Université de Rouen
UFR des Sciences et Techniques
Master 1 Sécurité des Systèmes informatiques
Projet annuel

Périphérique Bloc Virtuel Hybride SSD/HDD

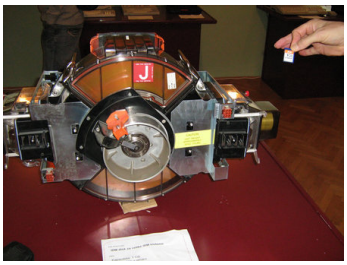
Zakaria Addi - Baptiste Dolbeau - Zineb Issaad -
Emmanuel Mocquet - Claire Smets

Rouen, le 17 mai 2012

Table des matières

1 Introduction

Depuis de début des ordinateurs, leurs capacités n'ont cessé d'augmenter, tant sur le plan de la rapidité que sur sur celui de la taille de la mémoire. Pour preuve un ancien disque dur IBM à côté d'une clé USB actuelle : chacun a pourtant une taille mémoire de 1 Go. Mais ni l'énergie consommée ni l'encombrement ne sont comparables.



Autrefois de quelques centaines de mega octets les disques durs font maintenant plusieurs centaines de giga octets. Mais la recherche du toujours plus performant est encore d'actualité. Pour ce faire, plusieurs techniques ont été développées pour les nouveaux types de périphériques de stockage de masse, dont entre autres les disques durs mécaniques (HDD) et les disque à mémoire flash :

Caractéristiques	HDD	SSD
Fragilité	Très fragile du fait de la tête de lecture	Peu fragile car pas de partie mobile à l'intérieur
Consommation en énergie	Assez gourmand en énergie : disques à faire tourner	Assez économe : pas de disque à faire tourner. En revanche, il faut alimenter les puces mémoires
Chaleur	Chauffe vite	Chauffe très peu
Prix	Assez peu cher : environ 1€pour 10 Go	Beaucoup plus cher : actuellement 1€pour 1Go

Il serait donc intéressant de pouvoir allier la rapidité du SSD à la grande capacité des HDD : créer un périphérique de stockage hybride SSD/HDD. C'est un produit déjà existant dans le commerce : une partie mécanique et une partie mémoire flash, le tout dans une seul boîtier et les deux parties étant complémentaires.

2 Description du projet

Pour notre projet nous disposons de deux périphériques séparés : un SSD et un HDD et notre logiciel consiste en :

un pilote permettant de relier les deux périphériques entre eux ;

un utilitaire permettant à l'utilisateur de maîtriser sa configuration.

Dans l'agrégat formé, le SSD jouera le rôle de mémoire cache non volatile : la taille du SSD sera obligatoirement plus petite de celle du HDD, et l'agrégat formé aura la taille du HDD.
Deux stratégies de gestion des disques seront développées :

le mode économies d'énergie : lors de la lecture d'un fichiers, les blocs lus seront importés sur le SSD. Cela permettra un gain de temps lors de la lecture suivante s'ils sont toujours présents sur le SSD. L'écriture se fera sur le SSD puis sur le HDD quand celui-ci sera vidé ;

le mode sécurité : le lecture se fera de la même façon. Ce sera l'écriture qui se fera de façon différente : elle se fera sur le SSD en priorité, puis sur le HDD en tâche de fond.

Lorsque l'espace libre sur le SSD sera inférieur à un certain seuil, une partie du contenu du SSD sera synchronisé avec le HDD puis supprimé du cache. Cette tâche sera effectuée en tâche de fond.

L'utilitaire se charge de la mise en place du "dialogue" entre l'agrégat formé et l'utilisateur du système hybride.

Plusieurs commandes sont à la disposition de l'utilisateur :

formation de l'agrégat : permet de définir par défaut les deux périphériques à intégrer à l'agrégat ;

activation de l'agrégat : permet de lancer manuellement l'activation de l'agrégat : les deux périphériques ne forment plus qu'un. SSD et HDD ne forment plus qu'un au vu de l'utilisateur ;

désactivation de l'agrégat : permet de désactiver l'agrégat : l'utilisateur peut de nouveau voir chacun des deux périphériques ;

choix de la stratégie de gestion du cache : économies d'énergie ou sécurité (cf opération ci-dessus) ;

demande de synchronisation du SSD et du HDD : permet d'être sûr que toutes les données du SSD sont aussi présentes sur le HDD ;

demande de "flush" : synchronise le SSD et le HDD puis vide le SSD.

3 Première livraison

4 Les différentes parties de l'hybridation

Notre projet se compose principalement de deux grosses parties :
- la première, très bas niveau, consistant à agréger deux segments de mémoire. Initialement, il était prévu d'agréger un SSD et un HDD. Cepdendant, de nombreuses difficultés ont été rencontrées, et non n'avons pu travailler que sur deux segments de mémoire ;
- le deuxième, plus haut niveau, consistant à effectuer un mapping entre des deux segements.

