État de l'art de la copie-sur-écriture

Master Systèmes et Applications Répartis 2ème année

Université Pierre et Marie Curie

Auteur :
Florian DAVID

Encadrants :
Gilles MULLER
Gaël THOMAS

Référent :
Fabrice KORDON

17 mai 2011

Table des matières

1 Introduction

Choco [CJLR06] [Tea10] est une bibliothèque Java permettant de résoudre des problèmes de satisfaisabilité et de programmation par contraintes. Choco est utilisé pour la recherche mais également dans l'enseignement ainsi que dans le monde industriel et a gagné le prix de la JSR (Java Specification Request) la plus innovante en 2010 ayant pour thème une interface de programmation pour la programmation par contraintes.

L'algorithme de résolution de contraintes est itératif et explore un espace de solutions. À chaque itération, l'algorithme teste une solution (un ensemble de valeurs) et continue l'exploration en raffinant les valeurs jusqu'à ce que celles-ci satisfassent le problème. Il arrive fréquemment que les valeurs tentées enfreignent les contraintes et que l'algorithme doive revenir à un ensemble antérieur de solutions.

Les domaines de variables et les contraintes servant à la résolution du problème sont regroupés dans un *environnement*. Un *monde* représente l'ensemble des valeurs constituant une solution à une itération de l'algorithme et plusieurs instances d'un monde cohabitent dans un environnement mais une seule instance est active à la fois. Les valeurs sont représentées par des données *backtrackables* : celles-ci permettent d'aller et venir entre les valeurs des différents mondes d'un environnement.

La manipulation d'un environnement est simple et ressemble à la gestion d'une pile, chaque élément de la pile étant une instance d'un monde. Le listing 1 illustre un exemple d'utilisation. Les lignes 1 et 2 créent un nouvel environnement et ajoutent une variable entière *anInt* de valeur 0 à cet environnement. La méthode *worldPush()* (l.3) permet de sauvegarder un monde (correspondant à un ensemble de solutions d'une itération) dans l'environnement. Les lignes 4 et 5 modifient la valeur de l'entier et la ligne 6 empile un nouveau monde dans l'environnement où la variable anInt vaut 1. La ligne 7 affecte la valeur 2 à anInt. La méthode *worldPop()* (l.8) dépile le monde courant pour laisser place au monde précédent dans la pile où anInt a pour valeur 1. La ligne 9 dépile encore un monde pour revenir au monde de départ où anInt vaut 0.

```
1  IEnvironment environment = new Environment();
2  IStateInt anInt = environment.makeInt(0);
3  environment.worldPush(); // Sauvegarde de la valeur au monde 0
4  anInt.add(2); // Modification de la valeur
5  anInt.add(-1); // anInt vaut maintenant 1
6  environment.worldPush(); // Sauvegarde de la valeur au monde 1
7  anInt.set(2); // anInt vaut 2
8  environment.worldPop(); // Restauration du monde 1
9  // anInt vaut de nouveau 1
10 environment.worldPop(); // Restauration du monde 0, anInt
       reprend sa valeur initiale 0
```

Listing 1 – Utilisation d'un environnement et des mondes dans Choco

L'objectif du stage est d'optimiser ces allers et venues entre les différents mondes. Chaque empilement d'un monde nécessite de sauvegarder l'ensemble des valeurs de celui-ci. Généralement, le nombre de valeurs modifiées à l'itération suivante est plus petit que le nombre total de valeurs et il peut être avantageux de factoriser les instances de valeurs qui ne sont pas modifiées. Si une valeur n'est pas modifiée entre 2 mondes, il est possible d'utiliser une seule instance de celle-ci au lieu de 2, ce qui permettrai d'économiser l'espace mémoire ainsi que le temps nécessaire à la copie de cette valeur.

Le mécanisme envisagé pour mettre en oeuvre ce principe est la copie-sur-écriture. La copie-sur-écriture aura pour but de permettre le partage d'une valeur entre plusieurs

mondes en minimisant le nombre de copies. Lorsque le monde courant modifiera une valeur partagée entre plusieurs mondes, le mécanisme assurera que la modification n'interfère pas avec les autres mondes et qu'une vision consistante des valeurs sera conservée pour tous.

La section 2 introduit en détail le fonctionnement de la copie-sur-écriture matérielle et logicielle, la section 3 illustre la mise en oeuvre de la copie-sur-écriture matérielle appliquée au checkpointing de processus, la section 4 présente la mise en oeuvre de la copie-sur-écriture logicielle appliquée aux systèmes de fichiers, la section 5 expose des domaines de l'informatique où la copie-sur-écriture est utilisée et la section 6 conclut l'état de l'art.

2 Copie sur écriture

La copie-sur-écriture (*copy-on-write*) est un mécanisme qui permet d'optimiser la copie d'une ressource entre agents, un agent pouvant être un thread d'une application multi-threadée ou encore une machine appartenant à un cluster. L'idée principale est que si plusieurs agents ont besoin de partager une ressource initialement identique pour tous, il est possible de leur faire partager une version commune de la ressource. Lorsqu'un agent vient à modifier la ressource, celle-ci doit être dupliquée afin que les changements ne soient pas visibles par les autres agents partageant cette ressource.

Prenons le cas d'un même fichier devant être utilisé par plusieurs programmes. Chaque programme souhaite disposer d'une copie privée de ce fichier pour y lire et y écrire. Tant que l'accès au contenu du fichier se fait en lecture, la ressource n'est pas modifiée et tous les programmes peuvent se la partager tout en gardant une vision cohérente. Lorsqu'un programme accède en écriture au fichier, celui-ci est dupliqué avant qu'il ne soit modifié. De cette manière, le programme modifiera une copie du fichier qui lui est propre et les autres programmes conserveront la version non-modifiée du fichier.

L'avantage de cette technique est que la ressource n'a pas besoin d'être copiée si le programme ne la modifie pas, ce qui permet d'économiser le coût de la copie. La copie est effectuée de façon paresseuse, au moment où il est nécessaire de l'effectuer afin que les ressources restent consistantes pour chaque application qui les partagent. Cela permet d'éviter des copies en série d'une même ressource lorsque plusieurs applications en ont besoin au même instant, provoquant ainsi un goulot d'étranglement.

Ce mécanisme peut être implémenté à 2 niveaux : le niveau matériel ou le niveau logiciel. L'implémentation au niveau matériel se fait conjointement avec le système d'exploitation et la MMU (*Memory Management Unit*) tandis que l'implémentation au niveau logiciel s'effectue avec du code spécialement dédié à ce mécanisme intégré directement dans l'application. Ces 2 types d'implémentations seront évaluées en fonction du mécanisme de détection des accès aux données, du surcoût induit pour les lectures et les écritures, de la granularité de la copie des ressources en copie-sur-écriture et de la complexité du code déployé pour mettre en oeuvre ce mécanisme.

2.1 Implémentation matérielle de la copie sur écriture

La virtualisation de la mémoire dans les systèmes est aujourd'hui omniprésente et permet principalement d'assurer la translation des adresses virtuelles vers les adresses physiques et la protection des accès à la mémoire. La MMU est le composant matériel en charge d'assurer ces fonctionnalités. La mémoire vive est fragmentée en *pages*, chacune identifiée par un numéro. Lorsqu'un processus accède à la mémoire, les bits de poids fort de l'adresse mémoire virtuelle permettent à la MMU de trouver la page correspondante. La MMU vérifie également que le processus dispose des droits suffisants pour y accéder.

Le mécanisme de copie-sur-écriture est mis en place en modifiant les droits d'accès à la page contenant la ressource. Les droits de la page où se situe la ressource partagée en copie-sur-écriture sont positionnés en lecture-seule. La figure 1 présente le traitement

effectué lors de l'accès en écriture à un objet marqué en copie-sur-écriture. Lorsque l'application accède en écriture la page contenant l'objet, la MMU signale une erreur de page (*page fault*) au système d'exploitation qui envoie un signal *SIGSEGV* (*SIGnal SEGmentation Violation*) au processus responsable. Le gestionnaire de signal associé est instrumenté pour vérifier que la faute provient d'une page contenant un objet en copie-sur-écriture. Il duplique alors la page et modifie l'entrée de la table des pages pour la faire correspondre à la page nouvellement allouée.

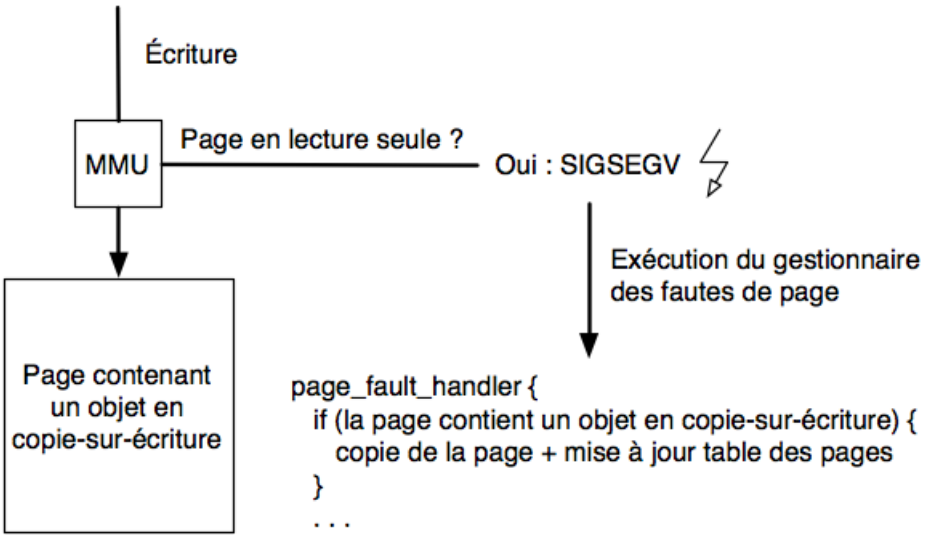

FIGURE 1 – Traitement d'un objet marqué copie-sur-écriture dans une implémentation matérielle.

La modification des droits de la page peut se faire par le biais de l'appel système *mprotect()* en mode utilisateur ou directement en modifiant les bits spécifiant le mode d'accès dans les entrées de la table des pages en mode noyau.

Chaque ressource partagée en copie-sur-écriture doit maintenir un compteur de référence correspondant au nombre d'agents qui la partage. Dans le cas contraire, la ressource originale persistera si toutes les applications la dupliquaient suite à une écriture, provoquant ainsi une fuite mémoire. La figure 2 illustre ce problème. L'objet *O* est marqué en copie-sur-écriture et les 2 threads *T1* et *T2* le partage (A). Dans (B), *T1* et *T2* ont modifié l'objet et *O* a été dupliqué pour obtenir *O1* et *O2* pour chaque thread. *O* n'est plus référencé par aucun thread et provoque une fuite mémoire.

