Sorbonne Université

Master Informatique

SAR

Rapport PSAR

Etudiants :
Smail Aider
Nathan Maurice
Clément Therry

Encadrant :
Julien Sopena

30/05/2019

Table des matières

Chapitre 1

Présentation

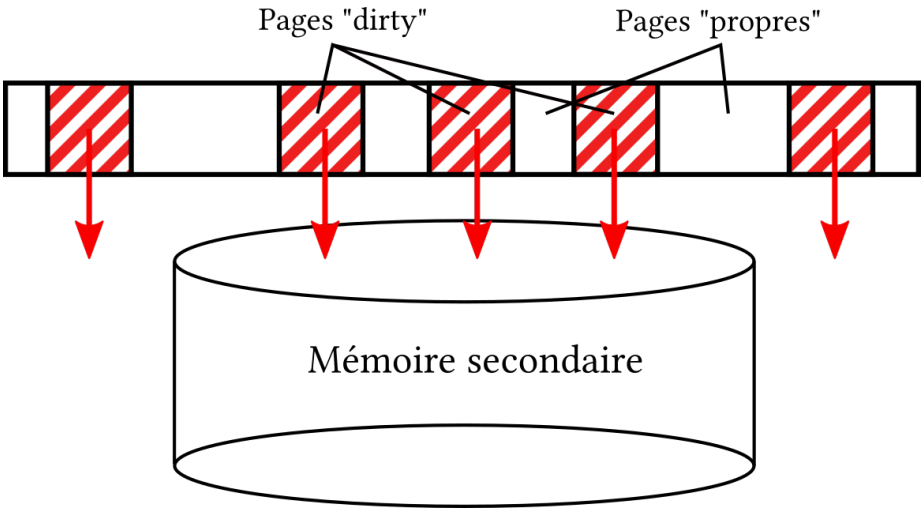
Ces dernières années, l'explosion de la capacité des mémoires de stockage a permis la collecte et le traitement des données à grande échelle dans le cadre du Big Data. Ces développements permettent de réaliser des analyses de plus en plus approfondies dans un large ensemble de domaines (prédiction de comportements, traitement de données scientifiques, ...)

Cependant, le traitement de telle quantité de données requiert des optimisations précise permettant de limiter les temps d'attentes sur les entrées sorties. En effet, un accès au disque est une opération lente ($\sim$1 million de fois plus lente qu'un accès à la mémoire principale). Pour cela, la plupart des systèmes d'exploitation modernes implémentent des caches en mémoire permettant de limiter le nombres d'accès disque effectués.

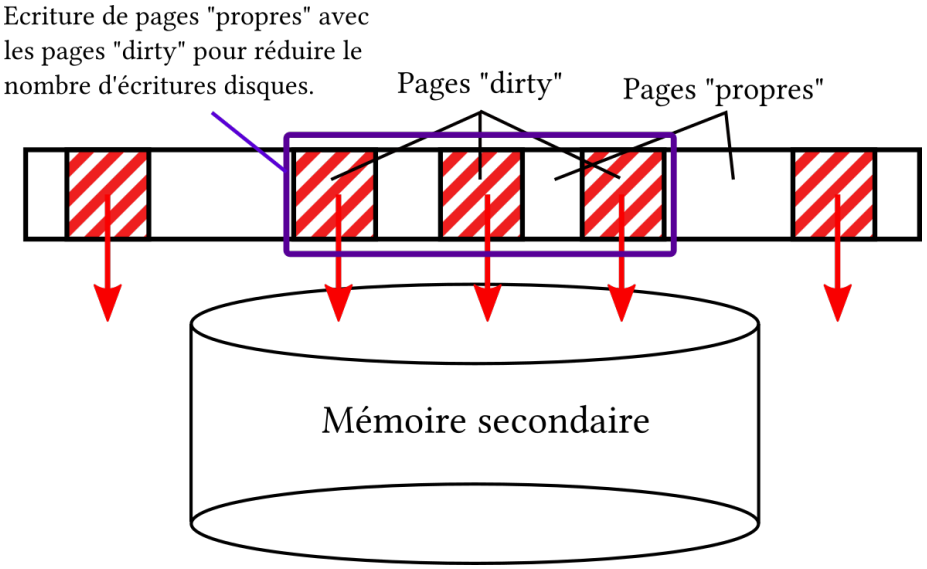
1.1 Problème

On se situe ici dans le noyau Linux. Lors d'une écriture dans un fichier, le noyau met à jour le bloc dans le buffer cache et marque le bloc comme "dirty". Une procédure viendra ensuite effectuer la synchronisation de ces blocs "dirty" vers le disque.

Pour réduire le nombre d'entrées/sorties, la procédure de synchronisation groupe les blocs "dirty" adjacents.



Cependant, on peut envisager de grouper les blocs dirty "proches", c'est à dire séparés par des blocs non dirty. En effet, le nombre d'entrées sorties serait réduit sans pour autant compromettre les données des blocs non dirty, les données de ces derniers étant identiques à celles du disque.



1.2 Définitions techniques

1.2.1 Page cache

Le page cache est un cache en mémoire contenant des données lues depuis le disque et ayant pour objectif de limiter la durée des accès aux fichiers. Si on lit pour la première fois un fichier, il ne se trouve pas dans le page cache et donc doit être lu sur disque. Si on le relit juste après, les pages du fichiers se trouvent déjà dans le page cache et donc l'accès est plus rapide.
Le page cache est composé de deux files à priorité LRU (Least Recent Use) :

— La liste active.

