FFSMark : Un Benchmark pour Systèmes de Fichiers Dédiés aux Mémoires Flash

Pierre Olivier, Jalil Boukhobza

Compas'2015 : Parallélisme/ Architecture / Système
Lille, France, du 30 juin au 3 juillet 2015

FFSMark : Un Benchmark pour Systèmes de Fichiers Dédiés aux Mémoires Flash

Pierre Olivier*, Jalil Boukhobza*

*Université Européenne de Bretagne, Université de Bretagne Occidentale,
UMR 6585 Lab-STICC
20 Avenue Le Gorgeu, 29200 Brest - France
pierre.olivier@univ-brest.fr, jalil.boukhobza@univ-brest.fr

Résumé
La mémoire flash de type NAND est le principal média de stockage dans l'embarqué. L'un des moyens pour intégrer cette mémoire dans les systèmes informatiques est d'utiliser des systèmes de fichiers dédiés aux mémoires flash (*Flash File Systems*, FFS). Dans ce domaine, les benchmarks sont des programmes permettant de réaliser des études de performances et de comparer différents systèmes entre eux. Nous montrons dans cet article qu'un benchmark pour FFS doit prendre en compte les caractéristiques spécifiques des mémoires flash, d'une part dans son comportement, et d'autre part en ce qui concerne les métriques de performances disponibles en sortie. Dans ce contexte, nous proposons *FFSMark*, un benchmark ciblant les FFS, sensible aux spécificités des mémoires flash. FFSMark est dédié à être exécuté sous Linux, un système d'exploitation supportant les FFS les plus populaires. Nous présentons également une étude de cas, utilisant FFSMark pour comparer les performances des FFS JFFS2, YAFFS2 et UBIFS sur une plate-forme embarquée.

Mots-clés : Mémoire flash NAND, Systèmes de fichiers, Benchmarking, Linux

---

1. Introduction

Les mémoires flash sont aujourd'hui fortement présentes dans les systèmes informatiques. Alors que les disques à base de mémoire flash remplacent petit à petit les disques durs, ces mémoires sont depuis longtemps le principal média de stockage secondaire dans l'embarqué. Elles offrent en effet de nombreux avantages : résistance au chocs, faible consommation énergétique, bonnes performances d'entrées / sorties (E/S). Néanmoins, ces mémoires présentent des contraintes d'utilisation qui leur sont propres, en particulier (1) l'impossibilité de mettre à jour des données en place, et (2) l'usure de la mémoire au fil des écritures. Ainsi, des systèmes de gestion spécifiques sont intégrés aux systèmes de stockage flash, pour permettre leur utilisation en satisfaisant les contraintes et en les masquant au système d'exploitation (*Operating System*, OS) hôte. Dans l'embarqué, les systèmes de fichiers dédiés aux mémoires flash (*Flash File Systems*, FFS) sont un type de systèmes de gestion fortement utilisé. Les FFS les plus populaires sont JFFS2 [17], YAFFS2 [18] et UBIFS [4]. Ils sont tous trois supportés par Linux, l'un des principaux OS embarqués [16].
Les benchmarks sont fortement utilisés dans le domaine des systèmes de fichiers pour la mesure et la comparaison de performances, la compréhension du fonctionnement de certains sys-

tèmes ou mécanismes, ou encore la validation de nouveaux systèmes ou de propositions d'optimisations de systèmes existants. Une grande partie des benchmarks pour systèmes de fichiers actuels supposent un disque dur comme média de stockage. Nous montrons dans cet article qu'un benchmark ciblant les FFS doit prendre en compte les spécificités du média de stockage flash (l'impossibilité de mettre à jour des données en place, et l'usure de la mémoire) afin de donner des résultats exhaustifs concernant les performances. Pour répondre à cette problématique, nous proposons FFSMark, un benchmark développé pour mesurer les performances des FFS sous Linux, sensible aux contraintes spécifiques des mémoires flash de type NAND. Nous donnons également des résultats d'un cas d'application de notre benchmark, montrant l'importance de la prise en compte des spécificités du média de stockage flash.
Cet article est organisé comme suit : dans une première section sont présentées des notions générales concernant les mémoires flash et leur gestion via systèmes de fichiers dédiés sous Linux. La section suivante présente les travaux en relation dans le domaine du benchmarking de FFS. Dans une troisième section, le benchmark FFSMark est présenté, et un cas d'étude est donné dans la quatrième section. Enfin, une conclusion est donnée en cinquième section.