L'implémentation matérielle de la copie-sur-écriture s'effectue en mode noyau et repose principalement sur la MMU et le gestionnaire des fautes de page. La MMU détecte les écritures sur les pages contenant un objet en copie-sur-écriture et le gestionnaire des fautes duplique la page et met à jour la table des pages. Il n'y a aucun surcoût lors de l'écriture ou de la lecture des objets en copie-sur-écriture car l'accès aux objets est direct. En contrepartie, l'implémentation matérielle est *gros-grain* : la duplication se fait toujours en fonction de la taille de page du système même si les objets en copie-sur-écriture sont de petites tailles. Le travail d'ingénierie nécessaire à l'implantation de

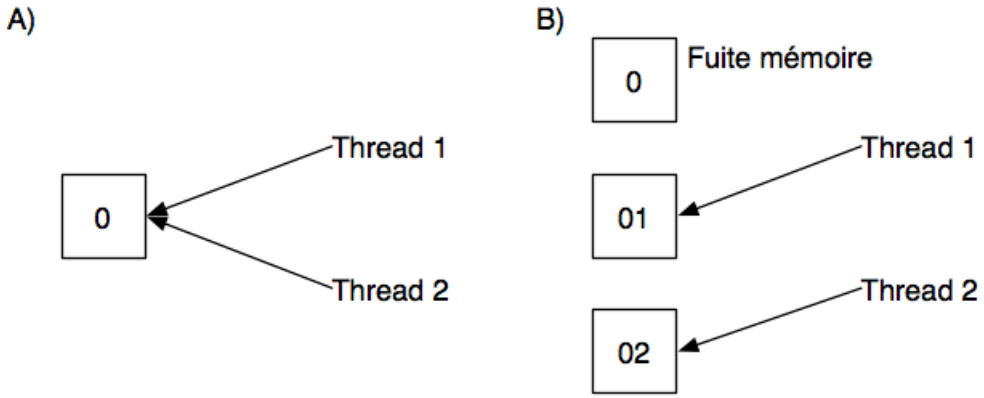

FIGURE 2 – Exemple de fuite mémoire due à l'absence de compteur de références pour un objet en copie-sur-écriture.

ce mécanisme est simple car la détection des écritures des objets en copie-sur-écriture est pris en charge par la MMU.

2.2 Implémentation logicielle de la copie sur écriture

L'implémentation logicielle de la copie-sur-écriture est au contraire complètement indépendante du matériel et de la MMU. Cette implémentation est restreinte au mode utilisateur et l'application voulant utiliser cette technique doit remplir le rôle qu'effectue la MMU dans l'implémentation matérielle, à savoir surveiller les accès en écriture aux objets marqués en copie-sur-écriture et les dupliquer à ce moment.

Le principe de base est d'instrumenter les écritures des objets de l'application. Lors d'une écriture, l'application vérifie que l'objet en question est un objet copie-sur-écriture et le duplique si c'est le cas. Là où la MMU remplaçait le numéro de la page par celui de la page nouvellement allouée dans la table des pages, l'application doit assurer que l'accès à l'objet soit consistant pour tous.

La méthode la plus couramment utilisée pour atteindre ce but est d'interposer une structure d'accès aux objets en copie-sur-écriture entre les agents et les objets (typiquement un pointeur d'indirection par objet). La référence d'un objet en copie-sur-écriture est modifiée dans la structure lorsqu'un agent écrit sur cet objet.

La figure 3 présente l'exemple d'un programme multi-threadé ; dans ce contexte, chaque agent est un thread. Supposons que la structure d'accès aux objets en copie-sur-écriture se fasse par l'intermédiaire d'un tableau contenant des pointeurs d'indirection vers ces objets. Chaque thread possède une version de ce tableau qui lui est propre (A). Quand un objet *O1* partagé en copie-sur-écriture est ajouté aux threads, chaque tableau est mis-à-jour et référence l'objet (B). Si le thread 2 modifie l'objet *O1*, le programme doit copier l'objet puis mettre à jour le tableau du thread 2 (C).

L'implémentation logicielle de la copie-sur-écriture s'effectue dans l'application souhaitant utiliser ce mécanisme. L'accès aux données en copie-sur-écriture se fait à l'aide d'une structure d'accès intermédiaire. Cette structure induit des indirections lors de l'accès aux données en fonction de la structure d'accès utilisée. Ces indirections entrainent un surcoût lors d'une lecture ou d'une écriture. La granularité de l'implémen-

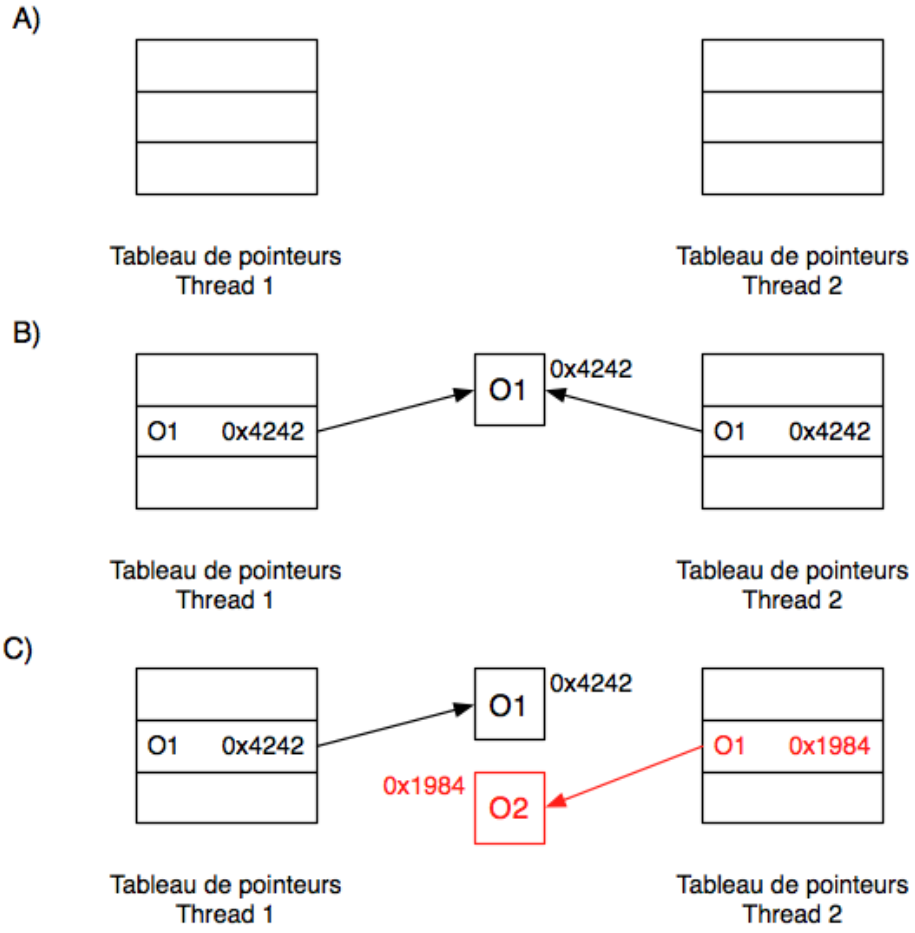

FIGURE 3 – Traitement d'un objet marqué copie-sur-écriture dans une implémentation logicielle.

tation logicielle est plus fine que l'implémentation matérielle car les accès sont tracés au niveau de l'objet. Seul l'objet sera copié lors d'une duplication et non une page qui contient également des données qui ne sont pas en relation avec l'objet. L'implantation de ce mécanisme nécessite l'utilisation d'une structure d'accès aux objets en copie-sur-écriture et chaque lecture ou écriture vers ces objets se sert de cette structure. Le développement de cette structure d'accès implique un travail d'ingénierie important une modification profonde du code si les objets sont accédés à de multiples endroits dans l'application.

2.3 Récapitulatif

Cette section présente la copie-sur-écriture ainsi que l'implémentation matérielle et logicielle de ce mécanisme. L'implémentation matérielle repose sur l'utilisation conjointe de la MMU et du système d'exploitation. L'implémentation logicielle est implantée directement en mode utilisateur dans l'application se servant de ce mécanisme. Le tableau 1 récapitule les propriétés de ces 2 mécanismes selon 4 critères :

– *Détection d'une modification sur un objet copie-sur-écriture* : L'utilisation du matériel permet de détecter les accès aux objets en copie-sur-écriture et de laisser le gestionnaire de fautes de page effectuer les traitements tandis que l'implémentation logicielle impose l'utilisation d'une structure d'accès à ces objets pour surveiller leur utilisation.
– *Accès aux données en lecture / écriture* : La copie-sur-écriture logicielle implique un surcoût lors de l'accès aux objets dû aux indirections de la structure d'accès. Le matériel permet de conserver un accès direct aux objets.
– *Granularité d'une copie* La modification d'un objet en copie-sur-écriture implique pour l'implémentation matérielle de dupliquer une page complète quelque soit la taille de l'objet. L'implémentation logicielle duplique uniquement l'objet et met à jour une entrée dans sa structure d'indirection.
– *Complexité du développement* : La détection des objets en copie-sur-écriture faite par le matériel permet un développement plus rapide contrairement à l'implémentation logicielle car la structure d'indirection doit implémenter cette détection.

L'implémentation matérielle est donc plus rapide lors des accès en lecture / écriture car elle n'implique pas d'indirections supplémentaires et est plus facile à développer car la MMU prend en charge la détection des accès aux objets par rapport à l'implémentation logicielle. La copie des objets dans l'implémentation logicielle est plus efficace car la granularité est plus fine : seul l'objet est dupliqué et nécessite une mise à jour dans la structure d'indirection tandis que l'implémentation matérielle duplique au minimum une page complète.

	Implémentation matérielle	Implémentation logicielle
Détection d'une modification sur un objet copie-sur-écriture	MMU	Structure d'accès aux objets
Accès aux données en lecture / écriture	Rapide : accès direct	Lent : indirection(s)
Granularité d'une copie	Gros-grain (page)	Grain fin (objet)
Complexité du développement	Simple	En fonction de la complexité de la structure d'accès

TABLE 1 – Tableau récapitulatif de l'implémentation matérielle et logicielle de la copie-sur-écriture

3 Checkpointing de processus

3.1 Utilisation

Le *Checkpointing* consiste à sauvegarder un processus de telle sorte qu'il puisse être redémarré ultérieurement dans le même état. Le checkpointing est principalement utilisé pour la migration de processus et la tolérance aux fautes.

La migration de processus permet l'équilibrage de la charge dans un réseau de machines distribuées et il est donc nécessaire de pouvoir transférer un processus d'une machine avec une charge de travail importante vers une autre machine ayant des ressources de calcul disponibles. Un mécanisme de redémarrage des processus est également nécessaire en plus du checkpointing (*Checkpoint/Restart*).

La tolérance aux fautes consiste à ajouter des mécanismes à un système lui permettant de détecter l'apparition d'une faute. Le système commence alors une procédure de recouvrement afin de revenir à un fonctionnement nominal ou au minimum à assurer un service dégradé. Le checkpointing permet le recouvrement d'une faute en faisant revenir l'application à un état antérieur stable et en redémarrant le calcul.