4.1 Formation de l'agrégat

4.2 Mapping

Pourquoi ? Dans notre agrégat, l'un des deux segments de mémoire sert de cache à l'autre. Cependant, comment savoir quelles informations sont sur le

cache ?
La solution naïve serait de parcourir tous les blocs du cache à la recherche de l'information. Avec beaucoup de chance, elle serait très vite trouvée. Mais il est beaucoup plus probable que l'information ne sera pas immédiatement trouvée, voir même ne sera pas présente dans le cache. Si le cache fait quelques méga-octets cela peut rester envisageable. Mais très vite, avec l'augmentation de la taille du cache, cette solution n'est plus viable.
Pour cela, nous avons dû implanter un mécanisme faisant un compromis entre efficatité temporelle et espace mémoire utilisé.

Solution apportée Pour cela, nous avons décidé d'utiliser un mécanisme assez connu : les tables de hachage.
Soit h(lab) = y une fonction de hachage prenant en paramètre une adresse et renvoyant la valeur de hache correspondant.
À chaque information demandée à l'agrégat, celui-ci transmettra la valeur lba et l'entrée correspondante dans la table sera trouvée. À chaque entrée correspondra une liste : la fonction h() produira plusieurs fois la même valeur pour des paramètres différents. En effet, son but est simplement de diminuer le temps de recherche d'une information. Lorsque l'entrée y aura été trouvée, on saura que si l'information est contenue dans la table, c'est à la ligne y. Si l'information n'est pas sur la ligne y, alors elle n'est nulle part dans la table. En supposant que h() soit bien "répartie", le temps de recherche est donc divisé par le nombre d'entrées de la table.

Implantation Comme dit précédemment, l'implantation ne doit pas être trop gourmande en teps ni en espace. Pour cela, nous avons décidé de ne pas importer de blocs mais des lignes. La taille d'une ligne a été fixée à 100 blocs, ce qui correspond à environ 1 Mo. Lorsqu'une ligne sera stockée dans le cache, une nouvelle cellule sera ajoutée à la table de hachage, en début de liste. Cette cellule comportera :

- l'adresse de la ligne sur le cache ;
- si le bloc a été modifié depuis son importation ;
- un pointeur vers les autres lignes ayant le même hache ;

Comme mentionné ci-dessus, lorsqu'une ligne est ajoutée au cache, l'insertion se fait en début de liste. Cela permet de ne pas gérer de liste doublement chaînée tout en pouvant supprimer facilement les plus anciennes lignes de la liste.

La question de l'insertion dans la table de hachage et été résolue. Mais le cache est généralement de taille inférieur à la mémoire principale. Il est donc indispensable de pouvoir vider le cache, au moins partiellement. Il serait très désagréable pour l'utilisateur que ce transfert s'effectue quand il souhaite utiliser l'agrégat. Pour cela, des seuils maximal et minimal sont mis en place : quand le seuil maximal de remplissage est dépacé, le cache se vide jusqu'à ce que le seuil minimal soit atteint. Cette tâche s'effectue en tâche de fond et n'empêche pas l'utilisateur d'utiliser l'agrégat.
À chaque transfert, ce sont les fins de listes qui sont suprimées de la table. Si la ligne a été modifiée depuis son transfert vers le cache, alors elle est réécrite sur la mémoire principale. Sinon rien n'est écrit.

Nous avions pensé à des listes doublement chaînées, mais le traitement de la table est lourd à chaque insertion et délétion.

Lignes libres sur le cache Lors de l'importation d'une ligne de la mémoire principale vers le cache, il faut pouvoir désigner l'emplacement à allouer. Le mécanisme n'est pas géré par le VFS : nous nous situons à un niveau plus bas. Deux possibilités ont été examinées pour acéder facilement aux lignes vides :

- bitmap du cache
- liste des lignes libres

La bitmap est peu coûteuse en espace ni en temps lorsqu'une ligne est allouée. Mais la recherche d'un ligne vide peut être longue : parcours de la bitmap. Plus le cache est grand, plus le parcours est long.

La liste des lignes libres est légèrement plus coûteuse en espace. Mais Lors d'une demande de ligne vide, le temps de réponse est constant. Plus le cache est long, plus la liste est longue et coûteuse en mémoire aussi. Mais chaque cellule de la liste ne comporte que l'adresse de la ligne, donc le coût reste raisonnable.

5 Annexes

5.1 Biblographie

- http ://www.2dayblog.com/2007/09/24/the-pass-and-the-present-1gb-drive/
- http ://www.lostinbrittany.org/blog/2007/11/05/nostalgie-informatique-et-disque-dur-de-10-mo/
- http ://uthash.sourceforge.net/userguide.html
- http ://fr.wikipedia.org/wiki/Disque_dur