2. Mémoires flash et gestion sous Linux

2.1. Notions de base concernant les mémoires flash

Il existe plusieurs types de mémoire flash. Dans ce travail, on s'intéresse uniquement à la mémoire flash de type NAND, dédiée au stockage de données non volatile. Une puce de mémoire flash NAND possède une architecture hiérarchique : elle contient un ou plusieurs plans. Chaque plan contient un certain nombre de blocs, et chaque bloc contient des pages. La taille d'une page sur les puces actuelles est de 2, 4 ou 8 Ko. Le nombre de pages par bloc est généralement de 64 ou 128 [2]. Une puce supporte trois opérations principales : la lecture, l'écriture et l'effacement. La lecture et l'écriture s'effectuent au niveau d'une page. L'effacement est quant à lui réalisé au niveau d'un bloc complet. Les temps d'accès varient selon le modèle de puce [2]. Des exemples de latences sont : 25 µs pour une lecture de page, 220 µs pour une écriture, et 500 µs pour un effacement de bloc [10].

2.2. Contraintes et gestion

L'usage des mémoires flash dans les systèmes informatiques implique de composer avec les différentes contraintes spécifiques à ce type de mémoire. L'une des principales contraintes est la règle *effacer avant d'écrire*. Il n'est pas possible d'écrire dans une page qui contient déjà des données : il faut au préalable effacer le bloc la contenant. Une seconde contrainte est l'*usure* de la mémoire. Un bloc de mémoire flash ne peut supporter qu'un nombre limité d'effacements. Passé un certain seuil, le bloc ne peut plus être utilisé. Du fait de ces contraintes, des systèmes de gestion sont intégrés aux systèmes de stockage à base de mémoire flash. Ces systèmes permettent d'utiliser la mémoire tout en respectant les contraintes. Cette gestion est réalisée de manière transparente pour l'OS hôte. Les systèmes de gestion mettent en œuvre les concepts suivants :

1. La règle *effacer avant d'écrire* est gérée via la mise en place d'une traduction d'adresses logiques (vues par l'OS) vers des adresses physiques (en mémoire flash). Cela permet d'écrire les données d'une page flash mise à jour dans une autre page flash : il s'agit de mise à jour de données hors place. Les pages contenant les anciennes versions des données ne sont pas effacées directement, mais marquées *invalides*. L'effacement survient plus tard, lors de l'exécution d'un processus nommé le *ramasse-miettes*, qui recycle de l'espace flash invalide en espace flash libre. Le ramasse-miettes fonctionne comme suit :

un bloc *victime* contenant des pages invalides est sélectionné. Les pages toujours valides sont recopiées à un endroit différent, puis le bloc est effacé ;

2. L'usure de la mémoire est gérée via l'implémentation de mécanismes de *répartition de l'usure*, qui répartissent les cycles d'écritures / effacements sur l'intégralité de la mémoire flash afin d'en maximiser la durée de vie.

On distingue deux types de systèmes de gestion : la FTL (*Flash Translation Layer*) est une solution matérielle et logicielle contenue dans le contrôleur de périphériques de stockage à base de mémoire flash tels que les clé USB, disques SSD ou encore les cartes SD/MMC. Les systèmes de fichiers dédiés (*Flash File Systems*, FFS) représentent quant à eux une solution purement logicielle, implémentée au niveau de l'OS de systèmes embarqués complexes tels que les smartphones, tablettes, set-top boxes, etc. Ces FFS gèrent des puces flash directement soudées (dites puces flash *embarquées*) sur les cartes mères des systèmes concernés. Dans cet article on s'intéresse uniquement aux FFS et aux puces flash embarquées. Linux est l'un des OS embarqués les plus répandus, et il supporte les FFS les plus populaires.

2.3. Linux et les systèmes de fichiers dédiés aux mémoires flash

Linux supporte les principaux FFS. Le stockage à base de FFS peut être vu sous Linux comme une pile : on retrouve en haut de la pile les applications qui réalisent des accès aux fichiers via des appels systèmes tels que `open`, `read`, `write`, etc. Ces appels systèmes sont reçus au niveau du noyau par le système de fichiers virtuel (*Virtual File System*, VFS), qui est une couche d'abstraction pour tous les systèmes de fichiers supportés par Linux. VFS transfère les accès au système de fichiers concerné, dans notre cas un FFS. Enfin, le FFS fait appel au pilote de la puce de mémoire flash pour réaliser des opérations sur la mémoire. Sous Linux, toutes les puces de mémoire flash supportées utilisent un pilote générique nommé *Memory Technology Device* (MTD). Au niveau de VFS, le *page cache* est un cache de données qui tamponne tous les accès aux fichiers réalisés par les applications. Le page cache utilise potentiellement l'intégralité de la mémoire vive qui n'est pas utilisée par les programmes.