3.2 Classification

Dans [SPD+05], les auteurs proposent une classification des mécanismes de checkpointing selon 3 caractéristiques : le contexte, l'agent qui fournit la fonctionnalité de *Checkpoint/Restart* et les spécificités de l'implémentation. La figure 4 présente cette classification. Au plus haut niveau de la classification, l'implémentation peut être soit au niveau *utilisateur* ou au niveau *système*.

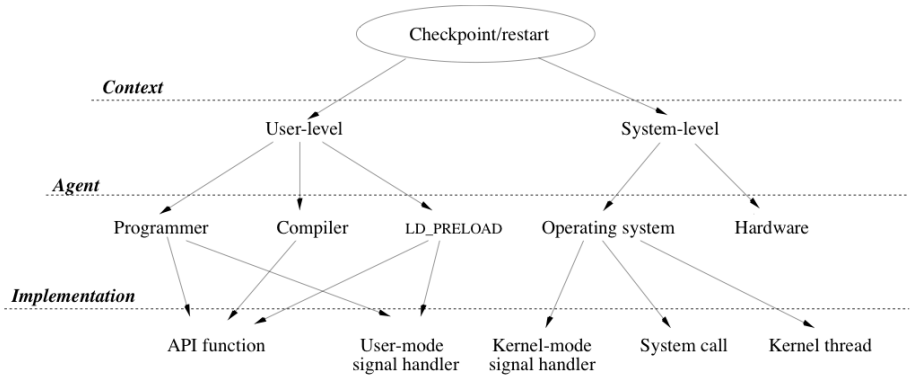

FIGURE 4 – Classification des implémentations de Checkpoint/Restart (Figure extraite de [SPD+05]).

Lors d'une implémentation au niveau utilisateur, l'insertion du code de checkpointing est effectuée directement par le programmeur ou par un compilateur. Les fonctions de checkpointing sont fournies par une bibliothèque. La bibliothèque peut être liée à l'application lors de l'édition de liens ou chargée dynamiquement grâce à la variable d'environnement LD_PRELOAD et à des gestionnaires de signaux fournis dans la bibliothèque.

Au niveau système, l'implémentation peut se faire avec du matériel spécifique dédié au checkpointing ou au niveau du système d'exploitation. Les mécanismes utilisés dans

le système d'exploitation sont :

- **appel système** : Un nouvel appel système est créé qui effectuera le checkpointing de l'application de façon synchrone. Seule l'application décide quand le checkpointing doit s'effectuer en appelant la fonction. L'application peut décider du moment le plus judicieux pour effectuer un checkpointing mais le code de l'application doit être modifié pour pouvoir appeler la fonction. Le checkpointing s'effectue de façon synchrone ce qui diminue le temps de réponse de l'application.
- **signal** : Un nouveau signal est ajouté au système d'exploitation. Lorsque l'application reçoit le signal, le système d'exploitation effectue le checkpointing de celle-ci. Le signal peut être envoyé à la demande de l'utilisateur en utilisant la commande *kill* ou bien par l'application en utilisant l'appel système *kill()*.
- **thread noyau** : Un thread noyau est créé et se charge d'effectuer le checkpointing en mode noyau de manière asynchrone. Le thread noyau peut être contrôlé de 3 manières différentes. Un nouvel appel système peut être ajouté au système, l'utilisation de cet appel permettant de notifier le thread qu'un checkpointing est demandé. Un fichier peut être créé dans le système de fichiers virtuel */proc* pour contrôler le checkpointing. Un périphérique virtuel peut également faire le lien entre le thread noyau et l'application à l'aide des appels système *read*, *write* et *ioctl*.

3.3 Libckpt

Libckpt [PBKL95] est une librairie de checkpointing au niveau utilisateur. Le programmeur doit changer une ligne de code et recompiler son application pour l'utiliser. Libckpt peut s'utiliser de 2 manières différentes. L'utilisateur peut déclencher le checkpointing à certains points qu'il aura fixé dans l'application. Libckpt peut également s'exécuter de manière transparente en utilisant un signal périodique pour déclencher le checkpointing. La sauvegarde du processus peut s'effectuer de manière incrémentale ou en utilisant l'appel système *fork()*.

Dans le cas de la sauvegarde incrémentale, celle-ci est mise en oeuvre en utilisant l'appel système *mprotect()* pour positionner l'accès des pages mémoire en lecture seule. Lors d'un accès en écriture qui modifie le contenu d'une page, le signal est intercepté et la page est mémorisée comme ayant été modifiée depuis la dernière sauvegarde. Lors du prochain checkpoint, les pages mémorisées qui seront les seules à avoir été modifiées depuis le dernier checkpoint seront sauvegardées sur disque. La sauvegarde incrémentale est synchrone car le processus ne peut s'exécuter tant que toutes les pages n'ont pas été recopiées sur le disque. Ce mécanisme permet donc de ne sauvegarder que les données modifiées depuis le dernier checkpoint mais doit s'exécuter de façon synchrone réduisant ainsi le temps de réponse de l'application.

Dans le cas de la sauvegarde du processus avec l'appel système *fork()*, celle-ci s'effectue en exécutant un code dans le processus fils qui sauvegardera l'ensemble des pages sur le disque. L'implémentation de Linux du *fork()* utilise la copie-sur-écriture sur toutes les pages du processus parent. Une page de l'espace d'adressages est dupliquée dès que le père ou le fils la modifie. De cette façon, le père peut continuer à s'exécuter et le fils peut sauvegarder le processus de manière asynchrone. Ce mécanisme s'exécute de manière asynchrone et ne réduit pas le temps de réponse de l'application mais effectue une sauvegarde complète du processus et non les modifications effectuées depuis le dernier checkpoint.

3.4 Diskless Checkpointing

Le *Diskless Checkpointing* [PLP98] est une méthode de checkpointing dans un système distribué qui propose de remplacer la sauvegarde sur support stable (disque dur) par la mémoire vive de certaines machines du système. Cette technique permet de supprimer le surcoût de l'écriture sur disque mais le système ne peut pas supporter la reprise du calcul après la panne simultanée de toutes les machines : il est donc conseillé de la coupler avec un système de checkpointing sur support stable. Lorsqu'une machine devient défaillante, une autre machine du réseau utilise le checkpoint sur support stable pour reprendre l'exécution et les autres machines se synchronisent en redémarrant depuis un checkpoint cohérent qui se situe en mémoire et non sur support stable. Les auteurs proposent 4 algorithmes de checkpointing sans disque dont 2 utilisent la copie-sur-écriture.

Le premier algorithme repose sur une technique de checkpointing incrémentale. Une partie de la mémoire de la machine est dédiée au checkpoint. Lors du checkpoint, toutes les pages sont mises en lecture seule. Lorsque l'application tente d'écrire sur l'une des pages, la MMU déclenche une faute de page. Le système de checkpointing copie alors la page dans la partie de la mémoire dédiée et repositionne le mode d'accès de la page en lecture-écriture (cf. Figure 5). Le checkpoint est constitué des pages en lecture seule de l'espace d'adressage de l'application et des pages copiées dans l'espace dédié. Lors du recouvrement, le processeur copie ou fait correspondre les pages sauvegardées dans l'espace d'adressage du processus.

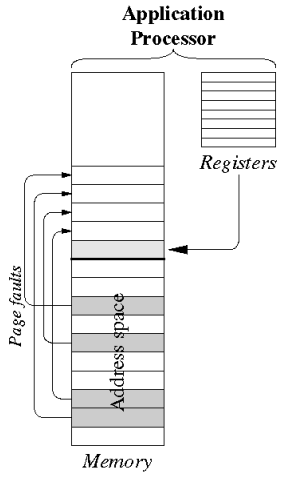

FIGURE 5 – Checkpointing d'une application avec l'algorithme incrémental (Figure extraite de [PLP98]).

Le deuxième algorithme utilise l'appel système *fork()* pour effectuer le checkpoint. L'application se duplique elle-même en utilisant *fork()* comme sur la figure 6. Le processus fils représente le clone de l'application. Lors du recouvrement, l'application peut remplacer son état (l'ensemble des registres du processeur et l'espace d'adressage) par celui du clone ou bien le clone peut reprendre l'exécution à la place de l'application. Cet algorithme de checkpointing utilise la copie-sur-écriture car les systèmes d'exploitation implémentent quasiment tous la copie-sur-écriture des pages de l'espace d'adressage lors de la création d'un processus fils. L'avantage est que la copie des

pages est entièrement prise en charge par le système d'exploitation et ne requiert pas de droits privilégiés pour changer le mode d'accès des pages mémoires contrairement à l'algorithme précédent.

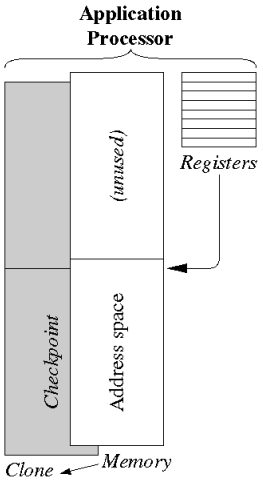

FIGURE 6 – Checkpointing d'une application avec l'algorithme utilisant *fork()* (Figure extraite de [PLP98]).

3.5 Zap

Zap [LN07] [OSSN02] [LBP+07] est un système qui sauvegarde de multiples processus dans un état consistent. Une application est souvent constituée de plusieurs processus qui ont des dépendances avec le système d'exploitation. Zap permet de sauvegarder de telles applications et également les ressources qui y sont associées telles que les IPC, les relations père-fils entre les processus, les fichiers, les périphériques ou encore les connections réseaux.

Zap est un module noyau introduisant une fine couche de virtualisation entre le système d'exploitation et les processus. Les processus sont encapsulés dans un *pod* (*PrOcess Domain*), une capsule fournissant un espace de noms virtualisé et privé et également l'interface entre le système et les processus. Cette capsule peut être isolée du système afin d'être sauvegarder sur un disque ou être envoyée sur une autre machine pour effectuer de l'équilibrage de charge. La manipulation des capsules (sauvegarde, restauration, migration) se fait à l'aide d'un utilitaire qui communique avec un périphérique virtuel.

Lors du checkpointing, le mécanisme de copie-sur-écriture est utilisé pour garder une référence sur les pages mémoires au lieu d'explicitement sauvegarder chacune d'entre elles afin de réduire la pression sur la mémoire et d'éviter la dégradation des performances du cache. De cette manière, les pages constituant le checkpoint restent disponibles et cohérentes même si l'application modifie les pages au moment où le pod reprend son exécution et avant que celles-ci ne soient écrites sur disque.