— La liste inactive.

Lorsqu'on accède à une page non présente dans le cache, elle est placé en tête de la file inactive. Si la liste inactive était déjà pleine, la page en queue de liste est sortie de la liste. Si cette page est propre, il n'y a rien à faire (le contenu de la page est le même que la copie sur le disque) , alors que si elle est dirty, on l'écrit sur le disque puis on la remet en tête de la liste inactive et elle sera libérée une fois qu'elle aura re-parcouru toute la liste inactive en restant propre. Il ne faut pas oublié que lorsqu'on écrit sur le disque une page dirty, on regarde les pages voisines direct pour voir si elles aussi sont dirty ou pas. Si on cherche à accéder à une page déjà présente dans la liste inactive, la page est placée en tête de liste active. Les pages sont retirés de la liste actives lorsque la mémoire est soumise à une grosse charge. Par exemple lorsque il y a beaucoup de processus en cours, et on a besoin d'avoir de la place pour les pages de ces processus. Les pages correspondant au information d'un processus (code, pile, etc...) sont appelé pages anonymes et ne sont pas dans le page cache. [**1**, **3**]

1.2.2 Arbre radix

L'arbre radix est une structure d'arbre de recherche utilisé par le noyau Linux pour l'indexation des pages d'un fichier. Les clés utilisées sont la valeur de l'offset dans le fichier et les adresses physiques des pages sont stockées dans les feuilles de l'arbre. L'efficacité de l'arbre radix tient au fait qu'on ne stocke qu'une partie P_0 d'une clé dans chaque noeud. Cette partie est telle qu'elle préfixe toutes les clés des sous arbres. De plus, le noyau ne place pas les pages non chargées dans cet arbre. En conséquence, non seulement l'arbre a une empreinte mémoire faible mais de plus la hauteur de cet arbre est peu important. [**1**] Par exemple, dans linux, un fichier de 4 To peut être contenu dans un arbre radix de hauteur 5.

1.2.3 XArray

A partir de la version 4.20 du noyau, l'API du page cache est changée et l'arbre radix, bien que toujours présent, n'est plus accessible directement et est situe derrière une couche d'abstraction appelée XArray.

Plutôt que de manipuler le cache comme s'il s'agissait d'un arbre, on le représente comme un tableau et on utilise les fonctions fournies par l'API pour

le manipuler.

L'API est composée de deux sous-API :

— Une API simple où la gestion des verrous est gérée par les fonctions appelées. Ces fonctions ont pour préfixe $xa\_$

— Une API complexe permettant une manipulation a grain fin mais ou la gestion des verrous doit être effectuée par l'appelant. Ces fonctions ont pour préfixe $xas\_$

Celle nouvelle API a plusieurs intérêts

— Une représentation en tableau est plus instinctive qu'un arbre pour le stockage des données.

— L'API est plus simple a utiliser mais permet d'effectuer les mêmes opérations qu'avec l'ancienne des arbres radix.

Chapitre 2

Benchmarks

Afin de déterminer les conditions de validation du projet, il est nécessaire d'établir des tests préliminaires permettant d'estimer l'étendue des optimisations possibles avec l'implémentation de ces écritures "propres". Ces tests permettront également de déterminer les gains de performances de cette implémentation par rapport à une configuration avec absence d'optimisation.

On réalise alors des benchmarks sur différentes stratégies d'écritures. Ces benchmarks s'effectueront avec un programme en mode utilisateur.

2.1 Outils utilisés

Le langage de développement utilise est le langage C. Celui ci permet de une gestion fine des écritures et est adapte aux interactions avec le noyau. De plus, compiler un programme C statiquement est simple a réaliser ce qui permet d'utiliser l'exécutable sur un système d'exploitation sans recourir a l'installation de bibliothèques tierces.

On utilise le module perf pour mesurer le nombre de requêtes effectuées. Perf est un module spécialement conçu pour le noyau Linux et permet de tracer une grande variété de métriques. Ici, nous mesurerons le nombre de requêtes d'écritures effectuées par l'exécution du programme de test.

Ces mesures seront exportées au format CSV. En effet, ce format est a la fois facile a lire pour un humain et est facile a parser. Pour les visualisations, nous utiliserons la bibliothèque Python matplotlib qui permet la création de graphiques et qui est hautement configurable.

La gestion des versions sera effectuée à l'aide du programme git et l'hébergement des sources se fera sur la plateforme gitlab. Un makefile sera utilisé pour l'intégration continue.

2.2 Tests écritures séquentielles

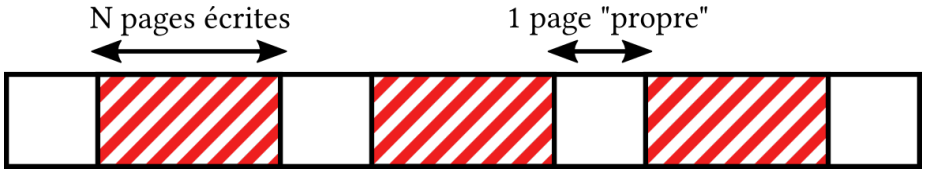
2.2.1 Hypothèse testée

On souhaite tester l'hypothèse selon laquelle le système d'exploitation optimise les écritures de pages consécutives mais ne regroupe pas les groupes de page "dirty" lorsque celle ci sont séparée par au moins 1 bloc "propre".

