TOUT SAVOIR SUR LES SSD (SOLID STATES DRIVES)

Mewtow

29 octobre 2015

Zeste de savoir

Table des matières

1 Introduction 5

2 Architecture d'un SSD 7
2.1 Support de mémorisation 7
2.2 Circuits annexes 7
2.3 Résumé 8

3 Support de mémorisation 9
3.1 Micro-architecture 9
3.1.1 Pages 10
3.1.2 Accès mémoire 10
3.1.3 Optimisation du support de mémorisation 13

4 Flash Transaction Layer 17
4.1 Correspondance entre adresses physiques et logiques 17
4.1.1 Correspondance statique 17
4.1.2 Correspondance dynamique 18
4.2 Corriger les défauts de la correspondance dynamique 20
4.2.1 Sur-approvisionnement 20
4.2.2 Garbage collector 20
4.2.3 TRIM 21
4.2.4 Effacement sécurisé 21
4.3 Wear leveling 22
4.3.1 Wear leveling dynamique 22
4.3.2 Wear leveling statique 23
4.4 Gestionnaire de requêtes 23
4.4.1 Dynamic write mapping 23
4.4.2 Native Command Queueing 24
4.4.3 Write Buffer/Coalesing Buffer 24

5 Conclusion 25

1 Introduction

Tout ordinateur contient obligatoirement une mémoire de masse, qui lui permet de mémoriser des données même quand celui-ci est éteint. Dans une mémoire de masse, les données sont conservées en permanence, et ne s'effacent pas quand l'alimentation électrique de l'ordinateur est coupée. De plus, ces mémoires de masse peuvent retenir une grande quantité de données.

Il y a de cela quelques années, les mémoires de masse disponibles sur le marché étaient des mémoires magnétiques : le stockage des informations s'effectuait sur un support magnétisable. Par exemple, on peut citer les fameux disques durs, ou les mémoires à bande magnétique.

De nos jours, une nouvelle forme de mémoire de masse a vu le jour : les **Solid State Drives**, ou SSD. Ces SSD sont des mémoires de masse dont le support de mémorisation n'est pas magnétique, mais électronique. Dans la majorité des cas, ces SSD sont fabriqués avec de la mémoire FLASH, une forme évoluée d'EEPROM, mais certains SSD assez anciens utilisaient de la mémoire DDR-SDRAM, comme on en trouve dans nos PC. Ces SSD sont évidemment l'objet de ce tutoriel.

2 Architecture d'un SSD

Un SSD contient un support de mémorisation, et des circuits annexes qui permettent de gérer ce support. Pour résumer, on peut voir un SSD comme un disque dur classique dont on aurait remplacé les plateaux magnétiques par des mémoires FLASH.

2.1 Support de mémorisation

Le **support de mémorisation** d'un SSD est, comme je l'ai dit plus haut, basé sur de la mémoire FLASH. Si une clé USB contient le plus souvent une seule mémoire FLASH, un SSD en contient un grand nombre.

Cette mémoire FLASH peut se décliner sous plusieurs versions : NOR, NAND, etc. Je ne vais pas rentrer dans les détails bas-niveau du fonctionnement de ces mémoires, mais il faut savoir que les premières mémoires FLASH étaient accessibles en lecture ou écriture via un bus série : on ne pouvait lire ou écrire qu'un seul bit à la fois. De nos jours, les mémoires FLASH peuvent communiquer avec l'extérieur via des bus parallèles : on peut lire ou écrire plusieurs bits à la fois.

2.2 Circuits annexes

Un SSD communique avec le reste de l'ordinateur via un bus, généralement un bus S-ATA, P-ATA, ou SCSI (comme les disques durs). En conséquence, le SSD contient obligatoirement un circuit qui se charge de gérer les communications sur le bus : c'est le **contrôleur de bus**.

A coté de ce contrôleur de bus, on trouve un **contrôleur de mémoire FLASH** qui reçoit des requêtes de lecture ou d'écriture, et se charge de commander les mémoires FLASH. Ce contrôleur fait en sorte que le SSD soit vu comme un disque dur par le processeur, et non comme un amas de mémoire FLASH. Le contrôleur de mémoire FLASH est souvent appelé le **Flash Transaction Layer**.

Ce contrôleur est souvent un processeur auquel on a rajouté une mémoire DRAM ou SRAM (voire les deux). Pour contenir le programme que doit exécuter ce processeur, le SSD incorpore parfois une mémoire ROM. Mais dans d'autres cas, une portion de la mémoire FLASH du disque dur sert à stocker le Firmware.

Les données reçues par le SSD sont des données de plusieurs bits : on est obligé de faire la conversion entre des données parallèles et l'interface série des mémoires FLASH. Un SSD contient donc, en plus du contrôleur de mémoire FLASH, des **convertisseurs parallèle <-> série**, et plus précisément un convertisseur série -> parallèle, et un convertisseur parallèle -> série. Ces convertisseurs sont composés d'une mémoire tampon, couplée à un multiplexeur ou un démultiplexeur.

Ce contrôleur et ces convertisseurs sont aussi secondés par des circuits de correction et de détection d'erreur, intercalés entre l'interface série des boitiers de FLASH et le convertisseur.

2.3 Résumé

L’architecture d’un SSD ressemble donc à ceci :