Étant donné que Zap sauvegarde de multiples processus pouvant appartenir à la même application, il est possible que les fils, petits-fils,.... de certains processus appar-

tiennent au même checkpoint. L'appel système *fork()* utilise déjà la copie-sur-écriture lorsqu'il crée un processus fils : celui-ci mets toutes les pages en copie-sur-écriture au lieu de copier la totalité des pages de l'espace d'adressage du père. Certaines pages mémoires peuvent donc appartenir à plusieurs processus et Zap mets en oeuvre son propre mécanisme de copie-sur-écriture en parallèle de celui du *fork()* et sauvegarde également les relations d'appartenance des pages en copie-sur-écriture entre les processus ayant des relations de parenté.

3.6 Évaluation d'algorithmes de checkpointing de programmes parallèles dans un système multi-processeurs à mémoire partagé

Dans [LNP94], les auteurs évaluent 4 implémentations d'algorithmes de checkpointing dans un système multi-processeurs à mémoire partagé. L'évaluation est faite selon 3 critères : le temps total passé à sauvegarder le processus sur disque, le temps pendant lequel l'exécution du processus checkpointé est stoppée et le surcoût de temps provoqué par l'algorithme sur le processus sauvegardé.

L'algorithme *séquentiel* consiste à stopper le processus, à sauvegarder toutes les données sur disque puis à le redémarrer. Le second algorithme appelé *mémoire principal* sauvegarde le processus dans un second espace d'adressage et le redémarre. Un thread se charge alors d'écrire le contenu de l'espace d'adressage sur disque. Le troisième algorithme appelé *copie-sur-écriture* utilise le mécanisme de protection de pages de la mémoire virtuelle : le processus est stoppé, toutes les pages sont mises en lecture seule puis le processus redémarre. Un thread *copieur* se charge de copier toutes les pages dans un nouvel espace d'adressage et de les remettre en lecture-écriture. Une fois toutes les pages copiées dans l'espace, celles-ci sont écrites sur disque. Le dernier algorithme appelé *concurrent, low-latency* (CLL) améliore l'algorithme précédent en pré-allouant un pool de pages pour la copie de l'espace d'adressage qui servira de buffer. Le thread *copieur* se charge de remplir ce pool et un thread *écrivain* écrit les données sur disque de façon concurrente. La figure 7 illustre le déroulement de chaque algorithme.

Ces algorithmes sont évalués en fonction du temps total nécessaire au checkpointing du processus et du surcoût induit par rapport au temps d'exécution du processus. Les 2 meilleurs algorithmes par rapport au temps total nécessaire pour checkpointer l'application sont l'algorithme séquentiel et CLL. Cela est dû au fait que la sauvegarde sur disque est immédiate contrairement aux algorithmes mémoire principal et copie-sur-écriture qui doivent d'abord recopier les données en mémoire avant de procéder à l'écriture sur disque. En ce qui concerne le surcoût du temps d'exécution de l'application, les algorithmes utilisant la copie-sur-écriture sont les plus efficaces. Ces 2 algorithmes stoppent le processeur moins longtemps et améliorent la concurrence entre le checkpointing et l'exécution de l'application par rapport aux 2 autres.

3.7 Récapitulatif

L'ensemble des solutions de checkpointing de cette section se sert de l'implémentation matérielle de la copie-sur-écriture utilisant le mécanisme de protection de la mémoire virtuelle comme expliqué à la section 2.1. Aucune utilisation logicielle de la copie-sur-écriture n'a été recensée : cela s'explique facilement par le fait que l'instrumentation des écritures sur l'espace d'adressage complet peut devenir très vite coûteuse et une granularité si fine n'est pas judicieuse car la taille de l'espace d'adressage à sau-

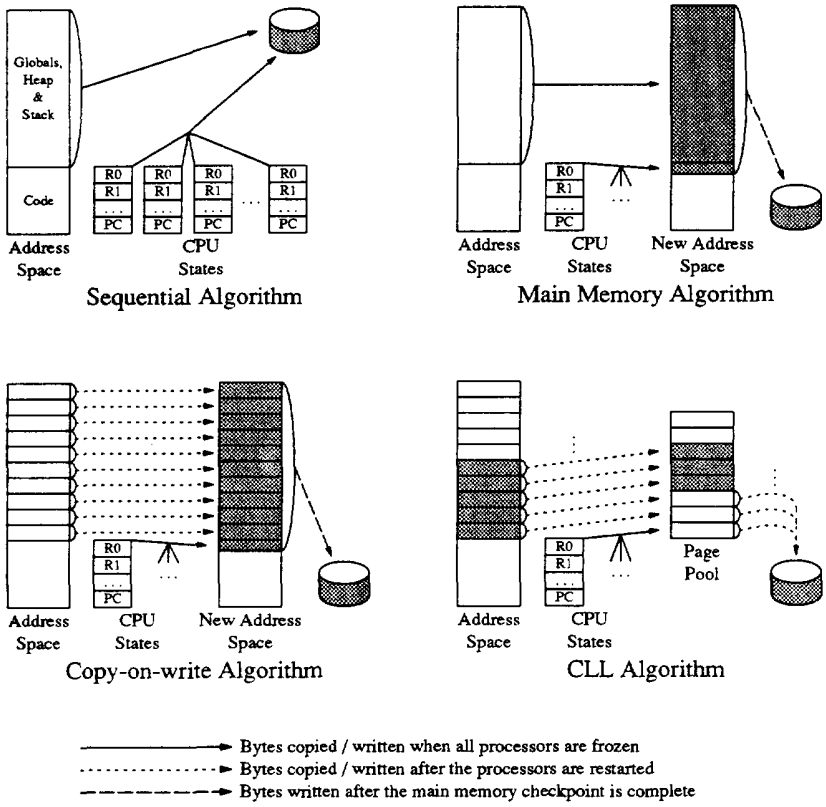

FIGURE 7 – Principe d'exécution des 4 algorithmes. CPU States représente l'état des registres des processeurs (Figure extraite de [LNP94]).

vegarder est importante. Le tableau 2 récapitule les 3 solutions de checkpointing de cette section (Libckpt, Zap et Diskless Checkpointing) selon 4 critères :

- Le *niveau de l'implémentation* définit si le mécanisme est implémenté dans le système d'exploitation ou dans l'espace utilisateur.
- Le support de stockage indique le(s) support(s) où les processus checkpointés seront sauvegardés.
- L'*algorithme* indique si la méthode employée est incrémentale ou totale. Un algorithme total sauvegarde toutes les données concernant le processus pour effectuer un checkpoint. L'algorithme incrémental sauvegarde uniquement les modifications effectuées depuis le dernier checkpoint.
- Le *format* indique le moyen employé pour distribuer le mécanisme de checkpointing en vue de son utilisation.
- Le *déclenchement* est le moyen utilisé pour notifier le mécanisme de checkpointing qu'un checkpoint doit être effectué.

	Libckpt	Diskless Checkpointing	Zap
Niveau de l'implémentation	Utilisateur	Système	Système
Support de stockage	Disque dur	Mémoire vive	Disque dur Réseau
Algorithme	Incrémental ou Total	Incrémental ou Total	Incrémental
Format	Bibliothèque	Bibliothèque	Module noyau
Déclenchement	Appel de fonction par l'application et Signal	Signal	Périphérique virtuel

TABLE 2 – Tableau récapitulatif des solutions de checkpointing

4 Systèmes de fichiers

4.1 Utilisation

Un système de fichiers permet de stocker et classer des informations de manière hiérarchique sous la forme de fichiers et de répertoires. Il s'occupe de la gestion des ressources permettant le stockage des informations (disques durs, mémoires, . . .) et du placement de ces ressources, de leur manipulation, de leur organisation et de leur accès par le système d'exploitation au travers d'une interface. Les systèmes de fichiers présentés dans cette section utilise la copie-sur-écriture pour pouvoir capturer un instantané du système de fichiers à un instant donné (*snapshot*) afin de pouvoir récupérer des données effacées par erreur ou réparer un système de fichiers corrompus. La copie-sur-écriture sert également à implémenter un modèle transactionnel de mise à jour des fichiers afin de garantir la cohérence du système de fichiers lors de défaillances.

4.2 Andrew File System

Andrew File System [HKM+88] (AFS) est un système de fichiers distribué pouvant passer à l'échelle à plus de 5000 stations de travail. AFS utilise un ensemble de serveurs appelé *Vice* qui présente un espace de noms de fichiers indépendamment de la location physique des données sur le réseau à tous les clients.

Le système d'exploitation de chaque client intercepte les appels systèmes concernant les opérations sur les fichiers et les transmet à *Venus*, un processus utilisateur chargé de mettre en cache les fichiers et de stocker les modifications apportées à ceux-ci. Les opérations de lecture et écriture se font uniquement en local permettant ainsi un travail en mode déconnecté en cas de panne du serveur. Venus informe Vice lorsqu'un fichier est ouvert ou fermé. En échange, il lui notifiera toute modification du fichier par un client et qui commitera les modifications apportées lors de la fermeture.

AFS utilise le *Volume* comme unité de stockage de données. Un volume est une collection de fichiers formant un sous-arbre de l'espace de nom de Vice, est situé dans une seule partition d'un disque et plusieurs volumes (ainsi que plusieurs partitions) peuvent appartenir au même disque. Les volumes sont reliés ensemble par des points de montage pour former l'espace de noms complet. Un point de montage est une feuille du sous-arbre de l'espace de noms d'un volume qui spécifie le nom d'un autre volume dont le noeud racine est relié à cette feuille.

L'équilibrage de l'espace disque disponible est réalisé en migrant les volumes parmi les partitions sur le(s) disque(s) d'un (de plusieurs) serveur(s). AFS crée un instantané (*snapshot*) appelé *clone* du volume en mettant tous les fichiers en copie-sur-écriture (i.e. en lecture seule) afin de minimiser le temps d'immobilisation des données de celui-ci. Les modifications faites aux fichiers sont autorisées et enregistrées pendant que s'effectue la migration du clone. Si le volume est modifié entre-temps, les différences entre le volume et le clone sont envoyées de façon incrémentale au nouveau site. Pour finir, le volume est brièvement désactivé, les derniers changements incrémentaux sont envoyés et les serveurs mis-à-jour pour rediriger les requêtes vers la nouvelle location du volume.

Les clones servent également à effectuer des sauvegardes du système de fichiers. La restauration des fichiers du dossier personnel des dernières 24 heures d'un utilisateur est possible à partir d'un dossier sans aide de l'administrateur. Les sauvegardes usuelles sur bande sont également possible. L'utilisation de la copie-sur-écriture sur

les clones permet d'économiser de l'espace disque en utilisant une stratégie de stockage incrémentale des clones. Les fichiers communs entre 2 clones sont marqués en copie-sur-écriture et seules les différences avec le précédent clone sont sauvegardées.

4.3 WAFL

Write Anywhere File Layout [HLM94] (*WAFL*) est un système de fichiers conçu spécifiquement pour fonctionner sur un serveur de fichiers en réseau (NFS). WAFL a été conçu pour fournir un service NFS rapide, pour supporter des fichiers de grande taille (supérieur à 10 Go), pour être performant tout en supportant de la redondance matérielle (RAID) et pour pouvoir redémarrer rapidement même après une coupure de courant ou un crash.