2.2.2 Démarche

On écrit les pages successivement en laissant un vide d'une page toute les N pages. On mesure alors le temps nécessaire pour toutes les écritures.

Nous utilisons les pages comme unités d'écriture. En effet, un seul octet modifié sur la page signifie que la page sera marquée "dirty" et sera donc entièrement écrite sur le disque. Utiliser les pages comme unités d'écriture et déplacement permet de garantir l'alignement des écritures c'est à dire qu'une écriture d'une taille T multiple d'une page garantira T écritures logiques.



On s'attend à voir une durée d'exécution maximale pour N = 1. En effet, d'après l'hypothèse 2.2.1, le noyau ne sera pas capable d'optimiser les requêtes écritures physiques.

Lorsque N augmente, on s'attend à voir la durée d'exécution diminuer. En effet, le noyau pourra optimiser les sorties en les regroupant, voir Figure 2.1.

2.2.3 Nombre de requêtes sur le périphérique

IOPS (Input/output operations per second) est l'unité standard de mesure du nombre de lectures et d'écritures effectuées sur une mémoire de stockage.

Dans cette section, nous allons reprendre l'expérience développée dans la démarche 2.2.2 afin de mesurer le nombre de requêtes (**après fusion**) émises sur le périphérique(device). Un transfert est une requête d'entrée/sortie sur le device. Plusieurs requêtes logiques peuvent être combiner dans une seule requête. Un transfert est d'une taille indéterminée.

Nous utiliserons pour cela, la commande **iostat** - report Central Processing Unit (CPU) statistics and input/output statistics for devices and partitions.
Le résultat correspond à une écriture de 8GO de données au total sur un disque dur SSD (Solid State Drive). Plus le nombre de pages consécutives augmente, plus le nombre de requêtes sur le périphérique diminue. Voir Figure 2.2.

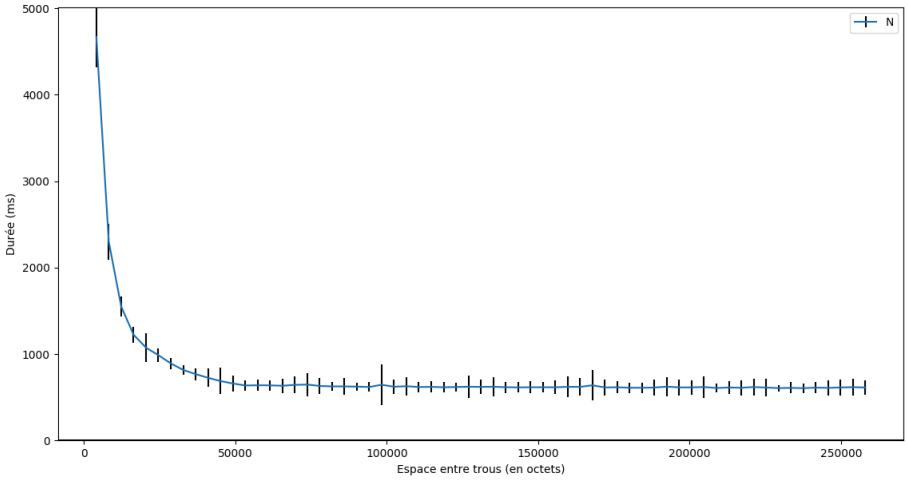


FIGURE 2.1 – Courbe avec intervalle de 95% pour écritures consécutives séparées avec trous.

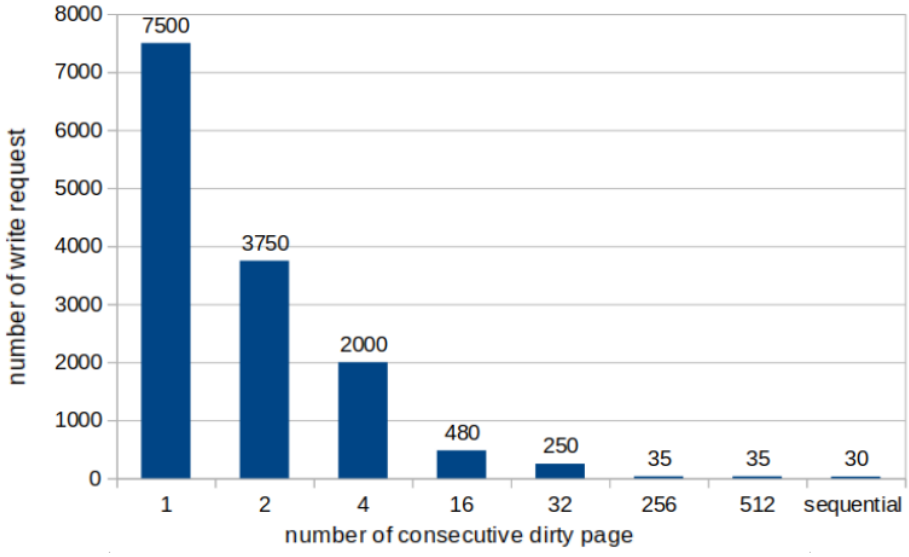


FIGURE 2.2 – Number of write requests(after merges) completed per second for the device according to the number of consecutive dirty page.