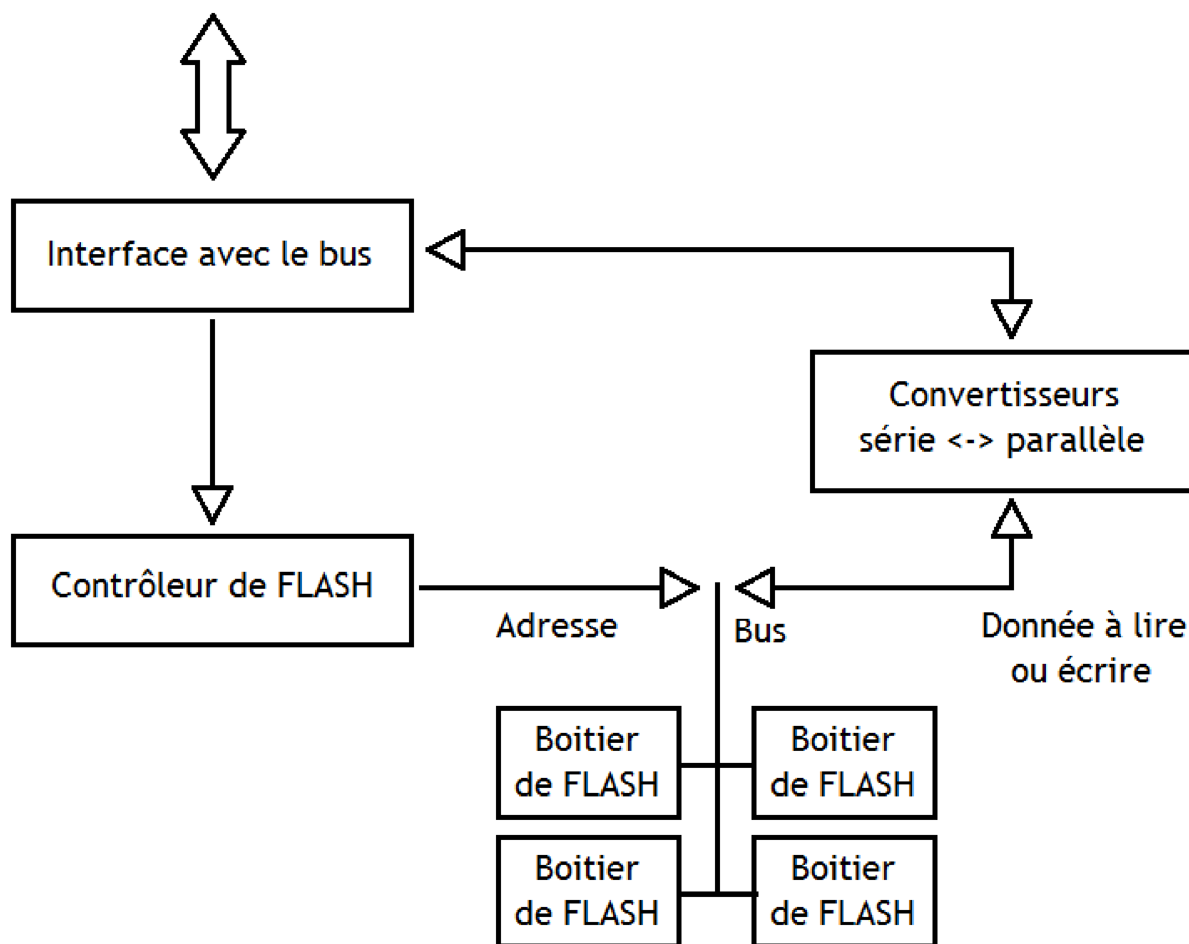

Figure 2.1 – Architecture interne d’un SSD

Évidemment, on peut rajouter des mémoires FIFO pour mettre en attente des données lues ou à écrire, afin d’accepter un grand nombre de requêtes de lecture/écriture pendant que les mémoires FLASH sont encours d’utilisation. Quelques circuits annexes se chargent de la gestion de ces files d’attente, files d’attente qui seront abordées à la fin de ce tutoriel.

3 Support de mémorisation

L'interface d'une FLASH autorise plusieurs opérations :

- la lecture ;
- l'écriture : on dit plutôt qu'on reprogramme une FLASH ;
- l'effacement.

3.1 Micro-architecture

Chose intéressante, la taille du Byte n'est pas la même pour les lectures/écritures et pour l'effacement. L'effacement d'un octet individuel n'est pas possible. Les FLASH sont découpées en **blocs**, l'effacement s'effectuant par blocs complets. Et cela est vrai aussi bien pour les mémoires FLASH NOR que pour les NAND.

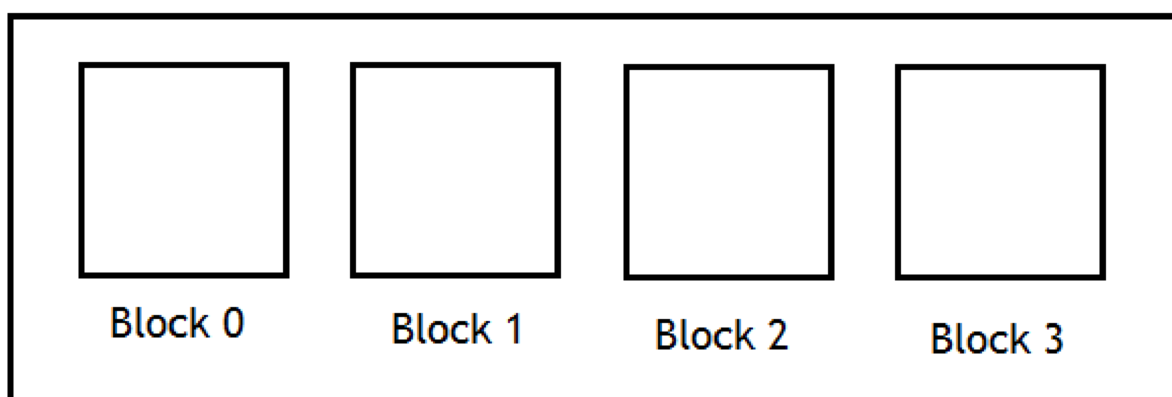

Figure 3.1 – Blocs

Ce blocs sont souvent rassemblés dans des ***planes***. On peut considérer que ces *planes* sont aux FLASH ce que les banques sont aux barrettes de RAM.

Ces *planes* peuvent elle-mêmes être regroupées en ***dies***, des groupes de planes reliés chacun à une liaison d'entrée-sortie série. Pour simplifier, ces *dies* sont aux FLASH ce que les barrettes de RAM sont aux SDRAM et aux DDR. Cette organisation permet quelques optimisations :

- il est possible d'accéder à plusieurs *planes* en parallèle sans pertes de performances ;
- plusieurs *planes* d'un même die peuvent effectuer la même opération en même temps.

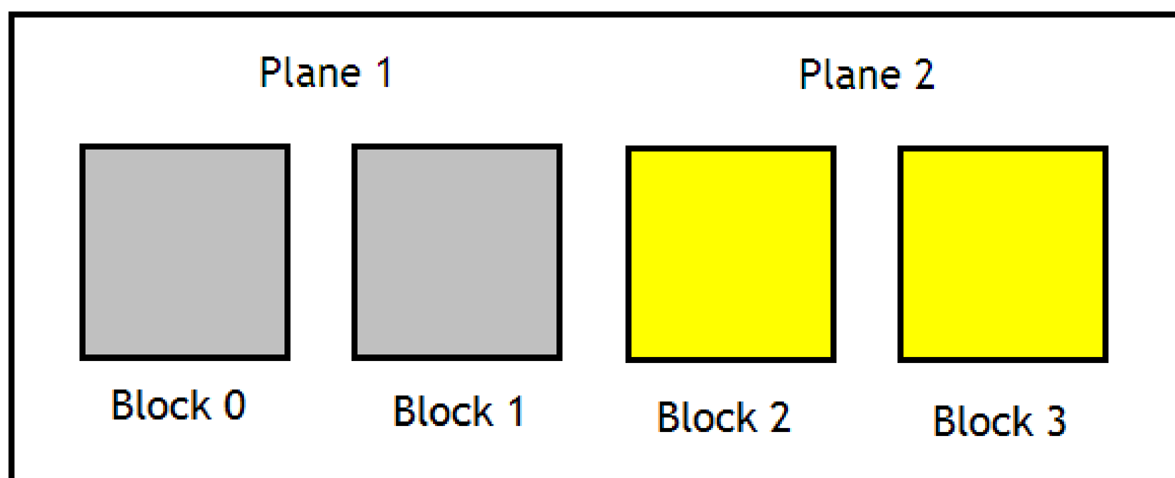

Figure 3.2 – Planes

3.1.1 Pages

Dans les mémoires NOR, il est possible de lire ou d'écrire octet par octet, chaque octet ayant une adresse. Mais sur les NAND, on ne peut pas : on est obligé de lire ou d'écrire par paquets d'octets relativement importants. Même si vous ne voulez modifier qu'un seul bit ou un seul secteur, vous allez devoir modifier un paquet de 512 à 4096 octets. Ces paquets sont appelés des **pages**.

Chacune de ces pages est composée de deux morceaux : un qui sert à stocker le contenu du secteur, et un autre qui sert à mémoriser des données de contrôle, comme des bits de correction d'erreur.

3.1.2 Accès mémoire

Chaque page d'une mémoire FLASH a une adresse, qui permet de l'identifier et de la sélectionner. Cette adresse est composée de plusieurs sous-adresses, qui ont chacun une utilité :

— une sous-adresse pour indiquer dans quel *die* se situe la page ;
— une adresse pour sélectionner la *plane* dans le die ;
— une adresse pour indiquer le bloc en question dans la *plane* ;
— et une dernière qui fournit le numéro de la page dans le bloc.