Les trois principaux FFS utilisés aujourd'hui sont JFFS2 [17], YAFFS2 [18] et UBIFS [4]. JFFS2 (*the Journaling Flash File System version 2*) est nativement intégré au noyau depuis 2001 (Linux 2.4.10), et est par conséquent relativement mature. YAFFS2 (*Yet Another Flash File System*) date également de 2001, et est intégré à Linux par l'intermédiaire d'un patch. UBIFS (*the Unsorted Block Image File System*) est quant à lui le plus récent des FFS, il est intégré nativement à Linux depuis 2008 (Linux 2.6.27).

Pour des raisons d'espace, on ne donne pas ici les détails d'implémentation de ces 3 systèmes de fichiers. On peut néanmoins s'attarder sur les différences majeures qui les caractérisent. Premièrement, UBIFS étant relativement récent, son implémentation s'inspire des limitations de JFFS2 et de YAFFS2. En particulier, UBIFS utilise des arbres comme structure d'indexation pour les fichiers gérés, alors que JFFS2 et YAFFS2 utilisent des tables. Cela fait que l'empreinte RAM et le temps de montage de UBIFS croissent de manière logarithmique avec la quantité d'espace flash géré, alors que pour ses concurrents cette croissance est d'ordre linéaire [4]. De plus, UBIFS supporte l'écriture différée depuis le page cache (*write back*), ce qui lui permet de profiter de la localité temporelle des accès aux fichiers. Les écritures de JFFS2 et de YAFFS2 sont quant à elles uniquement synchrones. Enfin, UBIFS et JFFS2 supportent la compression / décompression des données à la volée lors des écritures et lectures. YAFFS2 ne supporte pas la compression.

3. Travaux en relation : benchmarking de systèmes de fichiers et mémoires flash

Dans le domaine des systèmes de fichiers, les benchmarks sont des séries de tests utilisés pour mesurer diverses métriques de performances. Ils sont utiles pour comparer des systèmes entre eux, mais aussi pour comprendre le comportement d'un système de fichiers, ou encore évaluer l'impact d'une optimisation apportée à un système donné.
On distingue trois types de benchmarks pour systèmes de fichiers [15]. Les *macro-benchmarks* appliquent une charge d'E/S constituée de multiples opérations, censée être représentative d'un environnement applicatif particulier. Les *micro-benchmarks* se concentrent quant à eux sur un nombre réduit d'opérations (une ou deux) pour tenter d'en inspecter les performances. Enfin, la ré-exécution de *traces* permet de rejouer sur un système de stockage une charge d'E/S enregistrée sur un autre système.
Dans le domaine des FFS, les benchmarks sont majoritairement utilisés pour comparer les performances de différents FFS [7, 8, 9, 5, 13]. Certaines études présentent des comparaisons de performances de FFS existants. D'autres travaux, proposant un nouveau FFS ou une optimisation de FFS existant, comparent leur proposition à l'existant via des benchmarks. La plupart de ces études utilisent des micro-benchmarks *ad-hoc* (développés dans le cadre de l'étude). En effet, il n'existe pas de macro-benchmark standardisé dédié au domaine d'application des FFS : celui des systèmes embarqués. Le macro-benchmark *Postmark* [6] fait en quelque sorte figure d'exception : bien que la charge d'E/S produite (serveur mail) ne soit pas forcément représentative d'une application embarquée, il est fortement utilisé dans le domaine de l'évaluation de performances de FFS [7, 8, 9, 5]. La forte utilisation de Postmark dans l'embarqué s'explique d'une part par sa grande popularité, et d'autre part par la simplicité de la charge d'E/S produite (décrite en section suivante). Si cette dernière n'est pas représentative d'une application embarquée, elle est suffisamment généraliste pour offrir un indice de performances pour un système de fichiers donné, et permettre la comparaison des performances de plusieurs systèmes de fichiers.
On peut constater que la plupart des benchmarks pour systèmes de fichiers supposent un disque dur comme média de stockage [15]. Il n'existe pas, à notre connaissance, de benchmark considérant la mémoire flash. Il est montré en section précédente que la mémoire flash NAND possèdent des caractéristiques qui lui sont propres. Premièrement, plusieurs études ont montré l'impact important sur les résultat d'une étude de performances de *l'état de départ* du système de stockage testé, c'est à dire la quantité d'espace flash valide / invalide / libre [1, 11]. En effet, la présence de données invalides en début de benchmark déclenche l'exécution du ramasse-miettes pendant les tests, ce qui perturbe les temps d'accès des E/S relatives au benchmark. La présence de données valides fait que le ramasse-miettes doit recopier un certain nombre de pages valides au cours de son travail. Ces accès flash supplémentaires, réalisés pendant le benchmark, impactent les résultats en termes de performances.
Deuxièmement, dans le cadre des FFS, il est important de prendre en compte cette caractéristique spécifique qu'est l'*usure* de la mémoire flash, et d'ajouter la répartition de l'usure aux métriques de performances classiques disponibles en sortie d'un benchmark.