D'après la courbe 2.1 et l' histogramme 2.2, on note que des écritures sur des pages consécutives offrent des performances significativement meilleures que pour des pages fragmentées. Par conséquent, avec une stratégie d'écriture des pages propres, on s'attend à une différence moins prononcée entre une configuration avec une alternance page propre/page dirty et une configuration où l'on a que des pages dirty. En effet, dans le premier cas, on s'attend à n'écrire que 2 fois plus de blocs sur le disque par rapport au cas où toutes les pages dirty sont consécutives. Comme les écritures physiques seraient consécutives dans les deux cas, on s'attend à ce que le premier cas dure 2 fois plus longtemps que le second, contre un rapport de 6 sur la figure 2.1

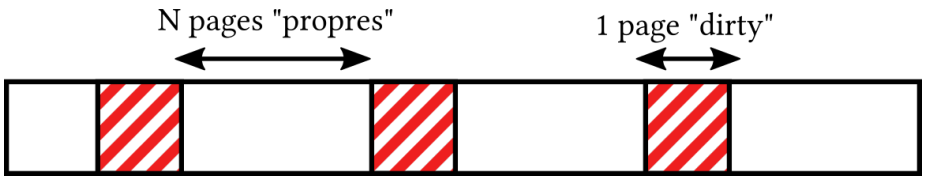
2.3 Écritures avec trou

2.3.1 Hypothèse testée

On souhaite également tester l'hypothèse selon laquelle la taille des trous n'influe pas sur les performances d'écriture. En effet, comme le noyau n'optimise pas les écritures non séquentielle, la taille des trous ne devrait pas influencer le résultats.

2.3.2 Démarche

On effectue 1 écriture pour marquer une page comme "dirty" puis l'on déplace l'offset du fichier tel que la page "dirty" soit suivie de N pages "propre". On répète cette opération un nombre fixé de fois.



Sans l'implémentation des écritures de pages propres on s'attend à voir une durée d'exécution minimale pour N = 0. En effet, il s'agit du cas où l'on effectue des écritures successives. Pour N > 0, on s'attend à voir une durée d'exécution constante mais plus importante que pour N = 0. En effet il s'agit des cas où le noyau ne peut pas optimiser les écritures, ces dernières étant toutes disjointes. Avec une stratégie d'écriture de M pages propres, on s'attend à ce que le temps d'exécution soit augmente linéairement pour une taille de trous inférieure à M pages puis à un temps d'exécution constant pour des tailles de trous supérieures. En effet, pour N < M, on écrit pour chaque page "dirty" M pages propres, d'où un résultat linéaire. Dans le cas où N > M, le temps d'exécution sera constant car les écritures de pages "dirty" seront effectuées séparément.

2.3.3 Nombre de requêtes disque effectuées

A l'aide d'un script bash et d'un exécutable de test, on exécute une série de mesures en faisant varier la taille des trous. Le script bash parse la sortie standard de perf et du programme et écrit les données en format CSV dans le fichier de sortie spécifié. On visualise ensuite les données sous forme de graphique en diagramme en barres avec un programme en python. Les bibliothèques matplotlib et seaborn sont utilisées pour la visualisation.

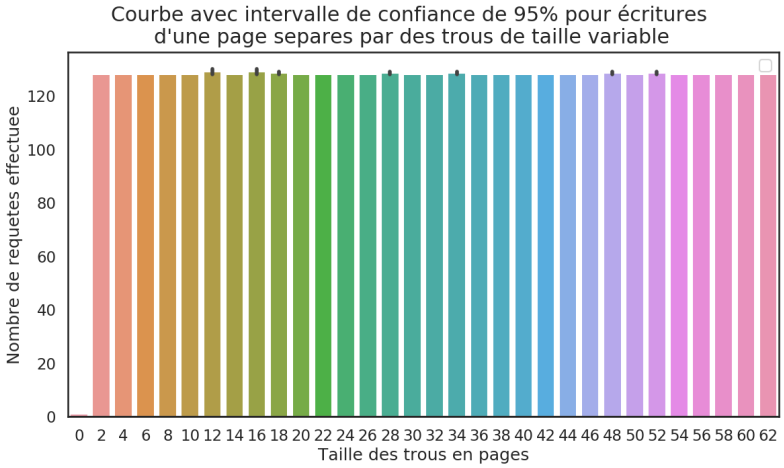


FIGURE 2.3 – Courbe avec intervalle de 95% pour écritures consécutives séparées avec trous. 128 pages sont écrites dans chaque cas