Dans les faits, le découpage peut être différent suivant les mémoires : certaines adresses peuvent être découpées en sous-adresses, ou des techniques d'optimisations peuvent intervertir des bits des différentes adresses. Mais le principe est toujours valable dans les grandes lignes.

Vu que les pages sont relativement grandes (plusieurs kibioctets), il est évident que les FLASH ne peuvent pas utiliser une interface parallèle : vous imaginez des bus de 4096 bits ? A la place, elles sont obligées d'utiliser des bus série pour envoyer ou récupérer les pages bit par bit. Pour accumuler les bits à lire ou écrire, chaque *plane* ou bloc contient un **registre de page**.

Voyons maintenant comment celui-ci est utilisé lors des lectures et écritures.

Boitier de mémoire FLASH

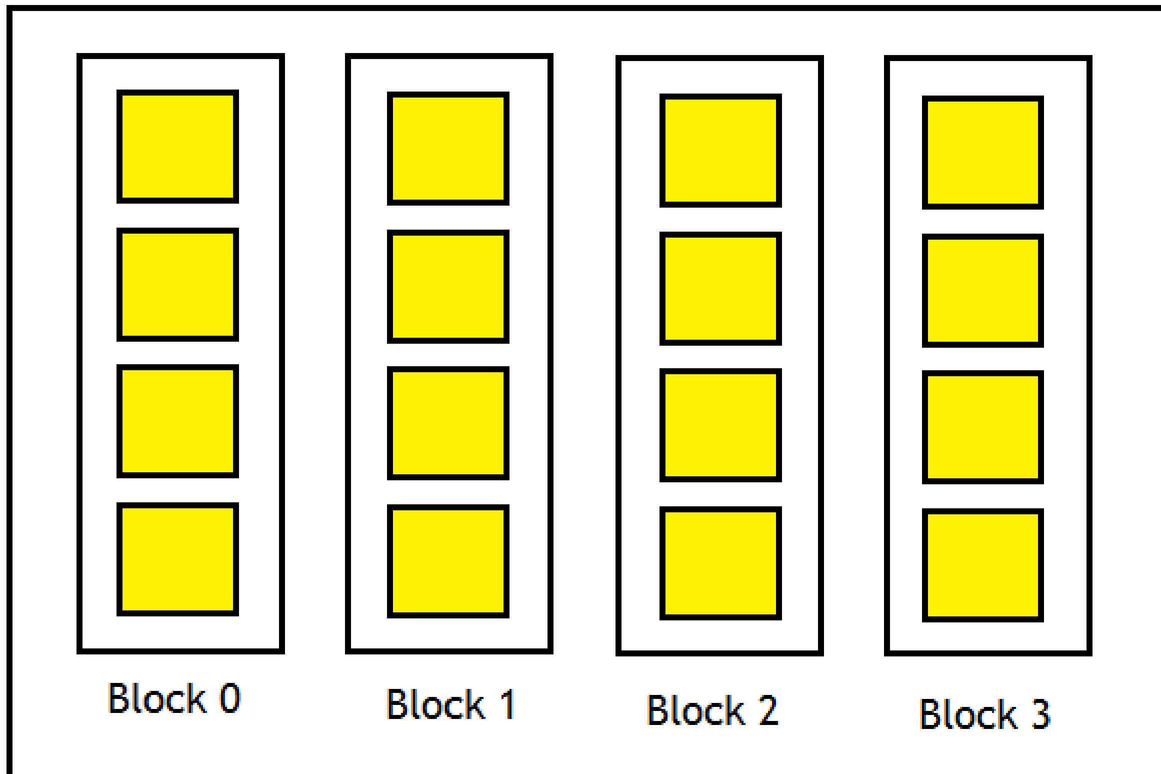

Figure 3.3 – Pages

Adresse du bloc	Adresse de la page	Adresse de la plane	Adresse du die

Figure 3.4 – Organisation d'une adresse mémoire de FLASH

3.1.2.1 Déroulement d'une lecture

La lecture d'une page se déroule comme suit :

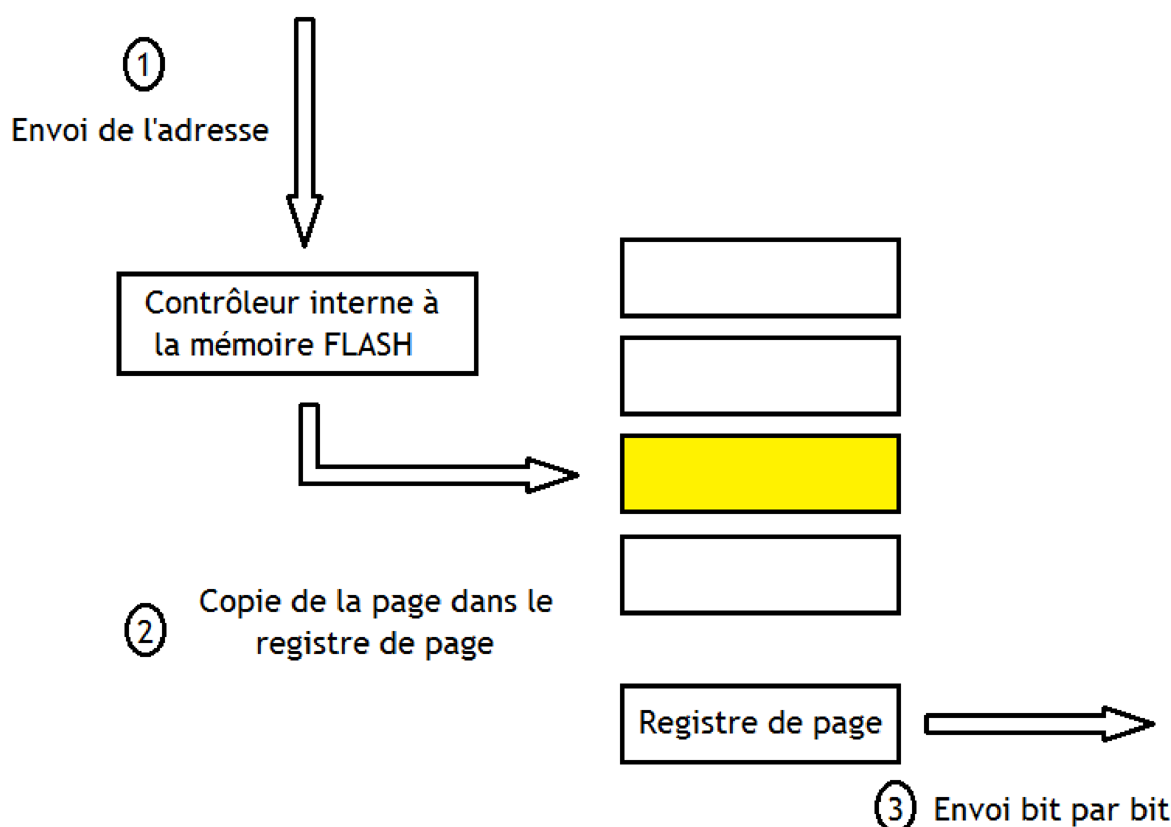

Figure 3.5 – Lecture sur une FLASH

Les bits lus sont ensuite accumulés dans une mémoire tampon, avant d'être envoyés dans la mémoire principale sur le bus P-ATA, S-ATA, ou SCSI.

3.1.2.2 Déroulement d'une écriture

L'écriture d'une page se déroule comme suit :

Seul problème : pour écrire dans une page, il faut impérativement que celle-ci soit intégralement effacée.