4. FFSMark : un benchmark pour systèmes de fichiers dédiés aux mémoires flash

4.1. Postmark

Postmark [6] est choisi comme base pour FFSMark car il est utilisé dans de nombreuses études concernant l'évaluation de performances de FFS [7, 8, 9, 5]. De plus, son implémentation est relativement simple, et il est aisément modifiable.

Un lancement de Postmark consiste en l'exécution de plusieurs phases. Premièrement, lors de la phase de *création*, un certain nombre de fichiers contenant des données aléatoires sont créés, possiblement répartis dans plusieurs répertoires. La seconde phase est la phase dite de *transactions*. Une transaction consiste en deux sous-phases : (A) soit une lecture d'un fichier existant, soit une écriture en fin de fichier existant (*append*) puis (B) soit la création d'un nouveau fichier, soit la suppression d'un fichier existant.
Postmark est lancé avec plusieurs paramètres : le nombre de fichiers initialement créés, leur tailles, le nombre de transactions, la chance de choisir une lecture plutôt qu'une écriture en sous-phase de transactions (A), etc. En sortie, diverses statistiques sont proposées : le temps d'exécution total, le temps d'exécution de chaque phase, les débits en lecture / écriture, etc.

4.2. FFSMark

L'objectif de FFSMark est l'adaptation de Postmark pour répondre aux limitations présentées en section 3 : pour un benchmark ciblant un système de stockage à base de mémoire flash, il est nécessaire de prendre en compte (1) l'état de départ quant à la quantité d'espace flash libre / valide / invalide et (2) la répartition de l'usure. FFSMark se base sur la version 1.5 de Postmark, et est écrit en langage C.
Gestion de l'état de départ - FFSMark permet de définir la quantité d'espace flash libre, valide et invalide présente sur la partition testée avant la phase de création. Pour ce faire, FFSMark crée sur la partition testée un fichier contenant des données aléatoires, qui est laissé tel quel durant l'exécution du benchmark. La taille de ce fichier représente autant de données valides en flash. Un autre fichier est créé, contenant lui aussi des données aléatoires. Ce second fichier est supprimé (via l'appel système `remove()`) juste avant l'exécution de la phase de création. La suppression provoque l'invalidation des données composant le fichier au niveau FFS.
FFSMark permet de paramétrer les tailles de ces deux fichiers en fonction de l'espace disponible sur la partition de test au moment du lancement du benchmark. Cela permet de maîtriser l'état de départ et de prendre en compte ce facteur impactant fortement les résultats d'une étude de performances. La présence de données invalides déclenche pendant le benchmark l'exécution du ramasse-miettes, ce qui impact les performances. Cet impact est d'autant plus important qu'un certain nombre de pages toujours valides doivent être recopiées au cours du lancement de ce mécanisme. Bien entendu, pour une maîtrise optimale, il est important de lancer FFSMark sur une partition flash fraîchement effacée : on s'assure ainsi que tout l'espace flash qui n'est pas dédié aux deux fichiers précédemment présentés est bien libre (effacé).
Statistiques sur les accès flash et la répartition de l'usure - FFSMark peut être exécuté conjointement avec Flashmon [12], qui est un module pour le noyau Linux permettant une trace des accès flash au niveau du pilote NAND générique MTD : lectures et écritures de pages, effacements de blocs. Il a été montré par la mesure que l'impact de Flashmon sur les performances des accès flash est négligeable (inférieur à 5%) [12].
FFSMark communique avec Flashmon et propose en sortie des statistiques sur les accès flash : le nombre de lectures et écritures de pages, d'effacement de blocs, le nombre moyen d'effacements par bloc, l'écart type de la distribution des compteurs d'effacements pour chaque bloc, ainsi que la différence en nombre d'effacements entre le bloc le plus effacé et le bloc le moins effacé. Les statistiques concernant les effacements permettent d'estimer la qualité de la répartition de l'usure effectuée par le FFS au cours du benchmark.
Amélioration diverses par rapport à Postmark - Dans [15], les auteurs notent un certain nombre de limitations relatives à Postmark, en particulier l'imprécision des mesures temporelles réalisées par le benchmark (via la primitive système `time()`). FFSMark utilise la primitive *gettimeofday()* pour mesurer le temps, ce qui lui donne une précision de l'ordre de la microseconde.