WAFL utilise 3 fichiers pour stocker les métadonnées associées à chaque fichier : le *inode file* qui contient tous les inodes du système de fichiers, le *block-map file* qui contient les blocs disponibles et le *inode-map file* qui contient les inodes disponibles.

Le système de fichiers WAFL est pensé comme un arbre de blocs (cf. Figure 8). À la racine de l'arbre se trouve l'inode racine qui décrit le *inode file*. Le *inode file* contient les inodes qui décrivent les fichiers y compris le *block-map file* et le *inode-map file*. Les feuilles de l'arbre sont les blocs de données des fichiers.

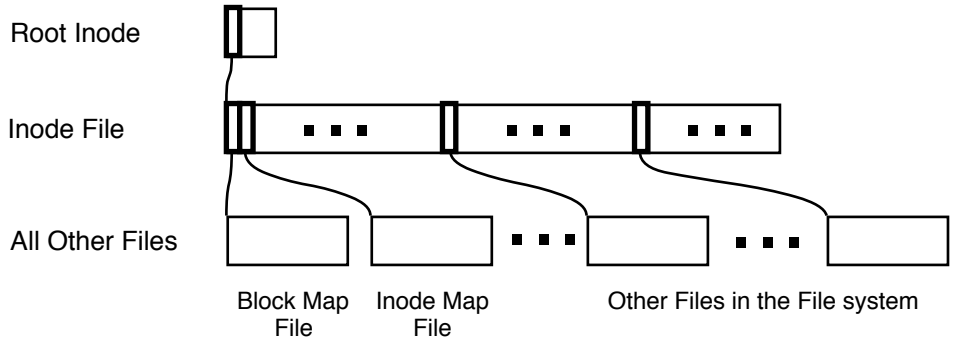

FIGURE 8 – Architecture de l'arbre des blocs de WAFL (Figure extraite de [HLM94]).

Pour créer un snapshot, WAFL duplique simplement l'inode racine afin d'avoir une référence sur l'arbre des blocs. La figure 9 (a) montre le système de fichiers avant le snapshot et la figure 9 (b) après le snapshot seul l'inode racine a été dupliqué pour référencer le snapshot. Lorsque le bloc de données D est modifié (Figure 9 (c)), WAFL utilise la copie-sur-écriture logicielle pour créer un nouveau bloc D' et modifie le système de fichiers actif pour qu'il pointe vers D'. L'inode racine du snapshot référence toujours le bloc D. Cet technique permet de créer un snapshot rapidement tout en consommant peu de mémoire.

La figure 10 illustre plus précisément la transition entre la figure 9 (b) et 9 (c). Quand un bloc est modifié, son un nouveau bloc est alloué et contenu y est écrit. Les blocs du chemin partant de l'inode racine du système de fichiers actif sont tous copiés pour que son arbre des blocs pointe vers le nouveau bloc. Cette technique laisse inchangé le chemin du snapshot menant à l'ancien bloc de données.

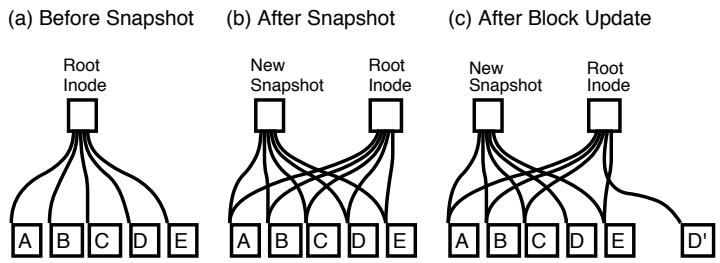

FIGURE 9 – Création d'un snapshot (Figure extraite de [HLM94]).

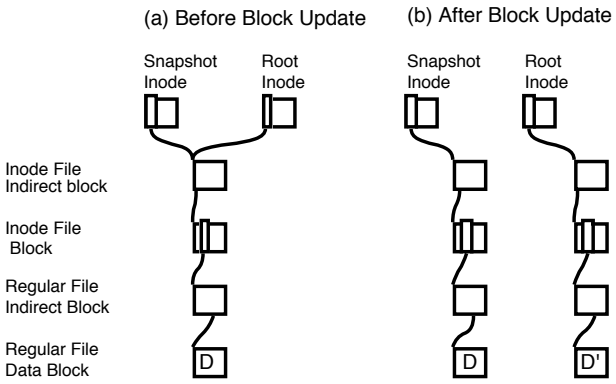

FIGURE 10 – Modification de l'arbre des blocs lors de l'écriture d'un bloc (Figure extraite de [HLM94]).

4.4 Google File System

Le système de fichiers Google [GGL03] (GFS) est un système de fichiers distribué passant à l'échelle conçu spécialement pour les besoins de Google pour les applications distribués nécessitant beaucoup de données pour s'exécuter. Il assure la tolérance aux fautes, est pensé pour être déployé sur du matériel usuel (même disques que pour un usage particulier), délivre des hautes performances pour un nombre important de clients connectés et assure une réplication des données pour la tolérance aux fautes.

L'architecture de GFS consiste en un serveur principal (le *maître*) et de multiples *chunkservers*. Les fichiers sont divisés en morceaux (*chunks*) de taille fixe. Chacun est identifié par un unique numéro (*chunk handle*). Les chunkservers stockent les chunks sur leurs disques locaux et se chargent de la lecture ou de l'écriture des données des chunks spécifié par des chunk handle. Le maître maintient toutes les métadonnées du système de fichiers. Il contrôle également les activités du système telles que l'espace de noms, les informations sur le contrôle d'accès aux fichiers, la correspondance entre un fichier et ses chunks et la localisation des chunks. Le code côté client est lié dans chaque application pour permettre l'utilisation de GFS mais n'est pas compatible POSIX.

La figure 11 illustre l'accès à un fichier. Le client contacte le maître en lui envoyant le nom d'un fichier avec le numéro du chunk souhaité (déduit à l'aide du code côté client). Le maître lui répond en lui renvoyant le chunk handle et les localisations de la donnée et de ses répliques que le client met en cache. Le client envoie ensuite le chunk handle et l'intervalle de données souhaité contenu dans le chunk au chunkserver le plus

proche disposant d'une réplique. Le chunkserver répond au client en lui renvoyant la donnée demandée. Lors d'une prochaine demande concernant le même chunk, le client n'aura pas à interagir avec le maître car les informations seront déjà présentes dans son cache.

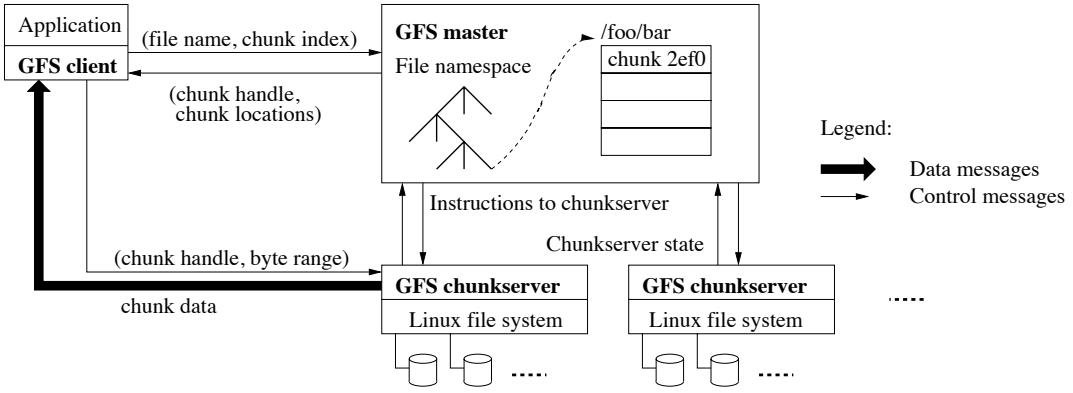

FIGURE 11 – Architecture de GFS (Figure extraite de [GGL03]).

Lors de la modification d'un chunk, le maître assigne un bail à l'une des répliques (appelé par la suite *primaire*) afin d'assurer la cohérence entre les répliques et de réduire la charge du maître. Le primaire ordonne les modifications reçues par les clients et fait en sorte que les répliques suivent le même ordre de modification afin que les répliques restent consistantes entre elles. Le bail a un timeout de 60 secondes pendant lequel le primaire peut renouveler le bail auprès du maître ou bien être révoqué par celui-ci (e.g. quand le maître veut interdire les modifications d'un fichier en train d'être renommé). Si le maître perd la communication avec le primaire, il peut réémettre un bail sans problème après 60 secondes à une réplique.

GFS fournit une opération de *snapshot* qui permet de faire une copie d'un fichier ou d'un répertoire quasiment instantanément en minimisant les interruptions des futures modifications des fichiers. Les utilisateurs se servent des snapshots pour copier des gros ensembles de données ou pour sauvegarder des données avant d'expérimenter des modifications qui pourront être confirmées ou annulées en revenant en arrière. Chaque chunk dispose d'un compteur de références qui comptabilise le nombre de snapshot référençant ce chunk. Le chunk est désalloué quand le compteur de références vaut 0.

GFS utilise une technique de copie-sur-écriture logicielle pour implémenter les snapshots. Lorsque le maître reçoit une demande, il révoque tous les baux associés aux chunks concernant le snapshot et s'assure que toutes les écritures en attente sur les chunks soient effectuées. Toute tentative ultérieure d'écriture devra passer par le maître. Le maître log ensuite sur disque les modifications et applique cet enregistrement en dupliquant les métadonnées pour les fichiers et les répertoires concernés qui pointent également sur les mêmes chunks que les fichiers sources. Le compteur de références de chaque chunk de tous les fichiers faisant partie du snapshot est incrémenté.

Lorsqu'un client souhaite écrire sur un des chunks faisant partie du snapshot, le maître s'aperçoit que le compteur de références du chunk est supérieur à 1. Il choisit donc un nouveau chunk handle et indique à tous les chunkservers possédant une réplique de dupliquer le chunk demandé. La modification se déroule ensuite de la même

manière que précédemment (attribution d'un bail et ordonnancement des modifications sur les répliques) sans que l'utilisateur ne s'aperçoive qu'un nouveau chunk a été créé.

4.5 Zettabyte File System

ZFS [RT03] [BAH+03] est un système de fichiers assurant l'intégrité des données ainsi que la possibilité de gérer une immense capacité de stockage (2^{128} blocs). ZFS fournit 3 composants qui s'insèrent entre le système de fichiers virtuel (VFS) et les drivers des disques de stockage (cf. Figure 12) : le *ZFS Posix Layer* (ZPL), le *Data Management Unit* (DMU) et le *Storage Pool Allocator* (SPA).