Cela ne pose pas de problème si cette page est vide : on peut écrire dans celle-ci sans problème, même si les autres pages du bloc sont occupées. Mais si la page contient déjà des données, on est obligé de l'effacer, ce qui demande d'effacer tout le bloc.

On est alors face à un problème : comment conserver les données des autres pages du bloc ? La solution est de sauvegarder temporairement tout le contenu du bloc à effacer, sauf la page à écrire. Dans le cas le plus simple, cette sauvegarde utilise une mémoire temporaire similaire au registre de page : le registre de page est simplement agrandit pour pouvoir stocker un bloc complet (les modifications d'une page s'effectuant directement dans ce registre de bloc). Mais dans le cas le plus courant, le bloc à effacer est simplement recopié dans un autre bloc, sans la page à modifier : la page à modifier est alors une page vide, qui peut être écrite directement

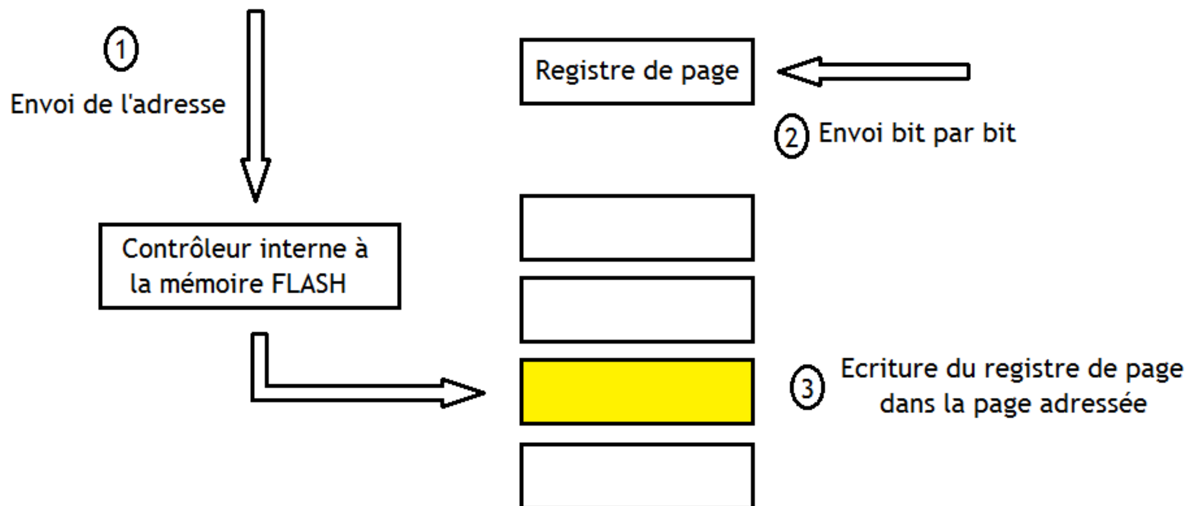

Figure 3.6 – Écriture sur une FLASH

3.1.3 Optimisation du support de mémorisation

Ce support de mémorisation peut être optimisé de manière à gagner en performances. Diverses techniques, aujourd'hui implémentées dans la majorité des SSD permettent de copier des pages très rapidement, ou d'utiliser efficacement le *pipelining*.

3.1.3.1 Migration rapide de pages

Certaines mémoires FLASH permettent d'effectuer des copies rapides entre pages, sans avoir à passer par le registre de page. Cela permet d'optimiser les migrations de donnée ou les copies de pages, opérations très courantes sur les SSD.

Mais cela a un coût : pour copier une page dans une autre directement, il faut que les pages en question soient reliées avec un lien série pour transférer les données directement de page en page. Et plus une mémoire FLASH contient de pages, plus le nombre d'interconnexions augmente. Pour limiter le nombre d'interconnexions, ces copies rapides entre page ne sont possibles qu'à l'intérieur d'une plane ou d'un *die* : impossible de copier rapidement des données mémorisées dans des *dies* ou *planes* différents.

3.1.3.2 *Pipelining*

Une mémoire FLASH telle que décrite plus haut a un défaut : il faut attendre qu'une lecture ou écriture soit terminée avant d'en envoyer une autre. Mais il est possible de pipeliner la mémoire FLASH : celle-ci devient alors capable de mettre en attente plusieurs requêtes de lecture/écriture supplémentaires. Généralement, les FLASH se limitent à une seule requête en attente : en mettre plus ne serait pas vraiment utile. Il devient possible d'envoyer une requête de lecture/écriture pendant que la FLASH effectue la précédente.

Pour cela, le contrôleur de la mémoire FLASH doit être modifié. Mais surtout, chaque plane ou bloc se voit attribuer un registre supplémentaire, en plus du registre de page : le ***cache register***.

Prenons l'exemple de deux écritures successives : ce registre de cache permet de mettre en attente une écriture pendant que la précédente utilise le registre de page. La donnée en cours d'écriture

est copiée du registre de page sur le support de mémorisation, tandis que le registre de cache accumule les bits de la prochaine donnée à écrire.