Paramètre	**Valeur**
Paramètres relatifs à Postmark	
Nombre de fichiers créés	5000
Nombre de sous-répertoires	50
Tailles des fichiers créés (octets)	aléatoire entre 512 et 10240
Nombre de transactions	15 000
Taille des lectures / écritures (octets)	4096
Chance de choisir une lecture plutôt qu'une écriture en sous-phase de transactions (A) (%)	50
Chance de choisir une création plutôt qu'une suppression en sous-phase de transactions (B) (%)	50
Fonctions de lecture et d'écriture	`read()` et `write()`
Paramètres relatifs à FFSMark	
Obtention de statistiques sur les accès flash via Flashmon	Oui
Vidage du page cache avant la phase de création	Oui
Vidage du page cache avant la phase de transactions	Oui
Espace flash valide avant la phase de création (% de la partition de test)	conf. *libre* : 0, conf. *âgée* : 50
Espace flash invalide avant la phase de création (% de la partition de test)	conf. *libre* : 0, conf. *âgée* : 25

TABLE 1 – Paramètres des configurations de FFSMark exécutées.

De plus, FFSMark offre la possibilité de vider le page cache (1) avant la phase de création et surtout (2) avant la phase de transactions. En effet, la phase de création peut être vue comme le chargement d'un état de départ pour le système, et la phase de transactions peut elle être vue comme l'exécution d'un régime permanent d'E/S. Il est donc naturel de vider les caches de données de l'OS entre ces deux phases.

5. Cas d'étude

Dans cette section, on présente un cas d'application de FFSMark. Il s'agit de la comparaison des performances des trois principaux systèmes de fichiers JFFS2, YAFFS2 et UBIFS, sur la carte de développement embarquée *Armadeus APF27*.

5.1. Méthodologie

Deux configurations de FFSMark sont définies pour notre étude. Ces configurations diffèrent uniquement dans l'état de départ donné à la mémoire flash via FFSMark. La configuration dite *système de fichiers libre* lance le benchmark sur une partition de test libre à 100% : la partition est complètement effacée avant le lancement de chaque test. La configuration *système de fichier âgé*, quant à elle, définit un taux d'espace valide de 50%, et un taux d'espace invalide de 25%. Ces valeurs sont choisies pour permettre d'observer l'écart de performances dû à l'état de départ. La taille de la partition de test est de 100 Mo.

Les autres paramètres de FFSMark sont définis pour la génération d'une charge d'E/S suffisamment importante pour stresser le système de fichiers. Concernant les paramètres introduits par FFSMark (par rapport à Postmark), le page cache est vidé avant la phase de création et avant la phase de transactions. La récolte de statistiques sur les accès flash est activée. La table 1 présente les valeurs pour les différents paramètres des deux configurations.

Les deux configurations sont lancées sur JFFS2, YAFFS2 et UBIFS, exécutés sous Linux 2.6.29,

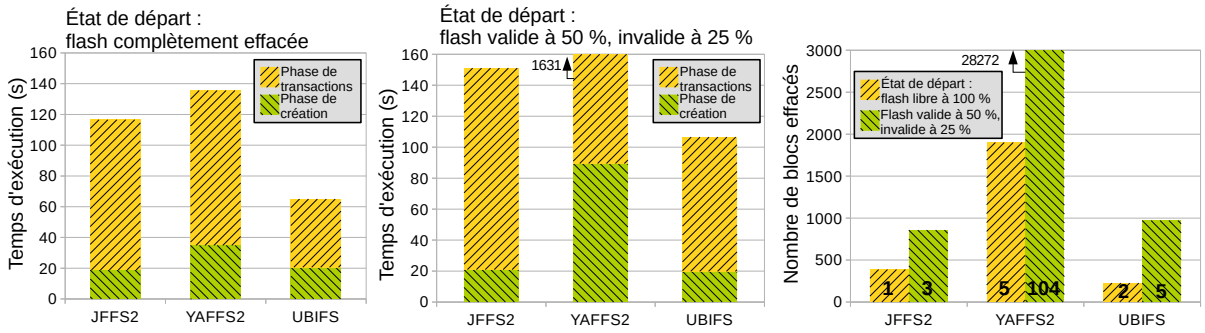

FIGURE 1 – **Impact de l'état de départ :** temps d'exécution des deux phases du benchmark (à gauche et au centre) et répartition de l'usure (à droite) en fonction de l'état de départ. Le nombre inscrit dans chaque histogramme de la figure de droite correspond à la différence entre le compteur d'effacement du bloc le plus effacé et le compteur du bloc le moins effacé. Certaines barres sont coupées pour YAFFS2 pour que le graphique reste lisible. Leur valeurs en ordonnée y sont inscrites textuellement.

lui même exécuté sur la carte *Armadeus APF27*. Cette carte contient un processeur Freescale i.MX27 à base de coeur ARM9, cadencé à 400 Mhz, et 128 Mo de RAM. La carte contient également 256 MO de mémoire flash NAND de marque Micron (pages de 2 Ko, 64 pages par bloc) [10]. Avant chaque lancement de FFSMark, la partition flash de test est effacée et un nouveau système de fichiers est créé.
UBIFS et JFFS2 supportant la compression, pour chacun de ces FFS chaque configuration est lancée avec la compression activée et désactivée. UBIFS est l'unique FFS qui supporte l'écriture différée (*write-back*) depuis le page cache, pour ce FFS chaque configuration est donc lancée avec l'écriture différée activée, et désactivée.