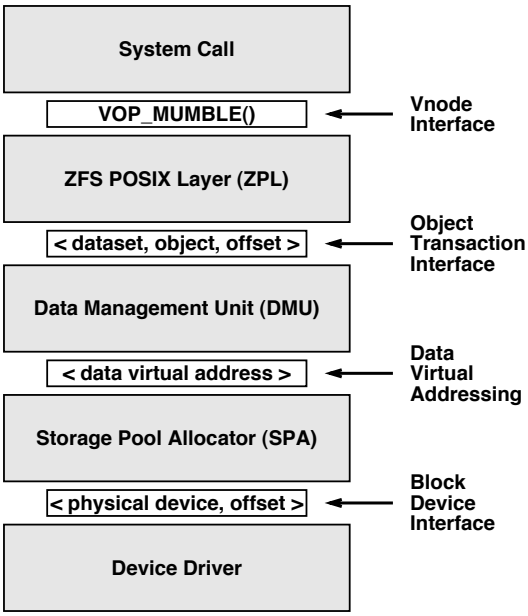

FIGURE 12 – Interface de ZFS entre le VFS et les drivers des périphériques de stockage (Figure extraite de [BAH+03]).

Le SPA alloue les blocs depuis tous les périphériques de stockage locaux dans un pool de stockage et présente une interface pour allouer et libérer les blocs au DMU. Le SPA utilise des adresses virtuelles de blocs (DVAs) pour adresser les blocs des périphériques et pour abstraire la notion de périphériques aux couches supérieures. Une DVA est stockée sur 128 bits ce qui permet à ZFS de stocker un nombre de blocs important. L'espace d'un système de fichiers ne dépend donc plus de la taille du périphérique utilisé car le SPA se charge de distribuer les blocs sur les périphériques de manière transparente.

Un arbre constitué de blocs et contenant les métadonnées permet l'accès aux fichiers. La racine de l'arbre est le **überblock** et les feuilles sont les blocs de données qui constituent les fichiers. Avant chaque écriture sur disque, le SPA calcule une somme de contrôle pour chaque bloc de données à partir de celui-ci afin de garantir l'intégrité des données. La somme de contrôle est calculée à partir du bloc et du parent de celui-ci est finalement stockée dans le parent afin que la somme soit auto-validante. Seul

le überblock n'a pas de parent pour calculer sa somme de contrôle. À la place, il est répliqué plusieurs fois sur le disque.

Le DMU se sert des blocs que le SPA lui présente. Il exporte les objets (i.e. les fichiers) au ZPL. Le ZPL est l'interface POSIX offerte au système par ZFS pour manipuler les fichiers, les permissions, implémenter les appels systèmes tels que *mmap()*,.... Le DMU préserve la cohérence des blocs en utilisant un modèle transactionnel de mise à jour des blocs reposant sur une technique de copie-sur-écriture logicielle. Lorsqu'une partie d'un bloc est modifiée, un nouveau bloc est alloué et entièrement copié en intégrant les nouvelles modifications (cf. Figure 13). Le chemin partant du bloc de données jusqu'à l'überbloc sont également copiés dans de nouveaux blocs et leur somme de contrôle recalculée (cf. Figure 14). Finalement, le überbloc est également copié et remplacé de façon atomique à la racine de l'arbre, remplaçant l'arbre précédent par le nouvel arbre à partir duquel les modifications sont visibles (cf. Figure 15). Une transaction est représentée par l'écriture de tous les blocs impliqués et par la réécriture de l'überblock . Si une erreur se produit lors du remplacement de l'überbloc, celui-ci est remplacé par l'une de ses répliques situées sur le disque. Le DMU regroupe plusieurs transactions ensemble pour minimiser les écritures sur le disque en ne réécrivant qu'une fois le überblock et les blocs indirects pour plusieurs transactions.

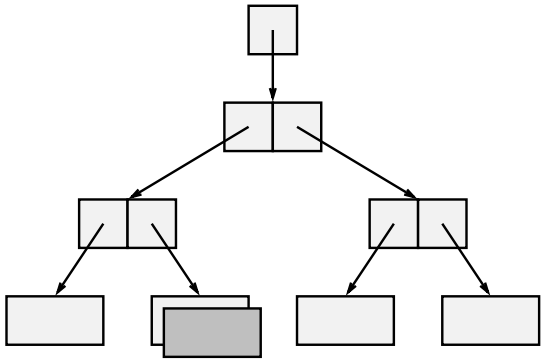

FIGURE 13 – Copie-sur-écriture du bloc de données (Figure extraite de [BAH+03]).

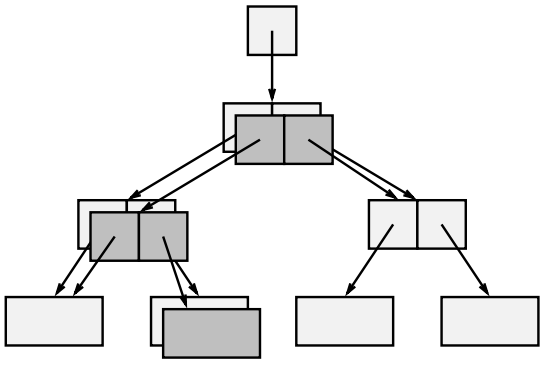

FIGURE 14 – Copie-sur-écriture des blocs indirects (Figure extraite de [BAH+03]).

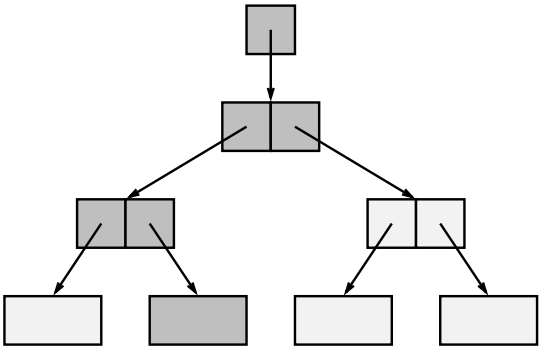

FIGURE 15 – Réécriture atomique de l'überblock (Figure extraite de [BAH+03]).

4.6 Récapitulatif

Les systèmes de fichiers utilisent le mécanisme de copie-sur-écriture pour créer des snapshots ou pour assurer la cohérence des données lors d'écritures sur disque. La copie-sur-écriture permet de laisser les fichiers faisant partie d'un snapshot intacts tout en partageant les blocs de données avec le système de fichiers actif. La cohérence des données pour ZFS est assurée par l'écriture de nouveaux blocs à chaque modification de fichiers puis par le remplacement des anciens blocs par les nouveaux. Il est évident que l'implémentation de la copie-sur-écriture ne peut être que logicielle dans ce cas : la MMU et le système d'exploitation peuvent surveiller l'accès aux pages mémoires d'un processus mais pas les droits d'accès aux fichiers du système de fichiers.

Le tableau 3 récapitule les 4 systèmes de fichiers présentés dans cette section selon 3 critères :

- *Serveur(s) de noms* désigne l'architecture des serveurs de noms du système de fichiers. Un serveur de noms centralisé connait l'emplacement de tous les fichiers dans le système. Un serveur de noms distribué est un regroupement de serveurs dont chacun d'eux possède une vue partielle du système de fichiers.
- *Serveur(s) de stockage* représentent la manière dont sont stockés les fichiers sur le système. Un serveur de stockage local indique qu'un seul et unique serveur possède l'ensemble des fichiers du système. Un serveur de stockage distribué indique que c'est un regroupement de serveurs qui possède chacun une partie des fichiers du système.
- Le *but de la copie-sur-écriture* indique à quoi sert la copie-sur-écriture dans le système de fichiers. La copie-sur-écriture est utilisée pour pouvoir prendre des instantanés du système de fichiers à un moment précis (Snapshot) ou bien pour garantir la cohérence des données lorsqu'une modification est faire sur un fichier (Cohérence).

1. Dans le cas d'AFS, les fonctionnalités de serveurs de noms et de serveurs de stockage sont remplies par les mêmes machines.

	AFS	GFS	WAFL	ZFS
Serveur(s) de noms	Distribués	Centralisé	Centralisé	Centralisé
Serveur(s) de stockage	Distribués [1]	Distribués	Local	Local
But de la copie-sur-écriture	Snapshot	Snapshot	Snapshot	Cohérence

TABLE 3 – Tableau récapitulatif des solutions de systèmes de fichiers

5 Mise en oeuvre de la copie-sur-écriture dans d'autres domaines

Cette section illustre l'utilisation de la copie-sur-écriture dans différents domaines ainsi que son illustration par au moins un exemple pour chaque domaine cité.

5.1 Exécution symbolique

KLEE [CDE08] est outil d'exécution symbolique capable de vérifier des programmes et de générer automatiquement des tests permettant une couverture importante du code. L'exécution symbolique permet d'analyser des programmes en les exécutant avec en entrée des valeurs "symboliques" (pouvant remplacer n'importe quelle valeur) plutôt que des valeurs concrètes et figées.

En pratique, le nombre d'états parcourus grandit très rapidement car à chaque branchement, le flot d'exécution se sépare en 2 (l'un où le branchement réussi et l'autre où il échoue) et dupliquer complètement l'état du programme pour que les 2 versions continuent leurs exécutions sans interférer est coûteux. KLEE utilise donc la copie-sur-écriture logiciel lors de la modification des objets faisant partie d'un état pour les représenter de façon compacte lors de la séparation du flot d'exécution.

Les objets de chaque état sont stockés dans un arbre équilibré et chaque noeud contient un objet et un compteur de références. Le compteur de références permet de savoir quand un objet n'est plus référencé par aucun état et peut être désalloué. Tous les objets sont en copie-sur-écriture et chaque modification, ajout ou suppression d'un objet implique d'en créer un nouveau. La figure 16 représente l'arbre des objets de l'état *E1* et de l'état *E2* juste après que leur exécution ait divergé. Les 2 états partagent le même arbre car aucune modification sur les objets n'a encore été faite. Supposons que l'état *E2* continue son exécution et modifie l'objet *O* ayant la valeur 24 par 42. Comme le montre la figure 17, un nouvel objet *O'* est alloué avec la valeur 42. *O'* référence les mêmes fils que *O* pour qu'ils soient toujours accessibles par l'état *E2* et leur compteur de références est incrémenté. Le chemin d'accès de la racine de l'arbre vers *O'* doit être copié afin que *E1* contienne *O* et que *E2* contienne *O'* dans leur arbre d'objets.

Cloud9 [BUZC11] est une plateforme de test automatisée distribuée. Cloud9 se base sur KLEE et s'en sert moteur d'exécution symbolique en le parallélisant pour pouvoir passer à l'échelle et analyser des logiciels que KLEE ne pourrait prendre en charge seul. Il peut traiter les programmes mono- et multi-threadés ainsi que les systèmes distribués et modélise un environnement qui permet de supporter les aspects majeurs de la bibliothèque POSIX.