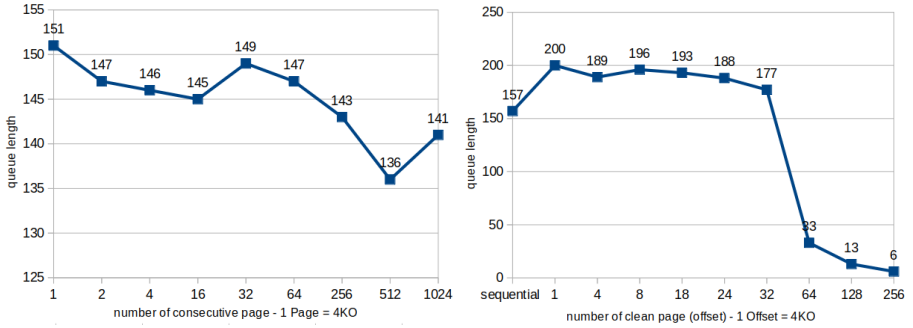
D'après la figure 2.3, le nombre de requêtes disques effectuées sans trous est proche de 1. En revanche, pour des tailles de trous supérieures a 1 page, le nombre de requêtes est constant et est proche de 128. On peut donc confirmer l'hypothèse selon laquelle le noyau peut optimiser les requêtes effectuées en continu mais ne peut pas effectuer de fusion de requêtes lorsque celles ci sont séparées par des pages propres.

2.3.4 Longueur file d'attente du périphérique d'entrée sortie

On peut aussi être amené à comparer la longueur de la file d'attente des requêtes émises sur le périphérique. Les résultats présentés ci-dessous sont issues d'une entrée sortie de 2GO sur un disque dur SSD.

La figure 2.4 représente la variation de la taille de la file d'attente du périphérique (en moyenne) par rapport au nombre de pages "dirty" consécutives (figure a) et le nombre de pages "propres" (figure b).

La file d'attente garde (en moyenne) une taille constante tout au long du transfert pour le cas des écritures consécutives (section 2.2.2), voir Figure(a). Sauf que, lors d'une entrée sortie avec une configuration qui consiste à laisser plus de 32 pages propres entre chaque écriture d'une page "dirty"(LSEEK de 25 PAGES), on observe une changement considérable (figure(b)).



(a) Increase the number of consecutive dirty pages (2.2.2)

(b) Make more and more offset between dirty pages (2.3.2)

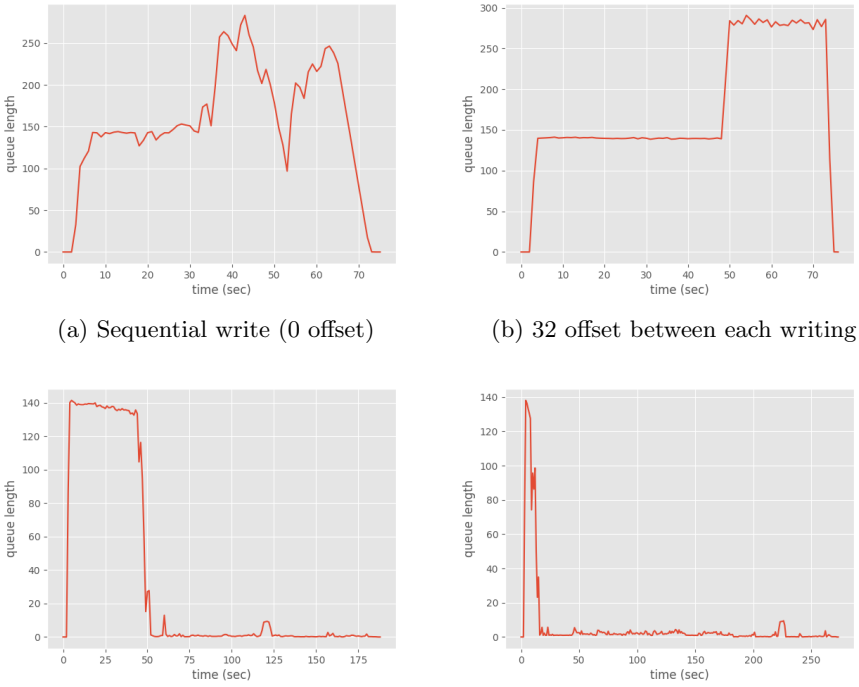
FIGURE 2.4 – The average queue length of the requests that were issued to the device according to the number of dirty/clean pages.

Pour plus de précisions, nous avons été amené à regarder le changement de la taille de la file d'attente sur certaines configurations, avant et après cette frontière (32 PAGES). Les résultats sont décris ci-dessous.

La figure (a) reflète le cas ou on effectue une écriture séquentielle, on voit plutôt une répartition uniforme.

Concernant les autres graphes, vu qu'on le laisse des décalages entre chaque écriture, le noyau essaye de faire des optimisations en regroupons les requêtes d'entrée sortie afin de gagner en performance.

Le temps d'exécution lui aussi augmente d'une façon substantiel, un facteur de 100 seconds en partant de la configuration 32 à 64 pages propres(70 -> 175 seconds, voir figure (a) et (b)), contre un temps d'exécution constant (∼70 seconds) pour toutes les autres configurations avec moins de 32 offsets.



(a) Sequential write (0 offset)

(b) 32 offset between each writing

(c) 64 offset between each writing

(d) 256 offset between each writing

FIGURE 2.5 – The average queue length of the requests that were issued to the device.