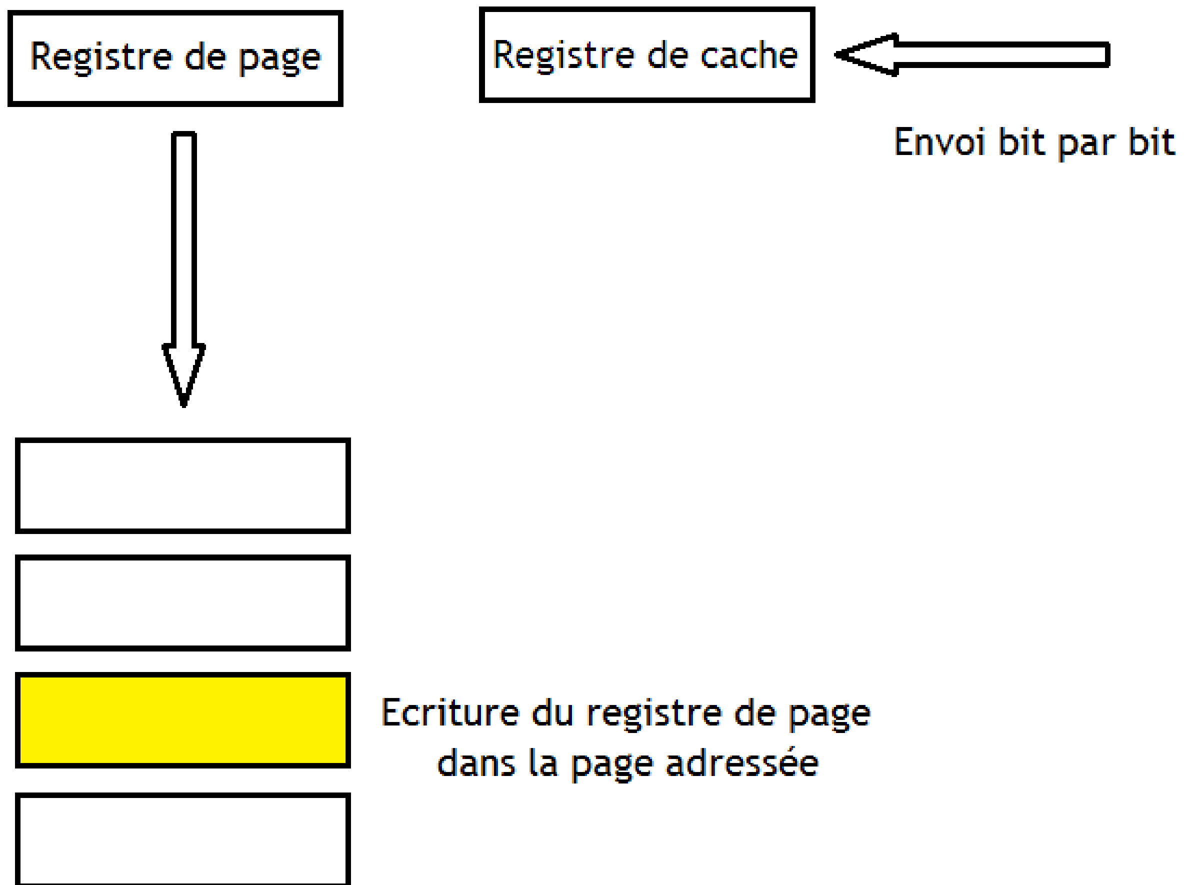

Figure 3.7 – Sans pipelining

Pour une lecture, c'est la même chose. Pendant qu'une donnée est chargée dans le registre de page, celui-ci est inutilisable et ne peut envoyer son contenu vers l'extérieur. Mais avec un registre de page, c'est différent : le contenu du registre de cache peut être envoyé sur la sortie série, pendant qu'une autre donnée est copiée du support de mémorisation vers le registre de page.

Évidemment, le contenu des registre de page et de cache est échangé à chaque début de lecture/écriture. Avec quelques optimisations, on peut se passer de recopies en échangeant le rôle des registres : le registre de page devient le registre de cache, et vice-versa.

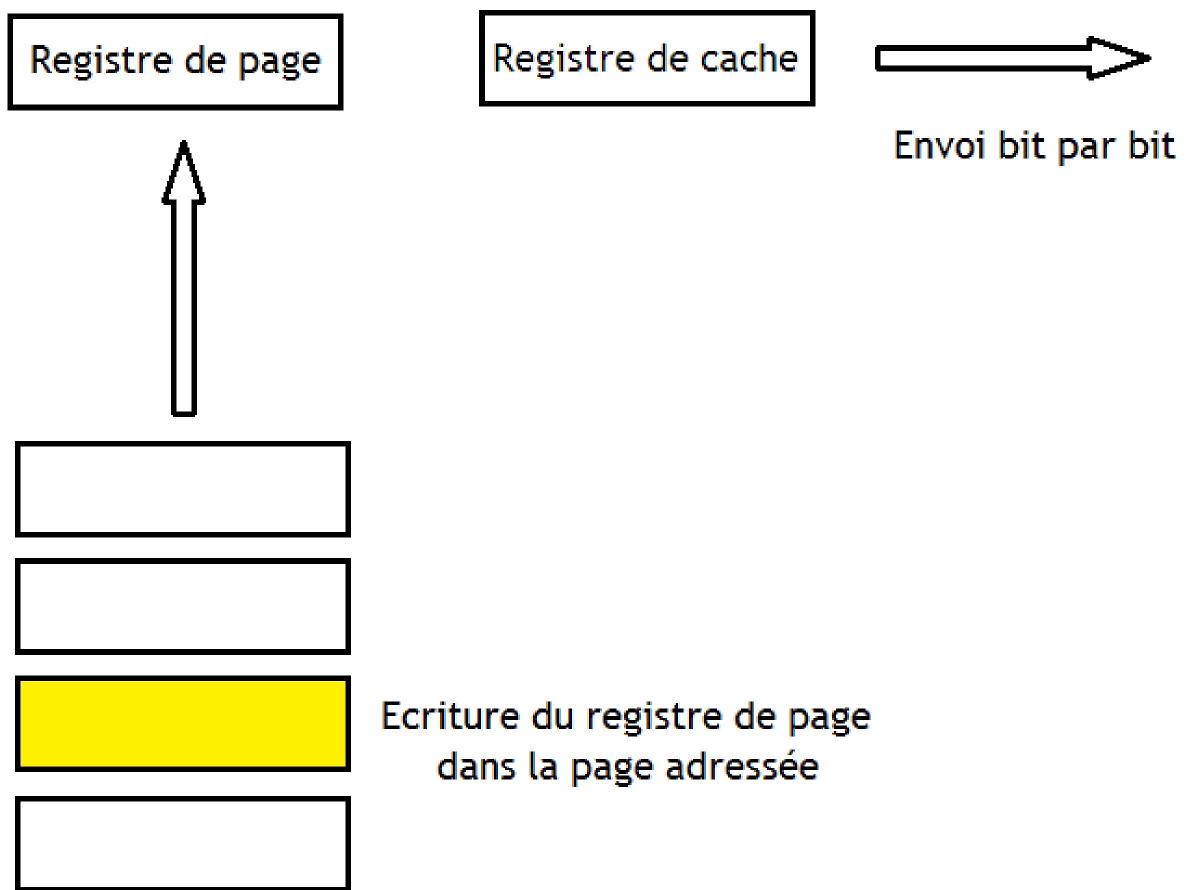

Figure 3.8 – Avec pipelining

4 Flash Transaction Layer

Le support de mémorisation est géré par deux contrôleurs. Le premier est un contrôleur qui joue le même rôle que le contrôleur mémoire pour les DRAM. Il gère les requêtes de lecture et d'écriture qui utilisent des adresses physiques, qui sont directement compréhensible par de la mémoire FLASH.

4.1 Correspondance entre adresses physiques et logiques

Mais le système d'exploitation ne peut pas gérer de telles adresses mémoires : il croit que le SSD est un disque dur comme un autre et communique avec lui comme avec tout disque dur. L'OS gère le stockage de donnée par secteurs : chaque secteur est adressé par une adresse LBA , comme le préconisent les protocoles ATA et S-ATA. Ainsi, les requêtes envoyées par le système d'exploitation doit être traduites en requêtes de lecture et d'écriture compréhensibles par la mémoire FLASH : c'est le rôle du contrôleur de SSD.

Le rôle principal du contrôleur de SSD est de traduire les adresses LBA, qui permettent d'adresser un secteur de disque dur, en adresse mémoire de FLASH. Dans les grande lignes, les adresses mémoires de FLASH permettent d'adresser des pages complètes. Pour se simplifier la vie, la majorité des SSD (si ce n'est tous) partent du principe qu'un secteur de SSD est égal à une page. Mais suivant la complexité du SSD, il existe différentes méthodes pour gérer cette correspondance.

4.1.1 Correspondance statique

Dans le cas le plus simple, la correspondance est directe. Chaque adresse LBA se voit attribuer une ou plusieurs adresses de mémoire FLASH, et la correspondance étant fixée une fois pour toute à la construction du SSD. On parle alors de **correspondance statique**.

Dans le cas le plus simple, l'adresse LBA est identique à l'adresse physique : pas besoin de traduction. Dans la réalité, il se peut que certains bits de l'adresse soient échangés, afin de répartir des adresses LBA consécutives dans des *dies*, *planes* ou blocs différents.

Cette correspondance statique effectue les écritures en place. Le SSD procède comme suit :

- il sauvegarde temporairement le contenu du bloc dans une mémoire RAM intégrée à la FLASH ;
- il efface le bloc ;
- les données sont mises à jour dans la mémoire temporaire ;
- et la totalité du bloc est réécrite, avec les données à jour.

Vu que l'on doit obligatoirement effacer un bloc de mémoire avant de pouvoir y écrire, il faut sauvegarder les pages du bloc qui ne sont pas écrites. En ajoutant à cela le fait que l'effacement du bloc prend beaucoup de temps, et on se retrouve avec des écritures très lentes.

