analyze

À titre d'exemple, voici quelques points sur lesquels vous pouvez réfléchir.

— Quel est l'effet du ratio *taille fichier/taille cache* (option **-r**) pour les différents tests ?
— Pourquoi le résultat du test 2 (avec les paramètres par défaut) donne-t-il toujours un taux de succès de 1 %, quelle que soit la stratégie ?
— Quel est l'effet du ratio *nombre de lectures/nombre d'écritures*, utilisé dans les tests 3 à 5 (option **-w**) ?

— Quel est l'effet du *nombre maximum d'itérations* dans les tests 2 à 5 ? Souvenez-vous que le paramètre correspondant (option **-l**) sera multiplié par le nombre d'enregistrements du fichiers (30 000 par défaut) pour donner le nombre total d'itérations.
— Le test 3 lit séquentiellement, à partir de l'enregistrement courant, un nombre (tiré au hasard entre 1 et `N_Seq_Access`) d'enregistrements. Quel est l'effet de ce paramètre `N_Seq_Access` (option **-s**) ?
— Quel est l'effet du nombre de *Working Sets* (option **-W**) dans les tests 4 et 5 ?
— Les tests 4 et 5 utilisent une *fenêtre de localité* (option **-L**). Quel est l'effet de ce paramètre ? Pourquoi note-t-on un effondrement du taux de succès à partir d'une largeur de fenêtre de 300 (quand tous les autres paramètres ont leur valeur par défaut) ?
— Enfin, pour approximer la notion de « récemment » dans la stratégie NUR, on remet à 0 le bit de référence (**R**) tous les `nderef` accès. Quel est l'effet de ce paramètre qui correspond à l'option **-d** ? Le choix de la valeur par défaut (100) vous semble-t-il judicieux ?

Tous ces points peuvent se traiter avec une utilisation raisonnable du script **plot.sh**, à l'exception du dernier. Pour voir l'effet de cette période de déréférençage (qui ne concerne que NUR) il vous faudra inventer vos propres commandes.