2.4 Limite de l'optimisation

2.4.1 Problème

On se doute bien que si le nombre de pages dirty à écrire n'est pas important, par exemple, disons que l'on doit écrire deux pages dirty non consécutives très éloigné l'une de l'autre, il n'est pas forcément plus rapide d'écrire toutes les pages intermédiaire. Dans ce cas, peut-être faut-il mieux écrire seulement ces deux pages et donc faire deux E/S. Nous allons donc dans ce projet chercher en plus à déterminer à partir de combien de pages consécutives, il est plus rapide d'écrire les deux extrémité que l'ensemble des pages d'une extrémité à l'autre. Ceci à pour but de prévoir la limite à partir de laquelle il n'est plus nécessaire de chercher dans les pages voisines des pages dirty lorsque nous changerons l'algorithme du noyau.

2.4.2 Démarche

On effectue tout d'abord des écritures consécutives de pages dirty de plus en plus grandes afin d'observer le temps en fonction du nombres de pages écrites consécutives (Voir figure 2.1). Ensuite il nous faut la valeur du temps d'écritures de 2 pages dirty écritent en deux requêtes distinctes. Pour cela, on écrit un nombre N de fois 2 pages espacé d'un trou et on fait la moyenne. L'objectif est d'obtenir la valeur, en nombre de page, pour laquelle l'une des stratégie dépasse l'autre.

2.4.3 Résultat

On obtient une valeur de 0.029 secondes pour l'écriture de 2 pages en 2 requêtes. Le nombre max de pages que l'on peut écrire consécutivement en moins de 0.029 secondes tourne autour de 100 pages. Cela veut dire que écrire 100 pages consécutives en 1 seul requête, est plus rapide que d'écrire 2 pages espacé en 2 requêtes.

Par exemple, écrire 50 pages espacé chacune de 1 page en 50 requêtes prend 2 fois plus de temps que d'écrire les 100 pages en 1 requête.

Chapitre 3

Détection trous dans le page cache

Afin de détecter la présence de trous dans le page cache associe a un fichier, on réalise un programme en mode noyau. On choisit de réaliser un module. En effet, on souhaite que l'utilisateur puisse afficher le nombre de trous pour un fichier donne, a un instant donne. Cela signifie donc que l'utilisateur doit pouvoir passer des paramètres a ce programme. Implémenter ce comportement est simple avec un module, en effet la gestion d'arguments depuis le mode user est très simple a utilise pour un module. En revanche, il est beaucoup plus complexe a réalisé avec un patch noyau. De plus, l'utilisation d'un patch noyau signifie que le mécanisme serait toujours charge en mémoire bien que l'utilisateur ne l'utilise pas.

3.1 Réalisation noyau

Dans la perspective de détecter les trous présents dans le cache, nous avons réalisé un module noyau permettant d'itérer sur le contenu de ce cache.

Ce module permet d'afficher le nombre de trous d'une largeur inférieur a une taille donnée pour un fichier passé en paramètre.

Nous utilisons la nouvelle API des xarray introduite a partir de la version 4.20 du noyau. En effet, son utilisation est simple mais reste complète et adaptée a notre cas d'utilisation. De plus, se familiariser avec cette API peut nous être utile a l'avenir, celle ci étant désormais l'API standard pour le page cache.

Afin de récupérer le xarray a partir du chemin du fichier on récupère la struct path associée avec la fonction noyau kern_path. On peut ensuite obtenir l'inode associé a cette struct path. L'inode obtenu contenant la struct address_space, on la récupère et on obtient la structure du xarray pointe par l'attribut i_pages de cette structure address_space.

On effectue l'affichage des résultats a travers le sysfs. Pour afficher les statistiques sur le fichier préalablement spécifié, l'utilisateur peut exécuter la com-

mande :

```
cat /sys/kernel/detection
```

3.2 Algorithme comptage de trous

count_holes

```
Input: hole
Output: hstat
Data: index = 0, prev = 0, lastDirty = 0, hasMark
xa = get_xarray()
foreach curseur ∈ xa do
    hashMark = xa_get_mark(cursor, PAGECACHE_TAG_DIRTY)
    if hasMark then
        hstats.dirty++
        if prev - lastDirty <= hole && prev - lastDirty >= 1 then
            hstats.holes++
        end
        lastDirty = index
    end
    hstat.clean++
    prev = index
end
```

3.3 Tests

3.3.1 Configuration

On utilise la version 5.0.6 du noyau Linux. Cette version utilise le page cache avec l'API des xarray. On ne se sert uniquement de l'API simple. En effet, on effet on souhaite avoir un accès exclusif au page cache pour itérer dessus. Cette fonctionnalité est remplie par la fonction xa_for_for_each(). On crée une partition ext4 de 500M et on place les fichiers de test sur cette partition. Il n'est ici pas possible de placer le fichier teste sur le ramfs ou sharefs. En effet, les écritures y seraient directes et aucune page ne serait marquée comme dirty, ce qui implique qu'aucun trou ne serait détecté.

3.3.2 Programme de test

On réalise un programme en mode user effectuant des écritures a des endroits arbitraires d'un fichier donne. Ce programme lis tout le fichier 2 fois pour s'assurer de son placement dans le page cache. Il effectue ensuite des écritures

sur les blocs du fichier suivants : 0, 2, 5, 9 et ajoute 2 blocs a la fin du fichier. Il y a donc 3 trous de 1 bloc, 2 blocs et 3 blocs respectivement ainsi qu'un trou entre le bloc 10 et la fin du fichier.