5.2. Résultats

5.2.1. Impact de l'état de départ

Les graphiques à gauche et au centre de la figure 1 représentent les temps d'exécution des deux phases du benchmark (création et transactions), en fonction du type de système de fichiers testé, et de l'état de départ (*libre* ou *âgé*). Le graphique de droite sur cette même figure 1 représente le nombre d'effacements de blocs constatés, et la différence entre le compteur d'effacement du bloc le plus effacé et le compteur du bloc le moins effacé. Pour ces tests, la compression est désactivée pour JFFS2 et UBIFS. L'écriture différée est activée pour UBIFS.
Une première observation importante est la forte augmentation des temps d'exécution sur la configuration *âgée*, par rapport à la configuration *libre*. Cela est particulièrement vrai pour le temps d'exécution de la phase de transactions. On constate concernant cette phase une augmentation de 95% pour UBIFS, 34% pour JFFS2, et plus de 1400 % pour YAFFS2 qui semble se comporter particulièrement mal sur la configuration âgée. Ces résultats montrent l'importance de l'état de départ lors d'une évaluation de performances (entre autres, du benchmarking) pour système de stockage à base de flash.
Si le pourcentage d'augmentation pour UBIFS est supérieur à celui de JFFS2, les temps d'exécutions restent meilleurs pour UBIFS. De manière générale, on peut noter que, indépendamment de l'état de départ, les temps d'exécutions avec UBIFS sont fortement inférieurs aux temps d'exécution de JFFS2 et de YAFFS2. Cela est dû au support de l'écriture différée depuis le page cache par UBIFS, qui lui permet de profiter de la localité temporelle des accès aux fichiers en

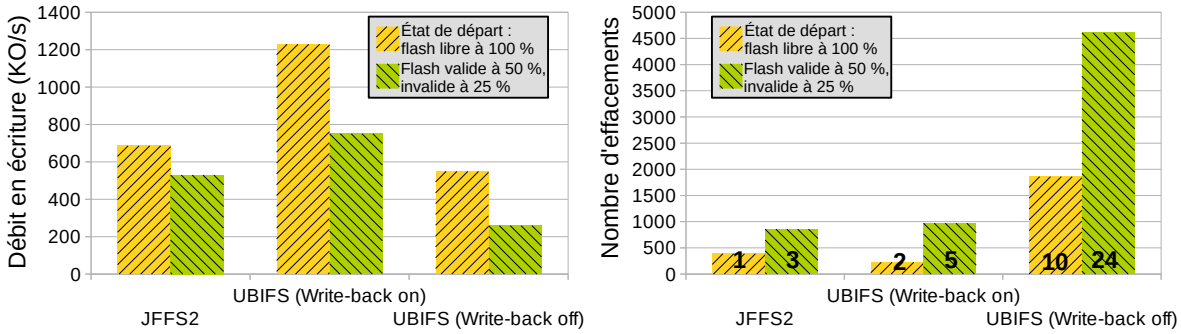

FIGURE 2 – **Désactivation de l'écriture différée pour UBIFS :** Débits en écriture (à gauche et au centre) et répartition de l'usure (à droite) pour JFFS2 et UBIFS avec écriture différée activé / désactivé.

écriture.
La différence des nombres d'effacements relevés pendant chaque test nous indique que c'est bien l'exécution du ramasse-miettes qui provoque une baisse des performances d'E/S durant les tests sur la configuration *âgée* : on compte 122 % d'effacements en plus pour JFFS2, 1400 % pour YAFFS2, et 329 % pour UBIFS. En ce qui concerne les différences de compteurs d'effacements, on peut voir que, indépendamment de l'état de départ, la répartition de l'usure est la meilleure sur JFFS2 (*libre* : 1 ; *âgé* : 3), suivi de près par UBIFS (2 ; 8). YAFFS2 (5 ; 104) semble quant à lui effectuer une assez mauvaise répartition de l'usure, en particulier sur configuration *âgée*.