Comme décrit précédemment, KLEE utilise la copie-sur-écriture pour permettre le partage d'objets entre les états symboliques. Cloud9 étend cette fonctionnalité en permettant à plusieurs processus appartenant au même état de pouvoir partager des objets entre eux en créant des domaines de copie-sur-écriture (*CoW domains*). Cette technique permet d'économiser de la mémoire lorsque le modèle de la bibliothèque POSIX de Cloud9 doit simuler l'appel système *fork()* : les objets des espaces d'états du père et du fils sont partagés en copie-sur-écriture.

Récapitulatif : KLEE et Cloud9 effectuent tous 2 de la vérification de programmes et utilisent la copie-sur-écriture afin d'économiser l'espace mémoire occupé par les

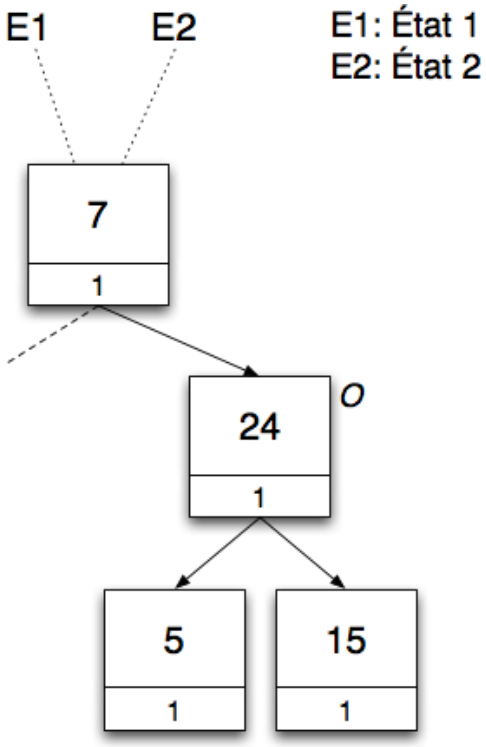

FIGURE 16 – Arbre des objets avant la modification de l'objet 24. Le trait en pointillé court désigne la racine de l'arbre des objets de l'état correspondant.

états lors de l'exécution symbolique. KLEE utilise la copie-sur-écriture pour partager les objets communs entre les états. Cloud9 étend ce mécanisme en partageant également les objets communs entre processus faisant partie d'un même état. L'implémentation de la copie-sur-écriture est logicielle car en dépit du coût des indirections lors de l'accès aux objets, cette technique économise plus de mémoire que l'implémentation matérielle grâce à la granularité qui s'effectue au niveau objet et pas au niveau de la page.

5.2 Exécution spéculative

L'exécution spéculative permet d'exécuter des tâches séquentielles de façon concurrente en prédisant le résultat d'une opération (e.g. une interaction avec l'environnement extérieur) afin de continuer son exécution. Lorsque l'opération est terminée, soit la prédiction est en accord avec le résultat et la tâche continue ou soit la prédiction est fausse et la tâche doit revenir à l'état précédent le début de l'exécution spéculative. Cette technique repose sur une *politique* qui spécifie quelles opérations peuvent être prédites et quelles valeurs seront choisies pour continuer l'exécution et sur un *mécanisme* qui supporte l'exécution spéculative en assurant le retour en arrière en cas d'annulation et en délayant les interactions avec l'environnement.

L'article [WCF11] effectue de l'exécution spéculative entre les applications et le système d'exploitation et montre comment découpler la politique et le mécanisme. Le

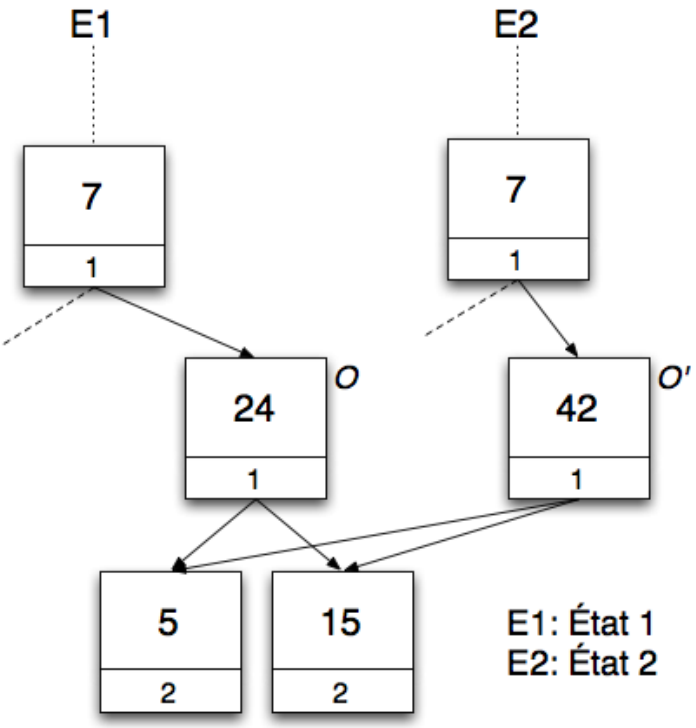

FIGURE 17 – Arbre des objets après la modification de l'objet 24. Le trait en pointillé court désigne la racine de l'arbre des objets de l'état correspondant.

mécanisme est implémenté dans le système d'exploitation pour pouvoir coordonner les exécutions spéculatives entre les applications et le noyau. Les décisions concernant la politique sont laissées à l'application qui dispose des informations sémantiques pour l'exécution spéculative. Lorsque la spéculation commence, le système doit s'assurer de pouvoir revenir en arrière si le résultat de l'opération n'est pas conforme à la prédiction. L'application est alors exécutée en copie-sur-écriture afin de pouvoir garder un checkpoint lors de l'annulation. Lorsque la spéculation s'exécute, les données modifiées sont dupliquées. Si la spéculation s'avère correcte, les données dupliquées remplacent les originales et l'exécution continue de façon non spéculative. Dans le cas contraire, les données spéculées sont supprimées et le checkpoint est restauré, l'application revient alors dans le même état qu'au point de spéculation et reprend son exécution en accord avec le résultat de l'opération.

Récapitulatif : L'exécution spéculative parallélise l'exécution d'un programme en anticipant la réponse du système d'exploitation lors d'un appel système. La copie-sur-écriture permet de checkpointer l'application et de l'exécuter spéculativement tout en gardant la possibilité de revenir en arrière au cas où la spéculation s'avèrerai incorrecte. Ce mécanisme est fortement analogue aux solutions de checkpointing présentée dans la section 3 et utilise donc une implémentation matérielle reposant sur le matériel de protection des pages.

5.3 Machine virtuelle Java

La JVM (Java Virtual Machine) est une machine virtuelle permettant d'exécuter des applications Java en interprétant le *bytecode* Java, un jeu d'instructions propre au langage. Les machines virtuelles au niveau langage permettent d'avoir un binaire unique pour une application et d'adapter la machine virtuelle pour chaque architecture.

Une application Java s'exécute dans son propre espace d'adressage et les fonctionnalités de la JVM sont typiquement disponible sous la forme d'une librairie partagée. Les espaces d'adressage ne sont pas partagés entre les applications Java et même si celles-ci utilisent les mêmes classes, il n'est pas possible de partager ces classes entre elles. Les serveurs hébergeant beaucoup d'applications Java pourraient diminuer la mémoire utilisée par ces applications en partageant entre elles les classes utilisées par plusieurs applications.

SLVM [WCD03] (Shared Library Virtual Machine) propose de factoriser l'espace occupé par les classes Java entre les différentes applications en les convertissant en bibliothèques partagées. SLVM convertit les représentations binaires des classes (le fichier *.class*) vers le format ELF (Executing and Linking Format) tout en conservant une représentation des données proche de celle de la machine virtuelle. SLVM optimise la construction de la librairie en maximisant les données en lecture seule (segment *.text* du fichier ELF) et en mettant le reste des données mutables ensemble (segment *.data*). La copie-sur-écriture est fournie par le mécanisme de chargement des bibliothèques partagées. Lorsqu'une application souhaite appeler une fonction, la JVM va ouvrir la bibliothèque associée avec *dlopen()* et charger les pages nécessaires à l'exécution. Le code (qui fait partie du segment *.text*) est immutable et sera chargé dans une page en lecture seule : lorsque d'autres applications feront appel à la même fonction, le code nécessaire sera déjà en mémoire et pourra être partagé. Les données (segment *.data*) associées à la bibliothèque seront quand à elles chargées dans une page en copie-sur-écriture. Lorsqu'une application modifiera l'une de ces pages, celle-ci sera dupliquée et associée uniquement avec l'application en question afin que les changements ne soient pas visibles par les autres applications de la machine virtuelle.

Récapitulatif : SLVM partage entre les classes de la librairie Java entre plusieurs instances d'une JVM afin d'économiser l'espace mémoire. SLVM transforme les classes Java en bibliothèques partagées et repose sur le système d'exploitation pour effectuer le partage entre les instances des classes Java. La copie-sur-écriture est donc effectuée par le système d'exploitation pour partager les bibliothèques en utilisant une implémentation matérielle.

5.4 Virtualisation du système d'exploitation

FVM [YGN+06] [YKLC08] (*Feather-weight Virtual Machine* : machine virtuelle poids plume) est une architecture de machine virtuelle système virtualisant le système d'exploitation (contrairement aux solutions telles que VMware ou Virtual Box qui virtualisent le matériel) pour Windows. FVM permet d'isoler l'exécution des applications afin de garantir que celles-ci ne corrompent pas le système d'exploitation. La technique principale de FVM est d'effectuer une virtualisation de l'espace de noms permettant d'isoler les systèmes de fichiers des machines virtuelles en renommant les ressources au niveau des appels systèmes.

Un répertoire privé est créé pour chaque machine virtuelle afin de virtualiser le système de fichiers. La copie-sur-écriture logicielle s'effectue en surveillant les appels

système demandant l'ouverture et la suppression des fichiers. Lorsqu'un fichier est ouvert en lecture, l'appel système renvoie un descripteur sur le fichier original tandis que s'il est ouvert en écriture, le fichier est copié dans le répertoire privé de la machine et le descripteur renvoyé pointe vers celui-ci. Un fichier créé par la machine virtuelle sera également placé dans le répertoire personnel. La détection de l'écriture se fait en analysant les arguments de l'appel système. Comme chaque lecture ou écriture se fait à travers le descripteur de fichier (qui désigne soit un fichier original du système, soit un fichier situé dans le répertoire personnel), il n'est pas nécessaire de surveiller les appels systèmes correspondants. La destruction d'un fichier du système d'exploitation est sauvegardé dans un fichier de *log* : lors d'une prochaine ouverture de ce même fichier, FVM indiquera qu'il n'existe plus (point de vue de la machine virtuelle) même s'il est présent physiquement sur le disque (point de vue du système d'exploitation).

La virtualisation de la base de registres (fichier où sont stockées de façon hiérarchique les options de configuration du système et des utilisateurs) s'effectue sur le même modèle. Un répertoire privé est créé dans la base pour chaque machine virtuelle et les appels systèmes relatifs à la manipulation des clés de la base de registres sont surveillés.