Cette technique a un autre défaut : la durée de vie du disque dur est relativement faible. En effet, les cellules de mémoire FLASH sont fragiles et ne peuvent supporter qu'un nombre limité d'effacements. Or, avec une correspondance statique, certaines cellules seront nettement plus souvent réécrites que d'autres : après tout, certains fichiers sont très souvent accédés, tandis que d'autres sont laissés à l'abandon. Certaines cellules vont donc tomber en panne rapidement, et devenir inutilisables.

4.1.2 Correspondance dynamique

Pour régler ces problèmes, certains contrôleurs utilisent une correspondance dynamique : ils font ce que l'on appelle du ***relocate-on-write***. Avec cette correspondance dynamique, la page à écrire est écrite dans un bloc vide, et l'ancienne page ou l'ancien bloc sont alors marqués invalides. L'effacement des blocs invalides est effectué plus tard, quand le disque dur est inutilisé.

Pour gérer cet algorithme convenablement, le SSD possède une liste des pages vierges, et/ou effacés, afin de savoir où il peut écrire la page mise à jour. De plus, il doit mémoriser dans une table la liste des blocs en attente d'effacement, afin de pouvoir effacer et réclamer ces blocs au besoin.

Cette méthode demande aussi de déplacer les données sur le disque dur à chaque écriture, sans que l'adresse LBA du secteur ne change. La position d'une adresse LBA dans la mémoire FLASH change donc à chaque écriture et le disque dur doit obligatoirement se souvenir à quelle page ou bloc correspond chaque adresse LBA. Cette méthode a besoin d'une table de correspondance qui va attribuer, pour chaque adresse LBA, l'adresse de page ou de bloc qui lui correspond. Cette table est mémorisée dans une mémoire non-volatile, comme une FLASH ou une EEPROM pour mémoriser cette table de correspondance. Sur certains SSD, cette table est stockée dans les premiers secteurs du SSD, dans des blocs réservés à cet usage.

Reste que cette correspondance d'adresse peut se faire de deux manières :

— soit le contrôleur travaille au niveau du bloc,
— soit il travaille au niveau de la page ;
— soit il utilise une technique hybride.

4.1.2.1 Par pages

Certains contrôleurs fonctionnent par page : un secteur peut se voir attribuer n'importe quelle page, tant que celle-ci est vide. Si un secteur n'a encore jamais été écrit, le contrôleur de SSD va simplement prendre une page vide, et écrire dedans : il suffit juste de mettre à jour la table de correspondance avec la page vide choisie, et de marquer cette page comme occupée par une donnée. Écrire demande donc :

— d'accéder à la liste des pages vides, et en choisir une ;
— d'écrire le secteur modifié dans la page choisie ;
— de mettre à jour la table de correspondance secteur <-> page.

Mais si l'écriture modifie un secteur qui contient déjà des données, alors la table de correspondance a déjà attribué une page pour ce secteur. Évidemment, cette page, qui contient l'ancienne version du secteur, est destinée à être effacée un peu après l'écriture. En attendant, elle est considérée comme invalide par le SSD. Cela demande d'effectuer une étape supplémentaire : ajouter l'ancienne page à la liste des pages occupées destinées à l'effacement.

Avec cette technique, des données consécutives peuvent se retrouver dans des blocs différents.

Cette technique a un gros désavantage : vu qu'il faut une correspondance pour chaque page, la table de correspondance a besoin de beaucoup de mémoire. Par exemple, un disque dur de 64 gibioctets contiendra 16 mebi pages : il y a donc 16 777 216 correspondances, ce qui fait une table de plusieurs centaines de mébioctets. Difficile à tenir sur les SSD actuels sans grosses pertes de performances.

4.1.2.2 Par blocs

D'autres contrôleurs travaillent au niveau du bloc pour gérer la correspondance entre adresses : la position de la page dans un bloc est déterminée statiquement, mais le bloc dans lequel placer la donnée change. Avec cette technique, des données consécutives peuvent se retrouver dans le même bloc, et sont alors placées dans des pages consécutives.

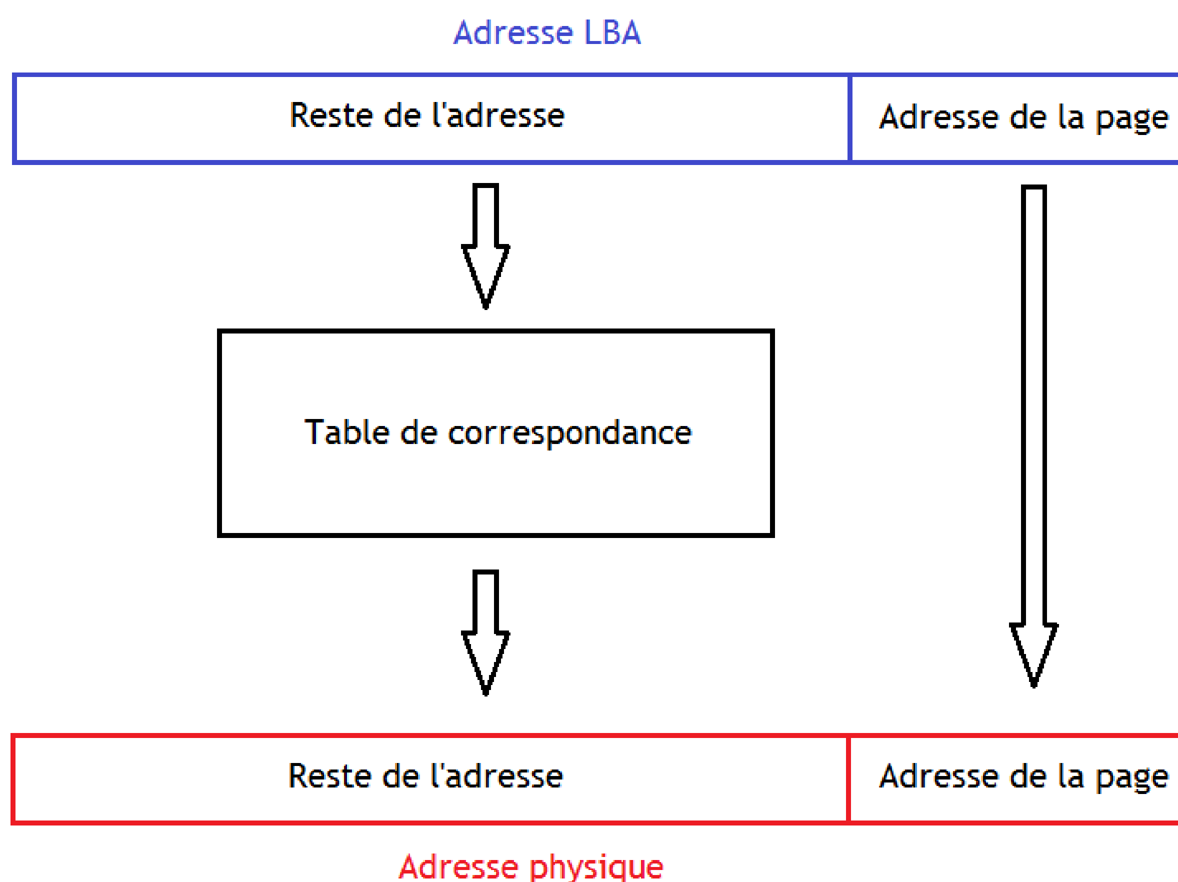

Figure 4.1 –

Vu qu'il y a plusieurs pages par blocs, la table prend donc nettement moins de place que son équivalent par page. Songez qu'on peut avoir environ 256 pages par bloc !