5.2.2. UBIFS et les écritures synchrones

Effectuer des écritures synchrones (non tamponnées) peut dans certains contexte s'avérer nécessaire : on peut penser au risque de perte des données tamponnées dans les systèmes embarqués sur batterie, ou encore au besoin de déterminisme des temps d'accès dans les systèmes temps-réel. Nous avons lancé FFSMark en désactivant le support de l'écriture différée pour UBIFS. Les résultats sont présents sur la figure 2. Dans ces tests la compression est désactivée.
En ce qui concerne les débits en écriture et la répartition de l'usure, lorsque l'écriture différée est activée pour UBIFS, ce dernier offre de meilleures performances que JFFS2. Néanmoins, lorsque l'on désactive cette fonctionnalité, JFFS2 devient plus performant que UBIFS. Cela est encore plus vrai sur la configuration *âgée*, pour laquelle JFFS2 offre un débit plus de deux fois supérieur à celui de UBIFS, et présente 5 fois moins d'effacements. Il est intéressant de noter que plusieurs études s'accordent classer UBIFS en tout point supérieur à JFFS2 [13, 3]. On peut voir que sous certaines conditions, JFFS2 propose de meilleures performances et une meilleure répartition de l'usure que son concurrent. En ce qui concerne les temps d'exécution des phases du benchmark (non présentés sur la figure), JFFS2 donne également de meilleures performances par rapport à UBIFS en mode synchrone sur configuration *âgée*. La chute de performances chez UBIFS en écriture synchrone trouve son explication au niveau des structures d'indexation des fichiers de ce FFS : ce sont des arbres dont les nœuds sont écrits en flash. Chaque mise à jour du système de fichiers provoque une mise à jour d'un nœud de l'arbre. Comme les mises à jour de données en place ne sont pas possibles en flash, tous les nœuds parents du nœud mis à jour doivent être modifiés et réécrits en flash, et ce jusqu'à la racine [4]. En écriture asynchrone (écriture différée activée), les modifications sont absorbées dans le page cache.

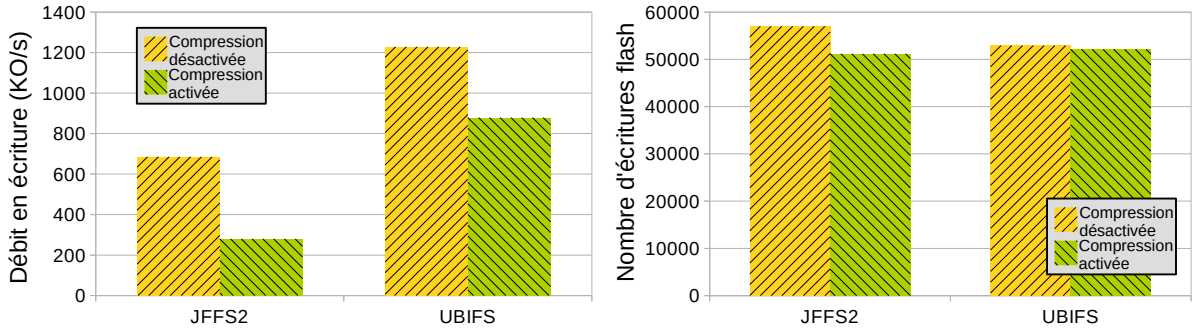

FIGURE 3 – **Impact de la compression :** Débit en écriture (à gauche et au centre) et nombre d'écritures flash (à droite) pour JFFS2 et UBIFS, compression activée / désactivée.

5.3. Impact de la compression

Nous avons lancé FFSMark en activant et désactivant la compression des données à la volée, une fonctionnalité offerte par JFFS2 et UBIFS. Les résultats sont présentés sur la figure 3. Ces tests concernent uniquement l'état de départ *libre*.

On peut voir que l'activation de la compression provoque une baisse de performances : le débit en écriture est 2,5 fois moins important avec JFFS2, 1,3 fois pour UBIFS. Néanmoins, en ce qui concerne le nombre d'écritures de pages flash constaté pendant le benchmark, on peut voir une légère baisse pour les deux FFS lorsque la compression est activée. Les fichiers manipulés par FFSMark contiennent des données aléatoires, elles sont donc difficilement compressibles. Ainsi, une certaine quantité de temps CPU est gâchée à tenter de compresser ces données, ce qui provoque la baisse de performances. Un tel constat à déjà été réalisé dans [14], où les auteurs proposent, pour JFFS2, un système de compression sélectif en fonction du type de fichier considéré. En ce qui concerne les débits en lecture et les temps d'exécutions, non présentés ici, ils suivent la même évolution que le débit en écriture.

6. Conclusion

Dans cet article, nous présentons FFSMark, un benchmark ciblant les systèmes de fichiers dédiés aux mémoires flash. Basé sur le populaire Postmark, FFSMark prend en considération les spécificités de la mémoire flash de type NAND, que sont (1) la règle effacer avant d'écrire qui impose la présence de données invalides en mémoire, et (2) l'usure propre à ce type de mémoire. FFSMark permet ainsi de définir un état de départ, en ce qui concerne la quantité d'espace flash libre, valide et invalide en début de benchmark. FFSMark permet également de récupérer en sortie de benchmark des statistiques sur les accès flash, en particulier concernant la répartition de l'usure, en supplément à des métriques de performances classiques.