3.3.3 Résultats

```
$ insmod detection-xarray.ko pathname=/mnt/file.bin hole=1
$ ./test /mnt/file.bin
$ cat /sys/kernel/detection
> dirty 7, clean 686, holes 1

$ insmod detection-xarray.ko pathname=/mnt/file.bin hole=2
$ ./test /mnt/file.bin
$ cat /sys/kernel/detection
> dirty 7, clean 688, holes 2
```

Les tests précédents correspondent aux résultats attendus. On peut donc affirmer qu'on a bien détection des trous.

Chapitre 4

Page Cache

4.0.1 Writing Dirty pages to Disk

Le noyau remplit le page cache avec des pages contenant des données sur disque. Lorsqu'un processus modifie une donnée, la page correspondante est marquée comme étant dirty(le flag PG_dirty est positionné). Le système Unix autorise les écritures déférées des pages dirty sur disque, parce-que cela améliore considérablement les performances du système.

Une page dirty peut rester en mémoire jusqu'à le dernier moment possible(système shutdown). Mais garder des pages aussi longtemps dans le page cache à deux inconvénients majeurs :

- Perte de données en cas de panne(crash).
- La RAM doit être suffisamment grande - même taille que le disque.

Par conséquent, les pages dirty sont flushées(écrites) sur disque dans les cas suivants :

- Le page cache devient plein et d'autres pages sont nécessaires, ou le nombre de pages dirty devient très grand.
- Beaucoup de temps s'est écoulé depuis qu'une page est restée dirty.
- Un processus à demander que les pages doivent être flusher(sync, fsync or fdatasync).

4.0.2 The pdflush Kernel Threads

Les versions précédentes de Linux utilisaient un thread de noyau appelé **bdflush** pour analyser systématiquement le page cache à la recherche des pages dirty à écrire, et ils ont utilisé un deuxième thread du noyau appelé **kupdate** pour s'assurer qu'aucune page ne reste dirty trop longtemps. Linux 2.6 a remplacé les deux par un groupe de threads du noyau à usage général appelé **pdflush**.

Quels types de travaux sont délégués aux threads du noyau pdflush ? Il y a quelques-uns, tous liés à vider les données dirty. En particulier, pdflush exécute généralement une des fonctions suivantes :

— background_writeout() : parcours le pages cache à la recherche de pages dirty.
— wb_kupdate() : vérifie qu'aucune page du page cache ne reste dirty pendant trop longtemps.

4.0.3 Looking for Dirty Pages To Be Flushed

Chaque arbre radix peut inclure des pages dirty à flusher. Récupérer toutes ses pages en un seul passage implique une recherche exhaustive. Par conséquent, Linux adopte un mécanisme sophistiqué qui divise l'analyse du page cache en plusieurs exécutions.

La fonction wakeup_bdflush() reçoit comme argument le nombre de pages dirty à écrire. Cette fonction invoque pdflush_operation() pour réveiller un thread et lui délègue la fonction background_writeout() à exécuter.

En analysant la chaîne d'exécution de background_writeout(), elle finit par appeler **writepages()** définit dans la structure **adresse_space_operation** de l'inode ou **mpage_writepages()** si elle n'est pas implémentée.

La fonction mpage_writepages() est remplacée par **write_cache_pages()** dans les versions récentes de Linux. Elle parcours la liste des pages dirty de la structures adresse_space passée en paramètre et écrit toutes les pages dirty.

write_cache_pages() utilise le flag *TOWRITE* pour identifier les pages à écrire sur disque. Pour cela, elle invoque la fonction **tag_pages_for_writeback()** pour ajouter ce flag aux pages marquées comme étant *DIRTY*.

Chapitre 5

Réalisation patch

5.1 Utilisation d'un patch

On souhaite modifier le comportement des écritures des pages dans le noyau de sorte que les trous inférieurs a une certaine taille soient écris vers le disque pour diminuer le nombre de requêtes.

Nous avions le choix entre un patch noyau ou un module (car la fonction que nous avons décidé de modifier est déjà exporté par le noyau). Avec un patch, il faut modifier le code de la fonction que l'on souhaite modifié dans les sources du noyau. Avec un module il faut réécrire une fonction dans notre module et faire pointer le symbole exporté par le noyau sur cette fonction.

On choisit de réaliser un patch pour implémenter ce comportement. En effet, celui ci est global et s'appliquera a toutes les pages du système et ce du démarrage jusqu'à l'arrêt. Implémenter ce comportement n'est donc pas possible avec un module. En effet, rien ne garantis que le module soit déchargé après la dernière écriture réalisée. De plus, un utilisateur pourrait le décharger, ce qui invaliderait la condition.

En revanche, un patch respecte ses contraintes. Le code d'écriture des trous serait toujours chargé en mémoire et appelé par le mécanisme de synchronisation et serait actif pour l'ensemble des pages.

5.2 Première solution

Notre première approche était d'avoir 2 curseurs sur le Xarray :

- Un qui parcourt toutes les pages qui sont taggué avec le tag PAGECACHE_TAG_DIRTY.
- Un qui rejoint le premier curseur en tagguant les pages avec le tag PAGECACHE_TAG_TOWRITE si le trou est inférieur à un seuil prédéfini.

Nous avons utilisés l'API avancée des XArray pour implémenter nos solutions.