Écrire un secteur consiste donc à :

- lire le bloc qu'on veut modifier et le copier dans le registre de page ;
- mettre à jour le contenu du bloc dans le registre de page ;
- choisir un bloc vide et y écrire le contenu du registre de page ;
- mettre à jour la table de correspondance secteur <-> page.

4.1.2.3 Hybrides

Il existe des techniques hybrides, qui vont utiliser une table de correspondance par bloc pour les lectures, avec un mise à jour par page lors de l'écriture.

La première technique hybride est celle du **log blocks**, décrite dans [1]. Cette technique groupe les blocs en deux catégories : les data blocks gérés avec un adressage par bloc, et des logs blocks gérés avec un adressage par page. Vu le faible nombre de log blocks, la table de correspondance associée est relativement petite. Quand un bloc est écrit, un log block vide est sélectionné pour accueillir la donnée écrite. A cette occasion, l'adresse LBA de la page est mémorisée dans les bits de contrôle de celle-ci. Pour une lecture, le FTL doit commencer par vérifier la table de correspondance des log blocks : on lit le log block qui correspond si une correspondance est trouvée, et on utilise la table de correspondance des data block dans le cas contraire.

Depuis, d'autres techniques hybrides ont vu le jour : de nombreux travaux académiques sur le sujet sont disponibles sur le net, et les brevets déposés par les industriels sont aussi relativement nombreux.

4.2 Corriger les défauts de la correspondance dynamique

La correspondance dynamique n'est pas parfaite, et pose un nouveau problème : chaque écriture va gaspiller un bloc ou une page, qui contient l'ancienne version de la donnée écrite. Si un secteur est écrit 6 fois de suite, 6 blocs ou 6 pages seront utilisés pour mémoriser ce secteur dont un seul contiendra une donnée valide : ce qui fait un gâchis de 5 blocs/pages. Un SSD tomberait rapidement à court de pages si les constructeurs n'avaient pas déjà pris la mesure du problème.

4.2.1 Sur-approvisionnement

Pour limiter la casse, certains SSD contiennent des FLASH en rab : on parle de **sur-approvisionnement**. Ces FLASH sont placées dans le SSD et sont invisibles du point de vue du système d'exploitation. Ces FLASH en rab peuvent aussi servir si jamais une mémoire FLASH tombe en panne après avoir été trop souvent écrite, il suffit de la remplacer par une FLASH en rab. Ce remplacement s'effectue simplement en changeant la correspondance entre l'adresse LBA et l'adresse physique : il suffit de remplacer l'adresse physique de la FLASH défectueuse par celle de la FLASH de rechange.

4.2.2 Garbage collector

Pour limiter ce problème, le SSD peut profiter des périodes d'inactivité du SSD pour réorganiser les données sur le disque dur afin de récupérer ces pages qui contiennent des données invalides. Pour cela, le SSD doit déplacer les données des pages de manière à obtenir des blocs totalement vides, qu'il pourra alors effacer. Ces méthodes de **garbage collection** permettent de récupérer de l'espace libre, dans une certaine limite.

La procédure pour récupérer les pages invalidées est très simple, et peut se décliner en trois versions. Dans la version totale, les pages valides du bloc sont copiées dans un bloc vide, et l'ancien

1. "A SPACE-EFFICIENT FLASH TRANSLATION LAYER FOR COMPACTFLASH SYSTEMS" Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min and Yookun Cho

bloc est effacé. Dans d'autres cas, les pages valides sont copiées dans les pages vides d'un bloc déjà occupé. Et enfin, si toutes les pages d'un bloc sont invalides, alors le bloc est simplement effacé sans copie de pages.

Cette garbage collection est effectuée dans un circuit spécialisé, qui peut commander la mémoire FLASH indépendamment du Flash Transaction Layer. Un SSD qui utilise une correspondance dynamique est donc composé au minimum d'un Flash Transaction Layer pour traduire les adresses LBA et numéro de page, et d'un Garbage Collector, ainsi que d'autres circuits annexes.

Ce garbage collector a toutefois un défaut : il est obligé de migrer des pages dans la mémoire FLASH, ce qui est à l'origine d'écritures supplémentaires. Ce phénomène s'appelle l'**amplification d'écriture**.

Il existe plusieurs algorithmes spécialisés pour effectuer cette garbage collection, et il existe un algorithme très simple qui permet de limiter le nombre d'écritures supplémentaires dues à la garbage collection. Cet algorithme est très simple :

— il retarde la garbage collection jusqu'au moment où l'espace libre descend en dessous d'un certain seuil (chose assez irréaliste pour un SSD normal) ;
— il choisit un bloc à la fois :
— il efface de préférence les blocs dont un maximum de page est invalide, afin d'éviter d'avoir à copier les pages valides du bloc (dans le meilleur de cas, tout le bloc peut être effacé, ce qui supprime la nécessité des recopies).

4.2.3 TRIM

Cependant, ces algorithmes de garbage collection ont une limite : ils fonctionnent au niveau du matériel. Or, une donnée peut parfaitement devenir inutile et être quand même présente sur le disque dur. C'est le cas sur beaucoup de système d'exploitation actuels : quand on supprime un fichier, ils n'effacent pas réellement les données, mais se contentent d'oublier leur localisation sur le disque dur. Les données sont toujours là, mais le système d'exploitation a modifié ses tables qui lui permettent de savoir où est le fichier sur le disque dur, rendant celui-ci inaccessible. Mais pour le SSD, ces données sont toujours présentes sur le disque dur, et les pages qu'elles occupent ne sont pas effacées : elles sont inutilisables et prennent de la place pour rien. Dans ces conditions, de nombreuses pages sont gâchées et l'espace libre du SSD diminue rapidement.

Pour éviter ce genre de problème, les constructeurs de SSD ont ajouté une commande au protocole de communication avec le disque dur : la commande TRIM. Celle-ci permet au système d'exploitation d'indiquer au SSD les secteurs qui correspondent à des fichiers supprimés. Ainsi, quand l'OS supprime un fichier, il peut avertir le SSD que les pages qui correspondent au fichier doivent, et peuvent être effacées. Cela permet d'éviter de gâcher des pages, et permet aux algorithmes de garbage collection de fonctionner plus efficacement.

4.2.4 Effacement sécurisé

L'amplification d'écriture est à l'origine d'un autre problème, qui concerne les situations où l'on veut effacer des données sans que qui que ce soit ne puisse les récupérer. Avec des technologies de pointes, il est parfois possible de récupérer physiquement des données présentes sur un disque dur, même si celui-ci est partiellement détruit ou que les données en question ont étés écrasées. Et cela pose quelques problèmes aux agences gouvernementales, aux agences militaires, ou aux

utilisateurs qui stockent des données sensibles : si vous ne voulez pas que qui que ce soit vous vole vos données, il vaut mieux savoir comment faire pour effacer les données de manière sécurisée.

Quand vous voulez effacer des données sur un disque dur, il vous suffit de réécrire des données aléatoires ou des zéros sur les secteurs concernés plusieurs centaines ou milliers de fois de suite. Mais si vous essayez de faire cela avec des SSD, cela n'effacera pas les données écrites dans la mémoire FLASH : cela se contentera de prendre des blocs vides pour écrire les données aléatoires dedans, sans toucher aux données à effacer. Cependant la réécriture complète d'un disque dur peut fonctionner. Mais un paramètre peut poser quelques problèmes : dans les procédures standards utilisées par certains gouvernements, le flux de bits à récrire est normé : on récrit tout le disque dur avec des zéros, puis avec des 1, et avec un caractère aléatoire. Seul problème : certains SSD compressent la donnée à écrire : si le flux de bits n'est pas aléatoire, il se peut que certaines pages ou certains bits ne soient tout simplement pas réécrits.