Nous présentons également une étude de cas d'exécution de FFSMark sur une plate-forme embarquée, comparant les performances de JFFS2, YAFFS2 et UBIFS. Les spécificités de FFSMark permettent d'observer l'important impact de l'état de départ du système de fichiers sur les performances et l'usure. Cette étude de cas met également en évidence des comportements non observés dans des études précédentes, notamment en ce qui concerne la différence de performances entre UBIFS en écriture synchrone et JFFS2 sur système de fichiers âgé.

Les sources de FFSMark sont librement disponibles en téléchargement à l'adresse suivante : http://syst.univ-brest.fr/~pierre/?page_id=209.

Bibliographie

1. Bjørling (M.), Le Folgoc (L.), Mseddi (A.), Bonnet (P.), Bouganim (L.) et Jónsson (B.). – Performing sound flash device measurements : some lessons from uFLIP. – In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, p. 1219–1222. ACM, 2010.
2. Grupp (L. M.), Caulfield (A. M.), Coburn (J.), Swanson (S.), Yaakobi (E.), Siegel (P. H.) et Wolf (J. K.). – Characterizing flash memory : anomalies, observations, and applications. – In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, p. 24–33. IEEE, 2009.
3. Homma (T.). – Evaluation of flash file systems for large NAND flash memory. – In *CELF Embedded Linux Conference*, San Francisco, Etats Unis, 2009.
4. Hunter (A.). – A brief introduction to the design of UBIFS, 2008. `http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf`.
5. Kang (Y.) et Miller (E. L.). – Adding aggressive error correction to a high-performance compressing flash file system. – In *Proceedings of the Seventh ACM International Conference on Embedded Software*, *EMSOFT '09*, EMSOFT '09, p. 305–314, New York, NY, USA, 2009. ACM.
6. Katcher (J.). – *Postmark : A new file system benchmark*. – Rapport technique, Technical Report TR3022, Network Appliance, 1997.
7. Kim (J.), Shim (H.), Park (S.), Maeng (S.) et Kim (J.). – FlashLight : a lightweight flash file system for embedded systems. *ACM Trans. Embed. Comput. Syst.*, vol. 11S, n1, juin 2012, p. 18 :1–18 :23.
8. Lim (S.) et Park (K.). – An efficient NAND flash file system for flash memory storage. *Computers, IEEE Transactions on*, vol. 55, n7, 2006, p. 906–912.
9. Liu (S.), Guan (X.), Tong (D.) et Cheng (X.). – Analysis and comparison of NAND flash specific file systems. *Chinese Journal of Electronics*, vol. 19, n3, 2010.
10. Micron Inc. – MT29F2G16ABDHC-ET :D NAND flash memory datasheet, 2007.
11. Olivier (P.), Boukhobza (J.) et Senn (E.). – Micro-benchmarking flash memory File-System wear leveling and garbage collection : A focus on initial state impact. – In *Proceedings of the 2012 IEEE 15th International Conference on Computational Science and Engineering*, *CSE '12*, CSE '12, p. 437–444, Washington, DC, USA, 2012. IEEE Computer Society.
12. Olivier (P.), Boukhobza (J.) et Senn (E.). – Flashmon v2 : Monitoring raw NAND flash memory I/O requests on embedded linux. *ACM SIGBED Rev.*, vol. 11, n1, 2014, p. 38–43.
13. Opdenacker (M.). – Update on filesystems for flash storage, 2008.
14. Song (H.), Choi (S.), Cha (H.) et Ha (R.). – Improving energy efficiency for flash memory based embedded applications. *Journal of Systems Architecture*, vol. 55, n1, janvier 2009, pp. 15–24.
15. Traeger (A.), Zadok (E.), Joukov (N.) et Wright (C. P.). – A nine year study of file system and storage benchmarking. *ACM Transactions on Storage (TOS)*, vol. 4, n2, 2008, p. 5.
16. UBM Tech. – Embedded market study, 2013. `http://images.content.ubmtechelectronics.com/Web/UBMTechElectronics/%7Ba7a91f0e-87c0-4a6d-b861-d4147707f831%7D_2013EmbeddedMarketStudyb.pdf`.
17. Woodhouse (D.). – JFFS2 : the journalling flash file system version 2. – In *Ottawa Linux Symposium*, Ottawa, Canada, 2001.
18. Wookey. – YAFFS - a NAND flash file system. – In *CE Linux Conference*, Linz, Autriche, 2007.