Récapitulatif : FVM virtualise le système d'exploitation pour donner l'illusion à des applications potentiellement dangereuses de s'exécuter directement sur le système tout en protégeant celui-ci. Les modifications apportées au système de fichiers et à la base de registres sont virtualisées en utilisant la copie-sur-écriture : les fichiers ou les clés sont copiés avant d'être modifiés par l'application. Ce mécanisme ressemble en tout point au mécanisme de snapshot de système de fichiers de la section 4 et repose sur une implémentation logicielle.

5.5 Migration de machines virtuelles

La migration de machines virtuelles systèmes dans un cluster est capitale pour assurer l'équilibrage de charge, la gestion des fautes ou encore la maintenance des machines. Cela implique cependant de suspendre les services fournis aux utilisateurs le temps que la migration soit effectuée. Les auteurs de [CFH+05] proposent une méthode de migration de machines virtuelles qui minimise le temps d'arrêt dû au transfert des données.

L'approche utilise une technique de *pré-copie* qui se déroule en 2 phases. Pendant la première phase, un maximum de pages est envoyé du site source vers le site destination pendant que la machine virtuelle continue de fonctionner. La deuxième phase consiste à stopper la machine virtuelle pour envoyer les pages restantes vers le site destination et à la redémarrer la machine virtuelle. Le problème réside dans le fait que pendant la première phase, certaines pages envoyées sur le site destination peuvent être modifiées sur le site source et doivent être réémises. Le mécanisme de copie-sur-écriture permet de détecter les pages modifiées qui ont déjà été envoyées au site destination. Lorsqu'une page est envoyée, son mode d'accès passe en lecture seule. Si cette page est accédée avant que la machine virtuelle ne soit stoppée et complètement migré, elle est mise en file d'attente pour être réexpédiée. La deuxième phase se déclenche une fois qu'un certain seuil de pages restantes à envoyer est atteint pour envoyer les pages qui sont fortement utilisées par le système et souvent modifiées.

Récapitulatif : La migration de machines virtuelles a pour but de migrer les machines virtuelles dans un cluster en minimisant le temps d'immobilisation du service. Les auteurs utilisent la *pré-copie* pour envoyer le maximum de pages de la machine virtuelle avant de la stopper et d'envoyer les pages restantes. La copie-sur-écriture permet de savoir si une page envoyée au site destination a été modifiée sur le site source et nécessite d'être réémise. L'implémentation de cet algorithme de copie est matérielle car le mécanisme de migration a besoin de détecter les modifications au niveau des pages de la machine virtuelle.

5.6 Optimisation de l'espace mémoire entre machines virtuelles

VMware ESX est un hyperviseur de type 1 permettant de multiplexer les ressources matérielles sans utiliser de système d'exploitation hôte parmi des machines virtuelles faisant tourner des systèmes d'exploitation non modifiés. [Wal02] présente plusieurs techniques permettant d'optimiser l'utilisation de la mémoire lors du fonctionnement en parallèle de multiples machines virtuelles sur le même système hôte.

L'une de ces techniques consiste à partager des pages identiques entre différentes machines virtuelles tout en garantissant l'isolation entre elles. Bien qu'à priori il puisse être difficile de trouver des données en commun entre les machines virtuelles, cela est possible notamment si celles-ci utilisent le même système d'exploitation. L'identification se fait en calculant une somme de contrôle sur chaque page toutes machines confondues. Lorsque 2 pages ont la même somme de contrôle, une vérification complète a lieu et si les 2 pages sont identiques, l'une d'entre elles est marqué en copie-sur-écriture (pour permettre de garantir l'isolation si une écriture survenait sur cette page) et l'autre est libérée.

Récapitulatif : VMware ESX est un moniteur de machines virtuelles de type 1 qui intègre des fonctionnalités de partage de mémoire entre machines virtuelles. Les pages ayant le même contenu (typiquement le code du noyau) sont partagées en copie-sur-écriture et copiées dès qu'une des machines effectue une modification sur l'une d'entre elles. La granularité du partage se situe au niveau des pages donc l'implémentation est matérielle.

5.7 Récapitulatif

Cette section expose des cas d'utilisation de la copie-sur-écriture dans des domaines de l'informatique tels que les machines virtuelles langages et systèmes, l'exécution spéculative d'applications, la vérification de programmes à l'aide de l'exécution symbolique et la virtualisation d'un système d'exploitation. Le tableau 4 récapitule les articles présentés dans cette section selon 3 critères :

- Le *thème de l'article* désigne le sujet dont traite l'article.
- Le *but de la copie-sur-écriture* rappelle la raison pour laquelle la copie-sur-écriture est utilisée dans l'article.
- L'*implémentation* présente si le type d'implémentation utilisée pour la copie-sur-écriture est matériel ou logiciel.

	Thème de l'article	But de la copie-sur-écriture	Implémentation
KLEE	Exécution symbolique	Partage inter-états d'objets	Logicielle
Cloud9	Exécution symbolique parallélisée	Partage inter- et intra-états d'objets	Logicielle
Exécution spéculative	Exécution spéculative	Checkpointing de processus	Matérielle
SLVM	Partage de mémoire entre machines virtuelles langages	Partage de classes entre machines virtuelles	Matérielle
FVM	Virtualisation du système d'exploitation	Virtualisation du système de fichiers et de la base de registres	Logicielle
Migration de VM	Migration de machines virtuelles systèmes	Détection des pages modifiées	Matérielle
VMware ESX	Partage de mémoire entre machines virtuelles systèmes	Détection et duplication des pages modifiées	Matérielle

TABLE 4 – Tableau récapitulatif des solutions de checkpointing

6 Conclusion

Ce document relate l'état de l'art de la copie-sur-écriture. La copie-sur-écriture est un mécanisme permettant d'optimiser le partage d'une ressource entre agents. La ressource est copiée uniquement lorsqu'un des agents souhaite la modifier afin que tous les autres agents gardent une vue consistante de la ressource. Cette technique permet d'économiser de la mémoire en minimisant le nombre d'exemplaires de la ressource et en délayant la copie au moment où l'application en a besoin.

Ce mécanisme peut être implémentée de façon matérielle ou logicielle. L'implémentation matérielle repose sur l'usage conjoint du matériel de protection des fautes de pages et du système d'exploitation. La MMU détecte l'accès aux pages contenant des objets en copie-sur-écriture et le gestionnaire des fautes de pages s'occupe de dupliquer la page lorsqu'elle est accédée en écriture. L'implémentation logicielle consiste à utiliser une structure d'accès aux objets marqués copie-sur-écriture afin d'instrumenter les écritures. La structure d'accès détecte les accès aux objets et les duplique lorsque ceux-ci sont modifiés.

L'implémentation matérielle est rapide dans l'accès aux objets car celui-ci est direct, est gros-grain car la copie d'un objet implique la copie d'une page et le développement est simple car la détection des accès aux objets est gérée par le matériel. L'implémentation logicielle est plus lente dans l'accès aux objets car la structure d'accès induit des indirections lors des lectures et des écritures, est à grain-fin car la copie s'effectue au niveau de l'objet et le développement est plus complexe à cause de l'implémentation de la structure d'accès.

La copie-sur-écriture est utilisée dans de nombreux domaines de l'informatique tels que le checkpointing de processus, les systèmes de fichiers, l'exécution spéculative, l'exécution symbolique, les machines virtuelles langages ou encore les moniteurs de machines virtuelles. Ce mécanisme est principalement utilisé afin d'économiser l'espace mémoire en partageant les objets utilisés entre les agents et pour améliorer le temps de réponse des programmes en repoussant la copie de l'objet au moment où il est nécessaire de la faire.

Bien que la copie-sur-écriture soit très répandue, elle n'est pas en soi un thème de recherche actuel. Elle est présente dans beaucoup d'articles de recherche mais souvent juste dans 1 ou 2 paragraphes et est utilisée comme moyen afin d'atteindre les résultats que les auteurs souhaitent obtenir. C'est pour cette raison que cet état de l'art n'explore pas un unique domaine de l'informatique en particulier mais plusieurs à la fois où la copie-sur-écriture est employée.

Références

[BAH+03] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum. The zettabyte file system. Technical report, Sun Microsystems, 2003.

[BUZC11] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 183–198, New York, NY, USA, 2011. ACM.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee : unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[CFH+05] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[CJLR06] Hadrien Cambazard, Narendra Jussien, François Laburthe, and Guillaume Rochart. The choco constraint solver. In *INFORMS Annual meeting*, Pittsburgh, PA, USA, November 2006.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[HKM+88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6 :51–81, February 1988.

[HLM94] Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.

[LBP+07] Oren Laadan, Ricardo A. Baratto, Dan B. Phung, Shaya Potter, and Jason Nieh. Dejaview : a personal virtual computer recorder. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 279–292, New York, NY, USA, 2007. ACM.

[LN07] Oren Laadan and Jason Nieh. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 25 :1–25 :14, Berkeley, CA, USA, 2007. USENIX Association.

[LNP94] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 5 :874–879, August 1994.

[OSSN02] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of Zap : a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36 :361–376, December 2002.

[PBKL95] J. S. Plank, M. Beck, G. Kingsley, and K. Li. **Libckpt** : Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.

[PLP98] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10) :972–986, October 1998.

[RT03] O. Rodeh and A. Teperman. zFS - a scalable distributed file system using object disks. In *Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, pages 207 – 218, april 2003.

[SPD+05] J.C. Sancho, F. Petrini, K. Davis, R. Gioiosa, and S. Jiang. Current practice and a direction forward in checkpoint/restart implementations for fault tolerance. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 8–pp. IEEE, 2005.

[Tea10] CHOCO Team. choco : an open source java constraint programming library. Research report 10-02-INFO, Ecole des Mines de Nantes, 2010.

[Wal02] Carl A. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the 5th symposium on Operating systems design and implementation Copyright restrictions prevent ACM from being able to make the PDFs for this conference available for downloading*, OSDI '02, pages 181–194, New York, NY, USA, 2002. ACM.

[WCD03] Bernard Wong, Grzegorz Czajkowski, and Laurent Daynès. Dynamically loaded classes as shared libraries : An approach to improving virtual machine scalability. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 38.2–, Washington, DC, USA, 2003. IEEE Computer Society.

[WCF11] Benjamin Wester, Peter M. Chen, and Jason Flinn. Operating system support for application-specific speculation. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 229–242, New York, NY, USA, 2011. ACM.

[YGN+06] Yang Yu, Fanglu Guo, Susanta Nanda, Lap-chung Lam, and Tzi-cker Chiueh. A feather-weight virtual machine for windows applications. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 24–34, New York, NY, USA, 2006. ACM.

[YKLC08] Yang Yu, Hariharan Kolam, Lap-Chung Lam, and Tzi-cker Chiueh. Applications of a feather-weight virtual machine. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 171–180, New York, NY, USA, 2008. ACM.