Dans le commentaire de la fonction tag_pages_for_writeback, il est dit : *thus it is important for this function to be quick so that it can tag pages faster than a dirtying process can create them.* Nous avons donc décider de faire un algorithme pas trop complexe, qui assure le contrat de taggué les pages que l'on veut, mais sans pour autant prendre trop de temps a s'exécuter.

tag_pages_for_writeback

```
Input: start, end
Data: index, last=start
XA_STATE(curseur1,start)
XA_STATE(curseur2,start)
foreach curseur1 ∈ page_i tagged PAGECACHE_TAG_DIRTY do
    index = page_i → index
    if index-last-1 ≤ NB_TROU_MAX then
        foreach curseur2 ∈ page_j from last to index do
            tag(page_j,PAGECACHE_TAG_TOWRITE)
        end
    else
        tag(page_i,PAGECACHE_TAG_TOWRITE)
    end
    last = index
end
```

Nous avons rencontré un problème dans l'implémentation de cette solution : L'utilisation de deux curseurs posait un problème de locks et donc nous avons du passer à une solution n'en utilisant qu'un seul.

5.3 Deuxième solution

Dans cette deuxième solution nous utilisons donc un seul curseur.

```
tag_pages_for_writeback
Input: start, end
Data: index, last=start
XA_STATE(curseur,start)
foreach curseur ∈ page tagged PAGECACHE_TAG_DIRTY do
    index = page → index
    if index-last-1 ≤ NB_TROU_MAX then
        XA_SET(curseur,last)
        foreach curseur ∈ page from last to index do
            tag(page,PAGECACHE_TAG_TOWRITE)
        end
    else
        tag(page,PAGECACHE_TAG_TOWRITE)
    end
    last = index
end
```

La différence entre cette solution et la première est que le curseur doit "revenir en arrière". Pour cela, on utilise la macro XA_SET qui refait une descente dans le XArray au lieu de faire des "xa_prev". Nous avons vérifier le bon résultat de notre algorithme à l'aide d'affichage et nous avons pu nous rendre compte que l'algorithme faisait bien ce que l'on voulait, c'est à dire, tagguer toutes les pages qui font des trous avec le tag PAGECACHE_TAG_TOWRITE.

Cependant, des tests supplémentaires à l'aide de perf on montré que le regroupement des pages pour en faire qu'une seule requête ne se fait pas sauf lors de la première écriture d'un fichier que l'on vient de créer.

5.4 Interprétation résultats

Nous avons pensé à plusieurs possibilités aui pourrait expliquer ce comportement. Tout d'abord, nous sommes sur que la fonction est appelé car nous avions mit un affichage lors de son appel. Cela veut donc dire que le noyau ne se sert pas des tags des pages pour décider quelles pages écrire ou pas. Peut-être que avant l'appel à cette fonction, il à déjà sélectionner les pages à écrire et les à retenu dans un buffer. Dans ce cas, pourquoi appellerais-t-il la fonction tag_page_for_writeback ? Il existe 3 raisons pour que le noyau lance l'écriture de page dirty sur le disque :

- Les pages sont dirty depuis trop longtemps, donc leur écriture est déclenché.
- Le page cache est plein, et le noyau à besoin de libérer de la place pour les nouvelles pages qui sont lu
- Les pages ont besoin d'être synchroniser sur le disque à la suite d'un

fsync par exemple.
Il est possible que la manière dont sont écrites les pages pour ces 3 raisons différentes diffère un peu. Comme la seul façon que nous avons testé est la troisieme raison, celle du sync, cela se pourrait que lors d'un sync, comme l'écriture ne concerne pas la granularité d'une page, mais plutôt celle d'un fichier, le noyau ne se comporte pas comme nous le pensions. Une autre différence est que les deux premiers point sont à l'initiative du noyau, alors que le sync est à l'initiative de l'utilisateur. Il faudrait donc tester les 2 autres raisons qu'a le noyau d'écrire une page sur le disque pour voir si on observe le même résultat ou pas.

Chapitre 6

Bilan

Au cours de ce projet nous avons pu mettre en évidence le coût en performances des trous dans le page cache à l'aide de programmes de benchmarks. Nous avons également pu créer un module permettant de déterminer le nombre de trous pour un fichier donné.

6.1 Améliorations possibles

A partir des fonctionnalités développées, il est possible de mettre en évidence plusieurs propositions d'extensions.

1. La correction du patch.
2. Une approche "lire pour écrire". Afin de limiter l'impact des trous, on effectue des lectures sur des zones propres pour les ramener en mémoire et permettre de limiter sur le nombre de requêtes disques supplémentaires provoquées par ces trous.

Bibliographie

[1] Maxime Lorrilère. *Caches collaboratifs noyau adaptés aux environnement virtualisés.* Février 2016 [*Université Pierre et Marie Curie*] *http ://www.ece.ubc.ca/ maximel/publications/thesis.pdf*

[2] Nima Hornamand. *Page Cache.* 2014 [*Stony Brook University*].

[3] Robert Love. *Linux Kernel Development (3rd Edition).* Juin 2010

[4] Daniel Bovet, Marco Cesati. *Understanding the Linux Kernel, 3rd Edition.* Decembre 2008

[5] *"Linux source code : (v5.0.6)", bootlin.* [En ligne]. Disponible sur : https ://elixir.bootlin.com/linux/v5.0.6/source.