Pour effacer de manière sécurisée un fichier unique, vous allez avoir de gros problèmes à cause des phénoménes d'amplification d'écriture. Dans ces situations, tous les protocoles d'effacement sécurisé utilisés par les gouvernements actuels ne fonctionnent tout simplement pas [2].

Pour régler ce problème, certains SSD modernes possèdent des commandes qui permettent de remettre à zéro l'intégralité du disque dur, ou d'effacer des pages ou blocs bien précis. Mais si jamais le SSD ne fournit que les commandes de lecture et d'écriture d'un bloc, vous devrez utiliser les commandes d'écritures normales et beaucoup d'ingéniosité pour réussir à effacer vos données.

4.3 Wear leveling

Certains contrôleurs de SSD utilisent une technique pour augmenter la durée de vie du SSD : le **wear leveling**, ou égalisation d'usure. L'idée est simple : on va chercher à répartir les écritures de manière à ce que tous les blocs soient "usés" de la même façon : on va faire en sorte que le nombre d'écriture soit identique dans tous les blocs. Pour cela, il suffit de déplacer les données fortement utilisées dans des blocs pas vraiment usés. Vu que les données fréquemment utilisées sont présentes sur des blocs très usés, qui ont un grand nombre d'écriture, on peut considérer que le wear leveling consiste à déplacer les données des blocs usés dans les blocs relativement intacts.

Les algorithmes de wear leveling sont particulièrement nombreux, et il serait difficile d'être exhaustif sans y passer beaucoup de temps. Aussi, je vais me contenter de décrire les deux grandes catégories d'algorithmes de wear leveling :

— le wear leveling dynamique ;
— le wear leveling statique.

4.3.1 Wear leveling dynamique

Le plus simple est le wear leveling dynamique. Avec celui-ci, seuls les blocs qui sont réécrits sont déplacés dans des blocs neufs. Quand on veut réécrire une donnée, on va choisir convenablement le bloc vide dans lequel on écrire la donnée à mettre à jour : il suffit de choisir un bloc qui a relativement peu d'écritures a son actif, le choix en lui-même étant effectué par un algorithme relativement compliqué.

2. " Reliably Erasing Data From Flash-Based Solid State Drives Michael Wei " , Laura M. Grupp, Frederick E. Spada, Steven Swanson.

Pour cela, le contrôleur de SSD doit mémoriser le nombre d'écritures de chaque bloc. Et pour cela, il contient une mémoire non-volatile qui stocke un compteur d'écriture pour chaque bloc : ce compteur est initialisé à 0 lors de la fabrication du SSD, et est incrémenté de 1 à chaque écriture.

4.3.2 Wear leveling statique

Avec ce wear leveling dynamique, les données qui ne sont pas réécrites souvent, ou qui sont en lecture seule, ne bougent quasiment jamais du bloc qui leur est assigné : celui-ci aura donc un nombre d'écriture particulièrement faible. Cette solution est loin d'être optimale : il vaut mieux déplacer des telles données sur des blocs très usés, et allouer leur emplacement initial à une donnée fréquemment mise à jour. C'est ce principe qui se cache derrière le **wear leveling statique**.

Avec le wear leveling statique, le contrôleur va choisir le bloc dans lequel réécrire la donnée parmi tous les blocs du disque dur : il va prendre un bloc qui possède relativement peu d'écritures, ce qui correspond à des blocs qui contiennent des données statiques, ou des blocs vides peu utilisés. Il va de soit que le contrôleur doit mémoriser la liste des blocs peu utilisés, en plus de la liste des blocs vides. Dans le cas où le bloc choisi contient une donnée :

- la donnée est déplacée dans un bloc vide qui a un fort nombre d'écriture ;
- le bloc est effacé avant d'être utilisé pour la réécriture.

4.4 Gestionnaire de requêtes

Le Flash Transaction Layer et le contrôleur mémoire de FLASH sont les circuits qui permettent de faire l'interface entre la mémoire FLASH et les requêtes envoyées par le processeur. Mais la gestion de ces requêtes n'est pas gratuite, et a un circuit dédié, similaire à celui que l'on trouve sur les disque durs.

Ce circuit est relativement simple : il s'agit le plus souvent d'une simple file d'attente dans laquelle les requêtes vont attendre que le support de mémorisation doit disponible. Ajouter cette file d'attente évite au processeur d'attendre qu'une requête soit terminée pour envoyer la suivante, et permet ainsi d'augmenter le nombre de requêtes traitées par seconde.

4.4.1 Dynamic write mapping

Ce gestionnaire de requête peut communiquer avec le Flash transaction Layer, histoire de fournir quelques optimisations assez intéressantes. Par exemple, il peut profiter du fait que des planes ou dies séparés peuvent être accédés en parallèle. Cela demande de répartir les données sur le disque dur, de manière à ce que des données consécutives soient réparties dans des planes/dies différents. Cette technique permet d'implémenter une forme de pipelining assez efficace du point de vue des performances du SSD.

Pour cela, la méthode est simple : il suffit de faire en sorte que des écritures consécutives atterrissent dans des dies/planes différents. Le Flash Transaction Layer doit donc communiquer avec le gestionnaire de requêtes afin de savoir si le processeur a envoyé des écritures consécutives au SSD : si c'est le cas, le FTL va sélectionner convenablement les pages dans lesquelles écrire les données de facon à ce qu'elle appartiennent à des planes/dies différents.

4.4.2 Native Command Queueing

Une autre solution consiste à réorganiser l'ordre des requêtes pour accéder le plus possible à des planes ou dies indépendants.

J'ai dit plus haut que les planes ont un certain degré d'indépendance : elles peuvent effectuer une même opération (lecture ou écriture) en même temps. On peut ainsi demander à plusieurs planes d'effectuer une écriture, la donnée à écrire étant différente pour chaque plane.

On peut profiter de ce genre de parallélisme en regroupant les requêtes de lecture et d'écriture dans la file d'attente, et en les fusionnant en une seule requête d'écriture/lecture envoyée aux planes en parallèle. C'est la technique du *Multi-plane rescheduling*.

4.4.3 Write Buffer/Coalesing Buffer

Une autre solution consiste à mettre les écritures en attente dans une mémoire tampon : le ***write buffer***. Cette mémoire cache sert juste à accumuler les écritures, et mémorise la donnée à écrire et l'adresse physique de la page à modifier. Ce tampon peut permettre de réorganiser les écritures de façon à ce que des écritures dans des pages consécutives aient lieu les unes à la suite des autres dans le temps, au lieu d'avoir des écritures éloignées dans le temps à des pages consécutives.

5 Conclusion

Pour les techniques de correspondance du Flash Transaction Layer, certains designs sont assez “renommés”, et devraient être connus de tous ceux qui s’intéressent aux SSD en profondeur :

— “LAST : Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems”, Sungjin Lee, Dongkun Shin, Young-Jin Kim and Jihong Kim
— “DFTL : A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings”, Aayush Gupta Youngjae Kim Bhuvan Urgaonkar
— “A Superblock-based Flash Translation Layer for NAND Flash Memory”, Jeong-Uk Kang Heeseung Jo Jin-Soo Kim Joonwon Lee